CLAWPACK Version 4.2
User's Guide

Randall J. LeVeque
University of Washington

December 20, 2003

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

Authors.

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

Randall 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 two-dimensional adaptive mesh refinement part of AMRCLAW was written by

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

and adapted to three dimensions with help from

David McQueen, Courant Institute, NYU
Donna Calhoun, University of Washington

The MPI and HDF routines were written by

Peter Blossey, University of Washigton

based on earlier contributions by

Sorin Mitran, Univeristy of North Carolina, Chapel Hill
Donna Calhoun, University of Washington

The three-dimensional MATLAB graphics routines were developed by

Donna Calhoun, University of Washington

Acknowledgments.

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, DMS-9505021,
DMS-96226645, DMS-9803442, and DMS-0106511
DOE Grants DE-FG06-93ER25181, DE-FG03-96ER25292, DE-FG02-88ER25053,
DE-FG02-92ER25139, DE-FG03-00ER2592, DE-FC02-01ER25474
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 **7**

1.1 Introduction . . . 7
1.2 Other references . . . 7
1.3 Versions . . . 8
1.4 Basic framework . . . 9
1.5 Obtaining CLAWPACK . . . 10
1.6 Getting started . . . 10
1.6.1 MATLAB graphics . . . 11
1.7 Using CLAWPACK — A guide through `example1` . . . 12
1.7.1 The main program (`driver.f`) . . . 12
1.7.2 The initial conditions (`qinit.f`) . . . 13
1.7.3 The `claw1ez` routine . . . 13
1.7.4 Boundary conditions . . . 13
1.7.5 The Riemann solver . . . 13
1.7.6 The input file `claw1ez.data` . . . 14
1.8 Other user-supplied routines and files . . . 17
1.9 Auxiliary arrays and `setaux.f` . . . 18
1.10 An acoustics example . . . 19
1.11 Two space dimensions (`claw2ez.f`) . . . 19
1.11.1 Riemann solvers . . . 20
1.12 Three space dimensions (`claw3ez.f`) . . . 21
1.12.1 Riemann solvers . . . 22
1.13 Restarting CLAWPACK . . . 25

2 Program output and graphics using MATLAB **27**

2.1 Output produced by `outN.f` . . . 27
2.2 HDF output produced by `outN_hdf.f` . . . 28
2.3 Plotting results using `plotclawN.m` . . . 30
2.4 Plotting parameters and `setplotN.m` . . . 31
2.5 Plotting AMRCLAW output . . . 35
2.6 The `UserVariable` option . . . 35
2.7 Creating 1d scatter and line plots from 2d and 3d data . . . 36
2.8 Nonuniform grids and the `MappedGrid` option . . . 37
2.9 Additional script files . . . 37
2.10 Changing plot characteristics directly from the command line . . . 38
2.11 Customizing `outN.f` . . . 38

3 Adaptive Mesh Refinement and the AMRCLAW Routines **41**
3.1 The input files `amr2ez.data` and `amr3ez.data` 41
3.1.1 Source terms and `src1d.f` 43
3.1.2 AMR parameters and the file `call.i` 44
3.1.3 Boundary conditions 44
3.2 Three dimensions 45
3.3 The adaptive algorithm 45
3.4 Error estimation and regridding 46
3.5 Comments and warnings 46

4 MPICLAW: an MPI version of CLAWPACK **49**
4.1 MPICLAW routines 50
4.2 Restarting MPICLAW 50

Chapter 1

The basic CLAWPACK software

1.1 Introduction

`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, including nonlinear systems of conservation laws. The software can also be used to solve nonconservative hyperbolic systems and systems with variable coefficients, as well as systems including source terms. The package includes an MPI version in which the domain can be distributed among multiple processors, and adaptive mesh refinement versions (AMRCLAW) in two and three space dimensions.

These notes describe many features of the software and ways in which it can be used, but only briefly review the numerical methods employed. A detailed description of the methods, with the same notation used here, can be found in the book *Finite Volume Methods for Hyperbolic Problems* [10]. This book also contains a general discussion of hyperbolic problems arising in several particular application areas. Numerous examples using CLAWPACK are presented in the book and source code for each can be found via the webpage

`http://www.amath.washington.edu/~claw/book.html`

Other applications of CLAWPACK can be found via the webpages

`http://www.amath.washington.edu/~claw/apps.html`

In most cases the easiest way to apply CLAWPACK to a problem of interest is to find an existing application to a similar problem, copy the relevant files to your own computer, and adapt them to your problem. The webpages above contain pointers to many directories that can be downloaded as tar files and contain everything needed for particular applications.

1.2 Other references

The one- and two-dimensional wave-propagation algorithms are described in the paper [8]. The three-dimensional algorithms are developed and analyzed in [6]. The ideas are presented for the relatively simple case of the advection equation in two and three dimensions in [7].

The adaptive mesh refinement routines used in `amrclaw` were developed with Marsha Berger. These are based on her work on adaptive refinement for conservation laws, particularly the Euler equations, [5], [2], [3]. 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 [4].

If you successfully use CLAWPACK in research that results in publications, please cite the webpage

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

and relevant papers on these algorithms. If you would like your publications, codes, or links listed on the `usage` webpage, please send email to `rjl@amath.washington.edu`.

1.3 Versions

Version 4.0 of CLAWPACK was introduced in 2000 with many substantial changes over previous versions. **Version 4.1** was installed in September, 2002 with only a few changes in the one-dimensional and two-dimensional routines.

More substantial changes were made in the basic three-dimensional routines to fix some bugs. No changes were made in calling sequences, except:

- The Riemann solver `rpn3.f` has the same calling sequence as before, but the arrays `auxl` and `auxr` are dimensioned differently, as described on page 22.
- A new Riemann solver `rptt3.f` has been added for "double transverse" propagation. This performs a function that used to be performed by `rpt3.f`, but splitting this off separately makes it easier to explain what it does and should make it easier to develop new Riemann solvers. See Section 1.12.1 for more details.
- All calls that previously had `rpt3` in the calling sequence now include `rptt3` as well.

A three-dimensional version of AMRCLAW was also introduced at this point, and some bugs were fixed in the two-dimensional AMRCLAW.

Check the webpage

`http://www.amath.washington.edu/~claw/changes.html`

for a summary of these and more recent changes.
Older versions can still be found at

`http://www.amath.washington.edu/~rjl/clawpack/`

Some applications that were implemented in earlier versions have never been converted to more recent versions, so this may still be of some value.

Version 4.2 was installed in September, 2003 with some bugs fixed and a number of new features:

- A new set of matlab graphics routines are available, particularly for 3d data; see Chapter 2.
- A restart facility has been incorporated in the basic clawpack routines (as was previously available in amrclaw, thought implemented somewhat differently), described in Section 1.13.
- Writing output files in hdf format rather than ascii has been simplified, see Section 2.2.
- The MPI versions of the clawpack routines have been substantially rewritten and are now in f77 form, more similar to the basic clawpack routines; see Chapter 4.

1.4 Basic framework

In one space dimension, the CLAWPACK routine `claw1` (or the simplified version `claw1ez`) can be used to solve a system of equations of the form

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

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.*,

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

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 $\mathcal{W}^p_{i-1/2}$ and speeds $s^p_{i-1/2}$ satisfying

$$\sum_{p=1}^{M_w} \mathcal{W}^p_{i-1/2} = Q_i - Q_{i-1} \equiv \Delta Q_{i-1/2}.$$

The Riemann solver must also return a left-going fluctuation $\mathcal{A}^-\Delta Q_{i-1/2}$ and a right-going fluctuation $\mathcal{A}^+\Delta Q_{i-1/2}$. In the standard conservative case (1.2) these should satisfy

$$\mathcal{A}^-\Delta Q_{i-1/2} + \mathcal{A}^+\Delta Q_{i-1/2} = f(Q_i) - f(Q_{i-1}) \tag{1.4}$$

and the fluctuations then define a "flux-difference splitting" as described in Chapter 4 of [10]. Typically

$$\mathcal{A}^-\Delta Q_{i-1/2} = \sum_p (s^p_{i-1/2})^- \mathcal{W}^p_{i-1/2}, \qquad \mathcal{A}^+\Delta Q_{i-1/2} = \sum_p (s^p_{i-1/2})^+ \mathcal{W}^p_{i-1/2}, \tag{1.5}$$

where $s^- = \min(s, 0)$ and $s^+ = \max(s, 0)$. 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

$$Q^{n+1}_i = Q^n_i - \frac{\Delta t}{\Delta x}\left[\mathcal{A}^+\Delta Q_{i-1/2} + \mathcal{A}^-\Delta Q_{i+1/2}\right], \tag{1.6}$$

assuming $\kappa \equiv 1$.

The Riemann solver must be supplied by the user in the form of a subroutine `rp1`, as described below. Typically the Riemann solver first computes waves and speeds and then uses these to compute $\mathcal{A}^+\Delta Q_{i-1/2}$ and $\mathcal{A}^-\Delta Q_{i-1/2}$ internally in the Riemann solver. The waves and speeds must also be returned by the Riemann solver in order to use the high-resolution methods described in Chapter 6 of [10]. These methods take the form

$$Q^{n+1}_i = Q^n_i - \frac{\Delta t}{\Delta x}\left[\mathcal{A}^+\Delta Q_{i-1/2} + \mathcal{A}^-\Delta Q_{i+1/2}\right] - \frac{\Delta t}{\Delta x}(\tilde{F}_{i+1/2} - \tilde{F}_{i-1/2}) \tag{1.7}$$

where

$$\tilde{F}_{i-1/2} = \frac{1}{2}\sum_{p=1}^{M_w} |s^p_{i-1/2}| \left(1 - \frac{\Delta t}{\Delta x}|s^p_{i-1/2}|\right) \widetilde{\mathcal{W}}^p_{i-1/2}. \tag{1.8}$$

Here $\widetilde{\mathcal{W}}^p_{i-1/2}$ represents a limited version of the wave $\mathcal{W}^p_{i-1/2}$, obtained by comparing $\mathcal{W}^p_{i-1/2}$ to $\mathcal{W}^p_{i-3/2}$ if $s^p > 0$ or to $\mathcal{W}^p_{i+1/2}$ if $s^p < 0$.

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

$$Q_i^{n+1} = Q_i^n - \frac{\Delta t}{\kappa_i \Delta x}\left[\mathcal{A}^+ \Delta Q_{i-1/2} + \mathcal{A}^- \Delta Q_{i+1/2}\right], \tag{1.9}$$

See Chapter 6 of [10] for discussion of this algorithm and its extension to the high-resolution method.

If the equation has a source term, a routine `src1` must also be supplied that solves the source term equation $q_t = \psi$ over a time step. A fractional step method is used to couple this with the homogeneous solution, as described in Chapter 17 of [10]. Boundary conditions are imposed by setting values in ghost cells each time step, as described in Chapter 7 of [10]. Some standard boundary conditions are implemented in the library routine `claw/clawpack/1d/lib/bc1.f`, or this can be modified to impose other conditions.

1.5 Obtaining CLAWPACK

The latest version of CLAWPACK can be downloaded 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 (see Chapter 2), 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/clawpack/1d/example1`

that illustrate the basic use of CLAWPACK. Many other examples can be found via the webpages mentioned in Section 1.1.

1.6 Getting started

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

Creating the directories

The files you download will be gzipped tar files. Before installing any of CLAWPACK, you should create a directory named `<path>/claw` where the pathname `<path>` depends on where you want these files to reside and the local naming conventions on your computer. You should download any CLAWPACK files to this directory. After downloading any file of the form `name.tar.gz`, execute the following commands in the directory `<path>/claw`:

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

This will create the appropriate subdirectories within `<path>/claw`.

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
```

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.

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.

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` that 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 file rather than everything.

To use the `Makefile`, simply type

```
unix> make
```

instead of `compile`.

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. In particular, if you type

```
unix> make program
```

a single file `claw1program.f` will be generated that contains the main program `driver.f` and all subroutines, giving a self-contained file. This may be useful for moving a program elsewhere, or to run some debuggers that cannot handle source code scattered between different files and directories.

1.6.1 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

```
>> 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.

See Chapter 2 for more information on the use of MATLAB for visualization, and about the format of output files that may be useful if using other graphics packages.

1.7 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

$$q(x, 0) = \exp(-\beta(x - 0.3)^2). \tag{1.10}$$

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 1.8

1.7.1 The main program (`driver.f`)

The main program for `example1` 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 value of these parameters depends 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 = 3` for the Euler equations.

