notes prepared for the VKI 29th CFD lecture series
23-27 February, 1998
lecture 1 of 3

Amrita – A Computational Facility (for CFD Modelling)

James J. Quirk
Graduate Aeronautical Laboratories
California Institute of Technology
Pasadena, CA 91125, USA.
jjq@galcit.caltech.edu

Abstract

Amrita is a software system for automating numerical investigations. The system is driven using its own powerful scripting language, *Amrita*, which facilitates both the composition and archiving of complete numerical investigations, as distinct from isolated computations. Once archived, an *Amrita* investigation can later be reproduced by any interested party, and not just the original investigator, for no cost other than the raw CPU time needed to parse the archived script. In fact, this entire lecture can be reconstructed in such a fashion. To do this, the script: constructs a number of shock-capturing schemes; runs a series of test problems; generates the plots shown; outputs the LaTeX to typeset the notes; performs a myriad of behind-the-scenes tasks to glue everything together. Thus *Amrita* has all the characteristics of an operating system and should not be mistaken for a common-or-garden code. In this first lecture I will attempt to describe *Amrita* from the ground up which, if successful, will be no mean feat given the scope of the system. Particularly, since much of the material strays quite far from traditional computational fluid dynamics into areas of heavy-duty programming. Hopefully, my second and third lectures will convince reluctant programmers that the excursion is worth the effort.

1 Introduction

Amrita was originally developed as a means of driving an Adaptive Mesh Refinement (AMR) algorithm so as to provide an interactive teaching aid which would allow students to explore the practical aspects of compressible, computational fluid dynamics (CFD). Hence the name – Adaptive Mesh Refinement Interactive Teaching Aid. Over time, however, *Amrita*'s mandate has become far broader and so its name is now best taken at face value[1].

In the context of this lecture series, *Amrita* can be viewed as a software system for automating CFD investigations; right down to the level of constructing documents which explain both the purpose and the outcome of a particular exercise. Automation is seem as the key to improving numerical reliability, repeatability and productivity to the point where algorithms could be improved through massed scrutiny. To reach this ideal, *Amrita* strives to provide a computational framework which is sufficiently attractive to both novice and expert alike that it might help erode the present cottage-industry mentality, and its concomitant vagaries, where

[1]By coincidence amrita (*am-rē'tä*) also happens to be the drink of the Hindu gods!

DISCLAIMER

CFD codes are crafted on a one-off basis. Specifically, the many latitudes introduced by mundane activities such as preparing input files and post-processing results, together with the sheer drudgery of orchestrating investigations by hand, ensure that there is no convenient, watertight basis for the exchange of practical information to feed back and improve the underlying CFD algorithms. As a result, important observations can be slow to percolate through the research community.

In an earlier work[18], I highlighted several pathological failings of Riemann solvers which took several years to become common knowledge, despite the popularity of the schemes concerned. Hence, personally speaking, I am reluctant to embrace new "improved" algorithms on the strength of results from one or two selected test problems, as typically appear in a journal article, for fear that the "improved" schemes contain their own as yet unidentified pathologies ("better the devil you know, than the devil you don't!"). Ideally, all new schemes would be subjected to a complete battery of approved, acceptance tests before any claims are made on their behalf, thereby streamlining the process of determining when it makes sense to employ a particular numerical scheme. But many difficulties, both practical and ideological, would have to be overcome before this could happen. One aim of this lecture is to demonstrate that *Amrita* removes a sufficient number of the practical difficulties, in a sufficiently impartial manner that you might be persuaded to help out with *Amrita*'s further development for the computational benefits that a properly, supported system would bring to the CFD community[2].

1.1 What is *Amrita* – Animal, Vegetable or Mineral?

Amrita is a system which not only spans several disciplines, but one which is designed to operate at several levels of sophistication. Consequently, it is impossible to describe the system's construction in an order which will keep all parties happy. First-off, *Amrita* is driven via its own scripting language, *Amrita*[3]. At one end of the spectrum, too early an introduction to *Amrita*'s power leaves reluctant programmers with the impression that the language is too sophisticated for their needs. At the other end, too late an introduction to its articulateness runs the risk that hardened programmers will dismiss *Amrita* as yet another scripting language. Similarly, in some quarters, too prominent a description of *Amrita*'s educational value, leaves the impression that the system is a mere tinker-toy. On the other hand, to underplay this role, in favour of emphasizing *Amr_sol*'s[4] mesh refinement capabilities, leaves *Amrita* open to accusations of algorithmic bias, which undermine its role as a neutral, numerical test-bed[5].

When all is said and done, *Amrita* stands or falls on its utility as a labour saving device. For this reason, no claims are made regarding its algorithmic originality or efficiency; nor should you feel *Amrita* is trying to undermine your intellectual creativity. *Amrita* is an open-ended system for composing numerical investigations; in much the same way that LaTeX is a open-ended system for composing documents. In view of this, it makes no sense to ask "What is *Amrita*?" or "What can *Amrita* do?" Instead, you should ask: "What do I need to provide, to enable *Amrita* to do *such and such*?" To make this distinction clear, and also show that *Amrita* is not vapour-ware, these notes rely heavily on worked examples which you can run for yourself; Appendix A explains how to get started.

[2]By design, all parts of *Amrita* are considered ripe for improvement, to the point where the ultimate development of the system rests with the generosity of its users.

[3]The typographic difference between *Amrita* and *Amrita* is defined on p. 41.

[4]*Amr_sol* is the subject of lecture 2.

[5]Do not be fooled into thinking that *Amrita*'s horizons start and end with adaptive mesh refinement, just because of its name.

To gear up for this *Amritan* odyssey, the next section provides a gentle introduction to *Amrita* programming using the example of a shock diffracting around a corner. Do not be put off if you find the early pace slow, because the pace will soon quicken. However, to avoid losing the way too often, many of the details as to how *Amrita* goes about its full business are consigned to Appendices. For example, Appendix B reveals how *Amrita* conjures up a CFD code for you to run the shock-diffraction simulation. Given this information, plus an appropriate amount of effort on your part, there is no intrinsic reason why you could not eventually re-run the self-same simulation in conjunction with your own hand-crafted solver.

Here is the complete road map for the lecture:

Road Map

2 An *Amrita* Primer

Amrita is an interpreted language and so does not require separate compiling, linking and loading phases. Given a ***scriptfile***, the interpreter is invoked directly by typing:

unix-prompt> ***amrita [options] scriptfile***

where ***[options]*** is a list of switches which fine tune the behaviour of the interpreter (the option ***-h*** lists the other available options).

2.1 *my.script*

This first *Amrita* script produces the results shown in Figure 1:

```
TasteOfAmrita
plugin amr_sol
CornerProblem Ms=2, Xs=10
logfile logs/my.script
solver code/roe_fl
do phase=1,5
   def RefinementCriteria
      DensityGradient
      if($phase>1) ContactSurface
   end def
   march 30 steps with cfl=0.8
   flowout io/Corner$phase
end do
autoscale
postscript on
plotfile ps/schlieren.ps
   ShadeCorner
   SchlierenImage
plotfile ps/grid.ps
   ShadeCorner
   plot grids
```

```
amrcp  Chp2/my.script
amrita -html my.script &
amrps  ps/grid.ps
amrps  ps/schlieren.ps
```

Provided you have followed the instructions in Appendix A, you can type:

unix-prompt> ***amrcp Chp2/my.script***
unix-prompt> ***amrita -html my.script &***

to generate a directory called *ps* which contains the two *PostScript* files needed to produce the hardcopy shown (*my.script* also produces directories: *io*, *code*, *logs* and *html_files*, but more on these later). Try running *my.script* now, and check the resultant *PostScript* output using a previewer such as *GhostScript*. The script takes 90 seconds to run on an SGI Indigo2 machine (195 Mhz Mips R10000 processor) with 384 Mbytes of memory, but you may well have to wait longer for the results, depending on the power of your machine, relative to mine. If *my.script* fails to work[6], consult with your local *UNIX* expert to correct your shell-setup before proceeding, because the rest of this lecture assumes you are able to run worked examples such as the above.

[6]Unfortunately the vagaries of *UNIX* preclude the possibility of *my.script* working first time, for every user, on every platform. Some common teething problems, listed in order of increasing difficulty to fix, are: the environment variable PATH is set incorrectly; the file *.cshrc* (or equivalent) contains an error which causes it to terminate prematurely; a user has the wrong file access rights; *Amrita* has not been installed properly; *Perl* has not been installed properly, or is buggy on a particular computing platform (if in doubt, use *Perl4* in preference to *Perl5*).

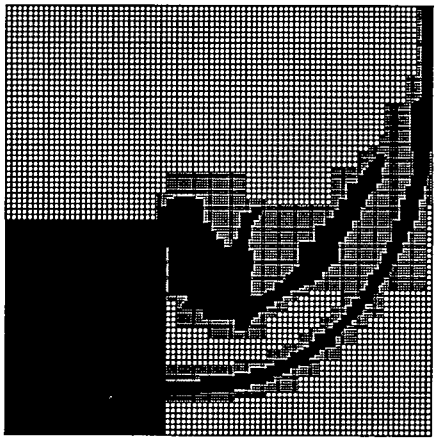
(a)  (b) 

Figure 1: Output from *my.script* depicting shock-diffraction around a 90° corner: frames (a) and (b) are produced by the files *ps/grid.ps* and *ps/schlieren.ps*, respectively. If desired, the solutions check-pointed using `flowout` can be retrieved for post-processing using `flowin`.

2.2 Autoload Procedures

Despite being only 21 lines long, this first *Amrita* script orchestrates a complete simulation and is representative of the effort reluctant programmers need expend to use *Amrita* profitably. *Amrita* scripts tend to be short-and-sweet, because a line such as `TasteOfAmrita` is not strictly a single command but a call to a procedure which contains the commands to be executed. When a procedure definition is missing from an executing script, *Amrita* automatically attempts to load it following some specified search path. By convention, autoload procedures have the first letter of every word capitalized to distinguish them from built-in keywords (i.e. commands) which are necessarily written in lower case. Although technically procedures, autoload routines are like commands in that they often come pre-supplied and do not appear in a user's script. The general idea is to have one individual craft a procedure which the wider *Amrita* community can then benefit from. Here, for example, the basic flow problem is set up by the procedure `CornerProblem`, leaving individuals to decide the choice of solver and the time period over which the problem is marched using the commands `solver` and `march`.

Amrita's procedure loading mechanism is a convenient means of building customized libraries, the only rule is that there is just one procedure per file and that the filename matches the procedure name with the extension *.amr*. Thus, the procedure `SchlierenImage`, which rendered the schlieren image shown in Figure 1, sits in a file called *SchlierenImage.amr*. There is nothing special about this procedure and given a little knowledge of *Amrita* you could well have written it yourself; it is supplied merely as a convenience. As another convenience, *Amrita* provides a command `showproc` which can be used to obtain both the location and source listing for an autoload procedure.

Here, you can type:

```
unix-prompt>amrita -c
```

to enter *Amrita*'s command mode, followed by:

```
amrita>showproc SchlierenImage
```

to obtain the listing:

```
autoload procedure: $AMRITA/stdlib/flowviz/SchlierenImage.amr
source listing<<
#
# Canned procedure to draw a schlieren image
#
proc SchlierenImage {
     exposure [0:1]     = 0.8  # darkness of image
     amplification [0:?]=  15  # magnification of weak features
     grid               = {G}  # select grid
   }
   DrhoDx       ::= (RHO[+i]-RHO[-i])/(X[+i]-X[-i])
   DrhoDy       ::= (RHO[+j]-RHO[-j])/(Y[+j]-Y[-j])
   schlieren    ::= sqrt(DrhoDx[]**2+DrhoDy[]**2)
   minmax schlieren[] -> min, max
   wt           ::= (schlieren[]-$min)/($max-$min)
   greyshading ::= $exposure*exp(-$amplification*wt[])
   plot image $grid m<greyshading[]>
end proc
>>end listing
```

You can then type:

*amrita>**quit***

to exit *Amrita*, or:

*amrita>**Show proc=SchlierenImage***

to produce an *HTML* listing of the procedure[7].

The above routine nicely demonstrates several features of *Amrita* procedures. Specifically, a procedure can be endowed with one or more parameters, each of which can be given valid range bounds and sensible default values, should an explicit value not be provided when the procedure is invoked. Thus `exposure` is a parameter restricted to lie in the range 0 to 1 and defaults to 0.8, and `amplification` is restricted to being greater than zero, but has no upper bound, and defaults to 15. When a procedure is invoked, parameters are supplied by name and so their ordering is unimportant. Therefore, all the following calls are acceptable:

```
SchlierenImage grid={G1-G2}
SchlierenImage exposure=0.5
SchlierenImage exposure=0.6, amplification=10
SchlierenImage amplification=5, exposure=0.9
SchlierenImage grid={G1-G2}, amplification=5, exposure=0.9
```

2.3 String Tokens

Amrita has no variables in the sense of a language like *Fortran* or *C*; a parameter such as `exposure` is nothing more than a token which identifies a string which is accessed by prefixing the token with a dollar symbol[8] e.g. `$exposure`. String tokens can be given explicit values using the command `set`, and can be viewed as containing anything from a number to a command, depending upon the situation in hand. Consequently, although contrived, the following script is valid:

[7] You can also type: ***Show keywords=**** to obtain a complete list of the commands available. The * is treated as a wildcard, thus you could use ***s**** to find all the keywords which begin with the letter ***s***.

[8] Strictly speaking, $ is an operator which expands a token; for details, see Chapter 3 of *An introduction to Amrita*[21].

```
...   preparatory script
set procedure   = LatexLabel
set number      = 2000
set string      = an Amritan oddity!
set number     #= $number+sin($number)**2+cos($number)**2
set parameters "= label=$number $string
$procedure $parameters
```

```
amrcp  Chp2/oddity.1
amrita run_space_oddity
amrps  ps/2001.ps
```

The = operator (sometimes written "=) directly assigns the string on its right to the token on its left, after stripping away leading and trailing white space, as distinct from #= which evaluates the string then assigns the result to the token. Thus here `$number` is incremented to `2001`, and since *Amrita* attempts to expand all string tokens before executing a line of script, the last line becomes:

```
LatexLabel label=2001 an Amritan oddity!
```

This version of the script:

```
...   preparatory script
set procedure   = LatexLabel
set number      = 2000
set string      = an Amritan oddity!
set number     #= $number+sin($number)**2+cos($number)**2
set parameters "= label=$number $string
$procedure $ parameters
```

```
amrcp  Chp2/oddity.2
amrita run_space_oddity
```

generates the error:

```
Error at line 14 of file run_space_oddity:
expected string token!

Line 14 is:
$procedure $ parameters

error near:
 parameters
```

because *Amrita* does not allow a space between the $ (i.e. the string expansion operator) and the token upon which it is supposed to act.

Programmers brought up on strongly-typed languages might balk at *Amrita*'s lax approach to things, but their fears are groundless. Generally speaking, *Amrita* scripts do nothing more than farm out requests to a `plugin` engine (here *Amr_sol*) to accomplish their tasks; they do not involve low-level code such as looping over the elements of an array. Consequently, because the context of a request is always clear-cut, the interpreter has no difficulty pinpointing errors (just as it did above). In the case of the procedure `SchlierenImage`, only two commands are required to do the work. The `minmax` command expects to be given an expression template, which if valid, is passed to the resident engine. The engine then grinds away to find the relevant minimum and maximum values, which it spits back as two strings which are then assigned to the tokens `min` and `max`. Hence the notation, `->`, which is suggestive of the logical flow of information from the engine to the script. Similarly, the `image` variant of `plot` expects to be given a shading template to render.

2.4 Template Expressions

Template expressions, such as `schlieren[]` and `greyshading[]`, essentially define functions which some command – further down the track – evaluates over the computational grid, as it sees fit. Thus:

```
DrhoDx        ::= (RHO[+i]-RHO[-i])/(X[+i]-X[-i])
```

defines a template which approximates the density derivative $\partial\rho/\partial x|_j$ using central differences[9]. The symbol `::=` is used, rather than a straightforward =, to emphasize the fact that an expression is being defined symbolically and that no assignment takes place. *Amr_sol* pre-defines `X[]` to return the x coordinate of the centre-of-gravity of the mesh cell (i, j). Similarly, `RHO[]` is a system function which returns the density within the cell (i, j); `RHO[]` is defined in the procedure `EulerEquations`[10], which is called from within `TasteOfAmrita`. *Amr_sol* allows expression templates to take offsets. For example, `RHO[+i]` would return the density in the cell $(i+1, j)$, and `RHO[+i-j]` would return the density for the cell $(i+1, j-1)$[11]. The precise syntax of an expression template is controlled by the `plugin` engine. Therefore, the fact that *Amr_sol* understands `+i-j`, has no bearing on what another engine might allow.

Once defined, a template may be used to help define further templates and so complicated expressions can be conveniently broken down into smaller sub-expressions which are more easily digested. For example, in `SchlierenImage` the function `wt[]` will clearly only return values between 0 and 1. Therefore `greyshading[]` will only return values between 0 and `$exposure`, and knowing that the `image` variant of `plot` shades the value 0 as black and the value 1 as white, the choice of the token name `exposure` becomes obvious: it controls the overall darkness or exposure of the rendered image.

Amrita uses expression templates in a host of commands which perform tasks ranging from prescribing initial flow conditions, through selecting refinement criteria, to extracting data along paths in space. This approach provides an extremely flexible, yet simple means of controlling the underlying computational machinery. Many of these commands are deemed to be specialist in the sense that they can only be used within a `def $mode` block such as the `def RefinementCriteria` block seen in *my.script*. Some other common `def` blocks are: `EquationSet`, `Domain`, `SolutionField` and `BoundaryConditions`.

There are no restrictions regarding the script complexity within `def` blocks, that is you are free to use logical constructs and invoke procedures (or even define new ones) and the like. They exist solely to allow *Amrita* to maintain some semblance of control over the order in which a simulation is set up. For example, it makes no sense to prescribe boundary conditions before a flow domain has been specified. Nor does it make sense to sprinkle refinement criteria at arbitrary places in a script, since the mechanics of grid adaption requires that all the desired criteria be specified up front. Therefore `def $mode` commands can be viewed as interlocks which turn certain commands on, and switch others off. Thus it is not possible to integrate a flow solution from within a `def RefinementCriteria` block, using the `march` command, since this could result in the grid being adapted using only a subset of the intended refinement criteria, leading to unexpected results. However, since `def` blocks can be repeated[12], it is possible to change refinement criteria midway through a simulation when desired. It just has to be done explicitly as is done in the program *my.script*; the reasons for doing so here are given at the end of the next section.

[9] This will only equal $\partial\rho/\partial x|_y$ on a uniform mesh.

[10] The construction of an `EquationSet` is described in lecture 2.

[11] The maximum allowable offset depends on the number of ghost-cells used by the engine, and will be discussed in lecture 2. But should you make a mistake, *Amrita* issues an informative error message.

[12] There are some restrictions concerning how `def` blocks are nested.

Although *Amrita* has the usual program flow-control constructs such as `do`, `while` and `foreach` looping commands, and logical constructs such as `if` and `switch`, these are not what sets it apart from other programming languages and so have been left out of this primer. *Amrita*'s usefulness stems from its ability to mix numerical tasks with document preparation tasks, and it is not intended to be a replacement for the likes of *C++*. Therefore, although the wisdom of endowing *Amrita* with its own programming language might not be immediately apparent, especially to reluctant programmers, *Amrita* does fill a niche which is not well catered for by other languages. Here, in case you have not already twigged, the `-html` option used in the running of *my.script* causes the interpreter to generate an *HTML* listing of *my.script* and any autoload procedure it activates. This provides users with a very convenient means of examining *Amrita*'s standard library in action, and so there is no excuse for not being able to follow the inner workings of the example scripts presented in these notes.

If you have not already done so, try typing:

unix-prompt>**`netscape html_files/my.script.html`**

and you will see that `CornerProblem` invokes:

```
proc ShockWave Ms=1.0, state1=quiescent, state2=post_shock
   gm ::= GAMMA'$state1
   gg ::= (gm[]+1)/(gm[]-1)
   c1 ::= sqrt(gm[]*P'$state1/RHO'$state1)
   p2 ::= (2*gm[]*$Ms*$Ms-(gm[]-1))/(gm[]+1)
   r2 ::= (gg[]*p2[]+1)/(gg[]+p2[])
   u2 ::= $Ms*(1-((gm[]-1)*$Ms*$Ms+2)/((gm[]+1)*$Ms*$Ms))*c1[]
   W'$state2 ::= W'$state1<RHO*=r2[],U+=u2[],P*=p2[]>
end proc
```

to compute the shock-jump relationships[13]:

$$\begin{aligned}
\frac{p_2}{p_1} &= \frac{2\gamma M_S^2 - (\gamma - 1)}{\gamma + 1} \\
\frac{\rho_2}{\rho_1} &= \left[\left(\frac{\gamma+1}{\gamma-1}\right)\frac{p_2}{p_1} + 1\right] \div \left[\left(\frac{\gamma+1}{\gamma-1}\right) + \frac{p_2}{p_1}\right] \\
\frac{u_2}{c_1} &= M_S\left[1 - \frac{(\gamma-1)M_S^2 + 2}{(\gamma+1)M_S^2}\right]
\end{aligned}$$

For the time being, the syntactic details of `ShockWave` are unimportant. Of more significance, is the fact that *Amrita* is equally at home executing such equation-based routines as it is executing the typesetting routines in the next section.

[13]When using expression templates, *Fortran* aficionados should note there is no computational advantage to be gained from replacing constant sub-expressions explicitly, because *Amrita* automatically reduces them down to a number as part of its internal optimization. For instance, do not introduce `gm1[] ::= gm[]-1` in the belief it will save the cost of the subtraction in a later expression. Here, `gg[]` was defined solely to improve the legibility of `ShockWave`. Once you have read §4, you can run this script (**`amrcp vki/sym.1`**):

```
plugin amr_sol
W'one ::= <RHO=1,U=0,V=0,P=1,GAMMA=1.4>
ShockWave state1=one,state2=two,Ms=2
exprA ::= P'two[]
W'one ::= <RHO=1,U=0,V=0,P=1,GAMMA=X[]>
ShockWave state1=one,state2=two,Ms=2
exprB ::= P'two[]
export exprA[],exprB[]
```

to gain a better appreciation of how expression templates work. Although *Amrita* is happy to use a variable `GAMMA`, the CFD `solver` may not be so forgiving.

3 Document Preparation

On a couple of occasions I have remarked that these notes can be reconstructed using *Amrita*. This section describes the rudiments of how this is done and then presents two realistic examples for you to cut your teeth on. The key new concept here is the notion of a program fold.

3.1 Program Folds

To facilitate the construction of top-down investigations, *Amrita* scripts can employ "program folds" along the lines used in the Occam Programming System (OPS)[5]. The new twist that *Amrita* adds is to allow for multiple fold-types. If *Amrita* can't process the fold itself, it farms the fold out elsewhere. Thus, if the application warrants it, the *Amrita* expert can go so far as to utilise several programming languages in the same script (a nice example is given in § 7). This next script, illustrates how you can use program folds to embed sections of LaTeXwithin *Amrita*, the end result being Figure 2.

```
PotentialFlowEquations
plugin amr_sol
...  generate figures
Latex2eHead
   ...  typeset title and background
   ...  typeset tabular of figures
LatexTail
Latex
```

```
amrcp  vki/potential.1
amrita potential_flow
amrps  cylinder/psi.ps
```

The script becomes more illuminating when it is unfolded once:

```
PotentialFlowEquations
plugin amr_sol
fold::amrita { generate figures
   ...  look in here for some more folds
}
Latex2eHead
   fold::print { typeset title and background
      fold>guard=|,dollar=*
      \pagestyle{empty}
      \vspace|*{-40pt}
      \centerline{\Large *text::title }
      *text::potential
      *text::ref
   }
   fold::print { typeset tabular of figures
      fold>guard=|,dollar=*
      \begin{center}
         \begin{tabular}{ccc}
            *fig::kappa0 & *hsep & *fig::kappa1 \\ \\[*vsep]
            *fig::kappa2 & *hsep & *fig::kappa3 \\ \\[*vsep]
            *fig::kappa4 & *hsep & *fig::kappa5
         \end{tabular}
      \end{center}
   }
LatexTail
Latex
```

Stream function ψ for flow around a cylinder with circulation

$$w(z) = \phi + i\psi = \underbrace{-Ve^{-i\alpha}z}_{\text{free stream}} \quad \underbrace{-Ve^{i\alpha}\frac{a^2}{z}}_{\text{cylinder}} \quad \underbrace{-\frac{i\kappa}{2\pi}\ln\frac{z}{a}}_{\text{circulation}}$$

For background details see §6.6[1] of:

BATCHELOR, G.K. 1967 *An Introduction to Fluid Dynamics*, Cambridge University Press.

