CLAWPACK Version 4.0
User's Guide

Randall J. LeVeque
University of Washington

DRAFT VERSION
July 23, 2002

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

©R. J. LeVeque, 2002

Authors.

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

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

The three-dimensional routines were written by

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

The MPI routines were written by

Sorin Mitran
Mathematics Department
University of North Carolina
Chapel Hill, NC 27599, USA

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

Acknowledgements.

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

Development of this software has been supported in part by

NSF Grants DMS-8657319, DMS-9204329, DMS-9303404, and DMS-9505021,
DMS-96226645, DMS-9803442, 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:

This software is made available for research and instructional use only. You may copy and use this software without charge for these non-commercial purposes, provided that the copyright notice and associated text is reproduced on all copies. For all other uses (including distribution of modified versions), please contact the authors.

This software is made available "as is" without any assurance that it will work for your purposes. The software may in fact have defects, use the software at your own risk.

Copyright by the authors, 1995–2002.

Status:

This documentation is being revised during the summer of 2002. Some changes are also being made in CLAWPACK and Version 4.1 should be available by the end of the summer.

There are only a few changes in the one-dimensional and two-dimensional routines. More substantial changes are being made in the basic three-dimensional routines to fix some bugs. A three-dimensional version of AMRCLAW is also being developed, and some bugs are being fixed in the two-dimensional AMRCLAW.

Check back later this summer for updates and refinements of this documentation.

Contents

1 The basic CLAWPACK software **7**
1.1 Introduction . . . 7
1.2 Other references . . . 7
1.3 Versions . . . 8
1.4 Basic framework . . . 8
1.5 Obtaining CLAWPACK . . . 9
1.6 Getting started . . . 9
1.6.1 MATLAB graphics . . . 11
1.7 Using CLAWPACK — A guide through `example1` . . . 11
1.7.1 The main program (`driver.f`) . . . 11
1.7.2 The initial conditions (`qinit.f`) . . . 12
1.7.3 The `claw1ez` routine . . . 12
1.7.4 Boundary conditions . . . 12
1.7.5 The Riemann solver . . . 12
1.7.6 The input file `claw1ez.data` . . . 14
1.8 Other user-supplied routines and files . . . 16
1.9 Auxiliary arrays and `setaux.f` . . . 17
1.10 An acoustics example . . . 18
1.11 Two space dimensions (`claw2ez.f`) . . . 18
1.11.1 Riemann solvers . . . 19
1.12 Three space dimensions (`claw3ez.f`) . . . 20

2 Program output and graphics using MATLAB **23**
2.1 One-dimensional output . . . 23
2.2 Two-dimensional output . . . 24
2.3 Plotting results using `plotclawN.m` . . . 24

3 Adaptive Mesh Refinement and the AMRCLAW Routines **27**
3.1 Two dimensions . . . 27
3.1.1 The input file `amr2ez.data` . . . 27
3.1.2 Boundary conditions . . . 29
3.1.3 Plotting results with MATLAB . . . 29
3.2 Three dimensions . . . 30
3.3 The adaptive algorithm . . . 30
3.4 Error estimation and regridding . . . 31
3.5 Comments and warnings . . . 32

4 MPI Versions for multiple processors **33**

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* [8]. This book also contains a general discussion of hyperbolic problems arising in several particular applications 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 interst 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 [7]. The three-dimensional algorithms are developed and analyzed in [5]. The ideas are presented for the relatively simple case of the advection equation in two and three dimensions in [6].

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, [4], [1], [2]. Some of the issues involved in coupling these codes together with CLAWPACK, and generalizing them to allow nonconservative systems, can be found in the paper [3].

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 should be available soon – see page 3.
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.

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 Section **??**. 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 introduced in Section 1.4,

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

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 [8]. These methods take the form

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

where

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

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 [8] 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 $q_t = \psi(q, \kappa)$ over a time step. A fractional step method is used to couple this with the homogeneous solution, as described in Chapter 17 of [8]. Boundary conditions are imposed by setting values in ghost cells each time step, as described in Chapter 7 of [8]. A few standard boundary conditions are implemented in the library routine `claw/clawpack/1d/lib/bc1.f`, but 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/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 deal with 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 values 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 [8]. 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 [8]. 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