`mwaves`: The number of waves produced in each Riemann solution, called M_w in the text. Often `mwaves = meqn` but not always.

`mbc`: The number of "ghost cells" used for implementing boundary conditions, as described in Chapter 7 of [10]. Setting `mbc = 2` is sufficient unless changes have been made to the CLAWPACK software that result in a larger stencil.

`mwork`: A work array `work` of dimension `mwork` is used internally by CLAWPACK for various purposes. The size of this array depends on the other parameters:
`mwork` $\geq$ `(maxmx + 2*mbc) * (2 + 4*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. This 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 approximation Q_i^n (a vector with `meqn` components) at each time t_n. The value of i ranges from 1 to `mx` where `mx` $\leq$ `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)`: Used for auxiliary variables if `maux > 0`. For example, in a variable-coefficient advection problem the velocity in the ith cell might be stored in `aux(i,1)`. See Section 1.9 for an example and more discussion.

If `maux = 0` then 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.

1.7.2 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 easiest to just evaluate this function at the center of the cell. This gives a value that agrees with the cell average to $\mathcal{O}((\Delta x)^2)$. 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 from claw1ez.

1.7.3 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 and 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 for this routine in the source code.

1.7.4 Boundary conditions

Boundary conditions must be set before each time step and claw1 calls a subroutine bc1 to accomplish this. The manner in which this is done is described in detail in Chapter 7 of [10]. 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 1.7.6 where the mthbc array is described).

1.7.5 The Riemann solver

The file claw/clawpack/1d/example1/rp1ad.f contains the Riemann solver. If claw1ez is used, then this subroutine must be named rp1. (More generally the name of the subroutine can be passed as an argument to claw1). The Riemann solver is the crucial user-supplied routine that 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 and then solve the Riemann problems between these values. This approach to high-resolution methods is discussed in Chapter 6 of [10].

Note that the Riemann problem at the interface $x_{i-1/2}$ between cells $i-1$ and i has data

$$\begin{aligned} &\text{left state: } Q_{i-1}^R = \texttt{qr}(\texttt{i}-1,:), \\ &\text{right state: } Q_i^L = \texttt{ql}(\texttt{i},:). \end{aligned} \tag{1.11}$$

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.

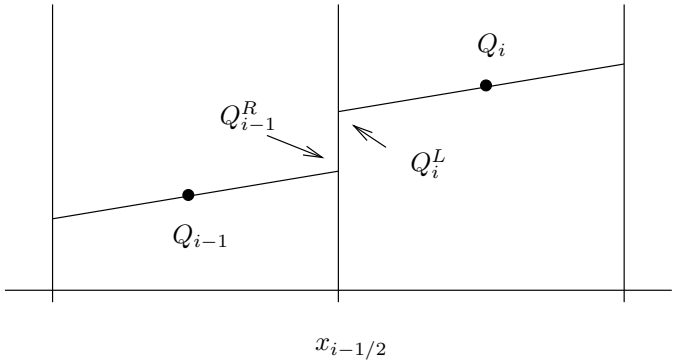

Figure 1.1: The states used in solving the Riemann problem at the interface $x_{i-1/2}$.

The Riemann solver `rp1` also has input parameters `auxl` and `auxr` that contain values of the auxiliary variables (see Section 1.9). Normally `auxl = auxr = aux` when the Riemann solver is called from the standard CLAWPACK routines.

The routine `rp1` must solve the Riemann problem for each value of `i`, and return the following:

`amdq(i,1:meqn)` The vector $\mathcal{A}^-\Delta Q_{i-1/2}$ containing the left-going fluctuation as described in Section 1.4.

`apdq(i,1:meqn)` The vector $\mathcal{A}^+\Delta Q_{i-1/2}$ containing the right-going fluctuation as described in Section 1.4.

`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 typically used in place of p.)

`s(i,p)` The wave speed $s_{i-1/2}^p$ for each 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) as described in Chapter 6 of [10].

The values in any auxiliary arrays (see Section 1.9) are also passed into the Riemann solver routine since these arrays typically contain spatially-varying information that is needed in solving the Riemann problem. For consistency with the `ql` and `qr` notation, two arrays `auxl` and `auxr` are passed in, but in the standard CLAWPACK implementation these arrays are identical and simply agree with `aux` in the main routine. So you can also use `auxl(i,:)`, for example, as the value in the ith grid cell.

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

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

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

1.7.6 The input file `claw1ez.data`

The `claw1ez` routine reads data from a file named `claw1ez.data`. Take a look at `claw/clawpack/1d/example1/claw1ez.data`, which is a typical input file. One or more values is read from each line of this file. Any text following these values 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 `t0 + (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. The value of `nout` is ignored. 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. This option is also useful for debugging if you want to force the solution to be output every time step, by setting `nstepout = 1`.

dtv(1): The initial value of Δt used in the first time step. If `method(1) = 0` below, 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 method (1.6) is used.

`method(2) = 2`: High-resolution correction terms (1.8) 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 which high-order correction terms are applied.

`method(4)`: The verbosity. This controls the amount of output printed by `claw1` on the screen as CLAWPACK progresses.

`method(4) = 0`: Information is printed only when output files are created.

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

`method(4) = L`: For AMRCLAW, if $L \geq 1$ then information is printed every time step for grids on level L and coarser. (Where level 1 is the coarsest grid — see Chapter 3.)

`method(5)`: Tells whether there is a source term in the equation. If so, then a fractional step method may be used as described in Chapter 17 of [10]. 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`. See Section 1.8 for more discussion.

`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 for reasons discussed in Chapter 17 of [10].

`method(6)`: Tells whether there is a "capacity function" in the equation, as discussed in [8] and [10].

`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 Chapter 6 of [10].

`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. The value set in `claw1ez.data` should agree with the value set in `driver.f`.

`mwaves`: The number of waves in each Riemann solution. This is often equal to `meqn` but need not be. The value set in `claw1ez.data` should agree with the value set in `driver.f`.

`mthlim(1:mwaves)`: The limiter to be applied in each wave family as described in Chapter 6 of [10]. 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 can be added by modifying the routine `claw/clawpack/1d/lib/philim.f`, which is called by `claw/clawpack/1d/lib/limiter.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. See Chapter 7 of [10] for more description of these and how they are implemented. 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; see Chapter 7 of [10].

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

1.8 Other user-supplied routines and files

Several other routines may be provided by the user but are not required. In each case there is a default version provided in the library `claw/clawpack/1d/lib` that does nothing but `return`. If you wish to modify this code to do something more interesting, copy the library version to the application directory, modify it as required, and also modify the `Makefile` to point to the modified version rather than to the library version.

`setprob.f` The `claw1ez` routine always calls `setprob` at the beginning of execution. The user can provide a subroutine that sets any problem-specific parameters or does other initialization.

As an example, for the advection problem solved in `example1`, this is used to set the advection velocity u. This value is stored in a common block in `setprob.f` that is also accessible from the Riemann solver `rp1ad.f`, where the value is needed. Similarly, the parameter `beta` is stored in a common block that is accessible from `qinit.f`, where it is used in setting the initial data according to (1.10).

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`. See Section 1.9.

`b4step1.f` Within `claw1` there is a call to a routine `b4step1` before each call to `step1` (the CLAWPACK routine that actually takes a single time step). The user can supply a routine `b4step1` in place of the default routine `claw/clawpack/1d/lib/b4step1.f` in order to perform additional tasks that might be required each time step. One example might be to modify the `aux` array values each time step, as described in Section 1.9.

`src1.f` If the equation includes a source term ψ as in (1.1), then a fractional step method can be used to alternate between solving the homogeneous hyperbolic problem and the equation $q_t = \psi$. In this case an appropriate routine `src1.f` must be provided in place of the default routine `claw/clawpack/1d/lib/src1.f`. This routine must solve the equation $q_t = \psi$ over one time step. Often this requires solving an ordinary differential equation in each grid cell. In some cases a partial differential equation must be solved. For example, if diffusive terms are included with $\psi = q_{xx}$, then the diffusion equation must be solved over one time step in the routine `src1.f`.

Alternative approaches to incorporating source terms might require modifying the Riemann solver rather than using a fractional step method. See, for example, [9] or [1].

1.9 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)`. See Chapter 9 of [10] for more discussion of variable-coefficient problems.

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. For some examples of the use of auxiliary arrays, see

```
claw/clawpack/applications/advection/1d/conservative/example1
claw/clawpack/applications/acoustics/1d/varying/interface
```

In some problems the values stored in the `aux` arrays must be time-dependent, for example in an advection equation of the form $q_t + u(x,t)q_x = 0$. The routine `setaux` is normally called only once at the beginning of the computation and is not called to modify values later. The user can supply a routine `b4step1` in place of the default routine `claw/clawpack/1d/lib/b4step1.f` in order to modify the `aux` array values each time step, or call `setaux` if desired. The `setaux` routine was originally designed to be called only at the initial time, and so the current time is not passed into this routine. If the value of `time` is required in `setaux`, it should be passed in a common block from `b4step1`, or use the common block `comxt` that is already set in `claw1.f` to contain the current time as well as the values of `dx` and `dt`. This common block is defined as

```
common /comxt/ dtcom,dxcom,tcom
```
and you can include a corresponding statement in any subroutine where you want access to these values. This may also be useful in the Riemann solver if the current time or mesh spacing are required, since these are not passed in.

1.10 An acoustics example

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

$$\begin{aligned} p_t + K u_x &= 0 \\ \rho u_t + p_x &= 0. \end{aligned} \tag{1.12}$$

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:

$$\begin{aligned} \lambda^1 &= -c, & \lambda^2 &= c \\ r^1 &= \begin{bmatrix} -Z \\ 1 \end{bmatrix}, & r^2 &= \begin{bmatrix} Z \\ 1 \end{bmatrix}. \end{aligned} \tag{1.13}$$

Solving the Riemann problem between states Q_{i-1} and Q_i gives $\alpha = R^{-1}(Q_i - Q_{i-1})$ with components

$$\begin{aligned} \alpha^1 &= (-(p_i - p_{i-1}) + Z(u_i - u_{i-1}))/2Z, \\ \alpha^2 &= ((p_i - p_{i-1}) + Z(u_i - u_{i-1}))/2Z, \end{aligned} \tag{1.14}$$

and the waves are $\mathcal{W}^1 = \alpha^1 r^1$ and $\mathcal{W}^2 = \alpha^2 r^2$.

1.11 Two space dimensions (`claw2ez.f`)

In two space dimensions the equation (1.1) is extended to

$$\kappa(x,y) q_t + f(q)_x + g(q)_y = \psi(q,x,y,t), \tag{1.15}$$

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

$$q_t + f(q)_x + g(q)_y = 0. \tag{1.16}$$

Again hyperbolic systems that are not in conservation form, *e.g.*,

$$q_t + A(x,y,t) q_x + B(x,y,t) q_y = 0, \tag{1.17}$$

can also be solved.

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 [8] and Chapter 21 of [10] 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)`.

1.11.1 Riemann solvers

Two Riemann solvers must now be provided, as described in [8] and Chapter 21 of [10]. The transverse solver `rpt2` is called only if `method(3) > 0` and a dummy routine could be provided otherwise, *e.g.*, the library routine `claw/clawpack/2d/lib/rpt2.f`.

`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 $q_t + A(x, y, t)q_x = 0$ from (1.17).

`ixy = 2`: Slice is in the y-direction and this routine should return the solution to the Riemann problem $q_t + B(x, y, t)q_y = 0$ from (1.17).

As in one dimension, two arrays `ql` and `qr` are passed in and normally `ql = qr` and contains the cell-centered averages. However, if you plan to use AMRCLAW at some point for adaptive refinement, it is important that you use `qr(i-1,:)` as the left state value in the `i`'th Riemann problem and `ql(i,:)` as the right state value. This is because `rpn2` is also called in a special way at each interface between grids at different refinement levels, and in this call `ql` and `qr` are not the same.

If `aux` arrays are being used, then two arrays `auxl` and `auxr` are also passed in to `rpn2`. As with `ql` and `qr`, these are identical in the standard CLAWPACK calls to `rpn2`, but for extension to AMRCLAW it is important to use `auxr(i-1,:)` as the auxiliary array in the left state and `auxl(i,:)` as the auxiliary array in the right state.