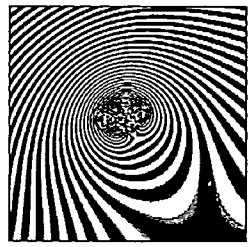
$\kappa = 4/3\pi aV \times 0$

$\kappa = 4/3\pi aV \times 1$

$\kappa = 4/3\pi aV \times 2$

$\kappa = 4/3\pi aV \times 3$

$\kappa = 4/3\pi aV \times 4$



$\kappa = 4/3\pi aV \times 5$

[1]Page 424 onwards, for the 1983 paperback edition.

Figure 2: Page produced by the script *potential_flow*. If you alter the title used by the script and then re-run it, you will notice that *potential_flow* does not waste the effort of regenerating the plots. One interesting experiment you can try is to run *potential_flow* using a coarse grid and observe the resultant *moiré* interference patterns between the stream function and the raster sampling of `plot image`; delete the old *PostScript* files first, otherwise the script will simply reuse them and ignore your investigation.

The `fold::print` construct works much as a "here-document" in *sh*, *csh* or *Perl*[28], in that it allows a text template, with unexpanded string tokens, to be embedded within a script. Lines which begin with `fold>`[14] contain directives which fine-tune the behaviour of the fold. Ordinarily, *Amrita* employs \ as a guard character, thus *Amrita* would attempt to expand `$a`, but not `\$a`. But within a `fold::print` block, the directive `guard` can be used to change the guard character to |, thereby eliminating the need to type \\ for every backslash intended for LaTeX. Similarly, the directive `dollar` can be used to change *Amrita*'s string expansion operator from $ to *, which comes in useful when typesetting mathematics[15]. The folding editor *amrgi* (short for *Amrita::Origami*) facilitates the construction and viewing of folded documents, but its use is not mandatory[16]. To see the utility of *amrgi*, type:

```
unix-prompt> amrgi potential_flow
```

Once you are in the editor you can type **`Alt-T`** to obtain a walk-through tutorial[17] which will explain how to browse the program folds within *potential_flow*[18].

If you have looked ahead to Figure 8, you will not be surprised to learn that this script can be used to typeset *potential_flow* in the manner shown earlier:

```
set script = potential_flow
set level  = amr::L1
fold::file $script (to $level) -> listing
LatexHead
   fold::print { latex script-listing
      fold>guard=|,dollar=*
      \begin{verbatim}
         *listing
      \end|{verbatim}
   }
LatexTail
Latex
```

```
amrcp  vki/fold.it
amrita fold_script
cd     latex_files
amrps  amrita.ps
```

The script obtained with **`amrcp vki/grab.it`** performs a similar trick to typeset the fold which defines the complex potentials:

```
fold::amrita'potentials { define complex potentials
   set a      = 15                    # radius of cylinder
   set V      = 1                     # velocity of free stream
   set alpha  = rad(30)               # angle    of free stream
   Z           ::= {X[]-$Xo,Y[]-$Yo}  # centre cylinder in domain
   free_stream ::= -$V*exp({0,-$alpha})*Z[]
   cylinder    ::= -$V*$a*$a*exp({0,$alpha})/Z[]
   circulation ::= -{0,1/(2*PI)}*ln(Z[]/$a)
   ...  typeset formulas
}
```

Now if you are wondering where all this is leading, the next section might convince you that there is at least some method to *Amrita*'s typesetting-madness.

[14] As a mnemonic, view `fold>` as a prompt for a directive which indicates how the fold should work.

[15] In *C* parlance, an *Amrita* string token is a pointer to a string whose value can be obtained using the indirection operator $; hence the use of the standby notation, * for $.

[16] When constructing a fold using a normal text editor, just make sure the closing brace } is in the same column as the `f` in `fold::print`.

[17] This is spawned as a separate window.

[18] Program folds are an advanced *Amrita* feature which reluctant programmers can take or leave as they see fit: *Amrita* is designed to allow an individual to find his or her own level of programming comfort.

3.2 A Montage of Flow Solvers

One reason why there is little consensus of opinion in the CFD community – concerning the relative merits of the various shock-capturing schemes in existence – rests with the sheer number of schemes to evaluate. Without some form of automation, no one person can hope to test more than a small subset of schemes, on a small subset of problems. Consequently it is not surprising that different workers, form different opinions, from different experiences. This script:

```
... redirect latex output
Latex2eHead pagesize=problem-sheet
   ... typeset title
   ... typeset figures
   ... typeset footnotes
LatexTail
Latex
```

```
amrcp  Chp2/montage.1
amrita run_montage
cd     doc/montage
amrps  solvers.ps
```

summarise some of my experiences. The script takes around 25 minutes to run[19], and generates three directory trees: *code*, *results* and *doc*, the last of which contains the *PostScript* for Figure 3.

If you browse the script with *amrgi*, you will notice that the shock-diffraction simulations are run by a sub-fold of the fold which typesets the figure, but only if the results have not already been generated by a previous invocation of the script (as was the case with *potential_flow*). The *Amrita* expert is not fettered by artificial notions of pre- and post-processing. Tasks, whatever their nature, can be dealt with in the order dictated by the investigation. Here, for example, `run_montage` goes so far as to call the library routine *BasicCodeGenerator* (see Appendix B) to craft the individual flow solvers needed for the investigation: *godunov_km*, *roe_fl*, *ausm_km*, *hlle_km* and *efm_km*. These solvers are not stand-alone CFD codes, but shared-objects which are sucked into the mesh refinement engine *Amr_sol* using dynamic-linking.

A brief introduction to the programming benefits of dynamic-linking is given in Appendix D, here the essential fact to grasp is that dynamic-linking allows a separation of the classical work of integrating a discretized set of partial-differential equations from the drudge work of crafting an investigation. For instance, this script:

```
... redirect latex output
Latex2eHead pagesize=problem-sheet
   ... typeset title
   ... lead in to body of article
   ... describe flowfield
   ... describe how images are drawn
   ... describe sensitivity study
LatexTail
Latex
```

```
amrcp  Chp2/schardin.1
amrita run_schardin
cd     doc/ausm_km
amrps  schardin.ps
```

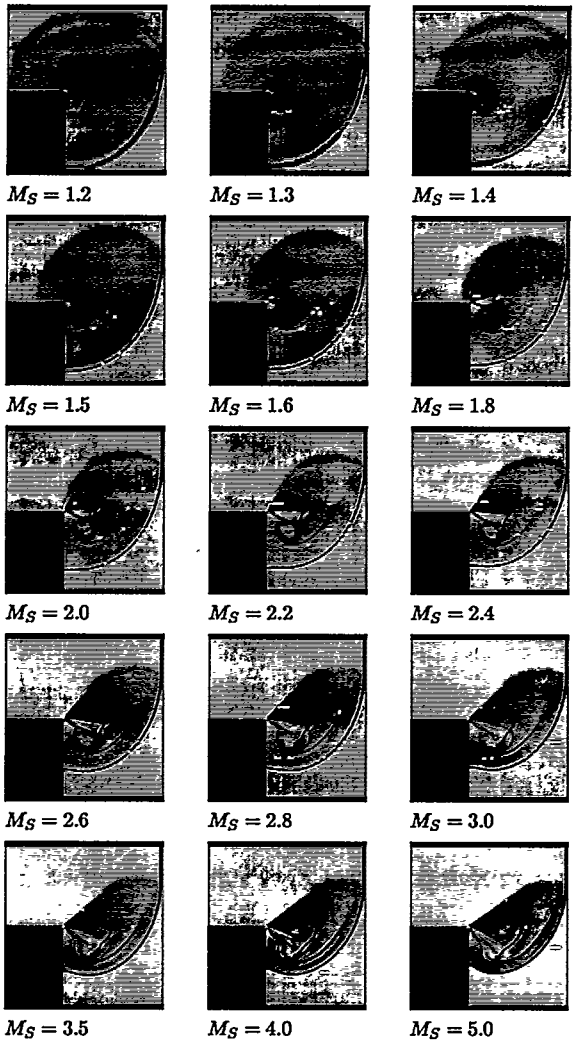
generates the two pages of *PostScript* shown by Figures 4 and 5. By default, the script employs the solver *ausm_km*, but the script-writer has made it possible for a different solver to be selected via a command line argument, say:

```
unix-prompt>amrita -a efm_km
```

But, given the wonders of dynamic-linking, any *Amr_sol* compatible solver could be chosen; even one that was not in existence when *run_schardin* was crafted.

[19]Recall, I am using an SGI Indigo2 machine (195 Mhz Mips R10000 processor) with 384 Mbytes of memory.

A Montage of Solvers[†]



`solver code/godunov_km`

- This solver is the most expensive of the five codes, but it works across the range of Mach numbers.
- Note the refinement criteria has lost track of the diffracted shock near the wall.

`solver code/roe_fl`

- This solver has the best resolution of the five codes.
- Unfortunately `roe_fl` cannot cope with Mach numbers much above 2[‡].

`solver code/ausm_km`

- This solver is arguably the cheapest of the five codes and has good resolution.
- Unfortunately `ausm` proves far from awesome for Mach numbers less than 1.5[‡].

`solver code/hlle_km`

- This solver is much more robust than `ausm` and almost as cheap.
- Unfortunately the improvement in robustness is offset by poor resolution of contact surfaces[§].

`solver code/efm_km`

- This solver is very similar in performance to `hlle`.
- Although, in some circumstances (e.g. slowly moving shocks) `efm` performs markedly better than `hlle`.

[†]CFD aficionados: please note the object of the present exercise is to look beyond the bedeviling debate of which numerical scheme is best (*Ans: it depends on the circumstances!*) so as to concentrate on the mechanics of crafting an automated investigation.

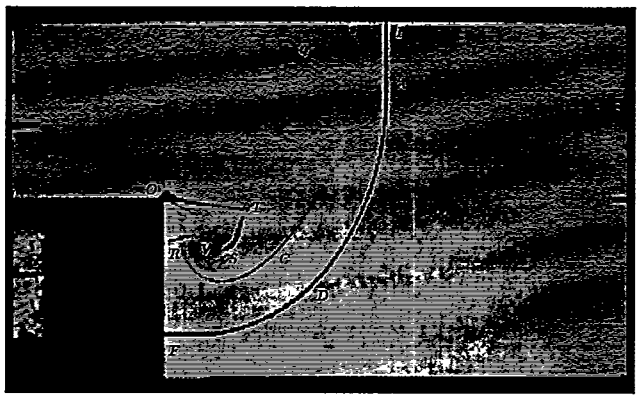
[‡]This observation, like the others on this page, is made in the context of the present shock-diffraction problem and may not apply in other situations ("All generalizations are dangerous, even this one." *fils* Alexandre Dumas).

[§]If you think the above images contradict this statement, you have forgotten to account for the increase in the strength of the contact surface with Mach number: the stronger a contact, the easier it is to resolve, the better it shows up in a schlieren.

Figure 3: Output produced by *run_montage*. The accompanying pithy remarks are not intended to diminish the presented schemes in any way: they simply reflect the author's view that compressible CFD can be an extremely frustrating business.

Comparison Against Experiment†

Amrita is sufficiently sceptical of its own capabilities, and numerical methods in general, that it goes to great lengths to allow CFD simulations to be vouchsafed down to the smallest detail. For example, here is an experimental shadowgraph† to show that the $M_S = 2.4$ simulation from "A Montage of Solvers" is at least qualitatively correct:



Briefly, the main features of the flow field are as follows. The diffraction of the incident shock wave (IA) around the corner gives rise to an expansion fan which emanates from the apex (O). The shape of this fan's lead characteristic (OQA) suggests the flow upstream of the shock is mildly supersonic. The interaction of the expansion fan with the incident shock gives rise to a disturbed shock front (ADF) which is curved. A contact surface (C) marks the boundary between fluid which has been induced into motion by the incident shock and fluid which has been processed by the disturbed shock front. The flow in the vicinity of the corner is detached and so a slip stream (OS) separates the expanded flow from a region of almost stationary gas, and the free end of this slip stream rolls up into a vortex (V). Two secondary shock waves (TS) and (OT) are needed to match the pressure of the flow accelerated by the expansion fan to that of the decelerated flow behind the diffracted shock front. Observe how the secondary shock (TS) is kinked as a result of its interaction with the slip stream (OS). A final shock wave (R) is needed to decelerate the reverse flow, within the separated region (OVR), down to zero velocity at the point of diffraction.

Now the above figure was produced using the *Amrita* procedure:

```
proc SchardinImage xo=-81.8,yo=-4.25,dx=83.7,drawkey
   pushmatrix
      rotate -90
      paste schardin.jpg in box $xo,$yo,$dx,?
   popmatrix
   if(token(drawkey)) SchardinKey
end proc
```

The file *schardin.jpg* contains a scanned image of the experimental shadowgraph, but the image is in portrait mode and lacks labels (try viewing it with *Netscape*). `SchardinImage` invokes *Amr_sol*'s graphics engine to draw this *jpg* file rotated by 90° to produce a landscape image. The `in box` part of the command ensures the shadowgraph is positioned and scaled to match the coordinate system used by `CornerProblem`. This

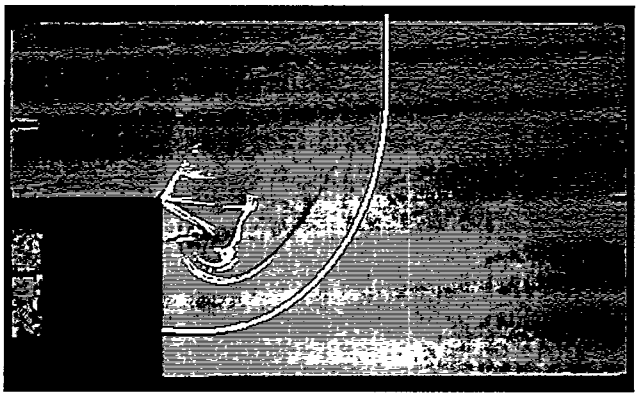
†SCHARDIN, H. 1965, *Proc. VII Int. Cong. High Speed Photog.* 1965, Damstadt, Germany. pp. 113-119. Verlag O. Helwich; picture courtesy of Dr. H. Reichenbach. Also, appears in: *An Album of Fluid Motion* (plate 243), assembled by Milton Van Dyke, Parabolic Press, 1982.

Figure 4: First page of output produced by *run_schardin*.

then allows `SchardinKey` to operate in the same way as `CornerSchematic`. For example, (O) is labelled with:

```
LatexLabel label=\$O\$,xo=26,yo=41,height=$ht
```

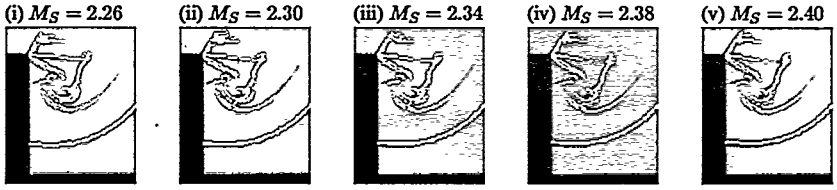
Similarly, it is possible to overlay the computed front positions on top of the experimental image[‡], However, for reasons given below, we choose to do this for $M_S = 2.34$ rather than the experimentally reported $M_S = 2.4$:



Comparing numerical results against experiment is not a cut and dried exercise and often involves a measure of judgement based on common sense and physical reasoning. Here, for example, a close examination of the experimental image suggests the corner is blunted. But this is an optical illusion caused by the refracton of light in the vicinity of the corner apex[§]. At another level, the inviscid flow model used for the simulation cannot be expected to mimic the full viscous behaviour of the experiment. While this can be raised as a criticism of the numerics, it can also be raised as a drawback of the experiment. Viscous drag acts to slow the incident shock down during the course of the experiment, consequently the effective incident Mach number is almost certainly different from the reported value.

Frames (i)–(v), below, show the results of a numerical sensitivity study aimed at determining the variation in the diffraction pattern to small changes in the incident Mach number:

(i) $M_S = 2.26$ (ii) $M_S = 2.30$ (iii) $M_S = 2.34$ (iv) $M_S = 2.38$ (v) $M_S = 2.40$



Observe how the secondary shock (TS) moves away from the apex (O) with increasing Mach number, while the foot (F) of the diffracted shock moves closer to the apex with increasing Mach number. Based on this study, arguably the closest agreement between numerics and experiment occurs for $M_S = 2.34$, which suggests an experimental uncertainty in shock speed of 2.6%.

[‡]Check out the procedure `OverlayNumericalFeatures` – `plot image` treats negative shades (e.g. `m<-1>`, `M<-1>`, `rgb<-1,0,0>`, `RGB<-1,0,0>` etc.) as transparent.

[§]If you are not swayed by this argument, examine plate 243 of *An Album of Fluid Motion*. It shows three snapshots of the diffraction process, each with a different amount of blunting. Moreover, the blunting decreases as the flow develops in time and so the variation cannot be blamed on some bizarre form of low-temperature ablation.

Figure 5: Second page of output produced by *run_schardin*.

3.3 Startup-Errors

Both *run_montage* and *run_schardin* weigh in at around 300 lines of *Amrita*, and so are not particularly onerous to construct. Nevertheless, they are sizeable enough to put an *Amrita* novice off, especially one struggling to get to grips with string expansions (see p. 52). Therefore, this next script drops down the programming scale to illustrate another problem which befuddles CFD:

```
... obtain shock-diffraction solution
... make interferogram schematic
LatexHead
... output raw latex for title
LatexNupFig iup=2,jup=3
foreach shift (0,0.2,0.4,0.6,0.8)
   set file = shift$shift.ps
   plotfile $amrita:latex::dir/ps/$file
etc ..
```

```
amrcp  Chp2/startup.1
amrita startup_errors
cd     startup
amrps  errors.ps
```

Even with the best will in the world, there is generally insufficient room for the CFD author to describe all the small innocuous details which might affect the reader's perception of the quality of the presented results[20]. For example, consider the shock-diffraction results shown in Figure 6. The raw solution-data is the same for each interferogram image[14]; all that changes is the reference density. But one's perception of the quality of the simulation varies dramatically[21] depending on the depicted strength of the tram-lines behind the shock. These tram-lines are "startup-errors" which arise because the shock-capturing scheme used for the simulation cannot propagate the perfect discontinuity used to prescribe the initial flow conditions. Therefore, instead of propagating information solely on the $(u+a)$ characteristic, small dips in the density field are introduced on the $(u-a)$ and u characteristics as the shock smears to the natural profile dictated by the numerics.

Startup-errors are self-similar with mesh spacing and so do not disappear under mesh refinement. As such, they are zeroth-order errors which cannot be eliminated[22] by increasing the order of the accuracy of the integration scheme; nor, for that matter, can they be eliminated by moving from a workstation to a massively-parallel computer. For inert flows, startup-errors are an annoyance which can be tolerated because they are small, localized glitches. For reacting flows, startup-errors can prove catastrophic, as the small decreases in density, correspond to small increases in temperature. These glitches can evolve under chemical amplification leading to "blow-up"[19] where the reaction proceeds many orders of magnitude faster than if the initial temperature perturbation were absent, giving rise to drastically different behaviour than expected. Thus a small local error, gives rise to a large global error.

Interestingly, adaptive mesh refinement diminishes startup-errors in a natural way, so long as the refinement criteria do not home in on the density glitches, as in [7] or Figure 9. It is for this reason that *my.script* does not employ `ContactSurface` for the first phase of the simulation. Startup-errors are low frequency errors, which explains why they are preserved so well by shock-capturing schemes that, universally, are designed to damp only high-frequency errors. In the case of *my.script*, the low-frequency startup-errors, which appear on the fine grid as the shock relaxes to a smeared profile, become higher-frequency errors as they drop off onto the coarse grid behind the shock. They are then damped in a natural fashion by the dissipation of the integration scheme.

[20] Let alone allow the reader to reproduce the results!

[21] At least for those with a critical disposition.

[22] I live in hope of being proved wrong.

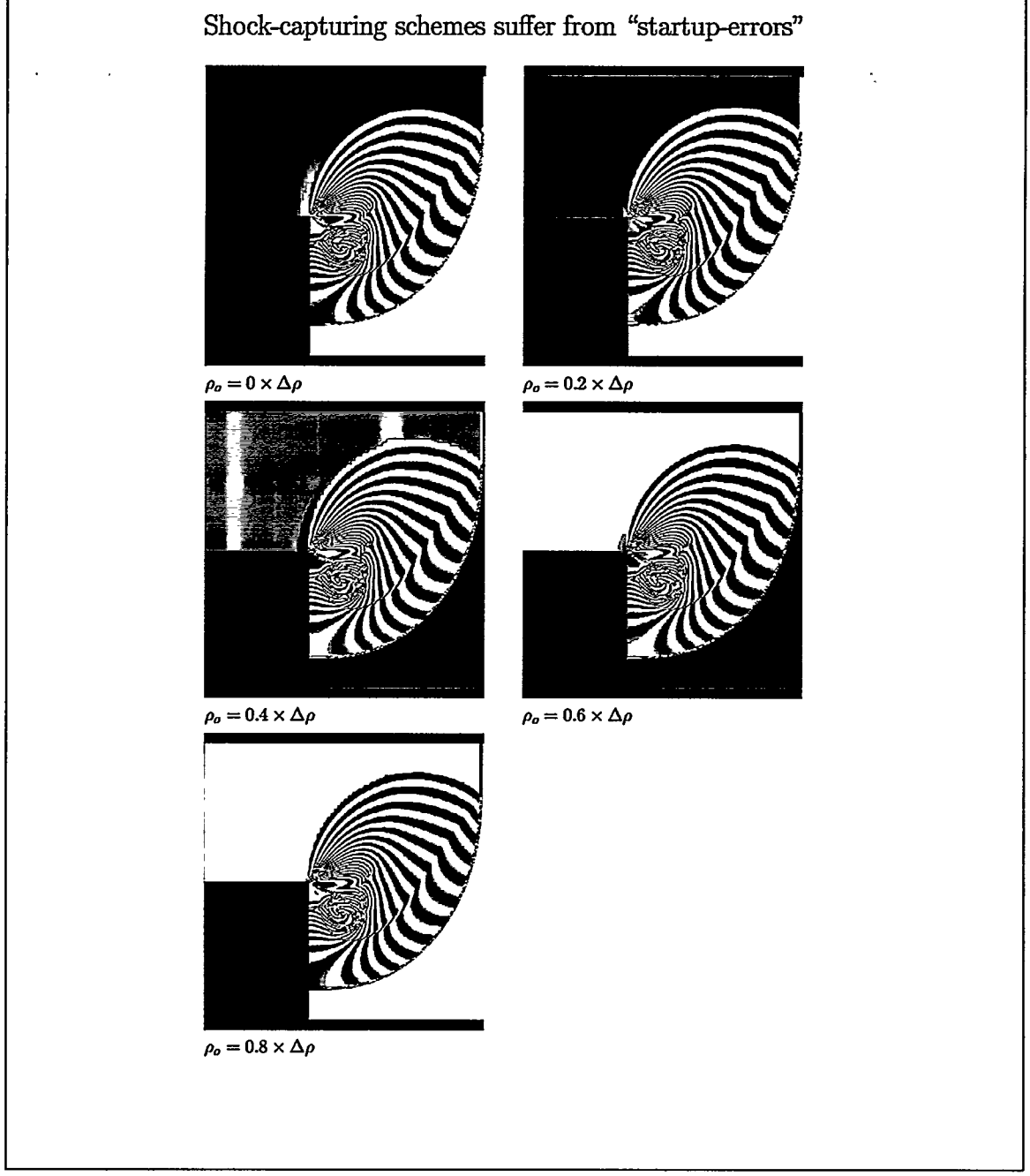


Figure 6: Page produced by the script *startup_errors*. The tram-lines do not appear for the $\rho_o = 0.6$ and $\rho_o = 0.8$ cases, because the erroneous density variation associated with the startup-error lies in the region $I > 1$ and `plot image` shades all values > 1 as white. The depicted strength of the tram-lines in the remaining cases depends on the slope of I for the nominal, density plateau behind the shock. If you re-run *my.script* with the refinement criteria: `setflags [ooo|oxo|ooo] 1` to force refinement everywhere (i.e. a uniform grid), the startup-errors will be more prominent than here.