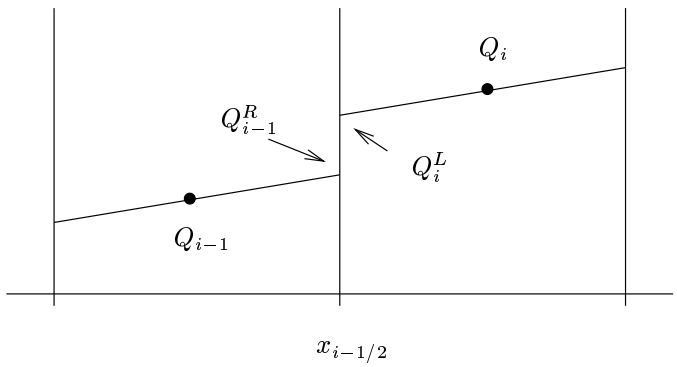

Figure 1.1: The states used in solving the Riemann problem at the interface $x_{i-1/2}$.

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 Section **??**.

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

$$\begin{aligned} &\text{left state: } Q^R_{i-1} = \texttt{qr}(\texttt{i}-1,:), \\ &\text{right state: } Q^L_i = \texttt{ql}(\texttt{i},:). \end{aligned} \tag{1.11}$$

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. 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 $\mathcal{W}^p_{i-1/2}$ 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 $s^p_{i-1/2}$ 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 [8].

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)`: 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(5)`: Tells whether there is a source term in the equation. If so, then a fractional step method is used as described in Chapter 17 of [8]. Time steps on the homogeneous hyperbolic equation are alternated with time steps on the source term. The solution operator for the source terms must be provided by the user in the routine `src1.f`.

- `method(5) = 0`: There is no source term. In this case the default routine `claw/clawpack/1d/lib/src1.f` can be used which does nothing, and in fact this routine will never be called.
- `method(5) = 1`: A source term is specified in `src1.f` and the "first order (Godunov)" fractional step method should be used.
- `method(5) = 2`: A source term is specified in `src1.f` and a Strang splitting is used.

The Godunov splitting is generally recommended rather than the Strang splitting for reasons discussed in Chapter 17.

`method(6)`: Tells whether there is a "capacity function" in the equation, as introduced discussed in [7] and [8].

- `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 [8].

`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. Several different limiters are provided in CLAWPACK (see Chapter 6 of [8]):

`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`, 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 [8] 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 [8].

`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 routine `src1` 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`.

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 [8] 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 called only once at the beginning of the computation and cannot be used 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. 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`.

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 [7] and Chapter 21 of [8] 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 [7] and Chapter 21 of [8]. The transverse solver `rpt2` is called only if `method(3) > 0` and a dummy routine could be provided otherwise.

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

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

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

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

`imp = 1:` `asdq` represents `amdq`, the fluctuation 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.

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`), 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 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 the 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 might want to apply "double-transverse" corrections as well as transverse corrections, as described in [5]. 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.)

`method(3) = 10`: Transverse propagation of the increment wave as in 2D. This method is unconditionally unstable.

`method(3) = 11`: Corner transport upwind of the increment wave. This method is unconditionally unstable.

`method(3) = 20`: Both the increment wave and the correction wave propagate as in the 2D case. Only to be used with method(2) = 2.

`method(3) = 21`: Corner transport upwind of the increment wave, and the correction wave propagates as in 2D. Only to be used with method(2) = 2.

`method(3) = 22`: 3D propagation of both the increment wave and the correction wave. Only to be used with method(2) = 2.

`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)`.

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

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

The parameter `icoor` indicates which of the transverse directions is to be used for the transverse splitting:

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

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

`ixyz=2, icoor=2`: Data is in y, split in z.

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

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

`ixyz=3, icoor=3`: Data is in z, split in y.

As in two dimensions, a parameter `imp` indicates whether `asdq` represents `amdq` or `apdq`:

`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.

Chapter 2

Program output and graphics using MATLAB

The CLAWPACK routines `clawNez.f` (for `N`= 1, 2, 3) are set up to call corresponding output routines `outN.f` at each time when output is desired. The output times are specified in the `clawNez.data` files, as described in Section 1.4. The default output routines located in `claw/clawpack/Nd/lib` write out the solution in a form that is suitable for use with the MATLAB graphics routines described later in this chapter. As with any CLAWPACK routine, you can customize the output routine for your own needs if the default version is not adequate (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.
- 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 pressure can be computed from the conserved quantities, for example. See Section **??**.)