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.

ixy = 1: Slice is in the x-direction, so asdq=$\mathcal{A}^*\Delta Q$ should be split into bmasdq= $\mathcal{B}^-\mathcal{A}^*\Delta Q$ and bpasdq= $\mathcal{B}^+\mathcal{A}^*\Delta Q$.

ixy = 2: Slice is in the y-direction, so asdq=$\mathcal{B}^*\Delta q$ should be split into bmasdq= $\mathcal{A}^-\mathcal{B}^*\Delta Q$ and bpasdq= $\mathcal{A}^+\mathcal{B}^*\Delta Q$.

Here $\mathcal{A}^*\Delta Q$ represents either $\mathcal{A}^-\Delta Q$ or $\mathcal{A}^+\Delta Q$. The parameter imp indicates whether asdq represents amdq or apdq (assuming ixy=1. Otherwise it represents bmdq or bpdq):

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

imp = 2: asdq represents apdq, the fluctuation that 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.

Note that in rpt2 only a single copy of each aux value is passed in, not two copies auxl and auxr as in rpn2, and auxN(i,:) gives the aux values in cell i.

1.12 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 claw3ez.data. The main changes from two dimensions are the following:

method(3): Setting method(3) = -1 gives dimensional splitting with the Godunov splitting, *i.e.*, a full time step is taken in x, then y, and then z. Strang splitting is not currently implemented in three dimensions.

In three dimensions, dimensional splitting is often much more efficient than the unsplit methods obtained with other choices of method(3). To obtain good results with the unsplit method it is often necessary to use full transverse and double-transverse propagation of all waves (*i.e.*, method(3) = 22 as described below), which results in a large number of transverse Riemann problems being solved in each grid cell every time step. For some problems this is worth doing, but often better results can be obtained with equal or less computational work by using dimensional splitting on a finer grid.

Setting method(3) = 0 gives the unsplit method but with no transverse splitting. This method is only first-order accurate and generally gives poor results.

If method(3) > 0 then the routine rpt3 is required and is used to do transverse splitting. In three dimensions the value of method(3) > 0 should be a two-digit integer. The first digit is 1 or 2 and specifies whether just the increment wave or both increment and correction wave are

transversely propagated. This digit plays the same role as the value of `method(3)` does in two dimensions.

In three dimensions one can apply "double-transverse" corrections as well as transverse corrections, as described in [6]. 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. This is indicated by the second digit of `method(3)`. This digit takes the value 0, 1, or 2 depending on whether no wave, just the increment wave, or both increment and correction waves should be propagated in the double-transverse sense.

To summarize, if `method(3) > 0` then it should take one of the following values. (Note that some of these choices lead to unconditionally unstable methods and are not recommended. At least they are unstable on linear problems by von Neumann analysis if no limiters are used. Limiters tend to stabilize them, as discussed in [6].)

`method(3) = 10`: Single transverse propagation of the increment wave as in 2D, but no double transverse propagation is performed. This method is unconditionally unstable when `method(2) = 2`. (`method(2)=1, method(3)=10` is stable for CFL $\leq 1/2$).

`method(3) = 11`: Double transverse propagation of the increment wave. The first-order method `method(2)=1, method(3)=11` is stable for CFL ≤ 1, and is the Corner Transport Upwind (CTU) method in 3D (see [10]). The method `method(2)=2, method(3)=11` is unconditionally unstable.

`method(3) = 20`: Both the increment wave and the correction wave are transversely propagated in each orthogonal direction separately, but no double transverse propagation is applied. Only to be used with `method(2) = 2` and CFL $\leq 1/2$.

`method(3) = 21`: Double transverse propagtion of the increment wave, but only single transverse propagation of the correction wave. Only to be used with `method(2) = 2` and CFL $\leq 1/2$.

`method(3) = 22`: Double transverse propagation of both the increment wave and the correction wave. Only to be used with `method(2) = 2` and CFL ≤ 1.

`zlower`: The front edge of the computational domain.

`zupper`: The back edge of the computational domain.

`mthbc(5)`: The type of boundary condition to be imposed at the boundary `z = zlower`. The same values are recognized as described above for `mthbc(1)`.

`mthbc(6)`: The type of boundary condition to be imposed at the boundary `z = zupper`. The same values are recognized as described above for `mthbc(1)`.

1.12.1 Riemann solvers

Three Riemann solvers must be provided. The transverse solver `rpt3` is called only if `method(3) > 0`. The double transverse solver `rptt3` is called only if the second digit of `method(3)` is positive. Dummy routines are provided in `claw/clawpack/3d/lib`.

`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.

Two arrays `ql` and `qr` are passed in that contain the data along this one-dimensional slice, and as in one and two dimensions, the values `qr(i-1,:)` and `ql(i,:)` should be used as data for the Riemann problem between cells `i-1` and `i`. As in the two-dimensional routine `rpn2`, these arrays are normally identical in the standard CLAWPACK calls, but may differ if AMRCLAW is being used.

If aux arrays are being used, then a 1-dimensional slice of these arrays is also passed into rpn3 as auxl and auxr.

In Version 4.1, the auxl and auxr arrays are dimensioned just as in rpn1 and rpn2, *e.g.*, auxl(1-mbc:maxm+mbc, 1:maux).

Warning: This is a change from Version 4.0, where the auxl array is dimensioned as auxl(1-mbc:maxm+mbc, 1:maux, 1:3). This was simply the array aux2 described below, which is passed into rpt3.f and rptt3.f. In general only one slice of this array is needed in rpn3 and so to simplify the explanation of this routine, and for consistency with 1d and 2d, this has been changed. Now auxl(1-mbc:maxm+mbc, 1:maux) corresponds to aux2(1-mbc:maxm+mbc, 1:maux, 2).

rpt3: Solves the Riemann problem in the transverse directions. The fundamental input to rpt3 is an array of fluctuations asdq and the output is the two arrays bmasdq and bpasdq that result from decomposing the input fluctuation into transverse fluctuations in the appropriate direction.

This routine is analogous to rpt2 in two dimensions, but with a new input parameter icoor added. In three dimensions, for each coordinate direction there are two potential transverse directions, and icoor indicates which direction should be considered the transverse direction in this call.

The parameter ixyz indicates what direction the slice of data lies in, as in rpn3. This is called the "x-like direction" below. The terms "y-like direction" and the "z-like direction" then denote the two orthogonal directions according to the table below:

	x-like direction	y-like direction	z-like direction
ixyz = 1	x	y	z
ixyz = 2	y	z	x
ixyz = 3	z	x	y

The parameter icoor always takes the value 2 or 3 and indicates which of the transverse directions is to be used for the transverse splitting. If we think of the direction of the normal Riemann solve as the x-like direction, then icoor=2 means we are doing a transverse Riemann solve in the corresponding y-like direction, while icoor=3 means we are doing a transverse Riemann solve in the corresponding z-like direction. In other words:

ixyz=1, icoor=2: Slice is in x, split in y.
asdq= $\mathcal{A}^*\Delta Q$, bmasdq= $\mathcal{B}^-\mathcal{A}^*\Delta Q$, bpasdq= $\mathcal{B}^+\mathcal{A}^*\Delta Q$.

ixyz=1, icoor=3: Slice is in x, split in z.
asdq= $\mathcal{A}^*\Delta Q$, bmasdq= $\mathcal{C}^-\mathcal{A}^*\Delta Q$, bpasdq= $\mathcal{C}^+\mathcal{A}^*\Delta Q$.

ixyz=2, icoor=2: Slice is in y, split in z.
asdq= $\mathcal{B}^*\Delta q$, bmasdq= $\mathcal{C}^-\mathcal{B}^*\Delta Q$, bpasdq= $\mathcal{C}^+\mathcal{B}^*\Delta Q$.

ixyz=2, icoor=3: Slice is in y, split in x.
asdq= $\mathcal{B}^*\Delta q$, bmasdq= $\mathcal{A}^-\mathcal{B}^*\Delta Q$, bpasdq= $\mathcal{A}^+\mathcal{B}^*\Delta Q$.

ixyz=3, icoor=2: Slice is in z, split in x.
asdq= $\mathcal{C}^*\Delta Q$, bmasdq= $\mathcal{A}^-\mathcal{C}^*\Delta Q$, bpasdq= $\mathcal{A}^+\mathcal{C}^*\Delta Q$.

ixyz=3, icoor=3: Slice is in z, split in y.
asdq= $\mathcal{C}^*\Delta Q$, bmasdq= $\mathcal{B}^-\mathcal{C}^*\Delta Q$, bpasdq= $\mathcal{B}^+\mathcal{C}^*\Delta Q$.

As in two dimensions, a parameter imp indicates whether asdq represents amdq or apdq (assuming ixyz=1, for example):

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

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

It is necessary to check the value of `imp` only if the transverse Riemann solution procedure is different to the left and right of the interface, *e.g.*, if spatially-varying parameters stored in the `aux` arrays are used in solving the Riemann problem.

As in the two-dimensional transverse solver `rpt2`, three slices of the `aux` arrays are passed into `rpt3`, called `aux1`, `aux2`, and `aux3`. These contain slices of the `aux` arrays in the rows that are below, at, and above the current row of data (in the y-like direction), respectively. In three dimensions there is an additional z-like transverse directions, and so each of these arrays is now dimensioned as

```
auxN(1-mbc:maxm+mbc, 1:maux, 1:3)
```

The final index takes the values `1, 2, 3` for slabs of `aux` data that are below, at, and above the current slab of data in the z-like direction. Consequently a 3×3 rectangular cylinder of `aux` values are available centered on the row of grid cells providing the data for `ql` and `qr`.

If `icoor=2` then the `aux` data needed for transverse solves is in `aux1(:,:,2), aux2(:,:,2)`, and `aux3(:,:,2)`.

If `icoor=3` then the `aux` data needed for transverse solves is in `aux2(:,:,1), aux2(:,:,2)`, and `aux2(:,:,3)`.

`rptt3:` Solves a "double transverse" Riemann problem as described in [6].

The fundamental input to `rptt3` is an array of fluctuations `bsasdq` along one slice in the `ixyz` direction, and the output is two arrays `cmbsasdq` and `cpbsasdq` that result from decomposing the input fluctuation into transverse fluctuations in the appropriate direction. In this case the input fluctuations are already transverse fluctuations from a splitting in one of the transverse directions, and these should now be split into the remaining direction. As in `rpt3`, the value of `ixyz` indicates the direction of the slice, and the original normal Riemann solve, and `icoor` indicates the direction in which a transverse solve should now be performed. As in `rpt3`, `icoor=2` means we are doing the transverse Riemann solve in the corresponding y-like direction, while `icoor=3` means we are doing a transverse Riemann solve in the corresponding z-like direction.

`ixyz=1, icoor=2:` Slice is in x, split in y.
`bsasdq`$= \mathcal{C}^*\mathcal{A}^*\Delta Q$, `cmbsasdq`$= \mathcal{B}^-\mathcal{C}^*\mathcal{A}^*\Delta Q$, `cpbsasdq`$= \mathcal{B}^+\mathcal{C}^*\mathcal{A}^*\Delta Q$.

`ixyz=1, icoor=3:` Slice is in x, split in z.
`bsasdq`$= \mathcal{B}^*\mathcal{A}^*\Delta Q$, `cmbsasdq`$= \mathcal{C}^-\mathcal{B}^*\mathcal{A}^*\Delta Q$, `cpbsasdq`$= \mathcal{C}^+\mathcal{B}^*\mathcal{A}^*\Delta Q$.

`ixyz=2, icoor=2:` Slice is in y, split in z.
`bsasdq`$= \mathcal{A}^*\mathcal{B}^*\Delta Q$, `cmbsasdq`$= \mathcal{C}^-\mathcal{A}^*\mathcal{B}^*\Delta Q$, `cpbsasdq`$= \mathcal{C}^+\mathcal{A}^*\mathcal{B}^*\Delta Q$.

`ixyz=2, icoor=3:` Slice is in y, split in x.
`bsasdq`$= \mathcal{C}^*\mathcal{B}^*\Delta Q$, `cmbsasdq`$= \mathcal{A}^-\mathcal{C}^*\mathcal{B}^*\Delta Q$, `cpbsasdq`$= \mathcal{A}^+\mathcal{C}^*\mathcal{B}^*\Delta Q$.

`ixyz=3, icoor=2:` Slice is in z, split in x.
`bsasdq`$= \mathcal{B}^*\mathcal{C}^*\Delta Q$, `cmbsasdq`$= \mathcal{A}^-\mathcal{B}^*\mathcal{C}^*\Delta Q$, `cpbsasdq`$= \mathcal{A}^+\mathcal{B}^*\mathcal{C}^*\Delta Q$.