4 System Overview

You now have sufficient knowledge of *Amrita*, put to everyday good use, for me to be able to present an overview of how the system works. The basic design tenets are: repeatability, accessibility and extensibility. For example, repeatability requires robustness. In this regard, a simple well-engineered idea, outperforms the sophisticated approach which trips up on its own cleverness. Therefore do not be afraid to code-up a "spade," in place of a "manually operated, earth moving implement," if it sufficient to do the job. Equally, have the common sense not to use a "spade," when the job calls for a "JCB excavator."

4.1 AmritaSystem

```
amrcp   vki/system.1
amrita  show_system
cd      AmritaSystem
amrps   AmritaSystem.ps
amrps   AmritaIsl.ps
```

This one line script[23]:

```
AmritaSystem
```

calls the *Amrita* library procedure:

```
...  programmer notes
proc AmritaSystem {
     plugin      = amr_sol
     workdir     = AmritaSystem
   }
   ...    define internal procedures
   #
   # Depict Amrita system using the above internal procedures
   #
   pushcwd $workdir
   plugin  $plugin
   RunUserScript
   postscript on
   SetPage size=USletter,orientation=landscape
   plotfile AmritaSystem.ps
      DrawAmritaLogo
      DrawUserScript
      DrawScriptOutput
      DrawIsl
      DrawPipeLine
      DrawLibraryScript
      EndDrawing
   SetPage size=USletter
   plotfile AmritaIsl.ps
      DrawExplodedIsl
      EndDrawing
   popcwd
end proc
```

to generate Figures 7 and 8.

[23] An in-joke amongst *BCPL* programmers was that the language's only use was to write *BCPL* compilers[22]. Similarly, a critic could say that *Amrita*'s only use is to document *Amrita*. A package such *Adobe Illustrator* could have been used to prepare the two figures, but then the tie-up between the schematic and the system would likely soon become out of date. Here, because the system goes to the trouble of documenting itself, changes made to the system (say a restructing of *ISL*) are automatically reflected in the schematic, the next time it is requested. The only thing worse than no documentation, is wrong documentation.

Figure 7: Schematic showing the basic organization of *Amrita*. High-level scripts are parsed by *Amrita* to produce *ISL* instructions which drive a `plugin` computational engine such as *Amr_sol*. The user script here is representative of the ones you will use in lecture 2 to gain first-hand experience of the strengths and weaknesses of heuristic refinement criteria.

ISL – Intermediate Scripting Language

```
isl::L0
... amr_sol:plot::flags
```

```
isl::L1
amr_sol:plot::flags {
   ... gridlist
   ... RefinementCriteria
}
```

```
isl::L2
amr_sol:plot::flags {
   gridlist {
      1 2 2 1
   }
   RefinementCriteria {
      ... setflags
      ... setflags
   }
}
```

```
isl::L3
amr_sol:plot::flags {
   gridlist {
      1 2 2 1
   }
   RefinementCriteria {
      setflags {
         ... gridlist
         ... mask
         ... expr
      }
      setflags {
         ... gridlist
         ... mask
         ... expr
      }
   }
}
```

```
isl::L4
amr_sol:plot::flags {
   gridlist {
      1 2 2 1
   }
   RefinementCriteria {
      setflags {
         gridlist {
            0 0 0 1
         }
         mask {
            0 0 0
            0 1 1
            0 0 0
         }
         expr {
            1
            v 601
            o 1 0
            v 601
            o 0 0
            n 13
            f 100
            v 601
            o 1 0
            v 601
            o 0 0
            b 12
            b 16
            v 580
            o 1 0
            v 580
            o 0 0
            n 13
            f 100
            v 580
            o 1 0
            v 580
            o 0 0
            b 12
            b 16
            b 21
            v 601
            o 1 0
            v 601
            o 0 0
            n 13
            f 100
            v 601
            o 1 0
            v 601
            o 0 0
            b 12
            b 16
            n 0.002
            b 21
            b 26
         }
      }
      setflags {
         gridlist {
            0 0 0 1
         }
         mask {
            0 1 0
            0 1 0
            0 0 0
         }
         expr {
            1
            v 601
            o 0 1
            v 601
            o 0 0
            n 13
```

Document Preparation – *Amrita* style

amr::L0

```
LatexHead
grab::isl amr_sol:plot::flags from uscript.isl -> isl
...  output title
foreach level (L0,L1,L2,L3,L4)
   fold::token isl (to isl::$level) lines {<75} -> listing
   ...  output listing
end foreach
LatexTail
Latex dvips=-E
```

amr::L1

```
LatexHead
grab::isl amr_sol:plot::flags from uscript.isl -> isl
fold::print { output title
   fold>guard=|
   \noindent\Large\amrprog{ISL} --
   Intermediate Scripting Language\\[20pt]
}
foreach level (L0,L1,L2,L3,L4)
   fold::token isl (to isl::$level) lines {<75} -> listing
   fold::print { output listing
      fold>guard=|
      \pagestyle{empty}
      \tiny
      \begin{minipage}[t]{2.8cm}
         isl::$level
         \begin{verbatim}
            fold>col=1
            $listing
         \end|{verbatim}
      \end{minipage}
   }
end foreach
LatexTail
Latex dvips=-E
```

Figure 8: The *ISL* instructions for the next-to-last line of the user-script in Figure 7 provide two pieces of information for *Amr_sol* to get the job done: a `gridlist` to fix which parts of the grid the engine should work on, and a shopping-list of `RefinementCriteria` by which to flag an individual cell. Here the user invoked `ContactSurface` to set-up a shopping-list of just two `setflags` tests; each of which needs a `gridlist` to restrict the extent of its operation and a `mask` of cells to flag if the symbolic `expr` is true. The *ISL* tree structure allows information in the leaf-nodes to be revamped, say an overhaul of `expr`, without the need to change the overall sequence of events.

4.2 *ISL* – Intermediate Scripting Language

The basic orchestration of *Amrita* follows the classical construction of a compiler[24]. In the case of a compiler, a front-end parses the user's source code to form an intermediate syntax-tree which a code generator then traverses to produce a target executable consisting of a sequence of machine code instructions[4]. In the case of *Amrita*, the *Amrita* front-end parses the user's script to form an *ISL* syntax-tree which is traversed by an *ISL* parser to schedule work for a set of user-supplied call-back routines which have been compiled to form a `plugin` engine[25]. Appendix E reveals how you can use the *Amrita* library routine `ClonePluginFoo` to generate the boiler-plate code needed to construct a new plugin. Here I will restrict myself to two concrete examples which demonstrate the design advantages of employing an *ISL* layer between the user written script and the engine which does the work.

Change to the directory where you ran *run_montage* and run this script[26]:

```
EulerEquations
plugin amr_sol
logfile logs/solvers
foreach scheme (godunov_km,roe_fl,\
                ausm_km,hlle_km,efm_km)
   solver code/$scheme
end foreach
```

```
amrcp  Chp2/solvers.1
amrita -debug solvers
amrgi  debug.isl
```

to get the *ISL* listing:

```
... fold::isl  copyright message
... amrita:plugin::amr_sol
... fold::isl  amr_sol defaults
... amrita::logfile
... amr_sol::solver
... amr_sol::solver
... amr_sol::solver
... amr_sol::solver
... amr_sol::solver
... amrita:unplug::amr_sol
```

Notice that the script keyword `solver` is really a contraction for `amr_sol::solver`, meaning that it is a keyword which belongs to (i.e. will be decoded by) *Amr_sol*. As explained in Appendix E, when an engine is plugged into *Amrita* it adds its own specialist keywords to the system, complete with the requisite parsing machinery to generate the specialist *ISL* for the

[24]This similarity is not coincidental as I originally taught myself to program by dissecting the source code for various compilers and interpreters: *BCPL*[22], *C*[13], *MOUSE*[11] and *PASCAL*[3, 16]. I recommend this route to anyone who wishes to improve their programming skills and rise above the debilitating language wars which rage on *USENET*: "*Fortran* bad, *C* good, *C++* better, *Java* best!" If you appreciate the basics of how programming languages are parsed, you will have no difficulty in mixing and matching languages to suit the end application, as is done in §7. Besides, external influences can dictate the choice of programming language, over and above personal preference. Back in 1988, the forerunner to *Amr_sol* was written in *Fortran* at the behest of the sponsoring government agency. Today, in 1998, large chunks of *Fortran* remain, but because of its design, *Amr_sol* is none the worse for its syntactically-humble start in life.

[25]In view of this, one could legitimately refer to *Amrita* as a compiler; one which produces *ISL* code. Thus circumventing the notion that an interpreted scripting language is inferior to a compiled programming language. Moreover, keen eyed readers will by now have noticed that *Amrita* is a pukka block-structured language, which allows procedures to be defined within procedures and so is more advanced than *C* in some respects.

[26]Here the \ acts as a line continuation marker.

new keywords, and adds the associated call-back routines for the parser to call when it stumbles across *ISL* it cannot decode by itself. Thus, in principle, *Amrita* is infinitely extendible.[27] The *ISL* parser decodes a sufficient number of system keywords (e.g. `amrita::logfile`) that a new plugin does not have to reinvent the wheel.

Try expanding the five `solver` folds, and you will see they do nothing more than locate shared-object files[28]:

```
amr_sol::solver {
    file ~CWD/code/~AMRSO/godunov_km.so
}
amr_sol::solver {
    file ~CWD/code/~AMRSO/roe_fl.so
}
amr_sol::solver {
    file ~CWD/code/~AMRSO/ausm_km.so
}
amr_sol::solver {
    file ~CWD/code/~AMRSO/hlle_km.so
}
amr_sol::solver {
    file ~CWD/code/~AMRSO/efm_km.so
}
```

which the *ISL* parser loads and links dynamically[29] at run time. Here, the only evidence that the solvers have indeed been activated is the log-file *logs/solvers*:

```
****************************************
*       SOLVER GENERATED BY BCG        *
****************************************
BCG ID: euler-code::2d-c-km-os
SOLVER: godunov_km
OWNER : James J. Quirk (aka jjq)
DATE  : Mon Feb  2 14:46:11 PST 1998
****************************************
*       SOLVER GENERATED BY BCG        *
****************************************

****************************************
*       SOLVER GENERATED BY BCG        *
****************************************
BCG ID: euler-code::2d-c-fl-os
SOLVER: roe_fl
OWNER : James J. Quirk (aka jjq)
DATE  : Mon Feb  2 14:46:21 PST 1998
****************************************
*       SOLVER GENERATED BY BCG        *
****************************************
etc ..
```

Prior to being passed the filename of a `solver`, *Amr_sol* has no knowledge of the code, thus it would view *BCG*'s efforts and your hand-crafted efforts with equal disdain[30]. Consequently, the system can accommodate codes: past, present and future; all on an equal footing. Note also that the *ISL* parser receives a logical filename and not a physical filename. Internally,

[27]But, as with any software, there are a number of practical reasons why this is not to be taken literally. Nevertheless, the statement stands in that *Amrita* can be extended seamlessly, well beyond its present boundaries.

[28]The *Amrita* parser has done all the hard work of checking that the files exits etc.

[29]Dynamic linking is discussed in Appendix D, but you might also like to type:
*unix-prompt>**man dlopen dlsysm dlclose**.*

[30]This does not imply your solver has no intellectual content, it simply means that *Amr_sol* views a `solver` as a lump of object-code it must link with in the *UNIX* sense of resolving external references in a link-load table. Once loaded, *Amr_sol* interrogates the `solver` to check that it is compatible with the resident equation set, thereby avoiding the anarchy which would ensue, say, if a `ShallowWaterEquations` solver were used on a problem set up for `BurgersEquation`. A `solver` is the one component of *Amrita* which can be equated to a classical CFD code. In lecture 2, I explain the constraints on the CFD codes *Amr_sol* can accept.

the *~CWD* part is expanded to be your current working directory, and the *~AMRSO* part is expanded to the architecture of the machine which runs the parser. For example, on my machine: *~AMRSO* expands to *AMRSO/serial/IRIX/64*, but if I were running on a cluster of workstations, then *mpi* would be substituted for *serial*.[31] The principal advantage though of having the *ISL* parser expand *~AMRSO*, is not to distinguish between *serial* or *mpi*, but to take account of the fact that one session you may be logged in to a *Solaris/sparc* machine, but the next session you could be using *Solaris/x86*, or *IRIX/32*, or whatever[32], and *Amrita* automatically insulates you from the underlying hardware. Thereby allowing you to work in an uninterrupted fashion.

Amrita is constructed in layers, not to bamboozle reluctant programmers, but to build in the necessary flexibility to allow the system to grow, while remaining considerate to the needs of its less computer-literate users. This script demonstrates how an *Amrita* expert can `replace`, or overload, a keyword with his or her own customized code:

```
amrcp  vki/overload.1
amrita run_overload
```

```
EulerEquations
plugin amr_sol
set dir = your/very/own/keywords
fold::amrcc {  ISL call-back routine
   fold>amrso ?= $dir/keywords
   fold>src   ?= $dir/keywords.C
   #include "AMRITA/isl.h"
   AMRVOID solver() {
      fprintf(stdout,"solver: %s\\n",ISL::get_file());
   }
}
fold::print { Amrita interpreter
   fold>file ?= $dir/solver.pl
   fold>guard=|,dollar off
   sub amr_sol'solver {
      unless($line =~ /^\s*([^\s]+)\s*/) {
         $error[1] = "expected name of a solver!";
         &amrita'report'error();
      }
      $line = $';
      &isl'put_ltag(0,'amr_sol::solver');
          &isl'put_file(1,$1);
      &isl'put_rtag(0,'amr_sol::solver');
   }
   1;
}
replace amr_sol::solver with $dir/{keywords:cc::solver,solver.pl}
foreach scheme (godunov_km,roe_fl,ausm_km,hlle_km,efm_km)
   solver code/$scheme
end foreach
```

[31] A computer scientist might ask of the *ISL* communication channel: does it use a "thick pipe" or a "thin pipe?" Conceptually *Amrita* does not care what form the pipe takes, as it only carries a small amount of scheduling data; the heavy duty input-output is handled by the operating system in the normal manner. Consequently, *serial* and *mpi* platforms can both use the same *ISL*. The internal working of the *ISL* parser functions differently in the two cases, but its interface to the back-end of *Amrita* remains the same. There is no need for the *Amrita* interpreter itself to run in parallel, because its work, although logically complex, is not labour intensive; *Amrita* provides the brain, the *plugin* provides the brawn.

[32] Look in the directory *$AMRITA/SYSTEM* to see what platforms your *Amrita* installation is set up for.

Do not worry, if you cannot understand the source code for the above example, as it is targeted squarely at system-level programmers[33]. The key point to grasp is that because both ends of the *ISL* pipe-line can be overloaded, an expert can develop and test incremental improvements to *Amrita*, in situ, without impacting on anyone else. Then when the upgrade is ready to ship, it can be slotted seamlessly into place. At that stage, a conscientious developer would not discard the old code, but would package it up in such a way that users could easily back-track to the old-version, should the need ever arise[34]. For instance, suppose `amr_sol::flowin` and `amr_sol::flowout` were revamped so as to take special advantage of an upgrade to the input-output hardware of a parallel computer. A small percentage of the users, may actually be negatively affected by the upgrade and wish to drop back to the old way of doing things[35]; `replace` would allow them to do so in a relatively pain-free fashion.

Although far simpler than *run_overload*, this next script shows an important design advantage of employing an *ISL* pipe-line:

```
EulerEquations
plugin amr_sol
solver amrita://www.amrita-cfd.com/vki/godunov
LatexHead
grab::info LatexDocument from solver
parse token LatexDocument
LatexTail
Latex
```

```
amrcp  vki/www.1
amrita run_www
cd     latex_files
amrps  amrita.ps
```

Because *Amrita* employs an explicit communication channel, as distinct from direct memory accesses, it can readily exploit the world-wide-web. The above script uses an *amrita:// URL*[36] to obtain a flow `solver` to link with *Amr_sol*, and is no more onerous for the user to write than had the solver been stored locally. At the system level, however, the story is quite different. Instead of using *NFS* to read the `solver` straight from disk, an *HTTP* request is sent to the remote server *www.amrita-cfd.com* which replies by sending back a *PGP*-authenticated *Amrita* script (see §C.1). This script, if it comes from a trusted user, is then run silently in the background to produce a shared-object *code/godunov* which is sucked into *Amr_sol* in the usual manner. Once the `solver` is installed, the `grab::info` command obtains an *Amrita* script[37] to generate a LaTeX document to produce the *PostScript* output *latex_files/amrita.ps*.

---

[33]The `fold::amrcc` constructs a four line *C* program and compiles it to a shared object. In the fold directives, the `?` preceding the = is optional and requests that no work be done should the target file already exist. Observe the use of the namespaced procedure call, `ISL::get_file()`. *Amrita* employs a pre-processor *amrpp* which maps such namespaced calls to standard *Fortran* or *C*, depending on the language being used; `AMRVOID` is a garden-variety, typedef which is defined in the header-file *AMRITA/isl.h*. A detailed explanation to the workings of *amrpp*, and why it is needed, can be found in any of the documents created by *BasicCodeGenerator*. The `fold::print` creates a small *Perl* file which is sucked directly into *Amrita*, using *Perl*'s `require` command, so the interpreter knows how to parse the overloaded version of `solver` in the user's script; the shared-object file is sucked into the `plugin` in the manner described in Appendix D.

[34]*Amrita* strives to ensure that user scripts are backwards compatible between software releases, but it feels under no obligation to do the same with all its system internals. For this reason, any code you write to link with *Amrita* is best recompiled when you upgrade to a new release in case the glue which binds user-code and system-code together has altered. As the source code for this glue is generated automatically by the system, this does not create work for the user.

[35]It is possible for a system to improve its peak efficiency, at the cost of lowering its off-design performance.

[36]Internally an *amrita:// URL* is translated to an *http:// URL*, and the file received is rejected unless it has a valid *PGP* signature from a trusted user. Therefore to run this example you must have *PGP* installed on your system, and *Amrita*'s public-key (see p. 53) must be installed on you public key ring.

[37]This script is embedded in the executable at compile time. The `parse` command executes the contents of a token, or a file, or an *amrita:// URL*, as if it were in-line *Amrita* script.

5 Repeatability

This next script, which outputs Figure 9, offers some light relief from the system stodge of the previous section:

```
amrcp  vki/surf.1
amrita surfs_the_web
amrps  ps/surf.ps
```

```
plugin amr_sol
postscript on
...  typeset title
set eccomas = http://www.amrita-cfd.com/vki/eccomas.ps.gz
set vki     = http://www.vki.ac.be/images/vkifront.jpg
set galcit  = http://www.galcit.caltech.edu/relief.gif
set icase   = http://www.icase.edu/images/transparentlogo.gif
set chester = http://www.chester.org/gif/chester-city-council.gif
paste $eccomas  in box  10, 20,125,?
paste $vki      in box 144,180, 60,?
paste $galcit   in box 155,120, 40,?
paste $icase    in box 150, 80, 50,?
paste $chester  in box 150, 20, 50,?
```

Although light-hearted, a number of common-sense observations pertinent to CFD software-design can be made. In fact, the initial motivation for *Amrita* is epitomized by the flow in Figure 9: the flexibility of my "research codes" had grown to the point where it was difficult for me to reproduce[38] pieces of work from week to week, let alone month to month, or year to year. The algorithm behind *Amr_sol* is sufficiently intricate that, even with stringent quality control, "features" (aka bugs) would creep in and destroy my confidence in the method as an investigative tool.

The first step to improving repeatability is automation: if you envisage having to do a job more than once, automate it! Doubly so, if the task concerned has anything to do with testing. The more a CFD code is tested: the quicker that bugs are flushed out; the more likely you are to spot conceptual errors; the sounder you sleep in bed at night. Testing by-hand is both time consuming[39] and prone to human errors which render the results useless. The time saved via automation can then be used profitably on the other important tasks you need to get done[40].

The second step to improving repeatability is self-sufficiency: the more you rely on third-party software, the more often you will come unstuck when said items go missing[41], or are upgraded in a backwards incompatible way[42]. For this reason, *Amrita* only uses third-party software which is either a de facto standard (e.g. *Perl*, *LaTeX* and *GhostScript*) or has source which I am prepared to maintain myself (e.g. *Origami*). Even if you do not have the software skills to be fully self-sufficient, you can help yourself in two simple ways: avoid using vendor supplied compiler switches and language extensions, which may be here today, but gone tomorrow; do not upgrade a piece of software at the first sight of a new release, let it bed down before taking the plunge[43].

The third step to improving repeatability is simplicity. The *HTTP* protocol[2] used to drive the web is a good example of how relatively simple software can prove revolutionary, whereas sophisticated software – ahead of its time – can all too easily fall by the wayside[44].

[38] This is used in the strictest sense, that is: bit for bit; byte for byte; word for word.

[39] Often to the point where it is ignored as being too much trouble.

[40] At least one of my colleagues would argue that I only automate programming jobs so as to free up the time needed to automate yet more jobs!

[41] You might change jobs and find your new place of employment does not have the graphics library you need.

[42] The implications of this advice in regard to you using *Amrita* will be discussed at the end of the lecture.

[43] *Perl*, the language used to write *Amrita*, underwent revisions 5.000 → 5.001 → 5.002 → 5.003 → 5.004 in a matter of a few months. Although to be fair, *Amrita* went through many more releases in the same period.

[44] A good example is *Algol68* which introduced operator over-loading, a feature now lauded in *C++*.