2.1 One-dimensional output

The default version of `out1.f` produces two files at each output time. These are called `fort.tXXXX` and `fort.qXXXX`, where `XXXX` gives the frame number, 0000 for to 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.t0000` 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. When using the basic CLAWPACK routines, `ngrids = 1`. With AMRCLAW there may be multiple grids at each output time,

one or more at each Level (see Chapter 3). The value of `meqn` is used in `plotclawN` so if you modify the output routine to print out only some solution values, this value should be changed accordingly.

`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.

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 `4e16.8` is used for printing the values. So roughly nine 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. 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 Two-dimensional output

The default routine `out2.f` behaves similarly. Again two files `fort.tXXXX` and `fort.qXXXX` are produced at each output time. The former is identical to what is described above. The latter file is essentially the same but also includes 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.

2.3 Plotting results using `plotclawN.m`

To plot results in MATLAB, use `plotclawN` for $N = 1, 2, 3$. This invokes the m-file in `claw/matlab/plotclawN.m`

Hitting `<return>` at the `plotclaw` prompt causes the next frame to of data to be read into MATLAB and plotted. Alternatively, one of several characters can be typed at the `plotclaw` 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 `plotclaw` execution.
- **r** Redraw current frame. You might want to do this after resetting some plot parameters using keyboard input, 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 `plotclaw`.

Various parameters used in `plotclawN` are initialized in the corresponding file `setplotN.m`. If no such file exists in the current working directory, then the default file `claw/matlab/setplotN.m` is used. The parameters are listed and documented in this file. Values can be changed by creating a modified version in your own directory, or dynamically during execution of `plotclaw` by typing `k` at the `plotclaw` prompt and then entering a new value.

See Section 3.1.3 for some additional information on using `plotclaw2` with AMRCLAW.

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`$= 2, 3$.

The boundary condition routine that sets ghost cell values is slightly more complicated for AMRCLAW, as described in Section 3.1.2, 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 3.1.1).

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.

3.1 Two dimensions

The basic two-dimensional routines can be found in `claw/amrclaw/2d/lib`. You must first do a `make` in this directory in order to create `.o` files for each library routine. For an example of the use of AMRCLAW see the directory `claw/clawpack/2d/example1/amr`. This solves the same problem as in `claw/clawpack/2d/example1` but with adaptive refinement. To use this code you will need to do a `make` first in `claw/amrclaw/2d/lib` and in `claw/clawpack/2d/example1` and then in this directory. This produces an executable `xamr`. Results can again be viewed in MATLAB using the `plotclaw2` script (see Section 3.1.3). Note that there are no user-supplied fortran routines in these directories; the `Makefiles` refer to the routines in the directory above.

3.1.1 The input file `amr2ez.data`

The input data file is different and is now called `amr2ez.data`. This has essentially the same form as the `claw2ez.data` file used by `claw2ez`, but for the adaptive routines some additional parameters must be specified.

A sample file can be found in `claw/clawpack/2d/example1/amr/amr2ez.data`. The parameters `mxnest` and `inrat` and all those from `iousr` onwards are required only in the `amr2ez.data` file. Deleting these lines would produce the corresponding `claw2ez.data` file.

The new parameters needed by AMRCLAW are as follows:

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

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

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

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

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

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

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

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

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

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

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

`tol`: Tolerance used in flagging grid cells which need to be refined. An estimate of the truncation error is computed and compared with this value, see Section 3.4. Smaller values will lead to more refinement.

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

`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` 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.

3.1.2 Boundary conditions

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

3.1.3 Plotting results with MATLAB

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

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

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

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

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

3.2 Three dimensions

Coming in CLAWPACK 4.1....

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 `bc2amr` routine is called to set ghost cells where needed.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

3.5 Comments and warnings

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

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

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

- **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 `b4step2` and the source term routine `src2` are called in this process as in any other time step. If the `b4step2` routine is used to adjust values of the solution or do other operations which should only be done once at each distinct time, this could be a problem.

  In the error estimation time steps, a fixed time step is used 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.

Chapter 4

MPI Versions for multiple processors

For users who have access to a parallel computer or cluster of workstations running MPI (Message Passing Interface), versions of the two- and three-dimensional CLAWPACK routines are available that use MPI to distribute the computational work between 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.

See the documentation in the routines `claw/clawpack/Nd/lib/mpiclawN.f90` for `N`$= 2, 3$. Note that the MPI routines are written in Fortran 90 and require an appropriate compiler.