`ixyz=3, icoor=3:` Slice is in z, split in y.
`bsasdq`$= \mathcal{A}^*\mathcal{C}^*\Delta Q$, `cmbsasdq`$= \mathcal{B}^-\mathcal{A}^*\mathcal{C}^*\Delta Q$, `cpbsasdq`$= \mathcal{B}^+\mathcal{A}^*\mathcal{C}^*\Delta Q$.

Often `rptt3` performs exactly the same action as `rpt3` and this routine can be simply replicated with only the name changed and the addition of another parameter `impt` (described below) to the calling sequence that need not be used if the equations are autonomous (no spatially-varying information is stored in `aux` arrays).

In Version 4.0 of CLAWPACK, only a single transverse solver `rpt3` was required, which took an input that could represent either `asdq` or `bsasdq`. In Version 4.1 this has been split into two

solvers because of the realization that, for problems with spatially-varying coefficients (typically stored in `aux` arrays), it may be necessary to specify more carefully where these fluctuations are located spatially in order to use the proper `aux` values.

As in `rpt3`, a parameter `imp` indicates whether `bsasdq` represents `bsamdq` or `bsapdq`. In addition, the new parameter `impt` indicates whether `bsasdq` represents `bmasdq` or `bpasdq`. In the description below we use *left* and *right* to refer to motion in the x-like direction and *down* and *up* to refer to motion in the transverse direction in which the transverse solve has already been performed by `rpt3`.

`imp = 1, impt = 1: bsasdq` represents `bmamdq`, the fluctuation that was propagating into the cell to the *left* of the interface has been split transversely and this is the *downgoing* portion.

`imp = 1, impt = 2: bsasdq` represents `bpamdq`, the fluctuation that was propagating into the cell to the *left* of the interface has been split transversely and this is the *upgoing* portion.

`imp = 2, impt = 1: bsasdq` represents `bmapdq`, the fluctuation that was propagating into the cell to the *right* of the interface has been split transversely and this is the *downgoing* portion.

`imp = 2, impt = 2: bsasdq` represents `bpapdq`, the fluctuation that was propagating into the cell to the *right* of the interface has been split transversely and this is the *upgoing* portion.

These parameters are typically needed only if `aux` arrays are being used to store spatially-varying information that must be properly accessed.

The same `aux1`, `aux2`, and `aux3` arrays are passed into `rptt3` as are passed into `rpt3`, as described above.

To understand how the `aux` arrays are indexed in the Riemann solvers, it is best to study an example, such as the spatially-varying acoustics equations. The Riemann solvers for this problem may be found in `claw/applications/acoustics/3d/rp` and are called `rpn3acv.f, rpt3acv.f`, and `rptt3acv.f`. On the web, see

`http://amath.washington.edu/~claw/applications/acoustics/3d/rp/www`.

1.13 Restarting CLAWPACK

As of version 4.2, CLAWPACK simulations may be restarted from old output files using the `restartN.f` (and `restartN_hdf.f`) routines supplied in `clawpack/Nd/lib` for `N` = 2 or 3. A small text file `restart.data` specifies whether or not the simulation should be restarted from an old data file and, if so, gives the frame number from which the simulation should be restarted. A sample `restart.data` file follows:

```
F     Restart from old fort.qXXXX or fort.qXXXX.hdf file. (T/F)
3     Frame number to restart from, i.e. specify value of XXXX.
```

This file is read within the `qinit` subroutine and, if a restart is requested, the `restart` routine is called. Examples showing the necessary changes to `Makefile` and `qinit.f` are given in `clawpack/Nd/example1/` and `clawpack/Nd/example1/mpi`. (For more information on HDF output, see Section 2.2.)

If a simulation is restarted, the initial condition will not be written to a file (to avoid over-wrtiting the old output file), and the frame number will increase from that specified in `restart.data`. If you wish to restart from the end of an old simulation, the values of `nout`, `tfinal` and/or `nsteps` in `clawNez.data` should be increased appropriately.

Note that the results of a restarted simulation may differ from those of a single simulation if variable time stepping is used, i.e. if `method(1)` $\neq 0$. The restarted simulation will start with a time step of `dtv(1)` (specified in `clawNez.data`), rather than one computed from the cfl number at the previous time step. This difference in time advancement may create discrepancies between continuing and restarted simulations.

Chapter 2

Program output and graphics using MATLAB

The CLAWPACK routines `clawNez.f` (for `N = 1,2,3`) are set up to call output routines `outN.f` at each time when output is desired. The output times are specified in the `clawNez.data` files, as described in 1.7.6.

The default output routines located in `claw/clawpack/Nd/lib/outN.f` write out the solution in a form that is suitable for use with the MATLAB graphics routines also supplied with CLAWPACK. As with any CLAWPACK routine, you can customize the output routine for your own needs if the default version is not adequate; see 2.11.

New to CLAWPACK version 4.2: The MATLAB graphics routines supplied with the CLAWPACK package is now thoroughly documented on-line via the MATLAB help system. For a quick summary of these routines, type

```
help clawgraphics
```

at the MATLAB prompt. This is the most complete and up-do-date description of all the graphics routines, so the user is encouraged to browse this help summary.

New to CLAWPACK version 4.2: As of CLAWPACK version 4.2, output also may be produced in HDF format, which can provide considerable savings in disk space (through the use of a portable binary file format that supports data compression). The HDF output routines `outN_hdf.f` take the place of the default output routines described below in section 2.1 and are fully compatible with the CLAWPACK MATLAB graphics routines. The use of the HDF output routines is described in detail in section 2.2.

2.1 Output produced by `outN.f`

The style of CLAWPACK output is similar in `N = 1,2,3` dimensions, and is also basically the same when AMRCLAW is used.

The default version of `outN.f` produces two files at each output time. These are called `fort.tXXXX` and `fort.qXXXX`, where `XXXX` gives the frame number: 0000 for the initial data, 0001 at the first output time, etc. Typical output files have the following form:

`fort.t0000:`

```
0.00000000E+00    time
2                 meqn
1                 ngrids
```

The file `fort.tXXXX` has only three lines giving the output time, the number of solution values printed from each grid cell, and the number of distinct grids at this time. The value of `meqn` is used in `plotclawN.m`, so if you modify the output routine to print out only some solution values, this value should be changed accordingly.

When using the basic CLAWPACK routines, the value of `ngrids` is set to 1. With AMRCLAW (in two and three dimensions) there may be multiple grids at each output time, one or more at each level (see Chapter 3).

The file `fort.qXXXX` has the following form in one dimension.

`fort.q0000:`

```
  1                 grid_number
  1                 AMR_level
200                 mx
  0.00000000E+00    xlow
  0.50000000E-01    dx

0.00000000E+00    1.00000000E+00
0.00000000E+00    1.00000000E+00
etc (198 more lines)
```

The file `fort.q0000` contains the solution on all grids at time $t = 0$. In the example above, there is only one grid. Information about this grid is contained in the first 5 lines and then the `meqn = 2` solution values at each of the `mx = 200` points on this grid are printed on the next 200 lines. Note that only the values at the interior points `i = 1, 2, ..., mx` are printed, not the values in the ghost cells.

In two dimensions these files are essentially the same, but the `fort.qXXXX` files also include the parameter values `my`, `ylow`, and `dy` in the header information for each grid. This header is followed by `mx*my` lines of data, and each line has the `meqn` solution values from one grid cell. The natural generalization extends this to three dimensions.

When AMRCLAW is used, there may be several grids at each output time. The values from each grid are all output to the same `fort.qXXXX` file. The data from each grid is preceded by 5 lines of information about this grid, in the same form as the first 5 lines of the sample `fort.q0000` file given above. Each grid has a distinctive `grid_number`. The `AMR_level` is the level as described in Chapter 3. `mx` gives the number of cells on this particular grid, with mesh width `dx`. This grid starts at `x = xlow` and ends at `x = xlow + mx*dx`.

The Fortran format `4e26.16` is used for printing the values. So roughly 16 significant figures are printed, with at most 4 values per line. If `meqn > 4` then the data from each grid cell will occupy more than one line of the file (but the MATLAB routines should still read the file properly). Before printing the values, any value that is less than `1d-99` is reset to zero. Otherwise Fortran will not print the `E` in the floating point number and MATLAB will not properly parse the file when reading it.

2.2 HDF output produced by `outN_hdf.f`

Starting with CLAWPACK version 4.2, output may also be produced in HDF format. HDF is a portable, binary file format developed by the National Center for Supercomputing Applications (NCSA). The CLAWPACK HDF output routines require software libraries developed by NCSA for working with HDF files. These libraries are available at:

`http://hdf.ncsa.uiuc.edu/`

(While these libraries are be installed in `/usr/local/hdf/lib` on many systems, the user may need to download and installed them him/herself.) The use of the HDF format can provide considerable savings in disk storage when compared to the ASCII output files described above. These savings result from

the use of a binary file format that supports data compression. **Note:** CLAWPACK makes use of version 4 of HDF. Support for version 5 of HDF is not planned at this time.

Quick Start Guide for HDF:

1. Go to the directory of the application you are interested in.
2. Set the proper path for the HDF libraries by editing the value of the `HDFLIB` variable at the top of the `Makefile` in that directory. (Note that for some operating systems, it may be necessary to enter the full path of each library, e.g. `HDFLIB=/usr/local/hdf/lib/libmfhdf.a ...`)
3. Type `make xclawhdf` to make the executable `xclawhdf`. (Similarly, type `make xamrhdf` for an AMRCLAW executable or `make xclawmpihdf` for an MPICLAW executable.)
4. Run executable `xclawhdf`.
5. Plot the results using CLAWPACK graphics. Before typing `plotclawN` in MATLAB, set the variable `OutputFlag = 'hdf'`. (This could also be set in a `setplotN.m` file in the application/example directory.) More details on the MATLAB graphics routines are given below starting in section 2.3.

The HDF output file contains the same information as the default (ASCII) output described above, although this information is arranged in a somewhat different fashion. Note that the behavior of the HDF output routines is similar when used with CLAWPACK, AMRCLAW and MPICLAW. The one difference is that the data compression feature of HDF is not used by the MPICLAW routines.

The default version of `clawpack/Nd/lib/outN_hdf.f` produces a single HDF file `fort.qXXXX.hdf` at each output time. Here, `XXXX` is the frame number: 0000 for the initial data, 0001 at the first output time, etc. The first variable stored in the HDF output file is a double-precision vector containing information about the simulation and grid. The entries of the vector are as follows:

1	grid number
2	number of dimensions
3	time of output
4	`meqn`: number of variables output at each cell
5	`level`: AMR level (defaults to 1 for CLAWPACK)
6-8	`mx`, `my`, `mz`: Number of grid points along each dimension
10-12	`xlower`, `ylower`, `zlower`: computational coordinates of lower left corner of domain
14-16	`xupper`, `yupper`, `zupper`: computational coordinates of upper right corner of domain
18-20	`dx`, `dy`, `dz`: Δx, Δy, Δz in computational coordinates

Following this vector, each of the `meqn` variables is added to the HDF file as a separate array of dimension `mx*my*mz` In total, the HDF file will contain `meqn+1` variables: the variable containing the grid information followed by the `meqn` variables containing the solution. Note that AMRCLAW HDF output files (produced by the routine `amrclaw/Nd/lib/valout_hdf.f`) which employ `ngrids` grids on various levels will contain `ngrids*(mqn+1)` variables (`meqn+1` variables for each grid).

The behavior of the one- and two-dimensional routines is similar, except that grid information only describes the appropriate axes, and the variables are output as arrays of dimension `mx` or `mx*my`.

The MATLAB graphics routines supplied with CLAWPACK will recognize the HDF output files if the variable `OutputFlag` is set to `'hdf'` either at the prompt (before typing `plotclawN`):

```
>> OutputFlag = 'hdf';
```

or in the file `setplotN.m` in the application/example directory. If the user wishes to use **hdf** output exclusively, the `OutputFlag = 'hdf';` command could be added to a `startup.m` file in the `claw/matlab` directory. The default value for `OutputFlag` is `'ascii'`.

The MATLAB commands `hdfinfo` and `hdfread` as well as the utility `hdfview` may be used to view the contents of an HDF file. These tools provide an ideal means for direct access to the data in an output

file, and (in the case of MATLAB) a simple means for comparison between the results of different CLAWPACK simulations. It is available from the NCSA website listed above. In addition to MATLAB, a wide range of software can read the data contained in the HDF output files. Some examples are given at `http://hdf.ncsa.uiuc.edu/tools.html`.