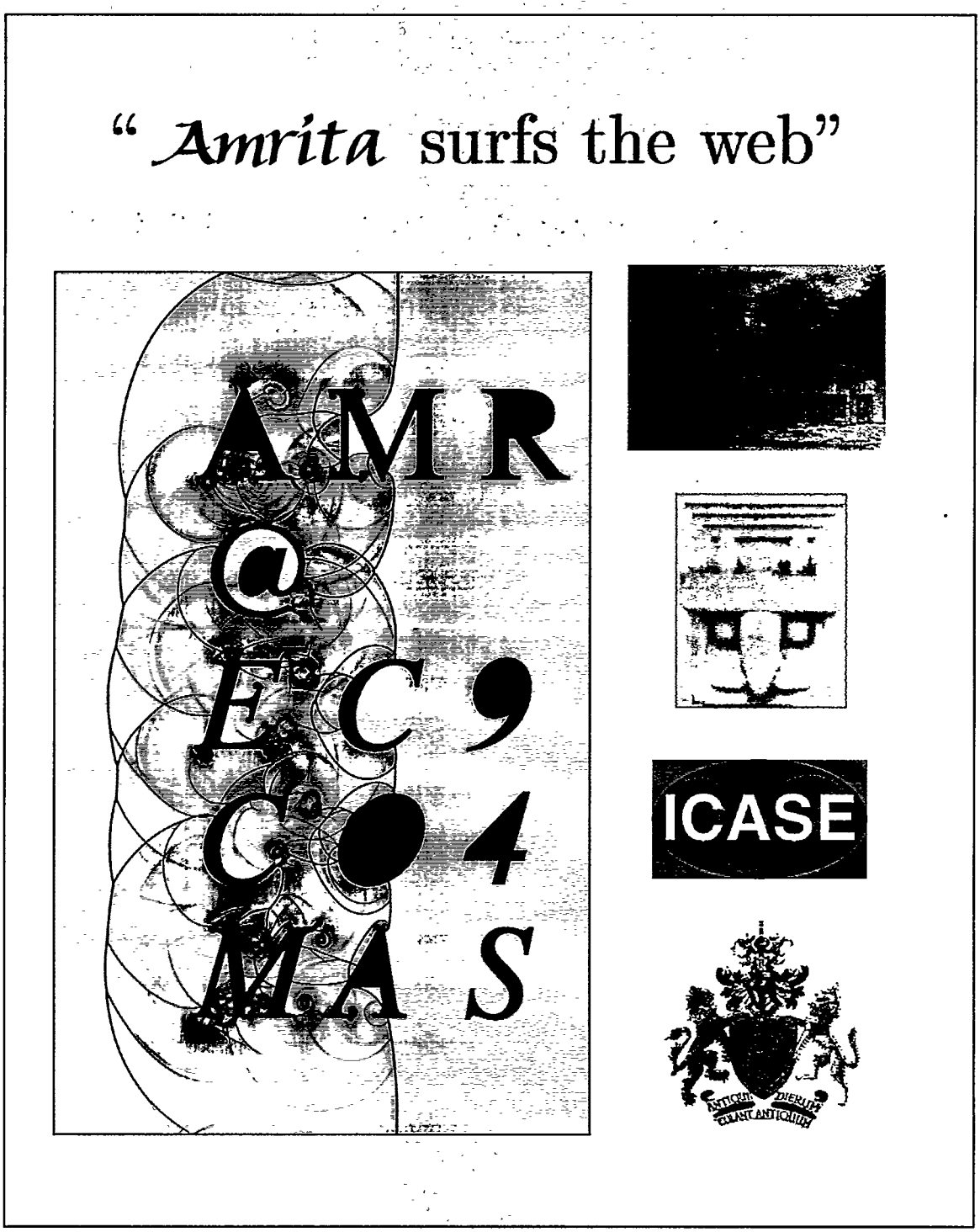


Figure 9: Page output by the script *paste_http*. *Amrita* was initially developed to reduce the effort needed to maintain the intricate software used to produce "AMR@ECCOMAS94"[20]. The right-hand images, from top to bottom, are: a painting of the von Karman Institute; a false-colour image of a relief sculpture located over the front entrance of the Guggenheim Aeronautical Laboratory (the original home of GALCIT); the logo of the Institute for Computer Application in Science and Engineering (where much of the spade work for *Amrita* was done); the city-council coat of arms for Chester (*Amrita*'s spiritual home).

6 Accessibility

Appendix C describes – *Amrita mailit* files – a simple idea which will likely out-live the rest of the system. This *mailit* was the very first one to see the light of day:

```
amrcp  vki/mailit.2
amrita run_cellular.mailit
amrps  ps/Cell1.ps
amrps  ps/Cell20.ps
```


```
-----BEGIN PGP SIGNED MESSAGE-----

AmritaMailit::run_cellular {
   origin {
      Amrita v1.35 R03-07-97
      user   James J. Quirk (aka jjq)
      date   Sun Jul  6 14:10:31 PDT 1997
   }
   resources {
      disk 57.09 MBytes
      cpu  3 hrs 34 mins 24 secs
   }
   operation {
      amrita -mailit -a 1step/A run_cellular
   }
   script {
      M'XL("'D\P@,''W)U;E]C96QL=6QA<BYT87('[1O]4]LX=G_V7Z&%3'MM/NR0
      MA$Z&=*>%+.6P[4%IMQPLTQ&VG&AQ+%>V*5F6__W>DRS;^>#CYEKV;O";%MO2
      M>T_2^Y8<RS3\XK(@2',J7_I!8-MVK],A>.W:CKHZO:ZZ*N@Z'6+W-GIPV^FU
      MN\1V.NV-S9^(_:,F5(8T3F#EMOW''U]OQPO/0_$M=)S-Y?UZ*22__I_'JK5*
      MR'9F',1CB0AIPD5(/,DO6$BL58NFB8AH,B;/'WYFO9*230_YGRPF[W?W+O\Q
      M>*$6;.V%//G'8I%*%[KBB+ILT-ZQ'C'R><'(7.-6[<_0Z_7C)/6FEAES7XJS
      M@$W(E44(.8#_9$'TW@&V#,LM0VSQRBV>=6W%(KA@DKC*8RTG3EAD12).8E?R
      M*"$EM#Q!PH%3;]M(/:'2'9..31'Q)M\XK,OU@X'=[&*W'XAO(DT(%RV<8"W$
      MQB@0B5I$%&>MS2C&CG?3O0D=,8N%'O&$]7?K\C^#EL?C\S8J@[-8"Z[UZH>,
      MA/[?N\7_VYT-]/_NIKUAPQ_T_T[7J?S_(2!F"5&>]#Y*8NU9[6ZS9^7MQRXA
      MNHUNZK4Y1><!,9U=NV@=YJU.J368T,ML@*+1*U";I685/C3NCE7Y_M<"[?$M
      ML/36?AK$$.?#D?%_C_DT@9+X1XQ[A_]O;&[T<O_?V,@\W[6[W<K_'P)6R<<Q
      MCXDR;[A&5,;,(V=3DHS!WJ5PF9=*UK=4G4@(3)97C8E'M'CN,9+9$+F800HU
      M@"\D>2JAO@1/?5HGR30J^!'ZD3RAI$')F'41R="@$_XAX41(9)A0'L1-I%H2
      M&T3(&AZ?L@"&@H4&1?@8T<F$+D:5NZ)5N;4<F&RK:!>>;\)DTZY@YPO\T\,_
      MG?K<<''!N'Y1%;KA"I<VW)7&OQ_:S(2R^U4B(/*I2JTTQ5(D-XB'&::3,XB2
      MPB>1X"@@\)",:>'W)*,N5GPWAWQ',3!XQ$V3AO'S/@7544[5*4WP:'Z]P'?'
      M-%R<YAP-&8${'$$01HR0E23*A1E*LABC5;FB1&)C@!F]QR5S$R&G!8<QHQY(
      M'3B@M>7M42+!J/K]@V_^F6C5JLEX:%H)^@*,-Q:!A]8>")<FK%PJ^U*4!&%X
      M^9)]52E0+PP74Q:Z4Z4"";*'Z+M'I(0"$G0T428![61'0DIJ,B0[1S&NYT4F
      MP)BI@/>-->Y@90,<I>&R,8^X<80:VK-7O",J=/T*D\$4'F1-6A$$DQ1'"4XZ9
      MO.@N/9S>6GV33((=X:;@S'G@/P=D7*<FZ(^A@SR'*\75LL;^]N>=EDE7R$^3
      M(:7]@($9?:8\*5BI$0<ZT&541><A4Z8='\)KZIZ/I$AQ7M@-P@0,?PQHVOK7
      M:@9Y72-D+I((HXXY[!@MG$\M(\X)10RT[PORNUKXE>F_[A,TI@FI&6XYA0K.
      M(-1SV)/-LL/B(YND;G@'(B+;8)(@R_@&668SK<T+N'7Z:**HFMA42/0-^'TI
      M'E^C64IP5KP,'MK4>[J\_Q!R:O*1GID%Q%&0RA$X-LC/%0%48QX8@)6@9X40
      MMSZ]?9D_'@R;70)BM'*4G*'O%Z,.0Z\TYGT4AU2O0A"8)"&=L,&*HF870\Z$
      MA/'Q:!>:O\?"<"W/YADO+&@I&HRX'QJ>O^6A,=D$E2/21TB5^ND]@^J71HQ8
      M=^L7MO2MN:UX$S"_:XUQQ_E/IV-OE.H_W/_UG&YU_O.@H$+S(4O2")(%3S@-
      MUIX(=E.AQW5,A*2BRT/1'D.,$P<"-E:&9.$(AUC*9%W,X<M@E4*9"(G/GRJ<
      MAHX:FLC4;"J](T'ME1%)3+.8FN*(N=SG+@1SFL2:@NJB$[O_RVF)KI?1%<61
      M:CA0F%W[U&!"P95A(C^H*53:T,C@FY&Q@+K0F9]!^!N9%90-;CX30@P)]/!)
      M.BD582/)/1*P"Q9DZY![N90,=<>LG_G@^Y@:M20T'99YV<HS@G;7@'?5RDUE
      M'@P:CI*QYA%&-'''+"X&?6$T5:H7,QS@M"NRM7IFI874S5I]?GG+B$?>+*'3
      MG2&$G;NR.ARM5&%IVHC'T/+%QZ-@>&PX0\?0)I*[YS@@A,E(;^H;D&@1HJ>:
      M]C@103XNEK5JNW]-MAHO=66H%H9Q$DS-G"Z'Q<.U=3"H';2&@]JPY0UJ'N)M
      MBPD4F.PX],@V,^<8U+1FH'@E[T'!KSP:J9"O1Z79(R%Y4V$P-56:D3E@J,D]
      M;%.G"\RW@/,=,:'<\,'9!Q@M80JK:FO0JJ%A9+T>E*L@Q;SNUIN.E<M^EJRF
      M;Y]E'YHQA9FG1LRP"M9XJE(S)E3P50UDZSD,^0VV-&.G]W*IZ_R:@D60HG1^
      MY2SP2DL@\:,V0!&U4QV>(62VU.\/R-')::-V9"@_/T4ZU;$%)?T8,XPUY-&$
      MAOJ1=W*Z7O^!F',.G]'^RNC0<'OZ;\J<%MN!:!GZ_CSW_5NY'\^C'QOTE_E*
      M/7"*5$(E!IOB?,%@!_5/'[O^&_S?AZ=N_7C@Y%JXQ+**@.%?BOI4U+W+NI=Y
      M*1F+Y$L<B42Q6J-G\=J_4++>M-5>WVJODR=/R!I.X.5&<;_57<^H"\7YJ$>M
      MA\7&\I3S$4].E]G%:RS$J9QNY^DH7[B.!K<HN5:*%\OE>X.Z[T-XL^)O-8';
      M3.'^(RXUAAE"H^@W9[';]E24Z&/R"*@VS+H.;^X:EKO'1B2-1('[W0"2@Y6!
      M,9'/9;P(-L:NY&<L5TA9BP3/6\^9TGNNT@]Y-MN&@I1)3DL6$M!13$Z$$']=
      M7E[^!==30D9X?/*<GS:.T?!K&.D73'7C''2C4AA5\3:[+R9!\B"D@FVQC+$>
      MH?ZQ_I\YU/ONU?^=]7_;Z>7U_\9FMX?GOYM.]?[G04'5\K/O;J^TK>A@-4B^
      MOY0@%RI'60FASV"AWMC1SQ/AL0"?J:ZJ\J(*GM0(;VD*V^QWPDOS'N*@*@\O
      MV!":Y/!K2O5N0^T@!K7B'+=^D@T=U/5;Y5IQ@&S\6_'%'>Z7N'@.RKGS&>RR
      MY%!M-S0]]]?48<I:>7'KZWCL'1;N5QPZ&V$LRH)'"<\*$A[BT6.):&N(/JO/
      MG[)SIK67P<Y28<3][%Z?],R<(9F7,\7<5TH'GRN$?24KR']EV?QO.IB:8TH6
      M3J7,>=QU?_Y0*J<HGTH9]**77?)DR0)A^@U&5LP\RHN9G_\2691_0[!,_DQ*
      MD+(6_:R40BAK/)5\O)_GI[5H+]MCYIX?JE>+);G__'0,(!9K*\7;QI4&JWE-
      M8^YN'\(N[@]I'O.YF@GI,13,DVR+/"!2L*=ZHLJ-9C'UN\T,4[_=+/=[F=!U
      M?RB*SNLEDD=ST!S[_?TW>^I5K%%5).1,YX)4R@XT'\[O(,$.OX;'OW@Y!,TS
      M,LR95?K_')_E?^K_%_E_RK_(U3Y__'%_RK_/V[!5_F_RO]5_J_R/T*5_Q]?
      M_*_R_^/6?Y7_J_Q?Y?\J_R-4^?_QQ?\J_S]N_6/^S5YC^P&97\.=WW_"@\G_
      M[4Y/??_9L:O\_Q"@;-U\QTC43X'WX9;$_$\V^'08L'225WT"Z8Z'`_T)]7X/
      MGO\2\-'=!YQ)%N8<\&M)*ADE;;O>M>N.K?YCCQOP*,)O1/2OA'(^^.OYJ]WV
      M]2(_]3/9P=6NT\@NW$DK__\O_;_X$O3OMK\**JB@@@HJJ****BJHH((*'@;^
      )#3W36S4'4'''
      `
   }
}

-----BEGIN PGP SIGNATURE-----
Version: 2.6.2

iQCVAwUBNMXI1LmgfLsjP/xJAQF1WAQAm+0zIC2jzelNkiaeyk+R3CNpiBgp/Ko9
RN7klAowGXJPJmTJ+CNHIWvuw8c3Fbomp+k5kuvp6Yp01h9scM2bjRWJkqJxbeNG
gipVNYzkoP5UZgzD98gJbse5aO4eEJR9VaDaGjD6ECYS1jYYiDPSH8gMLtMhUxeB
JJIYLap/TyA=
=6ndU
-----END PGP SIGNATURE-----
```


it runs a simulation to investigate the cellular instability of a detonation wave, see Figure 10; the recipient was Dr. Mark Short (`mshort@tam.uiuc.edu`).

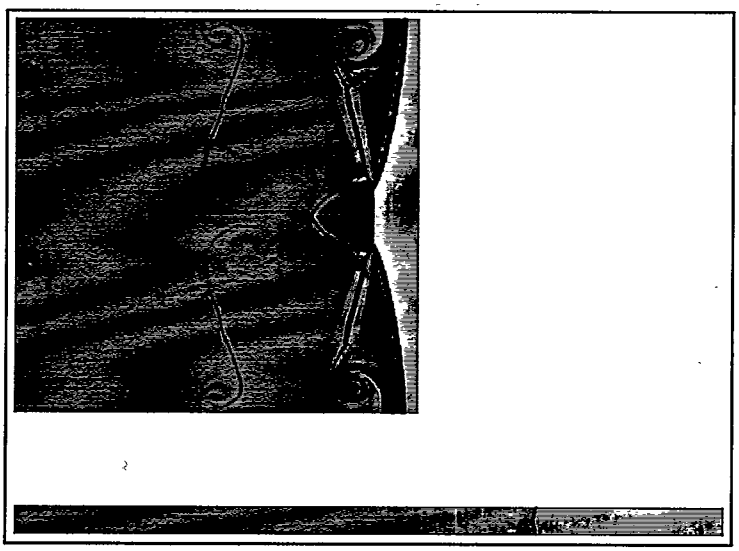


Figure 10: *PostScript* image *ps/Cell20.ps* produced by *run_cellular.mailit*. The bottom image shows the extent of the computational domain. The notional resolution of the grid is 3,200 by 256, but *Amr_sol*'s mesh refinement skills restrict the expense of this high-resolution to the local vicinity of the detonation front, that is the darker regions of the zoomed image.

Although the *mailit* was generated under *Amrita* v1.35, it runs faultlessly under v1.38, thus illustrating *Amrita* has some measure of backwards compatibility[45]. This reliability ultimately stems from *Amrita*'s high level of automation, which facilitates repeated testing. Another benefit of automation is that it improves accessibility to specialist fields. Here, instead of simply reading about the cellular structure of detonation waves(e.g. [24]), you are able to gain first-hand experience of how the phenomena develops. If you have not already done so, examine the files *ps/Cell1.ps* to *ps/Cell20.ps*. The simulation was started by prescribing a planar ZND wave[8] near the left-hand end of the domain. At the time shown in *ps/Cell1.ps*, the wave has just ingested a hot-spot located on the domain centre-line. The hot-spot seeds a physical instability which leads to a highly dynamic wave-pattern of which Figure 10 is just one snapshot[46]. This next script generates a 120 frame *mpeg* animation to illustrate the wave-dynamics involved[47]:

```
amrcp    vki/cell.1
amrita   -a 120 make_movie
netscape cell.mpg
```

```
... preparatory script
screen on
do n=1,$nframes
  march 1 step with cfl=0.5
  AmritaBlue
  filled rectangle $film::area
  SchlierenImage
  savescreen $film::area [snap 2x2] to sch/frame$n.jpg
end do
EncodeMpeg n1=1,n2=$nframes,mpeg=cell.mpg,dir=sch
```

[45]Do not be fooled by the closeness of the version numbers, because over 2,000 coding hours were spent revamping *Amrita* between these releases. Although the language is now sufficiently mature that no wholesale changes are planned to disrupt users, in the interests of consistency, changes are occasionally made which force minor script alterations be made. For instance, the v1.35 commands `splurge` and `slurp` have been superseded in v1.38 by `fold::print` and `fold::file`; the utility *amrsearch -vi* is useful for making wholesale edits following a syntax change. A software project the size of *Amrita* is never static, even if the look and feel suggests otherwise.

[46]Round-off errors are sufficient to trigger the physical instability; the hot-spot merely speeds up the process.

[47]If you are unsure how long this script will take, do a 1 frame animation and scale the time up.

7 Extensibility

On p. 2, I wrote:

> it makes no sense to ask "What is *Amrita*?" or "What can *Amrita* do?" Instead, you should ask: "What do I need to provide, to enable *Amrita* to do *such and such*?"

Amrita – like *UNIX* – has no set limits: it provides a foundation to build on. Consequently, the ultimate scope of the system rests with the industry, and perspicacity, of its users. In this regard, the flow over "AMR@ECCOMAS94" example is not as gratuitous as it first appears.

The solid-bodies for the computation in Figure 9 were obtained using the same construct which allows this script to output Figure 11:

```
amrcp  vki/swirl.1
amrita make_swirl
amrps  ps/swirl.ps
```

```
plugin foo
def Path'bodies
   ...  use PostScript to define bodies
   parse token postscript::output
end def
postscript on
plotfile ps/swirl.ps
extent of Path'bodies -> xo,yo,dx,dy
set r #= sqrt(($yo+$dy)**2+($xo+$dx)**2)
autoscale on -$r,-$r,2*$r,2*$r
do n=1,36
   rotate 10
   HLS<$n*10,128,128>
   stroke Path'bodies
end do
```

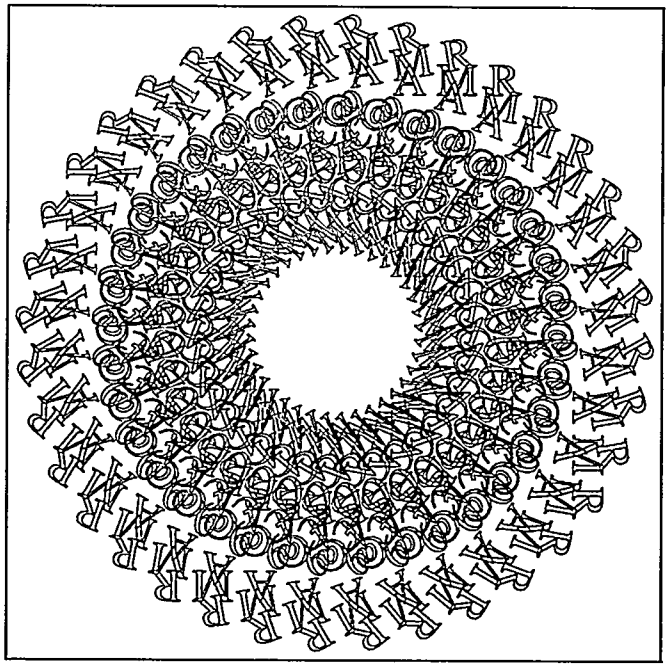


Figure 11: Output produced by the script *make_swirl*. For best effect, the image should be viewed on a 24-bit colour display. Try appending the line `echo $postscript::output` to see what the program `fold` produces.

7.1 def Path

Amrita provides a `def Path` construct which allows paths (i.e. segmented curves) to be defined using the *PostScript*-style operators: `newpath`, `moveto`, `rmoveto`, `lineto`, `rlineto`, `curveto`, `rcurveto` and `closepath`[48]. The program fold:

```
fold::postscript'bodies { use PostScript to define bodies
   fold>token=postscript::output
   /print-path {
      /prime-comma { /n 0 def 0 exch } def
      /print-comma { (, ) n 0 gt {print}{pop}
          ifelse n 1 add /n exch def } def
      /print-segment { print prime-comma dup -1 2
         {print-comma index 100 string cvs print} for
         1 sub {pop}  repeat (\n) print } def
      {3 (moveto    ) print-segment} {3 (lineto    ) print-segment}
      {7 (curveto   ) print-segment} {1 (closepath ) print-segment}
      pathforall
   } def
   /F1 {/Times-Roman  findfont 80 scalefont setfont} def
   /F2 {/Times-Italic findfont 80 scalefont setfont} def
   /C1 100 def /C2 180 def /C3 260 def /L1 420 def
   /L2 340 def /L3 260 def /L4 180 def /L5 100 def
   /M {moveto} def /P {true charpath} def
   F1 C1 L1 M (A M R) P
   F1 C1 L2 M (@) P
   F2 C1 L3 M (E) P F2 C2 L3 M (C) P F2 C3 L3 M (9) P
   F2 C1 L4 M (C) P F2 C2 L4 M (O) P F2 C3 L4 M (4) P
   F2 C1 L5 M (M) P F2 C2 L5 M (A) P F2 C3 L5 M (S) P
   print-path quit
}
```

contains a small *PostScript* program[49] which unravels the drawing operations used to `stroke` the outline of a `charpath`[1] and stores its findings[50] in the token `postscript::output`. The contents of this token is then parsed by *Amrita* to define the desired path, exactly as if you had typed the `moveto`, `lineto` and `curveto` commands by hand.

Do not worry, if you cannot understand the machinations of the above *PostScript* fold. The important point to grasp is that *Amrita* has hooks which allow experts to embed specialist programming languages into a script. Thus, instead of the present font-outlines, the fold could just as easily have contained a CNC program for machining a wind-tunnel model. The expert may need to fashion a fair amount of glue-code to incorporate such a fold, but the work is done as a one-off, allowing users to benefit time and time again. Here, for instance, `fold::postscript` can be used in conjunction with a solid-mechanics `plugin` called *Adlib* to produce Figure 12:

---

[48] These operators take the same arguments as their *PostScript* counterparts, but use infix rather than prefix notation e.g. `100 100 moveto` is written `moveto 100,100`. At the time of writing: `arc`, `arcn`, `arct` and `arcto` have not been implemented. The decision to replicate *PostScript*'s path model illustrates that *Amrita* does not reinvent new keywords for the sake of it. Regarding the flexibility of the path model: if you cannot draw it, you certainly cannot compute the flow around it!

[49] The program is farmed out to *GhostScript* to process, and the results are stored in the `token` specified by the `fold>` directive. *GhostScript* is also fed an error-handler which allows *Amrita* to assume control should the *PostScript* program abort.

[50] The *PostScript* `pathforall` operator allows the drawing instructions to be unravelled directly into *Amrita* script. The fact *Amrita* employs the same operator names as *PostScript*, is not critical to the conversion, but it does reduce the code involved.

Figure 12: Page output by the script *amrita@vki*, when run with `plugin` *Adlib*. Note when you run the script with `plugin` *Foo*, only the outlines of the letters are produced. Even if you do not understand how the script works, you should not find it too much trouble to substitute your own message in place of the current one. If you are willing to help dot some of *Amrita*'s i's, there is a sign-up sheet for volunteers at `http://www.amrita-cfd.com`.