Examples of the use of these routines can be found in

`claw/clawpack/Nd/example1/mpi`

Note that all the same user-supplied subroutines are used as needed for the standard CLAWPACK routines. The only changes are a new `Makefile`, which links in the appropriate MPI versions of some library routines, and a slightly-modified `clawNez.data` file. This file simply has `N` additional lines added to the end that indicate how the domain should be partitioned in each of the `N` dimensions.

Some of the applications in `claw/applications` also have an `mpi` subdirectory. Generating the appropriate files for any other application should be quite easy.

MPI is easy to apply because the explicit methods used for hyperbolic systems on a rectangular grid are "embarrassingly parallel". Most of the work takes place in updating the solution on each subdomain, which can be done based only on the local information on this grid (and its ghost cells), without reference to the solution on any other subdomain. Communication between subdomains 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 at the end 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 by a factor of nearly M when M processors are used.

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` can not 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, who also wrote the MPI versions of CLAWPACK. See

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

(This is still under development but a test version is available.)

Bibliography

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

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

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

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

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

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

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

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

Index

acoustics, 18
adaptive mesh refinement (AMR), 2, 7, 27
amdq, 13, 21
amr2ez.data, 27
amrclaw, 27
apdq, 13, 21
asdq, 19
aux, 12, 17
auxiliary arrays, 17
    with AMR, 32
auxtype, 28

b4step1.f, 17
b4step2.f, 32
bc1.f, 9, 12, 16
bc2amr.f, 27, 29
bearclaw, 33
bmasdq, 19
boundary conditions, 9, 12
    with adaptive refinement, 29
bpasdq, 19

capacity functions, 9, 15
cflv, 14, 32
citing CLAWPACK, 7
claw1.f, 12
claw1ez.data, 12, 14
claw1ez.f, 8, 12
claw1program.f, 10
claw2ez.data, 18
claw2ez.f, 18
claw3ez.data, 20
claw3ez.f, 20
compile, 10
copyright, 3
cutoff, 29

download, 9
driver.f, 11, 12
dtv, 14

environment variables, 10, 11
errf1.f, 31
error estimation, 31
errsp.f, 31
example1, 11

fluctuations, 8, 9, 13
flux-difference splitting, 8
fort.chk, 28

Godunov's method, 8, 13
graphics, 11, 23

ibuff, 29
icoor, 21
imp, 19, 21
initial conditions, 12
inrat, 28, 30
iousr, 28
ixy, 19
ixyz, 21

kcheck, 29, 31
keyboard input from plotclawN.m, 24

limiter.f, 16
limiters, 16

make program, 10
makefiles, 10
matlab, 9, 11, 23, 29
maux, 11
maxmx, 11
mbc, 11, 16
mcapa, 15
meqn, 11, 16
method, 15, 18, 20
MPI versions, 33
mpiclawN.f90, 33
mthbc, 19, 20
mthlim, 16
mwaves, 11, 16
mwork, 11
mx, 14
mxnest, 28
my, 18

NCAR, 29

`nout`, 14
`nstepout`, 14
`nsteps`, 14
`nstop`, 28
`nv`, 15

`outN.f`, 23
`outstyle`, 14

`philim.f`, 16
`plotclaw2`, 29
`plotclawN`, 24
`plotclawN.m`, 11
`PlotData`, 30
`PlotGrid`, 30
`PlotGridEdge`, 30
`PRINT`, 29

`q`, 12
`qinit.f`, 12
`ql`, 13
`qr`, 13

`restart`, 28
Riemann solvers, 8
    in one dimension, 12
    in three dimensions, 21
    in two dimensions, 19
    transverse, 19
`rp1.f`, 8, 12
`rpn2.f`, 19
`rpn3.f`, 21
`rpt2.f`, 19
`rpt3.f`, 21

`s`, 13
`setaux.f`, 17, 32
`setenv`, 10, 11
`setplotN.m`, 25, 29
`setprob.data`, 11
`setprob.f`, 11, 16
source terms, 9, 17
speeds, 8, 9
`src1.f`, 9, 17

`t0`, 16
tar files, 10
`tol`, 28
`tolsp`, 29
`tout`, 28

versions, 3, 8

`wave`, 13
waves, 8, 9
website, 9
`work`, 12

`xlower`, 16
`xupper`, 16

`ylower`, 19
`yupper`, 19

`zlower`, 20
`zupper`, 20