Note on compiler warning messages when using gcc-3 on linux: When compiling and linking with the HDF library on linux, the following warning message may be given when the compilation is complete:

```
/usr/local/hdf/lib/libmfhdf.a(error.o): In function ‘strerror’:
error.o(.text+0x2d): ‘sys_errlist’ is deprecated; use ‘strerror’ or ‘strerror_r’ instead
error.o(.text+0xe): ‘sys_nerr’ is deprecated; use ‘strerror’ or ‘strerror_r’ instead
```

While this error message may seem troubling, it has not caused any difficulty for the CLAWPACK developers in using the HDF version of CLAWPACK.

2.3 Plotting results using `plotclawN.m`

New to version 4.2: Please note that substantial changes have been made to the MATLAB graphics routines as of version 4.2. These changes have brought new features, especially for three-dimensional graphics, and better on-line help facilities. But, while every effort was made to maintain backward compatibility with previous versions of the CLAWPACK graphics routines, there are a few instances in which older scripts, (i.e. `setplotN.m`, `beforeframe.m` and `afterframe.m`) may not work as originally intended, because variable names have been changed. The old MATLAB graphics routines are available in the directory `claw/matlab41`.

`Hint:` To find a list of available routines and to get help on many aspects of CLAWPACK's MATLAB graphics routines, type `help clawgraphics`. The new MATLAB graphics routines can be found in `claw/matlab`.

To plot results in MATLAB, use `plotclawN` (with `N = 1,2,3` in the appropriate dimension). This should invokes the m-file `claw/matlab/plotclawN.m`. If MATLAB can't find this script, make sure your `path` is set properly in MATLAB; see 1.6.1

Hitting `<Enter>` at the `plotclawN` prompt causes the next frame of data to be read into MATLAB and plotted. Alternatively, one of several characters can be typed at the `plotclawN` prompt:

- **k** Keyboard input. Type any MATLAB commands you wish to execute at the resulting `K>>` prompt. Type the word `return` at the prompt to return to `plotclawN` execution.
- **r** Redraw current frame. You might want to do this after resetting some plot parameters. Note that this option doesn't re-read the `fort.qXXXX` file. Use `rr` to re-read the file and redraw.
- **rr** Re-read current frame from file and redraw. You might want to do this after re-running CLAWPACK, for example.
- **j** Jump to a particular frame. You will then be prompted for the frame number.
- **i** Print info about plotting parameters and the solution at the current time.
- **q** Quit out of `plotclawN`.

The script `plotclawN.m` cycles through over frames (individual output times). Before plotting each frame, it first calls `readamrdata.m` to actually read in the data. To plot the frame, it calls `plotframeN.m` and plots the data in the current frame, using plotting parameters set in `setplotN.m` and described below.

The function `readamrdata` is new to version 4.2 of CLAWPACK and is used by both CLAWPACK and AMRCLAW. To see how to use this function, type `help readamrdata` at the MATLAB prompt.

2.4 Plotting parameters and setplotN.m

The m-file setplotN.m is used to set the values of various parameters that are used in plotclawN.m (with N = 1,2,3 in the appropriate dimension). This script is called each time plotclawN is invoked (provided you answer yes or y at the prompt). The default version claw/matlab/setplotN.m will be used unless a modified version resides in the current working directory.

Values of plotting parameters can also be changed during execution of plotclawN by typing k at the prompt (see 2.3) and then entering a new value. If you have set new values and don't want these to be reset by setplotN the next time you run plotclawN, make sure you don't answer yes at the appropriate prompt.

Some of the parameters that can be set in the are described below.

In all dimensions:

mq: Indicates which component of the solution vector q should be plotted. In one dimension this can be a vector and several plots will be produced together in one figure using the subplot command. In more dimensions only a single value is supported.

UserVariable: A nonzero value indicates that some value derived from the components of q should be plotted rather than a component of q itself. See 2.6 for details.

MappedGrid: A nonzero value indicates that the physical grid is not simply a uniform Cartesian grid with spacing dx, dy, dz. See 2.8 for details.

MaxFrames: The maximum number of frames for the main loop in plotclawN. If this is undefined, you'll get a warning that you probably forgot to execute setplotN.

In one dimension:

PlotStyle: A string that is used in plotting the data with a command of the form plot(x,q,PlotStyle). You can set this to anything that you would normally send to the plot command to indicate the line style or color. Use the setplotstyle to specify a list of line styles and colors or even just a single style or color. For example, PlotStyle=setplotstyle('b-') plots a solid blue line while PlotStyle=setplotstyle('ro') would plot using red circles.

In two dimensions:

PlotType: Indicates what type of plot is desired. The current PlotType choices are:

PlotType = 1: A pseudo-color plot is generated. This is actually plotted as a flat three-dimensional surface because this is useful in plotting AMR results: the values on different refinement levels are plotted at different physical levels, with the finer grid data plotted above coarser grid data. Viewing from above then shows the finest grid data available at each point. (Rotating this figure using rotate3d sometimes gives an interesting view of the data.) **New to version 4.2:** Contour lines are drawn on top of the pseudo-color plot by default using the values in ContourValues. To suppress contour lines, set ContourValues = [].

If PlotGrid = 1 then grid lines are drawn along with the color plot. If the grid is very fine then these lines may completely obscure the color plot. If PlotGrid = 0 then the lines are not drawn. For AMRCLAW results, PlotGrid can be an array with an element for each level indicating whether grid lines should be drawn on that level.

Other plotting parameters are also used when AMR results are plotted; see below.

PlotType = 2: A contour plot is generated. The variable ContourValues should be set to the variable values at which the contour lines should be drawn. You may set ContourValues to any input valid for the MATLAB contour plotting routine contour. For example, set

`ContourValues = [0.05:0.1:0.95]` to show contour lines at values 0.05, 0.15, ..., 0.95. If you want to plot a contour line corresponding to a single value `c`, for example, set `ContourValues = [c c]`. You may also let the graphics routines choose the contour values automatically by setting `ContourValues = m`, where `m` is the number of contour values that should be chosen. This option, however, is not recommended for AMR plots, as the contour lines across patches will not necessarily match up across coarse/fine boundaries.

For `PlotType = 1,2,3` contour lines will be drawn whenever the variable `ContourValues` is non-empty. For `PlotType = 1`, contour lines will be drawn on top of the pseudo-color plot, whereas for `PlotType = 2`, the contour lines are drawn on a white background. For `PlotType = 3`, the Schlieren data will be contoured. To suppress contour lines, set `ContourValues = []`.

`PlotType = 3`: A Schlieren plot is generated. This is a grayscale plot of the gradient of the data with an exponentially varying colormap so that regions of large gradient are highlighted. This is particularly useful for viewing data that contains shock waves or other discontinuous waves. (For example, the lower figure on the cover of [10] was generated using `PlotType=3`.) This typically works well only when the grid is quite fine.

These are called Schlieren plots because they resemble Schlieren photographs that are often taken in experimental gas dynamics. Since the refractive index of a gas depends on its density, shock waves and other steep density gradient show up as streaks (Schlieren in German) in these photographs.

`PlotType = 4`: A scatter plot of the data is produced. Each pair `(r(i,j), q(i,j,mq))` is plotted as a single point. The value `r(i,j)` is the distance that the point `(x(i,j), y(i,j))` lies from an origin specified by the parameters `x0` and `y0` (which are set to $(0, 0)$ in the default `setplot2.m` script).

If you use the `MappedGrid` option, the distance `r(i,j)` is computed from physical coordinates, not computational coordinates. That is, `((x(i,j),y(i,j))` are mapped from computational coordinates to physical coordinates via a function `mapc2p.m`. See Section 2.8.

You may also specify the symbol used to plot in the scatter plot by setting `ScatterStyle` to the desired symbol and color. For example, `ScatterStyle = 'o'` will plot a small circle at each `(r(i,j),q(i,j,mq))` pair. For AMR data, you may specify multiple symbols if you would like a different symbol for each level of refinement. Multiple symbols and or/colors may be specified using for example

```
ScatterStyle = setplotstyle('x','o','+','s','^');
```

or

```
ScatterStyle = setplotstyle('rx','o','+','gs','^');
```

to specify that some symbols should have different colors.

This plot style is particularly useful for problems where the solution should be radially symmetric, in which case all the points plotted should lie along a single curve $q(r)$. The degree of scatter from this curve gives an indication of how non-isotropic the numerical solution is. See, for example, Figures 21.5 and 21.7 in [10].

Scatter plots may also be useful if the data should represent a plane wave with variation in only one spatial direction. By setting `(x0,y0)` to be a point far away from the physical domain in this direction, a scatter plot will be produced in which `r(i,j)` essentially corresponds to distance in this direction. For example, if the solution should vary only with x, setting `(x0,y0) = (-1000, 0)` will give a scatter plot of the data vs. $x + 1000$.

Another way to accomplish this is to define a user function `map1d`, which takes 2d data and converts it to 1d data in some fashion determined by the user. Type `help map1d` at the MATLAB prompt for more discussion of this, or see Section 2.7.

In three dimensions:

The CLAWPACK package now includes several MATLAB routines for visualizing 3d solutions. There are three main ways to visualize the 3d data: using scatter plots (one dimensional plots suitable for spherically symmetric data), isosurfaces (surfaces of constant value), and 2d slices through the data in planes of constant x, y, or z.

Many of the plotting types described below require the user to specify slices in one or more of the three coordinate directions. These slices can be specified in the `setplot3.m` file (or manually from the MATLAB prompt) using the variable names `xSliceCoords`, `ySliceCoords`, and `zSliceCoords`. For example, to create four slices through that the data at planes of constant `x = 0.5`, `y = 0.25`, `z = 0.25`, and `z = 0.75`, you should specify

```
xSliceCoords = 0.5;
ySliceCoords = 0.25;
zSliceCoords = [0.25  0.75];
```

Note that you can create multiple slices in any coordinate direction by specifying a vector of values rather than a scalar slice value. To suppress the generation of slices in one or more directions, set the appropriate coordinate direction variable to the empty matrix, *e.g.*, `zSliceCoords = [ ]`.

To view a slice in the y-z plane (constant x), type `view(xSlice)`. To view a slice in the x-z plane (constant y), type `view(ySlice)`, and to view a slice in the x-y plane (constant z), type `view(zSlice)`. These are defined in `claw/matlab/setviews.m`, which must be executed first.

To loop over several slices in a particular coordinate direction, type `sliceloop x`, `sliceloop y` or `sliceloop z`. This command will loop over the slices specified in corresponding `xSliceCoords`, `ySliceCoords` or `zSliceCoords` vector.

Another variable that is new to version 4.2 of CLAWPACK is the parameter `UserView`. This can be set in either the `setplot3.m` file, or at the `K>` prompt. By setting `UserView` to values which are valid for the MATLAB `view` command, the user overrides the default 3d view of the plot. Also, while cycling through time frames of data, the user may find that they want to fix a particular viewpoint (obtained, for example by using `rotate3d`). By issuing the command `UserView = view;` at the MATLAB `K>` prompt, the current view will be fixed for subsequent time frames.

Below are the details for creating various types of 3d plots. Note that several of them require that slice coordinates, described above, be set.

`PlotType`**:** Indicates what type of plot is desired. The current choices are:

- `PlotType = 1`**:** This is analogous to `PlotType = 1` in two dimensions, but now plotting pseudo-color plots on one or more slices of data. The parameters `xSliceCoords`, `ySliceCoords` and `zSliceCoords`, described above are used to indicate the locations of the slices.

  If desired, contour lines are also superimposed on the slices, by setting `ContourValues` as described below under `PlotType = 2`.

  If `PlotGrid = 1` then grid lines are drawn along with the color plot. If the grid is very fine then these lines may completely obscure the color plot. If `PlotGrid = 0` then the lines are not drawn. For AMRCLAW results, `PlotGrid` can be an array with an element for each level indicating whether grid lines should be drawn on that level.