7.2 `plugin` *Adlib*

Adlib employs an advancing-front grid generation technique[6] which allows the font-outlines to be meshed up with an unstructured, triangular grid[51]. Therefore, although *Adlib* comes out of the solid-mechanics community, its software organization is very similar to that of an unstructured grid, CFD code. Consequently, the *Amrita* glue written to drive *Adlib* could be re-cycled to drive the equivalent CFD code, should one be made available for distribution with *Amrita*[52]. Please note, *Adlib* is not bundled with the *Amrita* installation kit and so you will not be able to reproduce Figure 12, in its entirety. However, this *mailit* substitutes a `plugin` called *Foo*, see Appendix E, so that the script can at least be seen in action.

```
AmritaMailit::amrita@vki {
   origin {
      Amrita v1.38 R07-01-98
etc ..
```

```
amrcp  vki/mailit.3
amrita amrita@vki.mailit
amrps  ps/message.ps
```

Although `plugin` *Foo* has limited functionality, it has the exact same structure as *Adlib* (or for that matter *Amr_sol*). Therefore, if you can understand the construction of *Foo*, you will see why *Amrita* need not be too concerned with the inner workings of *Adlib* to be able to drive it[53]. In part, this is because each `plugin` brings its own specialist keywords to the programming table. The first set of *Adlib* keywords[54]:

```
KEYSPACE adlib:: {
   autoscale
   extent
   plot
   DEF BoundaryRepresentation {
     *addbody
      space
      globalspacing
      DEF Body {
         edge
        *edges
         name
         node
        *nodes
         DEF SubBody {
            path
            loop
           *loops
            material
            name
         }
      }
   }
}
```

were chosen to reflect the engine's internal notion of a Boundary Representation (BREP) grid. A BREP grid is viewed as a collection of bodies, where each body is made up from a collection of sub-bodies. Each sub-body is then thought of as consisting of a set of loops of edges, and each edge is defined by a set of nodes.

[51] *Adlib* can also produce unstructured, tetrahedral meshes. *Amrita* will drive any software which comes its way, and so it makes no sense to ask: "Is *Amrita* 2D or 3D?"

[52] CFD philanthropists can contact the author via *e-mail*.

[53] In fact, I have never seen the source for *Adlib*. The *Amrita* wrapper was written with knowledge of a handful of subroutine entry points then linked with two archived library files.

[54] The `plugin` is still under development.

Once the parsing machinery was written for the new keywords[55], the informed *Amrita* user could then write this procedure to mesh up a rectangular block:

```
proc MakeBRep {
     theta         = 0
     globalspacing = 0.04
   }
   def BoundaryRepresentation
      space         2D
      globalspacing $globalspacing
      def Body
         name simple
         nodes {
            V1 < -1.0,   0.0>
            V2 < -0.5,   0.0>
            V3 <  0.0,   0.0>
            V4 <  0.5,   0.0>
            V5 <  1.0,   0.0>
            V6 <  1.0,   0.5>
            V7 <  1.0,   1.0>
            V8 <  0.0,   1.0>
            V9 < -1.0,   1.0>
            V10< -1.0,   0.5>
         }
         edges {
            E1 <V1,V2 ,V3>
            E2 <V3,V4 ,V5>
            E3 <V5,V6 ,V7>
            E4 <V7,V8 ,V9>
            E5 <V9,V10,V1>
         }
         def SubBody
            name     loop1
            material unknown
            loops {
               L1 <E1,E2,E3,E4,E5>
            }
         end def
      end def
      addbody simple {
         rotate $theta
      }
   end def
end proc
```

[55]A fairly straightforward task for someone proficient at *Perl*. Note, however, that the *Amrita* keywords used to drive a `plugin` are written as one-offs, by some suitably qualified individual, in much the same way that LaTeX style files are written as one-offs. The usefulness of the `plugin`, to the end-user, rests in the *choice* of keywords and the amount of error checking they employ. The keywords: `nodes`, `edges` and `loops`, employ copious amounts of error checking so as to pinpoint the user's exact mistake. Nevertheless, the keywords are simply too cumbersome for the user to type in the description needed to produce Figure 12, and so I took the trouble to add a keyword `path` which would allow the user to simply specify the name of an *Amrita, PostScript*-style, `Path` and have the `plugin` do the work of deducing the `loops`, `nodes` and `edges`. *Amrita*'s golden rule: the more trouble the systems-level programmer goes to, the easier programming-life becomes for the applications specialist.

and throw in a driver script:

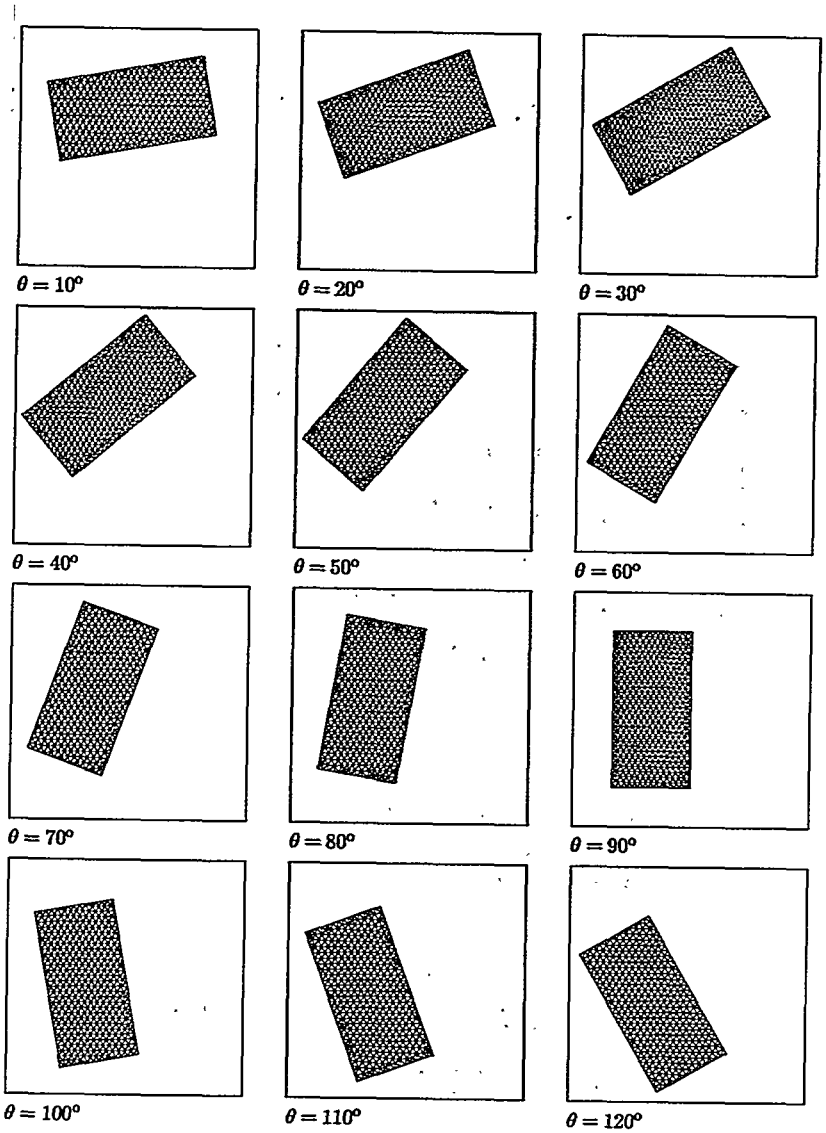
```
autopath +lib
plugin adlib
postscript on
set bbox = -1.5,-1.5,3,3
LatexHead file=$amrita::script.tex
LatexNupFig iup=3,jup=4
foreach theta (10,20,30,40,50,60,70,80,90,100,110,120)
   plotfile latex_files/ps/$theta.ps
   autoscale on $bbox
   rectangle $bbox
   MakeBRep theta=$theta degrees
   plot grids
   plotfile
   LatexNupFig {
      file    = ps/$theta.ps
      width   = 4.5cm
      caption = \$\\theta=$theta^{\\rm o}\$
   }
end foreach
...  output title
LatexTail
Latex
```

to produce the output shown in Figure 13.

There is no question that these results could be generated independently of *Amrita*, since *Adlib* does the basic work. However, *Amrita* streamlines the operation to make the *Adlib* user more productive[56]. The operational infrastructure developed to drive *Amr_sol* transfers directly to *Adlib* because it deals with day-to-day, programming needs and is relatively unconcerned by algorithmic issues. This infrastructure grew from a belief that modern CFD is poorly served by traditional means of scientific communication. The AMR method behind *Amr_sol* does not have a precise mathematical formulation and its implementation necessitates writing a large piece of sophisticated software. Consequently the "algorithm" cannot be reproduced exactly, by a third-party developer, because numerous small but nontheless essential details are missing from the open literature. Therefore, in the absence of purloining a code and reverse engineering it, many details have to be laboriously reworked. But given the inevitable variations in the success of specific AMR software implementations, a body of anecdotal evidence is bound to accumulate regarding the merits of the general approach, as it becomes more widespread.

To combat this malaise, *Amrita* is designed to provide a generic means of disseminating small, but important, algorithmic details in an unequivocal manner, so that the wider CFD community can benefit from the hard won skills and experience of individuals. In this regard, there is no difference between developing a CFD code, such as *Amr_sol*, and developing a solid-mechanics code, such as *Adlib*. *Amrita*'s software design principles – repeatability, accessibility and extensibility – can be put to good use no matter what the target application.

[56] In a similar vein, the argument for using *Amrita*, in preference to *Perl*, is that it streamlines the construction of a certain class of program; the same argument can be employed to justify either using *Perl*, in preference to *C*, or using *C*, in preference to machine-code. *Amrita* is not an attempt to out do *Perl* (try using the `fold::perl` construct), it just grew from a different necessity.



example1: *Amrita* checks *Adlib* for rotational invariance.

1

Figure 13: Because *Adlib*'s advancing-front algorithm marches in from fixed points on the block's boundary, it would be expected to produce grids which are invariant to the orientation of the block. The above output shows that this is indeed the case. However, the first time the script was run it unearthed a remnant left-over from a pre-*Amrita* exercise which broke the invariance. Programs which need re-wiring between user-problems, no matter how innocuous the changes involved, always run the risk of such "code rot."

8 An Open Invitation

These notes describe but a small part of *Amrita*, and propound an even smaller part of the underlying philosophy. However, enough of the system has been exposed for you to judge whether or not the approach has merit. Therefore, to close, I explore some possibilities for how interested parties might contribute to *Amrita*'s further development. But first, I build on some remarks made on p. 2.

Amrita stands or falls on its utility as a labour saving device. For this reason, no claims are made regarding its algorithmic originality or efficiency. The scope of the system is such that it touches upon a number of active research areas. Consequently, although these notes are undoubtedly self-absorbed, I recognize *Amrita* has much room for improvement[57]. The system is designed to allow experts to contribute specialist components, while at the same time remaining accessible to non-experts. The danger of this approach is that one-half of the target audience dismisses *Amrita* as being overly complicated, while the other half dismisses it as being old hat[58].

At a practical level, this makes it difficult to locate individuals who are willing to help maintain a system that runs to over 220,000 lines of code[59]. Half the target-audience members feel they do not possess the skills to contribute, the remaining audience members would rather reinvent the system wheel for themselves. This second response is fully consistent with the advice I offered on p. 27. If you develop your own software, you should always be reluctant to utilize unsupported, third-party software. Unfortunately, at the time of writing, *Amrita* is not a supported product, because its development and maintenance fall squarely on my shoulders. One motivation for developing *Amrita*, in the first place, is that I believe CFD has progressed to the point where it is no longer practical for one individual to develop and maintain a competitive "code." Even if you disagree with this sentiment today, given the developments of the last decade, it is clear that CFD is becoming reliant on ever larger and larger pieces of software. Consequently, unless something is done, you will eventually be forced to concur that the cottage-industry approach – one worker, one code – is outdated.

Therefore, although *Amrita* is currently unsupported, I feel a case can be made for a third-

[57]The security conscious might be concerned with the combination of digital-signatures and dynamic-linking. Since the source for *Amrita* is available for scrutiny, there is nothing to stop a malicious programmer from circumventing the built-in security features and wreaking havoc. This problem is not unique to *Amrita* and will undoubtedly receive widespread attention with the release of *Netscape 5.0*, whose source will similarly be open to public scrutiny.

[58]A computer scientist might classify *Amrita* as a problem solving environment, or PSE for short. But in my mind, having a fluids background, PSE stands for parabolized stability equation (e.g. [17]). The basic structure of *Amrita* has been in existence since 1989, and so can legitimately be considered old hat. *Amrita* was developed to cope with day-to-day programming needs, without regard to contemporaneous research activities. By releasing the system, now that it has become too useful to keep proprietary, in some quarters I expose myself to accusations of selective referencing. However, at this late stage, I feel it would be disingenuous to provide a post-natal PSE review. Instead, *Amrita* will soon feature a database server which will allow individuals to submit pertinent reference entries, and the requisite document-preparation keywords will be added to *Amrita* to allow users to access this bibliography in a transparent fashion.

[59]If it were not for *amrpp*, this number would be tripled. However, to quote from p.7 of *The C++ programming language*[25]: "*C++* was designed to enable larger programs to be constructed so that it would not be unreasonable for a single person to cope with 25,000 lines of code." – skip two sentences – "Naturally, the difficulty of writing and maintaining a program depends on the complexity of the application and not simply the number of lines of program text, so the exact numbers used to express the preceding ideas should not be taken too literally." – skip five sentences – "However, as programs get larger, the problems associated with their development and maintenance shift from being language problems to more global problems of tools and management." *Amrita* functions as a generic code-development tool, as such you should look beyond its present incarnation to envisage what it could become with your support.

party developer chancing an arm in the hope that a sufficient number of like-minded individuals band to form a critical mass of "*Amrita++*" developers. For my part, I am happy to modify any part of *Amrita* to facilitate the process. Moreover, if you feel you have a system to rival *Amrita*, and are willing to entertain the idea of a software merger, I would be happy to consider how this might be done. If the research community continues to go it alone, the pace of software development is such that a commercial[60] product in the manner of *Mathematica* or *Matlab* will eventually hold sway, and CFD will be all the intellectually poorer for it.

Below I indicate contributions which would benefit *Amrita* as it stands today. This is by no means an exhaustive list, and so should be considered a first-cut effort:

i. **Beta-Release Testers**
The primary development platform for *Amrita* is an *SGI Indigo2* machine running *IRIX 6.2*. Therefore, although the system runs on other *UNIX* platforms, installation teething problems often give reluctant programmers a bad first-impression. For instance, when v1.38 was released, the *Solaris* version ran fine on a *Sun SparcStation 5*, but choked on a *Sun Ultra*. The problem was easily fixed by adding `-lsocket` to the list of link libraries, and was a generic *Sun Ultra* problem for programs which used `-lX11 -lXext`. Nevertheless, in the eyes of the user, *Amrita* was mistakenly condemned as non-robust. To circumvent such problems, volunteers are needed to test beta releases of *Amrita* under: *AIX*, *HPUX*, *LINUX*[61] and *OSF1*.

ii. ***Amrita* Script Writers**
Few people in the CFD community are prepared to learn a new programming language for the sake of it. Hopefully, these notes will convince some that the time taken to learn *Amrita* will be recouped many times over by the improved productivity it brings. However, others will need more convincing. Therefore volunteers are needed to first learn *Amrita* then craft CFD applications to the standard of the linear-advection study obtained by typing:

```
unix-prompt>amrcp Ch7/la.mailit
```

It is important that such example scripts be of a reasonably high standard, because they will be held up as a programming standard for others to aspire to. At the same time, if the scripts are too sophisticated they run the risk of losing their target audience.

iii. **`plugin` Developers**
To demonstrate that *Amrita* is truly a neutral, numerical test-bed, volunteers are needed to develop the equivalent engine to *Amr_sol* in other CFD areas. For suitable candidates, I am willing to help write the required system-level code.

iv. ***BCG* Contributors**
Again, to demonstrate *Amrita*'s algorithmic neutrality, volunteers are needed to both solicit and install CFD codes into `BasicCodeGenerator`.

[60] To clear one common misconception, the *.com* domain extension in *www.amrita-cfd.com* does not signify that *Amrita* is a commercial venture. At the time of registering, the *.org* extension was inappropriate for an organization consisting of just one person. Similarly, *Quirk Research* obtained by typing:

```
unix-prompt>whois amrita-cfd.com
```

resides only in the mind of one Dr. Aure Prochazka (a former GALCIT student) who was kind enough to register and host the domain name for me.

[61] *Amrita*'s *Fortran* code requires the use of the *Absoft*, or *Portland Group*, commercial compiler.

v. **Technical Writers**
Many thankless, but nonetheless important jobs need doing if *Amrita* is to become a supported system. Therefore, although the infrastructure for an on-line manual is in place:

```
unix-prompt>amrita -c
amrita>Show keywords=*
```

volunteers are needed to help produce clear and precise, keyword documentation.

vi. **GUI Developers**
Although *Amrita* was designed to be batch driven using *Amrita*, the event driven nature of its `plugin` engines make them ideal candidates for being driven by a Graphical User Interface. Volunteers are needed to explore two approaches: (i) the GUI generates *Amrita* script to drive a `plugin`; (ii) the GUI generates *ISL* to drive a plugin.

Volunteers, for any of the above, can find a sign-up sheet at *http://www.amrita-cfd.com.*

Acknowledgements

This work was supported by Los Alamos National Laboratory – subcontract 319AP0016-3L under DOE Contract W-7405-ENG-36. All brand or product names used in these notes are the trademarks or registered trademarks of their respective holders.

A Getting Started

A.1 System Requirements

Amrita runs under the *UNIX* operating systems: *HPUX, IRIX, OSF, Solaris* and *SunOS*, and is self-contained apart from its language interpreter which is written in *Perl*. Consequently *Perl* must be installed before *Amrita* can be used. Now there is a good chance that *Perl* is already up and running on your computer system (check with your system administrator) as it has become a de facto standard on *UNIX* platforms. If not, consult the USENET newsgroup *comp.lang.perl* to see how *Perl* may be obtained via anonymous *ftp*. For its part, *Amrita* is available via from:

http://www.amrita-cfd.com

and comes complete with easy to follow installation instructions, written in *HTML*, and a set of acceptance tests to verify the installation process.

Amrita does not require any third-party software (other than *Perl*) to be used profitably, but to follow the examples in these notes you will need access to *Latex, Dvips, Ghostview, Gnuplot* and *Netscape*. Again, given their popularity, these packages should already be installed on your system.

A.2 Typographic Conventions

The following typographic conventions are used in these notes:

Amrita	is used to mean the complete system known as AMRITA: Adaptive Mesh Refinement Interactive Teaching Aid.
Amrita	is used to mean *Amrita*'s language interpreter.
Slant Font	is used for the names of third-party software packages and *Amrita*'s plugin engines.
Italic Font	is used for filenames.
`Constant Width Font`	is used for miscellaneous code fragments such as *Amrita* listings and *UNIX* commands, and also for program output.
`Constant Width Font`	is used in examples to identify variables which take context-specific values. For example, `filename` would be replaced by the actual name of a file.
`Constant Width Font`	is used both for commands that you are meant to type in verbatim and also for syntactic entities in the definition of commands.

A.3 New Users

To use *Amrita* from a *csh* window, add these two lines to your *.cshrc* file:

```
setenv AMRITA installation
setenv PATH "${PATH}:$AMRITA/tools"
```

where `installation` is the full pathname to the directory where *Amrita* is installed, say:

/usr/local/AMRITA

Once this is done, and you have typed:

```
unix-prompt>source .cshrc
unix-prompt>rehash
```

you can check which *Amrita* version is available by typing:

```
unix-prompt>amrita -v
```

This should produce a message along the lines of:

```
Amrita version 1.38 (release 28-01-98)
Copyright (C) 1988, 1998 James J. Quirk

Send bugs and suggestions to help@amrita-cfd.com
```

If it does not, or you are using an alternative command shell to *csh*, such as *bash*, check with your system administrator as to how you should access *Amrita*.

A.4 Worked Examples

These notes contains a number of worked examples such as this one taken from p. 14:

```
... redirect latex output
Latex2eHead pagesize=problem-sheet
   ... typeset title
   ... typeset figures
   ... typeset footnotes
LatexTail
Latex
```

```
amrcp  Chp2/montage.1
amrita run_montage
cd     doc/montage
amrps  solvers.ps
```

In each case, a text frame provides pertinent information such as the commands needed to run the example and the output files to look out for. Thus the above would be run by typing:

```
unix-prompt>amrcp  Chp2/montage.1
unix-prompt>amrita run_montage
```

The first command is a utility which unpacks the named file from *Amrita* into your filespace, to save you having to type it in, and the second invokes the *Amrita* interpreter to execute the program stored in the file *startup_errors*. Once the script is finished, the main output file of interest (relative to the directory in which you ran the script) would be:

doc/montage/solvers.ps

which you could then view with the *PostScript* previewer of your choice (by default *amrps* invokes *GhostScript*). Because the number of files produced by a single script can be quite large, you might want to consider using a separate work directory for each section of the notes so as to facilitate subsequent file management. Here, for example, you could use a directory called *Chp2*[62].

The script examples were tested using *Amrita* 1.38 (release 28-01-98) running on an SGI Indigo2 machine (195 Mhz Mips R10000 processor) with 384 Mbytes of memory.

[62]This example comes from Chapter 2 of *An introduction to Amrita*[21].

B *BasicCodeGenerator*

Whereas classical CFD doctrine has the computational universe revolving around the flow `solver`, *Amr_sol* views solvers as disposable items. For instance, if a code such as *roe_fl* does not meet your needs – recall this was used by *my.script* on p. 5 – it can be replaced by another code more to your liking[63], without disrupting the general orchestration of the investigations in which the substituted `solver` appears. This is possible, because *Amr_sol* takes care of all the generic work needed to perform a simulation, such as: file-handling; applying physical boundary conditions; applying mesh-refinement; post-processing results etc. When supplied the discrete solution $\{\mathbf{W}^n_{i,j}\}$ for an isolated rectangular patch[64], the `solver` is expected to perform one of just two tasks: (i) return a stable time-step for the integration process; (ii) return a discrete solution $\{\mathbf{W}^{n+1}_{i,j}\}$ at the next time level[65]. If needs be, the `solver` could read the new solution in from a file, or even grab it from the internet. As far as *Amr_sol* is concerned, the precise details of the integration process rest solely between the `solver` and its maker (but see below).

Given the division of labour, a `solver` is a light-weight piece of software[66]. For instance, the source for *roe_fl* weighs in at a shade over 300 lines of *Fortran*. If you change to the directory where you ran *my.script*, you can view this source by typing[67]:

```
unix-prompt>cd code
unix-prompt>amrgi code/roe_fl.src
```

Here, folded using:

```
fold::file code/roe_fl.src (to f77::L0) -> listing
```

the source appears as[68]:

```
#include "AMR_SOL/AMRITA"
      SUBROUTINE LOG_BCG_ID
#define HARTEN_entropy_fix
#define phi 2.0 D0
      SUBROUTINE INIT_BASIC_EULER_CODE
      AMRDBL FUNCTION PATCH_DT(IM,JM,NG,DX,DY,W)
      SUBROUTINE PRIME_I(GRD,J,IM,JM,NG,DX,DT,W)
      SUBROUTINE ISWP(GRD,IM,JM,NG,DX,DT,F,W)
      SUBROUTINE PRIME_J(GRD,I,IM,JM,NG,DY,DT,W)
      SUBROUTINE JSWP(GRD,IM,JM,NG,DY,DT,G,W)
      SUBROUTINE CALCULATE_WAVE_STRENGTHS(K1,K2)
      SUBROUTINE COMPUTE_EIGENVECTORS(K1,K2)
      SUBROUTINE MODIFY_WAVE_STRENGTHS(K1,K2)
      SUBROUTINE CALCULATE_DISSIPATION(K1,K2)
```

[63] In turn, *Amrita* views *Amr_sol* as a disposable item. However, given the software-engineering involved, it would be impractical to replace *Amr_sol* with same regularity that solvers are swapped.

[64] Details of how *Amr_sol* orchestrates the integration process will be give in lecture 2.

[65] *Amr_sol* decides the size of a master time-step based on the individual, patch time-steps returned by the `solver`. Again details are given in lecture 2.

[66] This does not imply that the intellectual content of the `solver` is slight.

[67] *Amrgi* automatically folds *Fortran* on `FUNCTION` and `SUBROUTINE` program units. Therefore you can use it to view your own folded-*Fortran* without needing to introduce explicit `fold` directives.