- `PlotType = 2`**:** Contour lines are generated on user specified slices. The slice color on all slices is set to white. The variable `ContourValues` should be set to the variable values at which the contour lines should be drawn. You may set `ContourValues` to any input valid for the MATLAB contour plotting routine `contour`. For example, set `ContourValues = [0.05:0.1:0.95]` to show contour lines at values 0.05, 0.15, ..., 0.95. If you want to plot a contour line corresponding to a single value `c`, for example, set `ContourValues = [c c]`. You may also let the graphics routines choose the contour values automatically by setting `ContourValues = m`, where `m` is the number of contour values that should be chosen. This option, however,

is not recommended for AMR plots, as the contour lines across patches will not necessarily match up across coarse/fine boundaries.

For `PlotType = 1,2,3` contour lines will be drawn whenever the variable `ContourValues` is non-empty. For `PlotType = 1`, contour lines will be drawn on top of the pseudo-color plot, whereas for `PlotType = 2`, the contour lines are drawn on a white background. For `PlotType = 3`, the Schlieren data will be contoured. To suppress contour lines, set `ContourValues = []`.

If you want to view the slices directly you could add the command `view(ySlice)`, `view(ySlice)` or `view(zSlice)` to your `afterframe.m` file to view the contour plot as if it were a 2d plot.

`PlotType = 3`: Creates a Schlieren type plot (greyscale plot of the norm of the gradient of desired quantity). This two-dimensional plots are created at slices of constant x,y and/or z, specified by setting `xSliceCoords`, `ySliceCoords`, or `zSliceCoords` above to desired slice values.

`PlotType = 4`: A scatter plot of the data is produced. Each pair `(r(i,j,k), q(i,j,k,mq))` is plotted as a single point. The value `r(i,j,k)` is the distance that the point `(x(i,j,k), y(i,j,k),z(i,j,k))` lies from an origin specified by the parameters `(x0,y0,z0)` (which are set to `(0,0,0)` in the default `setplot3.m` script).

If you use the `MappedGrid` option, the distance `r(i,j,k)` is computed from physical coordinates, not computational coordinates. That is, `((x(i,j,k),y(i,j,k),z(i,j,k))` are mapped from computational coordinates to physical coordinates via a function `mapc2p.m`. See Section 2.8.

You may also specify the symbol used to plot in the scatter plot by setting `ScatterStyle` to the desired symbol and color. For example, `ScatterStyle = 'o'` will plot a small circle at each `(r(i,j,k),q(i,j,k,mq))` pair. For AMR data, you may specify multiple symbols if you would like a different symbol for each level of refinement. Multiple symbols and or/colors may be specified using for example

```
ScatterStyle = setplotstyle('x','o','+','s','^');
```

or

```
ScatterStyle = setplotstyle('rx','o','+','gs','^');
```

to specify that some symbols should have different colors.

This plot style is particularly useful for problems where the solution should be spherically symmetric, in which case all the points plotted should lie along a single curve $q(r)$. The degree of scatter from this curve gives an indication of how non-isotropic the numerical solution is. See, for example, Figures 21.5 and 21.7 in [10].

Scatter plots may also be useful if the data should represent a plane wave with variation in only one spatial direction. By setting `(x0,y0,z0)` to be a point far away from the physical domain in this direction, a scatter plot will be produced in which `r(i,j,k)` essentially corresponds to distance in this direction. For example, if the solution should vary only with x, setting `(x0,y0,z0) = (-1000, 0,0)` will give a scatter plot of the data vs. $x + 1000$.

Another way to accomplish this is to define a user function `map1d`, which takes 3d data and converts it to 1d data in some fashion determined by the user. Type `help map1d` at the MATLAB prompt for more discussion of this, or see Section 2.7.

`PlotType = 5`: This `PlotType` option is no longer supported in the CLAWPACK graphics. Instead, isosurfaces will be generated for `PlotType = 1,2,3` any time the vector `IsosurfValues` is non-empty. See below for details.

New to version 4.2: For `PlotType = 1,2,3`, isosurfaces are drawn by default at the values specified by the vector `IsosurfValues`. To turn off isosurfaces, set `IsosurfValues = []`.

To make a plot with only isosurfaces and no slices, set `xSliceCoords = []`, `ySliceCoords = []` and `zSliceCoords = []`. You may also specify color and transparency values for each surface. Specify the colors using the vector `IsosurfColors` and the transparencies by setting the vector `IsosurfAlphas`. Values in `IsosurfAlphas` should be between 0 and 1, where 0 is transparent and 1 is opaque. In order to view transparent surfaces, you must be using MATLAB 6.0 or higher, and your system must support the OpenGL Renderer, supplied with MATLAB. Note that isosurface colors will be taken from the current colormap if `IsosurfColors = 'q'`.

If you are plotting AMRCLAW data, and if the vector `PlotData` has more than one value set to 1, then you will get multiple isosurfaces corresponding to the same constant value, coming from grids at different refinement levels. Depending on your data, this may lead to a confusing plot. It is recommended that you plot data at a single level of refinement only.

2.5 Plotting AMRCLAW output

The MATLAB routines `plotclaw2` and `plotclaw3` can also be used for viewing the AMRCLAW output (see Chapter 3). Most of the same plotting options used for CLAWPACK work also for 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 preceded 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.

`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 pseudocolor plots (`PlotType=1`), 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`. In three dimensions, `PlotGrid` and `PlotGridEdge` are used to determine what shows up on each slice where data is plotted (with `PlotType = 1, 2, 3`).

`PlotCubeEdge(L)`: In three dimensions, you can also specify that the edge of each grid patch at level `L` should be plotted as a three-dimensional object by setting `PlotCubeEdge(L) > 0`. This works with `PlotType = 1, 2, 3`.

2.6 The `UserVariable` option

The default `setplotN.m` files set `UserVariable = 0` and the value of `mq` is used to determine which component of q to plot. Instead of plotting one of the components directly, you may want to plot a value that can be computed from the components of q. For example, in a computation with the Euler equations, the components of q that are output by `outN.f` are the mass, momentum, and energy, but you may wish to plot the velocity, pressure, or Mach number. You could modify the `outN.f` routine to compute and print the desired quantity, but to view something different it would be necessary to re-run the code. Instead, the desired quantity can be computed in MATLAB and easily changed without repeating the entire computation. This is easily accomplished by setting `UserVariable = 1` and providing an m-file that converts an array `data`, which contains all the values read in from

`fort.qXXXX` (with `meqn` columns, corresponding to the components of q), into a vector that contains the desired quantity.

For example, computing the velocity from an N-dimensional Euler calculation can be accomplished by the following m-file `claw/matlab/xvelocity.m`. (In this case the second component of q is the x-momentum, ρu.)

```
function u = xvelocity(data)
% compute the x-velocity from the data
% for problems where the second component is the x-momentum and the
% first component is the "density"  (e.g. Euler, shallow water equations)
u = data(:,2)./data(:,1);
```

To use this m-file, it must be in the current directory or search path. In MATLAB, set

```
UserVariable = 1;
UserVariableFile = 'xvelocity';
```

The string variable `UserVariableFile` specifies the name of the m-file that computes the desired quantity from the data.

Other sample files can be found in `claw/matlab`, for example `pressure.m, mach.m`.

2.7 Creating 1d scatter and line plots from 2d and 3d data

, Using `PlotType == 4`, the user can create a 1d plot of their 2d or 3d data. There are now two ways of doing this. For radially and symmetric data, the user can specify values `x0`, `y0`, and (`z0` in three dimensions) in their `setplotN.m` file and a plot of r verses `q` will be produced.

For data which is not radially or spherically symmetric, but for which a 1d scatter or line plot might be appropriate, users can now define their own mapping function which takes 2d or 3d data and maps it to 1d. To specify that a user-defined 1d mapping function should be used, the user should first set `UserMap1d = 1` in their `setplotN.m` file. Then they should write a function `map1d.m` which takes three arguments (or four, in three dimensions) and returns two arguments, the vectors of the dependent and independent variables. For example, using the following `map1d` function, the user can plot their two dimensional data as a function of `x` only :

```
function [x,q] = map1d(xcm,ycm,qcm)
% xcm, ycm are results of meshgrid(xcenter, ycenter)

[mx,my] = size(xcm);
q = reshape(qcm,mx*my,1);
x = reshape(xcm,mx*my,1);
```

As another example, a user may wish to plot their solution along a given line or curve. The following `map1d` function illustrates how to plot data interpolated to a diagonal across a two-dimensional domain :

```
function [s,q] = map1d(xcm,ycm,qcm)
% xcm, ycm are results of meshgrid(xcenter, ycenter)
[mx,my] = size(xcm);
xmin = xcm(1,1);
ymin = ycm(1,1);
xmax  xcm(mx,my);
ymax = ycm(mx,my);
```

```
sx = linspace(xmin,xmax,100);
sy = linspace(ymin,ymax,100);
q = interp2d(xcm,ycm,qcm,sx,sy,'*linear');

% Plot q verses points along the diagonal
d = sqrt((xmax-xmin)^2 + (ymax-ymin)^2);
s = linspace(0,d,100);
```

For this example, the (s, q) data is sorted, and so it may be appropriate to plot it as a line plot, rather than a scatter plot. This can be done by setting `ScatterStyle` (or equivalently, `LineStyle`) to a line style such as `'-'` or `'--'`. Similar mapping functions can be written for three dimensional data as well.

The arguments to `map1d` are arrays of the independent variables, produced by `meshgrid`, and a conforming array of the dependent variable. The independent variables are in Cartesian, not physical, coordinates, so if the user has set `MappedGrid = 1`, then they should explicitly call `mapc2p` from their `map1d` routine.

It is often desirable to plot the solution to a 1d problem which has been transformed into radial or spherical coordinates, on top of the 2d or 3d scatter or line plots. To do this, one can create a subdirectory with the 1d computational results, and then from an `afterframe` file (see below), the user can call `readamrdata` to read the 1d data, and then `plotframe1ez` to plot the 1d data on top of the 2d or 3d lines or scatter plots. For more information on this function, type `help plotframe1ez` MATLAB prompt.

2.8 Nonuniform grids and the MappedGrid option

If the `plotclawN` parameter `MappedGrid` is zero then the data is plotted on a uniform grid with grid spacing `dx` in one dimension, on a Cartesian grid with spacing `dx` and `dy` in two dimensions, or with with spacing `dx`, `dy`, and `dz` in three dimensions.

If `MappedGrid` is nonzero then a grid mapping is applied before plotting the data. Version 4.2 of the CLAWPACK graphics now supports mapped grids in three dimensions.

A uniform computational grid denoted by `xc`, `(xc,yc)` or `(xc,yc,zc)` is first set up with uniform spacing as above. However, it is now assumed that the user has provided a MATLAB function `mapc2p.m` that maps these computational points into the corresponding physical points. This function has the form

```
[xp,yp] = mapc2p(xc,yc);
```

in two dimensions, and

```
[xp,yp,zp] = mapc2p(xc,yc,zc);
```

in three dimensions. The data is then plotted using these physical coordinates. The same plotting options and types are available as when plotting on a uniform grid. For more information, type `help mappedgrid` at the MATLAB prompt.

New to version 4.2: It is now possible to use the `MappedGrid=1` option in 3d. The user must supply a 3d `mapc2p.m` function, and all three dimensional graphical elements will be mapped to physical coordinates before being plotted.

To apply CLAWPACK on a nonuniform grid, see the discussions in Chapters 6 and 23 of [10].

2.9 Additional script files

Some other script files may be called from `plotclawN.m`. If you are plotting results obtained by running a code in one of the `claw/applications` or `claw/book` directories (or that you have obtained from some other source) and the plots don't appear as expected, check to see if one of these files exists and is modifying its appearance. In particular, these may need to be modified if you change the problem or initial data.

`setprob.m`: When the script `plotclawN` is invoked, it checks for a script called `setprob.m` in the current directory and executes it if found. You can create such a script if there are problem-specific parameters you need to set before starting to plot.

`beforeframe.m`: If a script with this name exists in the current directory, it is executed before plotting each frame. You can put commands in this script that specify the plotting region (using the `axes` command) for example. Or you might want to open a new plotting window for each frame so that you can compare each with previous frames.

`afterframe.m`: If a script with this name exists in the current directory, it is executed after plotting each frame. You can put commands in this script that modify the plot. For example, you might want to put a different title on the top in place of the default title that `plotclawN` generates, or add a `colorbar` to pseudo-color plots. A common use in one dimension is to apply the `axis` command to set the vertical scale so that each frame is plotted on the same axis (otherwise MATLAB will choose the scale for each frame based on the values of the data at that particular time.) `Hint:` If you are making changes to `afterframe.m` to adjust the appearance of your plot, add `clear afterframe` to the end of the file so that your new afterframe will be loaded automatically after each change you make to the file.

2.10 Changing plot characteristics directly from the command line

The typical way to change plot characteristics such as `PlotType`, visibility of grid lines or contour lines, patch edges (for AMR plots), and slice characteristics is to change the value of the corresponding `setplotN` variable (e.g. `PlotType`, `PlotGrid`, `PlotGridEdges`, etc), either in `setplotN` directly, or at the `K>` prompt. In either case, you have to re-generate the entire plot. For 2d graphics, this is usually not a problem, but for 3d plots, with many slices, it can be time consuming to always have to re-generate the figure whenever minor changes to plot characteristics are desired. To speed up this process, there have been a number of functions added to the CLAWPACK graphics which instantly make the desired changes to your plot. As an example, typing `showgridlines` at either the MATLAB prompt, or at the `K>` prompt will automatically make all grid lines on slice or surface plots visible. Typing `showgridlines(1:2)` shows grid lines on levels 1 and 2; typing `hidegridlines` hides all grid lines.

Several other similar functions are available. These are all documented in the online help, which the user can access by typing `help clawgraphics` at the MATLAB prompt.

2.11 Customizing `outN.f`

The default output routines located in `claw/clawpack/Nd/lib/outN.f` write out the solution in a form that is suitable for use with the MATLAB graphics routines described above. As with any CLAWPACK routine, you can customize the output routine for your own needs if the default version is not adequate. (Note that the HDF output routines and the MPICLAW output routines described in chapter 4 may be customized in a similar fashion.) Simply copy the library version to your application directory, modify it as desired, and modify the `Makefile` to point to the modified version. For example, you might want to:

- Change the format of the output to meet the requirements of some other graphics package.
- Print out more (or fewer) digits of the solution. (The HDF output routines could be customized to output simulations results in single-precision, rather than the default double-precision format.)
- Print out only some components of the solution rather than all `meqn` components. This may be desirable for a large system if only some components are of interest (in order to save disk space).

- Print out the solution only over part of the domain rather than everywhere.
- Print out some quantities derived from the solution values rather than the components of q itself. For the Euler equations, for example, you might want to print out density, velocity, and pressure rather than the conserved quantities. (Alternatively, there is a provision in the MATLAB routines to specify a function to apply to the solution values before plotting, so that the velocity or pressure can be computed from the conserved quantities, for example. See 2.6.)

Chapter 3

Adaptive Mesh Refinement and the AMRCLAW Routines

CLAWPACK 4.1 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.

These routines are now available in both 2 and 3 space dimensions. The basic AMRCLAW routines can be found in `claw/amrclaw/Nd/lib`, for `N`$= 2, 3$. 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 Chapter 2). Note that there are no user-supplied Fortran routines in these directories; the `Makefile` refers to the routines in the parent directory.

The boundary condition routine that sets ghost cell values is slightly more complicated for AMRCLAW, as described in Section 3.1.3, but the standard boundary conditions handled automatically in CLAWPACK (extrapolation, periodic, solid walls) are also implemented in the AMRCLAW routines `claw/amrclaw/Nd/lib/bcNamr.f` 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 3.1).

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.

The MATLAB routines `plotclaw2` and `plotclaw3` can be used for viewing the results. Most of the same plotting options described in Chapter 2 for CLAWPACK work also for AMRCLAW output. See Section 2.5 for some plotting parameters specific to AMRCLAW output.

3.1 The input files `amr2ez.data` and `amr3ez.data`

The input data file for the adaptive routines are slightly different and are now called `amrNez.data`. These have essentially the same form as the `clawNez.data` file used by `clawNez`, but for the adaptive routines some additional parameters must be specified.

A sample file can be found in `claw/clawpack/2d/example1/amr/amr2ez.data`. The parameters `mxnest` and `inrat` and all those from `ichkpt` 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. 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 in, so this line must 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:

`"xleft"`	a value associated with the "left" edge of the cell in the x-direction,
`"yleft"`	a value associated with the "left" edge of the cell in the y-direction,
`"zleft"`	a value associated with the "left" edge of the cell in the z-direction (`"zleft"` is an option only in `amr3ez.data`),
`"center"`	a cell-centered value,
`"capacity"`	a cell-centered capacity function.

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.

A cell-centered auxiliary value such as the density or impedance in a heterogeneous acoustics problem would have type `"center"`. On the other hand, in a variable-coefficient advection problem we may want to store the normal velocity at each edge of the cell. In two dimensions we might use one component of `aux` to store the value $u_{i-1/2,j}$ at the left edge, which would have type `"xleft"`, and another component of `aux` to store the value $v_{i,j-1/2}$ at the bottom edge, which would have type `"yleft"`.

In previous versions of the 2d AMRCLAW, `"xleft"` and `"yleft"` were called `"leftface"` and `"bottomface"` respectively. These terms can still be used, but the new terminology leaves less room for confusion when extending to three dimensions.

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.

`ichkpt`: A checkpoint file is dumped every `ichkpt` time steps on the coarse grid. These are binary files with names of the form `fort.chkXXXX` where `XXXX` is the step number. (This parameter was called `iousr` in Version 4.0.) Note: these files are typically very large!

The solution and grid structure is printed out 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 `ichkpt` 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. Setting `ichkpt` to a sufficiently large integer will cause a checkpoint file to be written only at the end of the computation.

If `ichkpt = 0` then no checkpoint files are generated, not even at the end.

restart: If `restart = T` then a restart is performed. Information read in from the file `restart.data` is used to resume a previous calculation. An appropriate checkpoint file `fort.chkXXXX` should be renamed `restart.data` in order to use this option.

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)`, refer to the total 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 that need to be refined. An estimate of the truncation error is computed and compared with this value, see Section 3.4. Smaller values will lead to more refinement.

tolsp: Another tolerance used in flagging grid cells that need to be refined. This is used in checking the magnitude of the spatial gradient, as computed by central differences. See Section 3.4 for more information about how `tol` and `tolsp` are used and what routines can be modified to change the refinement criteria. Like `tol`, a smaller value leads to more refinement.

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 3.4) 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 that 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 3.4). 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` or `plotclaw3` 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.

3.1.1 Source terms and `src1d.f`

If a fractional step method is used to incorporate source terms, then the routine `src2.f` must be provided as with the standard CLAWPACK. However, for the adaptive code, it is also necessary to provide another routine `src1d.f` that takes a single time step with the source term equation in the same manner as `src2.f`, but that takes a one-dimensional array of data as input instead of a two-dimensional array over a full grid patch. This routine is required for the process of performing a "conservative fix-up" at edges

of grid patches and is used to apply source terms over partial time steps to the coarse grid cell values used in solving Riemann problems at the interface between coarse and fine grids.

The dummy routine `claw/amrclaw/2d/lib/src1d.f` gives some more information about how this should be structured.

3.1.2 AMR parameters and the file `call.i`

Since the AMRCLAW routines are written in Fortan 77, all of the memory management is done within this code by splitting up one large work array `alloc(1:memsize)` into many pieces as needed for all the various arrays on each grid patch. The parameter `memsize` is the dimension of `alloc` and this limits the degree of refinement that will be possible without running out of memory.

The array `alloc` is declared in a file `claw/amrclaw/2d/lib/call.i` that is incorporated into most other Fortran files in this library using the Fortran `include` command. Many other parameters are also set in this file and some other smaller arrays are declared. In addition, a number of common blocks are declared that are used to pass information around between AMR routines. In particular, `alloc` is stored in the common block `calloc`.

Other particularly important parameters in this file are:

`maxgr`: maximum number of grids allowed.

`maxlv`: maximum number of levels allowed.

`max1d`: maximum number of grid cells in any direction on any grid.

`maxvar`: maximum number of components of `q` (equations in system)

`maxaux`: maximum number of components of `aux` (auxiliary variables)

If you want to change any of these parameters (*e.g.*, to solve larger problems), then after changing `call.i` it will be necessary to recompile all the library routines (by doing `rm *.o` followed by `make` in `claw/amrclaw/2d/lib`).

If you wish to have direct access to any of the common blocks or parameters from `call.i` in subroutines that are in your own application directory, then you may need to copy this routine to your directory. (Or do a link to the library version using the Unix command `ln`.)

In particular, if you copy any library routines to your directory in order to use modified versions (*e.g.*, `errf1.f` for error estimation; see Section 3.4), then you should check to see if these routines contain the statement

```
include 'call.i'
```

If so, you will need to copy this file to your directory.

3.1.3 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/Nd/lib/bcNamr.f` (with `N = 2` or 3) sets boundary conditions at the physical boundaries. It recognizes the same set of `mthbc` values as used in `clawNez`, 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 `bcNamr.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.

3.2 Three dimensions

Version 4.1 of CLAWPACK contains adaptive routines in three space dimensions, in the directory `claw/amrclaw/3d/lib`. If you understand how to use the two-dimensional AMRCLAW and the basic three-dimensional CLAWPACK routines, there should be no new surprises here.

3.3 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 `bcNamr` 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 that lie within adjacent grids at Level 2. Appropriate values are copied directly from the adjacent grid.

   (b) Those that 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 that lie outside the physical domain. For these cells `bcNamr` 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 that 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. A general "fix-up" is applied in all cases that is designed to maintain conservation for conservation laws. This is described in [4].

5. Every `kcheck` time steps on each level, error estimation and regridding is performed. This is done as described in Section 3.4.

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.

3.4 Error estimation and regridding

Every `kcheck` time steps on each level, the error is estimated in all cells on grids at this level. Cells where the error is above some tolerance are flagged for refinement. The norm used to measure the error can be adjusted, see below. The cells that have been flagged are then clustered into rectangular regions to form grids at the next finer level. The clustering is done in light of the tradeoffs between a few large grids (which usually means refinement of many additional cells that 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 `amrNez.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. The value `cutoff = 0.7` is usually reasonable.

Cells are flagged for refinement in the subroutine `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 maximizing the absolute value over all components `m`. A cell is flagged if this is greater than the parameter `tolsp` in `amr2ez.data`. This flagging 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.`. In the default version `claw/amrclaw/2d/lib/errf1.f`, the function statement

```
allowed(x,y,level) = .true.
```

appears. As an example of how this might be changed, suppose you want to allow refinement past level 2 only on the portion of the domain for $x > 0.5$. Then you could modify this to read

```
allowed(x,y,level) = level.lt.2 .or. x.gt.0.5d0
```

In three dimensions the corresponding function is `allowed(x,y,z,level)`.

3.5 Comments and warnings

For many problems the adaptive code should work immediately when a CLAWPACK code is converted, often simply by providing a new `Makefile` and amr2ez.data. However, there are several subtleties of the adaptive refinement procedure that can lead to problems. Here are some things to watch out for.

- **Using `ql` and `qr` appropriately.** When AMRCLAW is used, it is important that the normal Riemann solvers `rpn2` and `rpn3` use the values `qr(i-1,:)` and `ql(i,:)` as data for the Riemann

problem between cells `i-1` and `i`; see Section 1.11.1. This is because the AMRCLAW routine `qad.f` uses these arrays in a different way than the standard CLAWPACK routines. (This routine performs the "conservative fix-up" described in [4].) Rather than corresponding to data along a slice in one dimension, the data comes from sets of states along the edge of a refinement patch, with one array taken from the coarse grid and the other from the fine grid.

Similarly, if `aux` arrays are being used, the appropriate values to use are `auxr(i-1,:)` and `auxl(i,:)`.

- **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` (or `b4step3`), this will happen automatically on the new grids before the first call to `stepN`. In this case the `setaux` routine need only set the time-independent values in `aux` that are not set by `b4stepN`.

- **Error estimation time steps.** In the error estimation process, time steps are taken to estimate the error. This is not part of the main calculation. The routine `b4stepN` and the source term routine `srcN` are called in this process as in any other time step. If the `b4stepN` 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 based on the current time step, and `method(1) = 0` is set. This time step is not adjusted based on the observed Courant number.