[68] The file *roe_fl.src* is processed twice before it is actually compiled: *roe_fl.src* → *f77src/roe_fl.F* → *f77src/roe_fl.f* . The second of these pre-processing phases takes care of the `#include` and `#define` directives and expands the typedef `AMRDBL` to the appropriate *Fortran* type for a double precision variable.

B.1 Solver *Roe_fl*

If you are wondering precisely how you became the owner of the file *roe_fl.src*, the first line of *my.script* invokes the procedure `TasteOfAmrita` which in turn calls the *Amrita* library procedure `BasicCodeGenerator` (*BCG* for short) which constructed the `solver` for you. If instructed to do so, *BCG* will also document the code it produces. For instance, try running this script:

```
EulerEquations
plugin amr_sol
BasicCodeGenerator {
   solver    = Roe_fl
   scheme    = flux-limited'operator-split
   document  = yes
}
```

```
amrcp  Chp2/solver.1
amrita make_Roe_fl
cd     code
amrgi  Roe_fl.src
cd     code/doc
amrps  Roe_fl.ps
```

to create a clone of *roe_fl*, called *Roe_fl*; complete with the LaTeX document *Roe_fl.tex*, the first two pages of which are shown overleaf.

The `scheme` parameter instructs *BCG* to build a so-called *flux-limited, operator-split* code. Internally, *BCG* uses the `scheme` specification to traverse a tree of *Amrita* procedures which construct source code, subroutine by subroutine, using the document preparation skills described in §3. This provides for far greater flexibility, and ease of system maintenance, than if the code were merely regurgitated from a single, pre-prepared file. The `solver` parameter allows the user to select a name by which to refer to the resultant code and so has no bearing on the code content. Here the name was chosen to reflect the theoretical lineage of the solver (background references are given in *Roe_fl.ps*), but any filename would do[69].

To find out what other *BCG* codes can be used to integrate the `EulerEquations`, type:

```
unix-prompt>amrita -c
amrita>EulerEquations
amrita>plugin amr_sol
amrita>BasicCodeGenerator scheme=?
```

this will produce an *HTML* document which you can browse to find a naming-convention which reveals the available schemes. At the time of writing the allowed schemes are[70]:

```
def NamingConvention
   convention for euler-code is space'grid'recon'evol
   where space  = {1d:one-dimensional|2d:two-dimensional}
   where grid   = {c:cartesian|b:body-fitted:curvilinear}
   where recon  = {fo:first-order|km:kappa-muscl|cm:char-muscl}
   where recon .= {fl:flux-limited}
   where evol   = {os:operator-split|fv:finite-volume}
   exclude names 1d-b-*-*
end def
...  BCG latex documentation
```

[69] Try re-running the *make_Roe_fl* script with `solver` set to your initials (e.g. `jjq`) then check for differences between the source files *Roe_fl.src* and `jjq.src` using the *UNIX* utility, `diff`.

[70] In keeping with the rest of *Amrita*, the naming-convention for `euler-code` is programmable and so can be widened to accommodate user-supplied code. Thus `recon` could be extended to include `eno`, and `evol` could be extended to include `runge-kutta`. Because the naming-convention fixes the `scheme` names used to traverse *BCG*'s code-generating tree, and not the hard details of the codes themselves, its extension is trivial.

Amrita

BasicCodeGenerator

Made: *Roe_fl*

For : James J. Quirk (aka jjq)

On : Mon Feb 2 15:44:34 PST 1998

Correct Usage:

```
EulerEquations
plugin amr_sol
logfile logs/Roe_fl
solver code/Roe_fl
```

Preamble

This document[†] dissects the source code for the *Amr_sol* compatible solver:

Roe_fl

which was generated using:

```
BasicCodeGenerator {
   solver    = Roe_fl
   scheme    = flux-limited'operator-split
   document  = yes
}
```

It is assumed you have some familiarity with the operation of `BasicCodeGenerator` and understand how flow solvers are bound to *Amr_sol*. If this is not the case, you should read Chapters 6 and 7 of *An introduction to Amrita* before proceeding.

[†] If you spot an error in this document, or the associated source code *Roe_fl.src*, please take the time to file a bug report so that it can be corrected.

Figure 14: Page one of the document produced by *BCG* for solver *Roe_fl*.

Contents

Figure 15: Page two of the document produced by *BCG* for solver *Roe_fl*.

With this knowledge you can then choose a scheme (or schemes) to suit your preferences. For example, the *run_montage* script (see p. 14) essentially runs:

```
amrcp  Chp2/solver.2
amrita make_km_solvers
```

```
EulerEquations
plugin amr_sol
foreach flux (godunov, efm, hlle, ausm)
   BasicCodeGenerator {
      solver    = $flux`_km
      flux      = bcg/$flux
      scheme    = kappa-muscl'operator-split
   }
end foreach
```

to produce a collection of MUSCL-based solvers[71].

`BasicCodeGenerator` is designed to produce flow codes to meet the general needs of the *Amrita* community. As such, it operates at several levels, and leaves users to find their own level of programming comfort. At the lowest level, you can use *BCG* blindly, safe in the knowledge that it will craft a serviceable code. In this lecture the focus has been on the `EulerEquations`, but it is only a small leap of faith to run this script:

```
amrcp  Chp2/solver.3
amrita make_swe_solver
```

```
ShallowWaterEquations
plugin amr_sol
BasicCodeGenerator {
   solver    = my.swe.code
}
```

to acquire a code with which to integrate the `ShallowWaterEquations`, and so on for other sets of equations[72]. Then, if you are curious as to the inner workings of a particular solver, *BCG* will produce a code dissection to slake your curiosity; if you have not already done so, now would be a good time to view the *PostScript* file *Roe_fl.ps* (all 30 plus pages of it!).

At a more active level, the output from *BCG* can be used as boiler-plate for creating a customized `solver`. But as its name suggests, `BasicCodeGenerator` has no pretensions to representing the last word in `solver` sophistication. Therefore, if you are a CFD die hard, you may well feel you can do a better construction job than *BCG*. If this is the case, you can still use the output from *BCG* as a template with which to build your own *Amr_sol*-compliant code from the ground up. In particular, a document such as *Roe_fl*.ps will explain how to define the required `solver` bindings[73].

B.2 `def Solver`

Before *Amr_sol* can use a `solver`, it must have some knowledge of the code's internal layout. Specifically: which routine should it call to initialise the `solver`; which routine should it call to compute the stable time-step for an isolated mesh patch; which subroutine calls need it make to integrate the solution held by an isolated mesh patch. Therefore *Amrita* provides a `def`

[71] If you are not familiar with MUSCL schemes, activate the `document` parameter.

[72] For maximum flexibility, each `EquationSet` employs its own *BCG* naming-convention. In the case of the `LinearAdvectionEquation`, which provides the clearest setting in which to learn to build a `solver`, the naming-convention even allows the user to specify a choice of programming language (*f77* or *cc*).

[73] The complexity of the bindings is independent of the mathematical complexity of the target `EquationSet`. Thus lessons learned using the `LinearAdvectionEquation` transfer directly to codes written for full systems of partial-differential equations.

`Solver` block to allow the `solver` author to furnish *Amr_sol* with the information it needs, and a *BCG* document such as *Roe_fl.ps* provides a ready made example of its use, as do the documents for the `kappa-muscl` schemes.

The `Solver` block for *Roe_fl*, taken from *Roe_fl.mk* is:

```
def Solver
   gridreq    : cartesian (NG>=2)
   symmetry   : slab | cylindrical
   startup    : INIT_BASIC_EULER_CODE
   timestep   : PATCH_DT(IM,JM,NG,DX,DY,W)
   step Isweep (aka Li) : ISWP(GRD,IM,JM,NG,DX,DT,F,W)
   step Jsweep (aka Lj) : JSWP(GRD,IM,JM,NG,DY,DT,G,W)
   integration: Li*Lj
end def
```

and is dissected in §1.4 of *Roe_fl.ps*.

If you compile the code manually:

```
unix-prompt> amrita Roe_fl.mk
```

you will see a message:

```
creating Amrita bindings
```

The `CompileSolver` call at the end of *Roe_fl.mk* uses the `def Solver` information to craft a number of system-level, binding routines which allow *Amr_sol* to call *Roe_fl* at run-time. The output from this construction phase, *f77src/Roe_fl.F*:

```
#include "AMR_SOL/AMRITA"
      SUBROUTINE LOG_BCG_ID
#define HARTEN_entropy_fix
#define phi 2.0 D0
      SUBROUTINE INIT_BASIC_EULER_CODE
      AMRDBL FUNCTION PATCH_DT(IM,JM,NG,DX,DY,W)
      SUBROUTINE PRIME_I(GRD,J,IM,JM,NG,DX,DT,W)
      SUBROUTINE ISWP(GRD,IM,JM,NG,DX,DT,F,W)
      SUBROUTINE PRIME_J(GRD,I,IM,JM,NG,DY,DT,W)
      SUBROUTINE JSWP(GRD,IM,JM,NG,DY,DT,G,W)
      SUBROUTINE CALCULATE_WAVE_STRENGTHS(K1,K2)
      SUBROUTINE COMPUTE_EIGENVECTORS(K1,K2)
      SUBROUTINE MODIFY_WAVE_STRENGTHS(K1,K2)
      SUBROUTINE CALCULATE_DISSIPATION(K1,K2)
*********************************************************************
*  THE FOLLOWING AMR_SOL BINDING ROUTINE(S) WERE GENERATED BY AMRITA  *
*  USER: James J. Quirk (aka jjq)                                     *
*  DATE: Mon Feb  2 15:44:37 PST 1998                                 *
*********************************************************************
#include "AMR_SOL/AMRITA"
#include "AMR_SOL/BINDINGS.H"
      SUBROUTINE AMR_SOL::VALIDATE_SOLVER(CHAN)
      AMRDBL FUNCTION AMR_SOL::PATCH_DT(L,GRD)
      SUBROUTINE AMR_SOL::INTEGRATE_GRID(L,Nt)
      SUBROUTINE AMR_SOL::GRABINFO(CHAN)
*********************************************************************
*   THE ABOVE AMR_SOL BINDING ROUTINE(S) WERE GENERATED BY AMRITA    *
*********************************************************************
```

is then pre-processed by *amrpp* to produce a file *f77src/Roe_fl.f* which can be compiled by a standard *Fortran* compiler[74].

Reluctant programmers need not be phased by the above machinations, because all the hard work takes place automatically. For example, some of you may have noticed that the integration sequence:

```
integration: Li*Lj
```

is not formally second-order accurate. Fortunately, all that need be done to get pukka Strang-splitting[23], is substitute the line:

```
integration: 1/2Li*Lj*1/2Li
```

then recompile:

```
unix-prompt> amrita Roe_fl.mk
```

and the innards of the binding-routine `AMR_SOL::INTEGRATE_GRID` automatically adjust to the revised `integration` sequence[75].

Practical experience shows that Strang-splitting is a mathematical nicety *for the shock-diffraction problem run by my.script*[76] and so an Engineer might not be happy putting up with the extra cost of the integration for no tangible reward. For a uniform grid, the sequence:

```
integration: Li*Lj*Lj*Li
```

combines the economy of `Li*Lj` with the accuracy of `1/2Li*Lj*1/2Li`. However, when used with *Amr_sol*, it has the side-effect of halving the number of grid adaptions which increases the risk of introducing $O(1)$ errors, should a shock escape across a fine-coarse grid boundary. Such subtleties help explain why *Amrita* is endowed with a programmable interface rather than a GUI: *Amrita* allows a wider range of tastes to be catered for in a seamless fashion.

Here, the expert programmer need not feel fettered by *BCG*'s way of doing things. When all is said and done, with the current *Amrita* release an *Amr_sol* `solver` is a shared-object file which provides four system-level calls [77]:

i. `AMR_SOL::VALIDATE_SOLVER` allows *Amr_sol* to check that the `solver` is compatible with the current `EquationSet`. This prevents the sorts of chaos which would ensue from integrating the `EulerEquations` with a `solver` intended for a different `EquationSet`, say the `LinearAdvectionEquation`.

ii. `AMR_SOL::PATCH_DT` returns the stable time step for an isolated mesh patch.

iii. `AMR_SOL::INTEGRATE_GRID` updates the flow solution for a collection of mesh patches. Lecture 2 will describe some of the subtleties which dictate the internal machinations of this routine.

iv. `AMR_SOL::GETINFO` services *Amrita*'s `getinfo` command and is not mandatory.

[74]This statement is not strictly true, since two industry-standard extensions are needed: (i) ability to cope with long variable names; (ii) ability to copy with `POINTER` variables. But the statement stands in that the *Fortran* compilers provided by all major workstation vendors have the required extensions. Unfortunately, at the time of writing, the *Fortran* supplied with *Linux* does not cope with `POINTER` variables, and so you must purchase a commercial compiler which does (e.g. Portland Group *pgf77*).

[75]An explicit `integration` sequence can be specified when *BCG* is called, and so there is no need to feel inconvenienced by the default `Li*Lj`, should it not be to your liking.

[76]Please read the *Dumas* quote on p. 15.

[77]Future *Amrita* releases are not obliged to maintain the low-level binding structure. Therefore, unless you have good reason for doing so, do not circumvent the `def Solver` mechanism.

C *Amrita mailit* files

The primary purpose of *Amrita mailit* files is to allow *Amrita* scripts to be distributed via e-mail, hence the name "mailit". For instance, suppose person *A* sets out to develop an *Amrita* application for drawing fractals; with the express intention of showing that, despite its heritage, *Amrita* is not restricted to gas dynamics. Well, once the necessary scripts are written and tested, *A* need only invoke *Amrita*, thus:

```
unix-prompt> amrita -mailit fractal
```

to produce a file *fractal.mailit* which can then be e-mailed directly to person *B*; complete with a *PGP*[10] digital signature, courtesy of *amrmailit*, to authenticate the origin of the message.

All *B* need do, upon receipt of the *mailit*:

```
amrcp  vki/mailit.1
amrita fractal.mailit
amrps  ps/fractal.ps
```

```
-----BEGIN PGP SIGNED MESSAGE-----

AmritaMailit::fractal {
   origin {
      Amrita v1.38 R07-01-98
      user   James J. Quirk (aka jjq)
      date   Thu Jan  8 14:17:00 PST 1998
   }
   resources {
      disk 131 KBytes
      cpu  34 secs
   }
   operation {
      amrita -mailit fractal
   }
   script {
      M'XL("(!1M30''V9R86-T86PN=&$R'.U7;5/B2!#>S_R*KBNK%C1"$GEQ*>,6
      M(KA>*5J(MYXLM34F0T233$S""N?YWZ\["9'$U]T/WWMU=>D"9NAYNJ>G9Z:[
      M9^PQ/6#6FY])LBS7ZU60MB8K8:O4:V$;4J.F0EROUFIJE3X0*SNJ7'T#\D^U
      M*J:I'S'/3;P[>W0=Y]P[XM%1E,;+X]%28-G^1XA-'^&R8');EGE3Z$:GH8N_
      MPIN#SP/-9H[!K1M/!))CLYFFUFH%UYK>F0XPV_OL"ZMPP8-S=LO!-__BVN6%
      MQ8.'>Y+-/'1IBBP)S^1.P')3.)J%ZGR=N;S0"C_P=<]T'Q'.JA3!V+0XN'YE
      M'%E1=OT"')RP0,].V'VW8":TFBS-A:;LRM*$P[>30%.JTB>")<0BL/;I4\OV
      MS("]]2%>%L3K(O0YSG=B^H'IW'+-J\6320@D].!U4P_F0)ADS35'D%T737ENC
      M6R045>JAZ+M(5'Y9]-!CC[$XR2$D(1K-"N]"47(0\S0#="A]&C7ZTDA"!YF\
      M+9=W25HF8>I;X;85DKM>F0B;5[Z8+L==XV2,^.%G[!OW7]UI-);W7Z98H-1V
      MY$9^_W\%N9[04R?HJ1"?_=4]T&!;+:L17Y[F*^5ZQ#=F^$.4F^]'Q%<B)IT\
      ME"HQ<R=B>I#44"7F,_V,A64TFRR\L_!$L6?J0CX1/G?'91Y&B3'HPG8M/L/+
      MPIP>WWJ#C^%0V,QTEP''P0B#H63#F85VY247PYP^03U%4J1'BA_5VO[ZV%9V
      MC#LE34'_8D[&T#0*'I^Y7']\\/%JP8&8.0;SYFWAL"8%/G\5H\C6U\:1>C?&
      MG'!L:05-U!QXS!46!L(4ZN*[4)WO0GW\-BKAB<6>A3^TG0MA36D979-;1HS'
      M'1S37_CX-0ZN>QB>-F:"HM/&7(0>3BC-^E4/S"]H'-C<GX#'QZ;#;<PC8#-]
      M0OTHCL?SGR*F93"7;%A9S&(.)]GA^=OP5HSPE&[0[_J6+Q?87\[?1ONX9[+%
      M[#XTFWCB*^IVO,IF4\?=#8:CA!\L=NO#4'CQMYCA5X01"0Y'^S(\'<G/7P'.
      MEL'MU1QM8KN-[9Z\#I^E]6ZI=S'\+H9_W<VF\T7<KSDYGL)F]WR<V%)#0*4I
      M4M)9"S>O_F:%PKG%TIM4<83W<Y'5P+2I>G0Z4K:/0LW/\.'D8F^G+LN;69=*
      MBKI+W_#LV'O*_E+)K6<:8AI8N''?=46*(I2\0AE1H""3*/(5_O?I3_E_O7[Y
      ML3GP6_F_XLJ+_%^M-ZC^X\NJFN?_7T%A_L^4_7$)0.<K.F0:K!X!T1"]!.(A
      M>0]$</0CI>S?-&AY'IH?XLO'A][1\>QW;3=T2GBA8&][']Y8W)740QU0'/H4Y
      MGAX",>;S>%6TQ^!]Q/H"YWX%XGLZP'N$-!K8K620Y6Z,W5RT43.8F/A:$!YF
      M/P>P2][AOL\-N)E3HG==''1QC:G.TZ+)B5;SC.%Q8F)E0;J""18R\>*,M"PI
      M)_]126'LYX^Q7CEK:RIQMT[[AP<GT+WLM0?'9STX;?4..R<'_;-!\;S5+[V$
      ML7Y1;FZNC;7/K^!::F?9Q(T!]*3>:>LJ7:.0'_>'E"EI5>WH9+1/ST^NR(BB
      M6I*HV2FE<=<1KBA+<GJ0%PVL2C=\/D'[K#<X[EUV,H\]U"+!]>;U5CL[<-PM
      M%EL'%\7K4OFH4U9+Y;-^N=BC/IE>*L'00Z>7%4):>1'UOP3H=P:7_;6I3N_P
      MN)ME]DC%EI)B'YT-SB!^$+ZF$Q6F4E#Q8LJ^H?&Y'>,<AJ5F2.D;O$00+M.C
      M\>!I;!LF9B\?Y9FU$.</CA_V%08Z(GJF4_V/55K,Q?-^8W$;]7'=')MZ:C!9
      M_?W!:?I$J<47IJ;'4WB()YV2YM/5<+0U$YC6_\3.'#O/*]3'H3(*46.GF+F^
      MS>8&!B=I0\*0U.?%Z^&H)!W;89LX29&]\_'M0XKD]:%Y9BB1)Q/=90&[$"<1
      M*F;#9WI47J23?XB0%51"(\/Z9WD^Q_!QCV%0"8Z#9KH0^97[_V_GGYQRRBFG
      >G'+**:><<LHIIYQRRBFGG'+*Z>?1/_VWG?\'*'''
      '
   }
}

-----BEGIN PGP SIGNATURE-----
Version: 2.6.2

iQCVAwUBNLVQ4rmgfLsjP/xJAQEOWwP/XalBM9GW8C4QNRQZAKfx3+7FbHNcHaaP
njJVfQn0HqS2vRnN6T9xHTUzPVWknS8F3+suDVfnxwmeAqVGeKfW7TGvy1emwDVa
8gI0mPK1CQ7703RWK5J/G1rqajin8Z5Yf0v48J7EOA9IbIZHun2qfu5T7dYZdJz4
xQ6FFkY+4kA=
=dH4+
-----END PGP SIGNATURE-----
```

is type:

```
unix-prompt> amrita fractal.mailit
```

and *Amrita* will automatically unpack the archived script, check its authenticity, and invoke the interpreter with the exact same options used by person *A*. Thus *B* ends up with the same set-up as *A*, regardless of the complexity of the script involved[78].

[78]The system is not completely foolproof, that is determined users can break it!

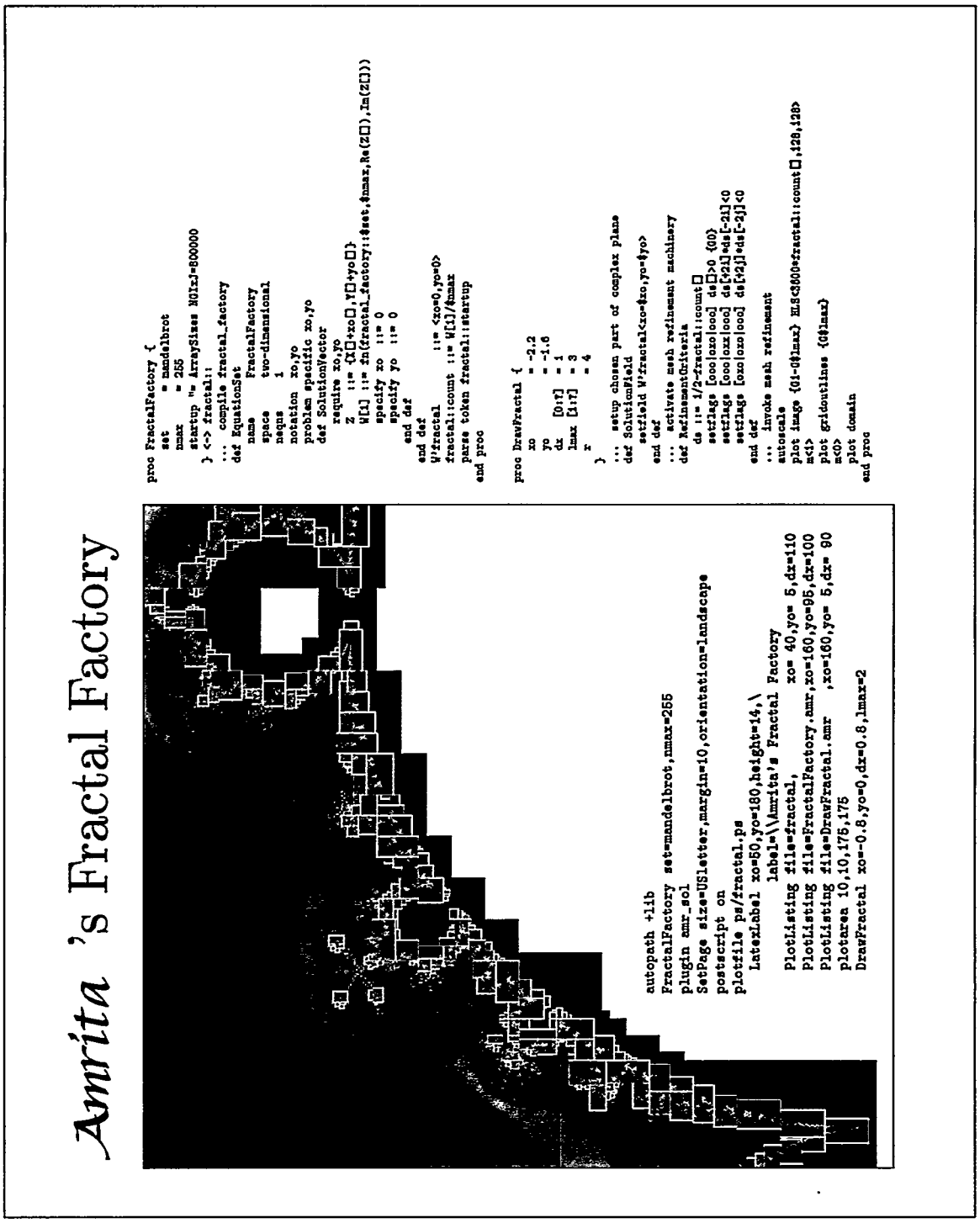


Figure 16: Output produced by *fractal.mailit*. The image is best viewed on a 24-bit colour, computer screen so that the full affect of the *HLS* colour shading[9] is seen near the fractal boundary. A *PostScript* previewer such as *Ghostview* will also allow you to zoom in on the boundary details. This *Amrita* example was inspired by the unwieldy source needed to produce Figure 5.9 of *Using MPI*[12]. Here – the three listings, barring the hidden program folds, constitute the entire *Amrita* script needed to produce the page shown. The largest of the hidden folds uses the dynamic-linking techniques discussed in §D: it weighs in at 4 *Amrita* script lines plus 24 lines of *Fortran*.