- **Exceeding the CFL limit.** If `method(1)=1` then CLAWPACK attempts to automatically adjust the time step to keep the Courant number near the value specified in `cflv(2)`. If the Courant number is above the limit specified in `cflv(1)`, then the initial data for this time step is restored and a smaller time step is taken. In `amrclaw` this is only partially true. At the end of each time step on Level 1 (the coarsest level), the next coarse time step is chosen based on the largest wave speed seen in the last step (maximizing over what was observed on all levels). If this value causes the Courant number on the Level 1 step to be larger than `cflv(1)`, then this coarse step is retaken with a smaller `dt`, just as in CLAWPACK. However, once a time step is accepted on the coarsest level, this same time step, divided by appropriate values of `inrat`, is used for all the finer level time steps within this single coarse time step. It may happen that within one of these finer time steps (*i.e.*, with `Level > 1`), the Courant number is observed to go above `cflv(1)`. This is ignored and the computation proceeds, with the time step adjusted only at the start of the next `Level 1` time step. Trying to adjust the time step at a finer level would be difficult as it would require going back and retaking the coarser steps.

- **Global coupling in source terms.** If a fractional step method is used to couple in source terms, and if these source terms are not simply applied pointwise, but involve global coupling of the solution across the entire grid (for example when the source term is a diffusive term or viscous term), then it will not generally be possible to use AMRCLAW in its current form.

Chapter 4

MPICLAW: an MPI version of CLAWPACK

For users with more than one processor at their disposal (through a multiple-processor workstation/server, a beowulf cluster or supercomputer), two- and three-dimensional versions of CLAWPACK are available which make use of MPI (the message passing interface) to distribute the computational work across the available processors. The domain is split into slices or an array of blocks, and ghost cells are used to pass information between these subdomains at the end of each time step.

Each of the `clawpack/Nd/exampleN` directories (as well as some of the `application` directories) contain `mpi` subdirectories which demonstrate the use of the MPICLAW routines. These routines require the same basic routines (e.g. `setaux.f`, `qinit.f`, etc.) that are used by standard, single-processor CLAWPACK. The only changes are a new `Makefile` and N additional lines at the bottom of the `clawNez.data` file which specify how the processors should be distributed across the N dimensions. Note that once an application has been set up for CLAWPACK little additional work is necessary to get the application running with `mpiclaw`.

The use of MPICLAW is encouraged for large simulations, because the methods used in CLAWPACK allow the distribution of work across a number of processors with a minimum of communication. Much of the computational work takes place in updating the solution on each subdomain, which can be done using only the local information on this processor (and its ghost cells). Communication between processors is only required to update the ghost cell values of other processors at the beginning of each time step. This is accomplished by copying the values from `mbc` rows of cells near the boundary of each subdomain into the ghost cells for the neighboring subdomains before the start of each time step. This communication typically has a trivial cost compared to the cost of applying Godunov-type methods over each subdomain. Because of this, one can typically expect to see speedup of $0.8M - 0.9M$ when M processors are used. The speedup is limited primarily by the ratio of the surface area of the subdomains to their volume: the larger surface area, the more time spent in communication between subdomains/processors.

Note that the MPI version may not apply to some problems if there is a more global coupling between values. For example, if an advection-diffusion equation is solved by using a fractional step method with an implicit method for the diffusion "source term", then the routine `srcN.f` cannot be applied independently on different subdomains.

There is no MPI version of the AMRCLAW routines, since it is more difficult to combine parallel processing with adaptive refinement. Users who desire this feature should consider the BEARCLAW software developed by Sorin Mitran. See `http://www.amath.unc.edu/Faculty/mitran/bearclaw.html`.

Note that the MPI implementation of CLAWPACK has changed substantially from the last release. In addition to bug fixes, the FORTRAN77 style of standard CLAWPACK was adopted for uniformity.

4.1 MPICLAW routines

The routines are located in the `claw/clawpack/Nd/mpilib` directory and include driver, boundary condition and output routines. Specifically, they are:

- `clawNez_mpi_driver.f`: Driver routine that read run parameters from `clawNez.data`, distributes that information to all processors and calls `clawNez_mpi`.
- `clawNez_mpi.f`: Contains the `clawNez_mpi` subroutine, which contains the main time-stepping loop and calls the output subroutines.
- `clawN_mpi.f`: Advances the simulation from one output time to the next. Calls the boundary condition routines as well as the routines that advance the simulation forward by a single time step (`stepNds` or `stepN`, depending on whether or not dimensional splitting is used.).
- `bcN_mpi.f`: Supplies ghost cell values for both physical boundary conditions (i.e. extrapolation or solid wall) and internal boundaries (i.e. boundaries between blocks of data distributed on different processors).
- `bcN_aux_mpi.f`: Supplies ghost cell values for aux arrays at internal boundaries (i.e. boundaries between blocks of data distributed on different processors). For some applications, aux arrays (which hold things like advection velocities and grid metrics) may vary with time. In such cases, the aux array values are updated in the `b4stepN.f` routine, and this new information should be shared with other processors by calling the `bcN_aux` subroutine just after `b4stepN`. To do this, copy `clawN_mpi.f` into the application/example directory, edit the `Makefile` to refer to this copy of `clawN_mpi` and uncomment the call to `bcN_aux` which follows `b4stepN` in `clawN_mpi.f`.
- `outN_mpi.f`: Produces ASCII output files numbered `fort.qXXXX.YY` where `XXXX` is the number of the output and `YY` is the number of the processor. **Important Note:** Type `make catfiles` in the directory containing the `fort.qXXXX.YY` files to concatenate these into a set of `fort.qXXXX` file suitable for use with the MATLAB graphics routines. If these files have already been processed to generate `fort.qXXXX` files, the following error message will be returned by make:

  ```
  make: *** No rule to make target 'fort.q0000.00', needed by 'fort.q0000'.  Stop.
  ```

  Note that `catfiles` is a PERL script residing in `claw/clawpack/3d/mpilib`. The first line of the script should contain the correct path to the PERL executable on your system. The default first line `#/usr/bin/perl` should be changed if necessary. Type `which perl` to find the location of the PERL executable on your system.
- `outN_mpi_hdf.f`: Generates HDF output files. A single `fort.qXXXX.hdf` file is written at each output step. The format of this file is identical to the single processor HDF format, providing a simple means for comparison between output from runs on different numbers of processors. (See section 2.2 for details on HDF output.) The MATLAB graphics routines will recognize these output files if the command `OutputFlag = 'hdf'` is given in MATLAB before calling `plotclawN`.

4.2 Restarting MPICLAW

As of version 4.2, MPICLAW simulations may be restarted from old output files using the `restartN_mpi_hdf.f` routine supplied in `clawpack/Nd/mpilib` for `N = 2` or 3. See Section 1.13 for details on the restart process. Examples showing the necessary changes to `Makefile` and `qinit.f` are given in `clawpack/Nd/example1/` and `clawpack/Nd/example1/mpi`.

Please note: Restarting from ASCII output (generated by `outN_mpi.f`) is not supported.

Bibliography

[1] D. Bale, R. J. LeVeque, S. Mitran, and J. A. Rossmanith. A wave-propagation method for conservation laws and balance laws with spatially varying flux functions. *SIAM J. Sci. Comput.*, 24:955–978, 2002.

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

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

[4] 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.

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

[6] J. O. Langseth and R. J. LeVeque. A wave-propagation method for three-dimensional hyperbolic conservation laws. *J. Comput. Phys.*, 165:126–166, 2000.

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

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

[9] R. J. LeVeque. Balancing source terms and flux gradients in high-resolution Godunov methods: The quasi-steady wave-propagation algorithm. *J. Comput. Phys.*, 146:346–365, 1998.

[10] R. J. LeVeque. *Finite Volume Methods for Hyperbolic Problems.* Cambridge University Press, 2002.

Index

3d slices, 33

acoustics, 19
adaptive mesh refinement (AMR), 3, 7, 41
afterframe.m, 38
alloc, 44
amdq, 14, 23
amr2ez.data, 41
amr3ez.data, 41
amrclaw, 28
amrclaw, 41
apdq, 14, 23
asdq, 21
aux, 12, 18
aux1, 21, 24, 25
aux2, 21, 24, 25
aux3, 21, 24, 25
auxiliary arrays, 18
    with AMR, 47
auxiliary variables, 16
auxl, 14, 20, 23
auxr, 14, 20, 23
auxtype, 42

b4step1.f, 18
b4step2.f, 47
b4step3.f, 47
bc1.f, 10, 13, 17
bcNamr.f, 41, 44
bearclaw, 49
beforeframe.m, 38
bmasdq, 21
bottomface, 42
boundary conditions, 10, 13
    with adaptive refinement, 44
bpasdq, 21
bsamdq, 25
bsapdq, 25
bsasdq, 24

call.i, 44
capacity, 42
capacity functions, 10, 16
center, 42

cflv, 15, 47
citing CLAWPACK, 7
claw1.f, 13
claw1ez.data, 13, 14
claw1ez.f, 9, 13
claw1program.f, 11
claw2ez.data, 19
claw2ez.f, 19
claw3ez.data, 21
claw3ez.f, 21
cmbsasdq, 24
compile, 11
comxt, 18
contour plots, 31
ContourValues, 31, 33
copyright, 4
cpbsasdq, 24
cutoff, 43

download, 10
driver.f, 12
dtcom, 18
dtv, 15
dxcom, 18

environment variables, 11
errf1.f, 46
error estimation, 46
errsp.f, 46
example1, 12

fluctuations, 9, 14
flux-difference splitting, 9
fort.chk, 42

Godunov's method, 9, 14
graphics, 11

HDF output, 27, 28, 38, 50
high-resolution methods, 9, 13

ibuff, 43
ichkpt, 42
icoor, 23, 24

imp, 21, 23, 25
impt, 25
initial conditions, 13
inrat, 42, 45
isosurface plots, 34
isosurfaces, 33
IsosurfAlphas, 34
IsosurfColors, 34
IsosurfValues, 34
ixy, 20, 21
ixyz, 22, 24

K>> prompt, 30
kcheck, 43, 45
keyboard input from plotclawN.m, 30

leftface, 42
limiter.f, 17
limiters, 17
LineStyle, 37

mach.m, 36
make catfiles, 50
make program, 11
makefiles, 11
map1d.m, 36
mapc2p.m, 37
MappedGrid, 31, 37
matlab, 10, 11, 41, 43
MATLABPATH, 11
maux, 12, 16
max1d, 44
maxaux, 44
MaxFrames, 31
maxgr, 44
maxlv, 44
maxmx, 12
maxvar, 44
mbc, 12, 17
mcapa, 16, 42
memsize, 44
meqn, 12, 16
method, 15, 20, 21
MPICLAW, 38, 49
mq, 31
mthbc, 20, 22
mthlim, 17
mwaves, 12, 16
mwork, 12
mx, 15
mxnest, 42
my, 19

NCAR, 43
nout, 15
nstepout, 15
nsteps, 15
nstop, 43
nv, 15

outN.f, 27
outN_hdf.f, 27, 28
outstyle, 15

philim.f, 17
plotclaw2, 41
plotclaw3, 41
plotclawN.m, 11, 30
PlotCubeEdge, 35
PlotData, 35
plotframe1ez.m, 37
PlotGrid, 31, 33, 35
PlotGridEdge, 35
PlotStyle, 31
PlotType, 31, 33
pressure.m, 36
PRINT, 43

q, 12
qinit.f, 13
ql, 13, 20, 22, 46
qr, 13, 20, 22, 46

readamrdata.m, 30
restart, 43
restarting CLAWPACK, 25
restarting MPICLAW, 50
Riemann solvers, 9
    in one dimension, 13
    double transverse, 24
    in three dimensions, 22
    in two dimensions, 20
    transverse, 21
rp1.f, 9, 13
rpn2.f, 20
rpn3.f, 22
rpt2.f, 21
rpt3.f, 23
rptt3.f, 24

s, 14
scatter plots, 32, 34, 36
ScatterStyle, 32, 34
Schlieren plots, 32, 34
setaux.f, 18, 47
setenv, 11

setplotN.m, 31
setprob.data, 12
setprob.f, 12, 17
setprob.m, 38
setviews, 33
source terms, 10, 16, 18, 43, 47
speeds, 9
src1.f, 10, 16, 18
src1d.f, 43

t0, 17
tar files, 10
tcom, 18
tol, 43
tolsp, 43
tout, 43

UserVariable, 31, 35
UserVariableFile, 36

verbosity, 16
    at different refinement levels, 16
versions, 8
view, 33

warnings, 23, 46
wave, 14
waves, 9
website, 10
work, 12

x-like direction, 23
xleft, 42
xlower, 17
xSliceCoords, 33
xupper, 17
xvelocity.m, 36

y-like direction, 23
yleft, 42
ylower, 20
ySliceCoords, 33
yupper, 20

z-like direction, 23
zleft, 42
zlower, 22
zSliceCoords, 33
zupper, 22