C.1 Digital Signatures

Garfinkel[10] provides a thorough introduction to the general need for tamper-proof, digital signatures and explains the basics of public-key encryption; the technology upon which DSA (digital signature algorithms) are based. This script suffices to illustrate the basic problem:

```
set ob          "= obfuscated
set obfuscated #= "\162\155"
set fu          "= Obfu
set scated      #= "scated"."\052"
set Obfuscated #= " \052\056"
clarify on
execute $$ob.$$fu.$scated
```

```
amrcp  Chp3/obfus.1
cd     safeguard
amrita run_obfuscate
```

It outputs, courtesy of *Amrita*'s built-in `clarify` command:

```
<<
SCRIPT: execute $$ob.$$fu.$scated
   execute $$ob.$$fu.{$scated}
   execute $$ob.$$fu.scated*
   execute $$ob.${$fu}scated*
   execute $$ob.$Obfuscated*
   execute $$ob.{$Obfuscated}*
   execute $$ob. *.*
   execute ${$ob} *.*
   execute $obfuscated *.*
   execute {$obfuscated} *.*
   execute rm *.*
>>
```

and illustrates one good reason why you should not use (or write!) obfuscated scripts: you can never be sure (or recall!) precisely what they do.

Programming languages which employ string interpolations, such as *Perl* and *Amrita*, are inevitably susceptible to Trojan Horse attacks through variable tainting. Here, the `clarify` command reveals the intricate expansion process which leads to the devastating bottom-line[79]. It also indicates the extent to which *Amrita* is happy to cater for reluctant-programmers[80]. If you fall in this category, do not be put off *Amrita* by the above example: you run the same risk of such an attack, every time you run a program executable you did not write yourself[81].

There are two basic methods for safeguarding against malicious attacks. The first is the so-called "padded cell" approach, favoured by *Java*[26] and *TCL*[27], where suspect programs (e.g. ones executed straight off the web) are run with draconian access restrictions, to prevent them running amok. The second approach, as used by *Amrita*, is based on trust. When *Amrita* receives a *mailit* from *John Doe*: it runs it, if and only if, you trust *John Doe* not to send out a Trojan Horse[82] and the attached digital signature verifies that the *mailit* was not tampered with in transit. Neither security approach is entirely satisfactory, but the "padded cell" style, because

[79] In fact, as a precaution against novice programmers shooting themselves in the foot, `execute` is disabled when `clarify` is activated: `execute` launches a *Bourne* shell, and so *csh* users who have sensibly aliased `rm` to `rm -i` would still have their files deleted by this obfuscated script, if it were not for `clarify`.

[80] Given the `clarify` command, there is no excuse for being unable to get to grips with *Amrita* string expansions, other than plain sloth.

[81] Try typing: *unix-prompt>**man unlink***.

[82] That is, *John Doe*'s *PGP* public-key is on your *PGP* key-ring of trusted associates.

it hampers both good and bad scripts alike, appears to be losing favour[83]. Ultimately, however, the security-buck stops on the desk of the user.

For those who are interested, *fractal.mailit* was sent out by[84]:

```
Amrita Mailit-Master <help@amrita-cfd.com>
```

using the public key:

```
-----BEGIN PGP PUBLIC KEY BLOCK-----
Version: 2.6.2

mQCNAzP156YAAAEEAMBGpiCv7LkkP8WDIO6HuucKoyQuFtaTRdwOSpRYQtD4u4k6
ftPjDJ4IfZ+s4gyD9VPiBsCT5qoW3V32MBEVCS/x04zzo9GC79Ofh1uWmiDA+XMW
jf/IEDIxo42g1XAvZHHok3Pyz4A5YEsJ31vmnRtpU/L8H2UirrmgfLsjP/xJAAUR
tCpBbXJpdGEgTWFpbGl0LUlhc3RlciA8aGVscEBhbXJpdGEtY2ZkLmNvbT6JAJUD
BRAz92NpuaB8uyM//EkBAbS1A/oDZKYC3Xw6X4hrWQFdh7sc45jQsgU084QX9Vpb
Kx9uyJ/CKN19pDF199s9pMFbvu3xacE6/V80aeS3AGC88W10JQdw4UCp+l2V7DYO
wvglSX7H9cr+WO+eB5c98WKoF3K++0Hz2ORDRLR1MY8aZFE/0pmb+hMnCf+bebME
fwq//Q==
=SVU7
-----END PGP PUBLIC KEY BLOCK-----
```

which has the *MD5* fingerprint:

```
D4 33 F8 7F 1F 80 C7 3A  00 13 FF A5 C9 FA 70 34
```

C.2 Bug Reports and System Updates

At another level, *Amrita mailits* were developed to facilitate the filing of bug-reports. Someone struggling to get to grips with *Amrita* string expansions might be convinced that this script:

```
set x #= 10
set y #= -5
set z #= $x-$y
```

```
amrcp  Chp3/error.2a
amrita run_careless
```

which outputs:

```
Error at line 3 of file run_careless:
error evaluating expression '10--5'!

Line 3 is:
set z #= 10--5

error near:
10--5
```

unearths a bug in the interpreter. Instead of bemoaning the fact, the individual can fire off a *mailit* to a more expert programmer, with a realistic hope that the problem will be tracked down. At least more so than had they merely submitted a vague written report[85]. In this instance, the problem lies with the user's script. Because *Amrita* fully expands a script-line before attempting to execute it, the user should have written:

```
set x #= 10
set y #= -5
set z #= $x-($y)
```

```
amrcp  Chp3/error.2b
amrita run_correct_way
```

[83] *Netscape* belatedly offers a *JavaScript* signing tool called "zigbert"[15] which produces *JAR* files that work along the same lines as *Amrita mailit* files.

[84] The library routine `GetPGPkeys` can be used to obtain a copy of *Amrita*'s public key.

[85] Nevertheless, you should *always* take the time to distill a buggy script down to the minimum which characterizes the problem before submitting the *mailit*. The smaller the *mailit*, the quicker the bug-fixer can get to grips with it, the quicker you get your problem solved.

At the system level, *amrmake* automates the process of producing *mailit* files which upgrade one version release of *Amrita* to another. For example, following a bug-fix, a "software tzar" runs:

unix-prompt>**amrmake tarfile**

to generate the definitive *Amrita* installation kit[86]:

AMRITAv1.38_R24-01-98.tar.gz

plus a series of *mailit* files which upgrade previous releases to the new release, say:

AMRITAv1.38_R17-11-97_to_R24-01-98.mailit

The upgrade *mailit* can then be shipped out to *Amrita*'s existing user-base, for recipients to type:

unix-prompt>**amrmake AMRITAv1.38_R17-11-97_to_R24-01-98.mailit**

to make the appropriate upgrade. As the whole process is completely automatic, the upgrade can be performed by individuals who are blissfully ignorant of the workings of *patch* and other *UNIX* system utilities.

For those who are interested, here is the first part of the upgrade file:

```
AmritaPatch::header {
   upgrade     AMRITAv1.38_R17-11-97_to_R24-01-98
   instigator James J. Quirk (aka jjq)
   date        Sat Jan 24 12:27:44 GMT 1998
}
AmritaPatch::delete { Amrita/keywords/gl/pasteimage.pl
   file Amrita/keywords/gl/pasteimage.pl
}
AmritaPatch::delete {
   file examples/Chp2/Schardin/LatexSchardin.amr
}
AmritaPatch::delete {
   file examples/Chp2/Schardin/SchardinInfo
}
AmritaPatch::delete {
   file examples/Chp2/Schardin/schardin.ps.gz
}
AmritaPatch::edit {
   file include/cc/AMRITA/isl.h 58759
   4a5,9
   > #include <stdio.h>
   > #include <string.h>
   > #include "AMRITA/typedef.h"
   > #include "AMRITA/errors.h"
   >
   22a28
   > AMRVOID   ISL::unplug              (AMRSTR *plugin);
}
AmritaPatch::add {
   file stdlib/system/GetPGPkeys.amr 14312
   proc GetPGPkeys
      ...  amrita:pgp::id
      ...  amrita:pgp::fingerprint
      ...  amrita:pgp::publickey
      foreach token (id,fingerprint,publickey)
         chop amrita:pgp::$token
etc ..
```

[86]This is far from being as simple as running *tar* on *Amrita*'s root directory, which is why the task is automated.

D Dynamic-Linking

Dynamic-linking enables a code to perform open-heart surgery on itself – as it runs – so as to fix a bug or obtain functionality that was not available when the code was constructed. Under *UNIX*[87], the dynamic linker (i.e. the surgeon) is controlled using four routines: `dlopen`, `dlsym`, `dlerror` and `dlclose`[88]. *Amrita* builds on these routine to provide users with a painless means of exploiting the programming-power of dynamic-linking.

D.1 Hello, World!

On my machine these two *Fortran* subroutines:

```
C
C     OUTPUT CLASSIC MESSAGE
C
      SUBROUTINE MSG1
      WRITE(6,*) 'HELLO,  WORLD!'
      RETURN
      END
C
C     OUTPUT AMRITA'S MESSAGE
C
      SUBROUTINE MSG2
      WRITE(6,*) 'HELLO, AMRITA!'
      RETURN
      END
```

```
amrcp  vki/dl.1f
cd     code
amrf77 greetings.f
```

can be compiled to form a so-called "dynamic shared object:"

AMRSO/serial/IRIX/64/greetings.so

On your machine, as explained on p. 25, the filepath between *AMRSO* and *greetings.so* might reflect a different architecture, but the net result is the same: the file *greetings.so* is ready to be sucked into any executing program which cares to make use of it.

For instance, change up out of the *code* directory[89] and run this three line *Amrita* script:

```
plugin foo
call code/greetings::msg1
call code/greetings::msg2
```

```
amrcp  vki/run_dl.1f
amrita print_greetings
```

to output:

```
 HELLO,  WORLD!
 HELLO, AMRITA!
```

The *Amrita* keyword `call` has the syntax:

`call` ***[<path>/]<package>[:<language>]::<procedure>***

The optional ***<path>*** locates a directory containing a shared-object ***<package>*** which contains a ***<procedure>*** to `call`, in an optionally declared ***<language>***[90] (by default *f77* is assumed).

[87] At least under the *UNIX* variants supplied by the major workstation vendors: dynamic-linking (DL) is not available under *UNICOS*! DL is now so pervasive in the design of operating-systems that you should not, not use DL out of some misplaced fear that it is non-standard. If *UNICOS* survives, a future release *will* support DL.

[88] *IRIX* provides a solid introduction to the use of dynamic shared objects: *irix-prompt>***man dso**.

[89] As a matter of good file-management, unless used for compilation purposes, I recommend you keep *Amrita* scripts separate from both *Fortran* and *C* source-code.

[90] Currently, ***<language>*** is restricted to *f77* or *cc*, but the generalization to other languages is straightforward.

Therefore, if you prefer programming in *C*, there is nothing to stop you from compiling:

```
#include <stdio.h>
/*
   output classic message
*/
void msg1(void) {
   printf("Hello,  World!\n");
}
/*
   output Amrita's message
*/
void msg2(void) {
   printf("Hello, Amrita!\n");
}
```

```
amrcp vki/dl.1c
cd    code
amrcc greetings.c
```

and running:

```
plugin foo
call code/greetings:cc::msg1
call code/greetings:cc::msg2
```

```
amrcp  vki/run_dl.1c
amrita print_greetings
```

to output:

```
Hello,  World!
Hello, Amrita!
```

The *Amrita* expert could even run:

```
...  make hybrid-package
plugin foo
call code/greetings:cc::msg1
call code/greetings:f77::msg2
```

```
amrcp  vki/run_dl.1
amrita print_greetings
```

to output[91]:

```
Hello,  World!
 HELLO, AMRITA!
```

D.2 Compiler Options

Amrita is designed to work transparently across multiple platforms. Therefore, unless you have good reason for doing so[92], the only compile options you should employ with *amrf77* and *amrcc* are: `-O` or `-G`, and `-serial` or `-mpi`. The first pair of options toggle between production mode (i.e. best possible optimization) and debug mode (see next section). The second pair toggle between the serial and parallel versions of *Amrita*. This minimalist approach works[93], because tools such as *amrf77* and *amrcc* tune themselves to your local platform by in-lining code from the directory structure *$AMRITA/SYSTEM*[94]. For instance, on my machine the bottom-line *Fortran* compilation uses *$AMRITA/SYSTEM/IRIX/64/amrf77*.

[91]The *Fortran* `WRITE` introduces a leading space, hence the mismatch in the justification of the messages.

[92]There is nothing to stop you from building shared-objects independently of *Amrita* using whatever tools or switches your system provides.

[93]Many of the reluctant programmer's woes stem from using computer systems which are overly flexible. Here, the generic options `-O` and `-G` are automatically mapped to the specific options required by the hardware.

[94]If present, the directory structure *$AMRITA_HOME/SYSTEM* takes precedence over *$AMRITA/SYSTEM*. Therefore, if you find yourself swimming against *Amrita*'s tide, change the flow direction!

D.3 Debugging

Although *Amrita* is designed to insulate users from the harsher aspects of *UNIX*, it does not prevent you from working with the operating-system when the need arises. For instance, this *Fortran* subroutine contains a deliberate floating-point error:

```
C
C      GENERATE FLOATING POINT ERROR
C
       SUBROUTINE FPE
       X = 1.0
       Y = 0.0
       Z = X/Y
       WRITE(6,*) 'Z = ',Z
       RETURN
       END
```

```
amrcp  vki/dl.2f
cd     code
amrf77 -G example.f
```

which causes this *Amrita* script:

```
plugin foo
call code/example::fpe
```

```
amrcp  vki/run_dl.2f
amrita -debug run_fpe
```

to output[95]:

```
output:plugin::foo {
 Z = Infinity
}
```

The **-debug** option instructs *Amrita* to save a copy of the *ISL* sent to *Foo* in a file *debug.isl*:

```
... fold::isl  copyright message
... amrita:plugin::foo
... fold::isl  foo defaults
... amrita::call
... amrita:unplug::foo
```

This file can then be fed directly into a symbolic debugger[96]:

```
unix-prompt>amrdbx amrita:plugin::foo
dbx version 7.0 May 28 1996 00:47:28
Executable $AMRITA/bin/serial/IRIX/64/G/foo/amrita:plugin::foo
(dbx)run
Process 16531 (amrita:plugin::foo) started
amrita:plugin::foo {
   str ok:23:3:62
}
Process 16531 (amrita:plugin::foo) stopped on signal SIGFPE:
(handler __catch) at [FPE:7 +0x8,0xffffe0834]    7  Z = X/Y
(dbx)quit
```

to find the exact location of the error[97], thereby eliminating the hit-or-miss approach of debugging with print statements.

[95] On your system, `Infinity` may appear differently e.g. `Inf`, `NaN`, `*****` etc.

[96] *Amrdbx* is a *Perl* wrapper to a standard symbolic-debugger such as *dbx* or *gdb*. Under *IRIX*, *amrdbx* sets the environment variable `TRAP_FPE` to `DIVZERO=TRACE(5); OVERFL=TRACE(5), ABORT(100); DIVZERO=ABORT`, to ensure floating point errors are trapped.

[97] If *amrdbx* complains that it cannot find *amrita:plugin::foo*, get your system administrator to run: *unix-prompt>***amrmake -G amrita**, then try again.

D.4 *ISL* Call-Back Routines

To gain an appreciation of the role of *Amrita*'s Intermediate Scripting Language, it is instructive to follow the sequence of events activated by the `call` command in the last script example.

Amrita parses the script line[98]:

```
code/example::fpe
```

to produce the *ISL*:

```
amrita::call {
   file ~CWD/code/~AMRSO/example.so
   str fpe_
}
```

which is then fired down the pipe-line (see Figure 7) to `plugin` *Foo*.

Internally, the `plugin` relies on an *ISL* parser to decode the incoming stream of instructions[99]. This parser works much like a GUI call-back interface in that it maintains a list of event-activated routines. However, instead of keyboard presses and mouse clicks, the events which invoke the call-back routines are *ISL* tag-names. For instance, this *C* code[100] primes the parser to recognize a series of tags in the keyspace `amrita::`, one of which is `call`:

```
#include "AMRITA/isl.h"
AMRVOID AMRITA::keywords(AMRVOID) {
   AMRVOID  CC:AMRITA::call();
   AMRVOID  CC:AMRITA::command();
   AMRVOID  CC:AMRITA::export_expr();
   AMRVOID  CC:AMRITA::export_path();
   AMRVOID  CC:AMRITA::export_token();
   AMRVOID  CC:AMRITA::import_token();
   AMRVOID  CC:AMRITA::replace();
   AMRVOID  CC:AMRITA::plugin();
   AMRVOID F77:AMRITA::PRINTFILE();
   AMRVOID F77:AMRITA::LOGFILE();
   ISL::add_keyword("amrita::call",          CC:AMRITA::call);
   ISL::add_keyword("amrita::command",       CC:AMRITA::command);
   ISL::add_keyword("amrita:export::expr",   CC:AMRITA::export_expr);
   ISL::add_keyword("amrita:export::path",   CC:AMRITA::export_path);
   ISL::add_keyword("amrita:export::token", CC:AMRITA::export_token);
   ISL::add_keyword("amrita:import::token", CC:AMRITA::import_token);
   ISL::add_keyword("amrita::logfile",      F77:AMRITA::LOGFILE);
   ISL::add_keyword("amrita::plugin",        CC:AMRITA::plugin);
   ISL::add_keyword("amrita::printfile",    F77:AMRITA::PRINTFILE);
   ISL::add_keyword("amrita::replace",       CC:AMRITA::replace);
}
```

[98]The location of the *Perl* script responsible for parsing `amrita::call`, relative to *Amrita*'s root directory, is *Amrita/keywords/basic/call.pl*. This you could have found by typing:

```
amrita>location <keyword> amrita::call wrt $amrita::AMRITA -> src
amrita>echo $src
```

[99]The source for this parser is located in the directory tree *$AMRITA/plugin/amrita/src*.

[100]This source is pre-processed by *amrpp* before it is compiled by an *ANSI* compiler. This pre-processing phase mangles the namespaces `ISL::` and `AMRITA::` down to a name, `AMRxxx_`, to reduce the possibility of name conflicts with routines users introduce via dynamic-linking. The qualifiers `CC:` and `F77:` allow the pre-processor to take care of the calling conventions between languages. The `call` command does a similar trick, which explains why `fpe` in the *Amrita* script reads `fpe_` in the *ISL*.

When the *ISL* parser reads the tag `amrita::call`, it skips over the opening brace '{' and the newline character, then invokes `CC:AMRITA::call` to parse the body of the command. The *C* procedure is short enough to be listed here:

```
#include "AMRITA/isl.h"
AMRVOID AMRITA::call() {
   AMRSTR *package, *procedure;
   package   = strdup(ISL::get_file());
   procedure = strdup(ISL::get_str());
   DL::call(package,procedure);
   free(package);
   free(procedure);
   ISL::done("amrita::call");
}
```

because it utilizes the `ISL::` routines `get_file()` and `get_str()` to grab the pertinent information needed by the procedure which does the dynamic-linking. The call to `ISL::done` sends information back down the pipe-line to inform *Amrita* that the operation completed normally. The *ISL* parser then checks for the closing brace '}', before moving on to decode the next keyword (here `amrita:unplug::foo`) to come down the pipe-line[101].

D.5 Import-Export Control

By design, the two ends of the *ISL* pipe-line need not reside on the same machine. Therefore a `plugin`, and any code linked to it, does not have direct access to the string tokens of an *Amrita* script. Instead an import-export control mechanism is used to exchange explicit packets of information, over and above that exchanged by the `plugin`'s built-in keywords.

This script shows how you can generalise the *print_greetings* example to output an arbitrary string token:

```
fold::amrcp { user instructions
   type {
      amrita export_msg
   }
}
fold::amrita { make package
   pushcwd code
   ...  compile some Fortran
   popcwd
}
plugin foo
set message = The quick brown fox ...
export message
call code/package::print_token
```

```
amrcp  vki/export.1
amrita export_msg
```

[101] A call-back procedure is free to parse the contents of on an *ISL* command in any way it sees fit. Its only obligation is to stop at the closing brace '}' to allow the parser to check for the end of the *ISL* block. Consequently there is nothing to stop a code-developer from embedding program sources, or even executables, within the *ISL* stream. Moreover, because the programmer controls both ends of the pipe-line (see p. 25), he or she is free to employ specialist hand-shaking should the need arise. This simple design, coupled with the fact that the logical pipe-line could be generalized to several physical lines connecting machines on different continents, ensures that *Amrita* will stand the test of time, at least over the next decade or so. Of course, ideas can often be cheap, and so the accuracy of this last statement rests in future graft and implementation details. Good software never dies – old components are phased out, as new improved components are phased in.

Here, I have deliberately chosen to show the program fold which provides the information used to typeset the shadow-box instructions in these notes[102]. The second fold[103]:

```
fold::amrf77'mycode { compile some Fortran
   fold>amrso = package
   fold>src   = package.f
   C
   C      OUTPUT AMRITA STRING TOKEN
   C
          SUBROUTINE PRINT_TOKEN
          CHARACTER*80 STR
          INTEGER      AMR_LEN
          CALL AMR_GET_TOKEN('message',STR)
          WRITE(6,*) (STR(I:I),I=1,AMR_LEN(STR))
          RETURN
          END
}
```

compiles a few lines of *Fortran* to produce the shared-object *package*. The *Amrita* keyword `export` fetches the contents of the string token `message` and fires an *ISL* packet down to *Foo*. Upon receipt, *Foo* squirrels the token away in an internal storage-heap, ready for when the *Fortran* code issues an `AMR_GET_TOKEN`[104]. The function `AMR_LEN` returns the length of a null terminated string, as used by *Amrita*, thereby allowing the `WRITE` statement to print out the requisite number of characters in the `message`.

Apart from the long variable names, the *Fortran* used above meets the *f77* standard. Therefore, although I might choose to write:

```
fold::amrf77'mycode { compile some Fortran
   fold>amrso = package
   fold>src   = package.F
   C
   C      OUTPUT AMRITA STRING TOKEN
   C
          SUBROUTINE PRINT_TOKEN
          AMRSTR*80 STR
          AMRINT    AMR::LEN
          CALL AMR::GET_TOKEN('message',STR)
          WRITE(6,*) (STR(I:I),I=1,AMR::LEN(STR))
          RETURN
          END
}
```

you are not forced to do so. However, this next script illustrates why it is safer to take advantage of the benefits afforded by `amrpp`:

```
amrcp  vki/export.3
amrita typedefs
```

```
...  make package
plugin foo
set string = three point one four one five nine two six five
set number = 3
set single = 3.141593
set double = 3.141592653589793
export string,number,single,double
call code/package::print_tokens
```

[102]To re-iterate an earlier sentiment – the one thing worse than no documentation, is wrong documentation. For those interested, the instructions are typeset by `LatexAmrcp`.

[103]Ordinarily, the *Fortran* code would live in a separate file, but here it was convenient for me to bundle it in with the *Amrita* script.

[104]For consistency purposes, `ISL::GET_TOKEN` may also be used.

The tokens `string`, `number`, `single` and `double` may look like they contain quantities of different types, but to *Amrita* they are all just character strings. Therefore, when `AMR::GET_TOKEN` is used to pull the tokens off *Foo*'s storage heap, the onus is on the programmer to specify the necessary type conversion. For instance:

```
fold::amrf77'mycode { compile some Fortran
   fold>amrso = package
   fold>src   = package.F
   C
   C      OUTPUT AMRITA STRING TOKENS
   C
          SUBROUTINE PRINT_TOKENS
          AMRSTR*80 A
          AMRINT    B
          AMRSGL    C
          AMRDBL    D
          AMRINT    AMR::LEN
          CALL AMR::GET_TOKEN('AMRSTR::string',A)
          CALL AMR::GET_TOKEN('AMRINT::number',B)
          CALL AMR::GET_TOKEN('AMRSGL::single',C)
          CALL AMR::GET_TOKEN('AMRDBL::double',D)
          WRITE(6,*) (A(I:I),I=1,AMR::LEN(A))
          WRITE(6,*)  B,C,D
          RETURN
          END
}
```

The typedefs `AMRSTR`, `AMRINT`, `AMRSGL` and `AMRDBL` provide a convenient means of providing both cross-platform and cross-language consistency.

Here is how you can send information back from a code, (this time written in *C*):

```
fold::amrcc'mycode { compile some C
   fold>amrso = package
   fold>src   = package.C
   fold>guard = |
   /*
         set some Amrita string tokens
   */
   #include "AMRITA/isl.h"
   AMRVOID set_tokens(AMRVOID) {
      AMRSTR *A = "three point one four one five nine two six five";
      AMRINT  B = 3;
      AMRSGL  C = 3.141593;
      AMRDBL  D = 3.141592653589793;
      AMR::set_token("AMRSTR::string",A);
      AMR::set_token("AMRINT::number",&B);
      AMR::set_token("AMRSGL::single",&C);
      AMR::set_token("AMRDBL::double",&D);
   }
}
```

to an *Amrita* script:

```
... make package
plugin foo
call code/package:cc::set_tokens
import string,number,single,double
echo $string
echo $number $single $double
```

```
amrcp  vki/import.1
amrita import_tokens
```

D.6 Grid Generation

To close this section, below is a small script to produce the polar-grid shown in Figure 17[105]:

```
... create code/f77/polar
EulerEquations
plugin amr_sol
def Domain
   set grid::NS = 4
   set grid::R1 = 1
   set grid::R2 = 3
   do n=1,$grid::NS
      patch <1,+,w25,h25>
   end do
   export grid::{*}
   grid   code/f77/polar
end def
... plot grid
```

```
amrcp  vki/polar.1f
amrita f77_polar
amrps  ps/polar.ps
```

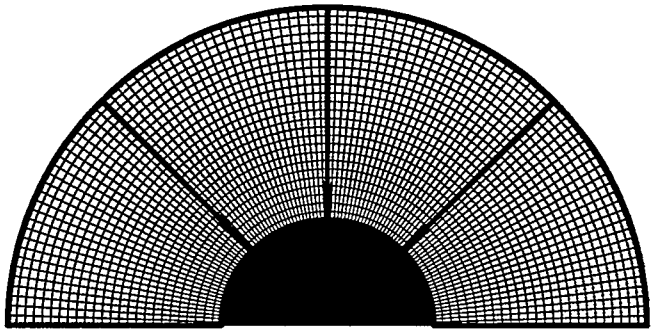


Figure 17: *Amr_sol* polar-grid generated using dynamic-linking.

The script illustrates how namespaced tokens[106] can be exported en masse. The line:

```
export grid::{*}
```

allows the *Fortran* code to access: the number of grid sectors, `grid::NS`; the inner-radius of the grid, `grid::R1`; the outer radius of the grid, `grid::R1`. However, here the attention of focus is the script-line[107]:

```
grid code/f77/polar
```

The keyword `grid`, or `amr_sol::grid` to give the full name, acts as a special version of `call`. The dynamic linker loads *polar* in the normal fashion, but instead of invoking a named procedure, it jumps directly to a binding routine `AMR_SOL::GENERATE_GRID`. This routine is a system wrapper, similar to `AMR_SOL::INTEGRATE_GRID` described on p. 49.

To compile *polar* manually, so as to see where the wrapper comes from, type:

```
unix-prompt>cd polar/f77
unix-prompt>amrita polar.mk
```

[105] If you prefer programming in *C*, use ***amrcp vki/polar.1c***.

[106] *Fortran* programmers unfamiliar with namespaces can view them as glorified common blocks.

[107] The `def Domain` block will be described in lecture 2.

to run the script:

```
NullEquationSet
plugin amr_sol
def Grid
   patch: GEN_SECTOR(NG,IM,JM,X,Y,IW,JS)
end def
CompileGrid grid=polar
```

to compile the file *polar.src*[108]:

```
#include "AMR_SOL/AMRITA"
      SUBROUTINE GEN_SECTOR(NG,IM,JM,X,Y,IW,JS)
      AMRINT NG,IM,JM,IW,JS,NS
      AMRDBL X(amrVpatch(IM,JM,NG))
      AMRDBL Y(amrVpatch(IM,JM,NG))
      AMRDBL R,THETA,PI,R1,R2
      PI = 4*ATAN(1.0 D0)
      CALL AMR::GET_TOKEN('AMRDBL:grid::R1',R1)
      CALL AMR::GET_TOKEN('AMRDBL:grid::R2',R2)
      CALL AMR::GET_TOKEN('AMRINT:grid::NS',NS)
      DO I=1-NG,IM+NG+1
         R = R2+(R1-R2)*FLOAT(I+IW-2)/IM
         DO J=1-NG,JM+NG+1
            THETA = ((2-J-JS)/FLOAT(NS*JM)+1.0)*PI
            X(I,J) = R*COS(THETA)
            Y(I,J) = R*SIN(THETA)
         END DO
      END DO
      RETURN
      END
```

The `def Grid` block serves a similar purpose to the `def Solver` block discussed in §B.2. The `patch` command identifies the subroutine which needs the wrapper. It also identifies the information that *Amr_sol* must provide the grid generator. The labels `IM`, `JM` etc. are mnemonics for variables, the quantities which are actually passed across depend on the choice of programming language.

Here it just so happens that for `GEN_SECTOR` I chose to use variable names to match the mnemonics. In practice, you write the *Fortran*, or *C*, using whatever names you want. Then, afterwards, you write the `def Grid` block to instruct *Amr_sol* what information must be passed across, and what order it must be supplied in[109]. One exercise, to try, is to rearrange the order of the `SUBROUTINE` parameters. So long as the `patch` mnemonics are similarly rearranged, the code can be compiled and will run just as successfully as before. The changes in the wrapper can be seen by examining the file *f77src/polar.F*.

[108] If you feel uneasy looking at this *Fortran*, try typing:

unix-prompt> **amrpp srcin=polar.src lang=f77 warn=yes**

to see the output from *amrpp*. For instance, `amrVpatch(IM,JM,NG)` dimensions an array large enough to store Vertex quantities for a patch *IM* by *JM* cells surrounded by *NG* layers of ghostcells.

[109] If *Amr_sol* only had to worry about user-supplied routines written in *C*, it could simply pass across a structure containing all the necessary information. However, apart from its multi-lingual capabilities, the present approach has the merit that the innards of *Amr_sol* can be changed without impacting on user-written code. Following a new release, the user need only recompile his or her code, keeping the original `def Grid` block the same, and the appropriate bindings are automatically generated.

E Anatomy of `plugin` *Foo*

The *Amrita* library routine *ClonePluginFoo* constructs a bare-bones *Amrita* plugin which provides two keywords: *com1* and *com2*. Although the functionality is limited, a *Foo*-cloned plugin has the exact same architecture as *Amr_sol* and so essentially provides the boiler-plate code for any new `plugin`, regardless of its sophistication or target application. To see how this might work in practice, consider the one line script:

```
ClonePluginFoo name=vki
```

```
amrcp  vki/foo.1a
amrita clone_foo
```

It produces a *Foo*-clone named *Vki* with directory root, relative to your home directory:

.amrita/plugin/vki

or if set, relative to the environment variable *AMRITA_HOME*.

Once the `plugin` is built, it can be put through its admittedly very limited paces:

```
amrcp  vki/foo.1b
amrita run_clone
```

```
plugin vki
com1 'com1' does nothing more exciting
com1 than echo its string argument to
com1 the screen
def VkiInterlock
   com2 {
      'com2' can only be used inside a:

                def VkiInterlock

                end def

      block, but is capable of outputting an
      entire block of text.
   }
end def
```

The remaining sub-sections correspond directly to program folds in `ClonePluginFoo`, which can be perused along with this text by typing:

```
unix-prompt>cd $AMRITA/stdlib/system
unix-prompt>amrgi ClonePluginFoo.amr
```

E.1 `ClonePlugin2Perl`

The keyword `plugin` instructs the *Amrita* interpreter to search the directories:

$AMRITA/plugin
$AMRITA_HOME/plugin

for a directory called *vki*. On locating the root of the plugin, the interpreter (which is written in *Perl*) parses the file *vki.pl* so as to obtain two new procedures[110]:

```
amrita'plugin'vki'
vki'copymsg
```

[110] *Perl* aficionados should note that *Amrita* is *Perl4* compliant.

which are short enough to be listed here in full:

```
sub amrita'plugin'vki {
   $ROOT'VKI = @_[0];
   $amrita'plugin{"keywords::vki"} = "$ROOT'VKI/keywords/KEYWORDS";
   $amrita'plugin{"defaults::vki"} = "$ROOT'VKI/defaults/plugin";
   &vki'copymsg();
}
sub vki'copymsg {
   $AMRITA'COPYMSG{"vki"} = <<COPY;
     plugin::vki Copyright (C) James J. Quirk (aka jjq)
COPY
}
#
1;
```

Only the first of these procedures is mandatory: it instructs *Amrita* where to locate the keywords *Vki* brings to the programming table, see §E.4; it identifies a defaults script, written in *Amrita*, which will be run once the `plugin` is activated, see §E.2.

The second routine shows that authors of plugins can daisy-chain their own copyright messages to those of *Amrita*'s[111]. Try running:

```
amrcp  vki/copy.right
amrita copyright_msg
```

```
plugin vki
command
```

to place *Amrita* into its command mode where script lines can be typed interactively; it demonstrates that *Amrita* is respectful of intellectual ownership[112].

E.2 CloneDefaults

The file *vki/defaults/plugin*:

```
#
# The following will be executed by vki on plugin
#
set defaults = $amrita::AMRITA/defaults
parse file $defaults/plugin
```

defers to a set of master defaults which fix the size of the graphics page etc. Additional *Amrita* script could be added as needed.

E.3 CloneIncludes

The file *$AMRITA_HOME/include/cc/VKI/AMRITA*:

```
#include <stdio.h>
#amrpp namespace VKI  f77{VKI_}  cc{vki_}
#include "AMRITA/isl.h"
#define SCREEN  stdout
```

is a header file (see §E.5) for inclusion by the *C* files: *keywords.C*, *com1.C* and *com2.C*. The #amrpp directive instructs *amrf77* and *amrcc* to map the namespace `VKI::` to `VKI_` and `vki_`, respectively. Additional namespaces could be added as needed.

[111] When you have run *run_clone*, your name will appear in the copyright message instead of mine.

[112] Please read the licence agreement by which you obtained *Amrita*: it does not lie in the public domain.

E.4 CloneKeywords

The file *vki/keywords/KEYWORDS*:

```
KEYSPACE vki:: {
   com1
   DEF VkiInterlock {
      *com2
   }
}
```

provides *Amrita* with a list of the keywords that `plugin` *Vki* can parse. Based on this information *Amrita* searches[113] the *vki/keywords* directory structure for three *Perl* files: (i) a file named *com1.pl*; (ii) a file named *vki.pl*, in a directory named `VkiInterlock`; (iii) a file named *com2.pl* which lives in the same directory as file (ii). An error is issued, should any of the files be missing, but no attempt is made to parse the *Perl* until it is needed. The * against `com2` informs the interpreter that the keyword spans multiple lines and so must be called even when it is inactive, as in:

```
if(0) then
   com2 {
      without the help of vki::com2
      Amrita would not know how to
      skip over this inactive command
   }
endif
```

E.4.1 com1

The file *vki/keywords/basic/com1.pl*:

```
sub vki'com1 {
   $line =~ s/^\s*//;
   &isl'put_ltag(0,'vki::com1');
      &isl'put_str(1,$line);
   &isl'put_rtag(0,'vki::com1');
   $line = '';
}
#
1;
```

is sucked into *Amrita*, as a one-off, when the interpreter first comes across a script-line which begins with `com1` or `vki::com1`. When this routine is called, the scalar variable `$line` contains the text of the *Amrita* script-line following the keyword `com1`. The three `isl` routines output to the *ISL* pipe-line and are defined in *$AMRITA/Amrita/isl.pl*. They are provided as a convenience, that is you are free to substitute your own routines should you so desire. A keyword must set `$line` to a null string before exiting, otherwise *Amrita* will complain that it expected an end of statement. Normally, `$line` is whittled down as the command is parsed, but here it is explicitly set to a null string.

[113] A full-blown search is only done, if the file locations depart from those used here. Moreover, the *KEYWORDS* file can contain directives to specify where to start the search for a particular keyspace.

E.4.2 def VkiInterlock

The file *vki/keywords/def/VkiInterlock/vki.pl*:

```
sub Enter'vki'VkiInterlock {
}
sub Exit'vki'VkiInterlock {
}
#
1;
```

is sucked into *Amrita*, as a one-off, when the interpreter first comes across a script-line which begins with `def VkiInterlock`. The `Enter` and `Exit` stubs can be used to control the program behaviour inside the `def` block. For instance, you could choose to turn `com1` off upon entering the interlock, and turn it back on again at exit. *Amrita* automatically restricts the visibility of `com2` to script-lines within the `def VkiInterlock` block.

E.4.3 com2

The file *vki/keywords/def/VkiInterlock/com2.pl*:

```
sub vki'VkiInterlock'com2 {
   local($active) = @_;
   local(@com2'strs);
   unless($line =~ /^\s*{/) {
      $error[1] = "expected '{'!";
      &report_error();
   }
   $line = $';
   &check_end_statement() if($active);
   undef @com2'strs;
   while (1) {
      $n = &get_line($INPUT_FHDL,1,1);
etc ..
```

is sucked into *Amrita*, as a one-off, when the interpreter first comes across a script-line which begins with `com2` or `vki::com2`. The *Perl* for this command is more involved than that for `com1`. The parameter `$active` is passed from *Amrita* and determines whether `com2` should output its content to the screen or silently skip on by. The routines: `report_error`, `check_end_statement` and `get_line` are direct calls to the *Amrita* interpreter[114]. Once the body of `com2` has been gathered up, it is a relatively straightforward matter to send the appropriate *ISL* down the pipe-line to the `plugin`.

```
   if($active) {
      &isl'put_ltag(0,'vki:VkiInterlock::com2');
          &isl'put_int(1,$#com2'strs+1);
          foreach $str (@com2'strs) {
             &isl'put_str(1,$str);
          }
      &isl'put_rtag(0,'vki:VkiInterlock::com2');
   }
etc ..
```

[114]These calls are soon to be cleaned up to read `amrita'report_error` etc. so as to better protect *Amrita* from a wayward *plugin*.

E.5 CloneSrc

The file *vki/src/vki.C*[115]:

```
#amrpp include "VKI/AMRITA"
AMRVOID VKI::args();
AMRVOID VKI::keywords();
main () {
   ISL::parser(VKI::args,VKI::keywords);
}
```

provides the main driver for *Vki*. The driver for a full-blown `plugin` need not be any larger than this, because startup and shutdown procedures are dealt with through the `ISL::parser`. The parser is fed two routines: (i) `VKI::args()` which decodes any system arguments that are passed to the `plugin`[116]; (ii) `VKI::keywords()` which adds a set of call-back routines to supplement the built-in ones.

E.5.1 *keywords.C*

The file *vki/src/keywords.C*:

```
#amrpp include "VKI/AMRITA"
AMRVOID VKI::keywords() {
   AMRVOID VKI::plugin();
   AMRVOID VKI::unplug();
   AMRVOID VKI::com1();
   AMRVOID VKI::com2();
   ISL::add_keyword("amrita:plugin::vki",     VKI::plugin);
   ISL::add_keyword("amrita:unplug::vki",     VKI::unplug);
   ISL::add_keyword("vki::com1",              VKI::com1);
   ISL::add_keyword("vki:VkiInterlock::com2", VKI::com2);
}
```

adds four call-back procedures using the library routine `ISL::add_keyword`. The first procedure `VKI::plugin` is called in response to the *ISL* tag `amrita:plugin::vki` which is generated at `plugin` time. `VKI::unplug` is called in response to `amrita:unplug::vki`, which is automatically generated when a script terminates. This provides the `plugin` with a chance to exit gracefully, should it need to flush any output buffers. The procedures `VKI::com1` and `VKI::com2` are called in response to the keywords `com1` and `com2`, stipulated in §E.4. Note the *ISL* tag for `com2` includes the name of the `def` block which activates the keyword.

Four coding-steps are needed to add new keywords to *Foo*: (i) modify the *KEYWORDS* file (§E.4) to inform *Amrita* of the new keywords; (ii) write the necessary *Perl* parsing routines, à la *com1.pl* (§E.4.1) and *com2.pl* (§E.4.3); (iii) append the appropriate `ISL::add_keyword` calls to *keywords.C*; (iv) construct the call-back routines, à la *com1.C* (§E.5.3) and *com2.C* (§E.5.4). In the case of *Amr_sol*, I found it convenient to add just one new keyword at a time. After each keyword was added, I would test and debug the `plugin` before moving on to the next keyword. This divide-and-conquer approach was far more productive than an abortive attempt at mass assimilation.

[115]The `#amrpp include` directive causes *VKI/AMRITA* to be in-lined by the pre-processing phase *vki.C* → *ccsrc/vki.c* so that the namespace `VKI::` is dealt with correctly. The plain `#include` contained by *VKI/AMRITA* does not come into effect until *ccsrc/vki.c* is compiled by an ANSI *C* compiler.

[116]These do not appear in a script but are sent using the environment variable `AMRITA_PLUGIN`. For instance, *amrdbx* (see p. 57) sends `-input debug.isl`.

E.5.2 *vki_lib.C*

The file *vki/src/vki_lib.C*:

```
#amrpp include "VKI/AMRITA"
AMRVOID VKI::plugin(AMRVOID) {
   ISL::plugin("vki");
}
AMRVOID VKI::unplug(AMRVOID) {
   ISL::unplug("vki");
}
AMRVOID VKI::args(AMRVOID) {
   ISL::args();
}
```

contains three routines which are needed for *Vki* to fulfill its system responsibilities towards the *ISL* parser. Here, the `plugin` defers to pre-defined `ISL::` routines, but if required, specialist code could be used instead.

E.5.3 *com1.C*

The file *vki/src/com1.C*:

```
#amrpp include "VKI/AMRITA"
AMRVOID VKI::com1() {
   AMRSTR *line;
   line = ISL::get_str();
   fprintf(SCREEN,"line: %s\n",line);
}
```

decodes the *ISL* generated by *com1.pl* (§E.4.1). The call `ISL::get_str` grabs a single `str` from the *ISL* stream. A full list of the pre-defined `ISL::` decoding routines are given in the header file *$AMRITA/include/cc/AMRITA/isl.h*. These routines, however, are merely provided as convenience, because a call-back procedure is free to parse the contents of an *ISL* command in any way it sees fit. Its only obligation is to stop at the closing brace '}' to allow the parser to check for the end of the *ISL* block.

E.5.4 *com2.C*

The file *vki/src/com2.C*:

```
#amrpp include "VKI/AMRITA"
AMRVOID VKI::com2() {
   AMRINT nstr,line=1;
   AMRSTR *str;
   nstr = ISL::get_int();
   while(line<=nstr) {
      str = ISL::get_str();
      fprintf(SCREEN,"line %3d: %s\n",line++,str);
   }
}
```

decodes the *ISL* generated by *com2.pl* (§E.4.3). The call `ISL::get_int` grabs a single `int` from the *ISL* stream. This provides the line count to see how many calls should be made to `ISL::get_str`. The file handle `SCREEN` is set to `stdout` in the header file *VKI/AMRITA*.

E.6 CloneAmritaBuild

The files:

vki/amrita.build
vki/src/amrita.make
vki/src/keywords/amrita.make

are used by *amrmake* to orchestrate the compilation of *Vki*[117]:

```
unix-prompt>amrmake -O -serial plugin::vki
unix-prompt>amrmake -G -serial plugin::vki
```

However, you are free to compile code in the normal *UNIX* fashion. *Amrita*'s only expectation is that the resultant binary for `plugin` *Vki* be called:

amrita:plugin::vki

and that it reside in a directory[118]:

$AMRITA_HOME/bin/~AMRSO/O/vki

if intended for production runs, or:

$AMRITA_HOME/bin/~AMRSO/G/vki

if intended for *amrdbx* purposes.

The *amrita.build* file contains all the information needed to locate the libraries used to link the `plugin`. At the time of writing, these are:

`-lvki`	see §E.5.2
`-lamrita`	*ISL* parser plus `amrita::` keywords
`-lamrita_gl`	`amrita:gl::` keywords
`-ljpeg`	see directory *$AMRITA/src3p/libJpeg*
`-lYgl`	see directory *$AMRITA/src3p/libYgl*
`-lXext -lX11`	standard *X11* libraries
`-lmpi`	standard *MPI* message passing library

The last library is not needed on `-serial` platforms.

[117] The default options to *amrmake* are: `-O -serial`.
[118] *$AMRITA_HOME* defaults to your *UNIX $HOME* directory.

References

[1] ADOBE SYSTEMS INC., *PostScript Language Reference Manual* (2nd ed.), Addison-Wesley, 1990.

[2] T. BERNERS-LEE, R. FIELDING & H. FRYSTYK, *Hypertext Transfer Protocol – HTTP/1.0*, RFC1945, http://info.internet.isi.edu:80/in-notes/rfc/files/rfc1945.txt, May 1996.

[3] R.E. BERRY, *Programming Language Translation*, Ellis Horwood, 1982.

[4] R. BORNAT, *Understanding and Writing Compilers*, Macmillan Press, 1979.

[5] K.C. BOWLER, R.D. KENWAY, G.S. PAWLEY, D. ROWETH, & G.V. WILSON, *An Introduction to OCCAM 2 Programming*, Chartwell-Bratt, Studentlitteratur, Sweden, 1989.

[6] G.T. CAMACHO & M. ORTIZ, *Adaptive Lagrangian Modeling of Ballistic Penetration of Metallic Targets*, Comput. Meth. Appl. Mechanics & Engineering **142**, 269–301, 1997.

[7] W.Y. CRUTCHFIELD & M.L. WELCOME, *Object-Oriented Implementation of Adaptive Mesh Refinement Algorithms*, Scientific Programming **2**, 145-156, 1993.

[8] W. FICKETT & W. DAVIS, *Detonation*, University of California Press, Berkeley, 1979.

[9] J.D. FOLEY, A. VAN DAM, S.K. FEINER & J.F. HUGHES, *Computer Graphics: Principles and Practice* (2nd ed.), Addison-Wesley, 1990.

[10] S. GARFINKEL, *PGP: Pretty Good Privacy*, O'Reilly & Associates, 1995.

[11] P. GROGONO, *MOUSE: A Language for Microcomputers*, Petrocelli Books, Princeton NJ, 1982.

[12] W. GROPP, E. LUSK & A. SKJELLUM, *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, MIT Press, 1994.

[13] HENDRIX, J. E., *The Small-C Handbook*, Reston Publishing Company, Reston VA, 1984.

[14] H.W. LIEPMANN & A. ROSHKO, *Elements of Gasdynamics*, Wiley, 1957

[15] NETSCAPE COMMUNICATIONS CORPORATION, *Object-Signing tools*, http://developer.netscape.com/products/zigbert/index.html, 1998.

[16] S. PEMBERTON & M.C. DANIELS, *Pascal Implementation: The P4 Compiler*, Ellis Horwood, 1982.

[17] D.C. PRUETT & C-L. CHANG, *Transitional High-Speed Flow on a Cone: PSE Versus DNS*. In *Transition, Turbulence and Combustion*, edited by M.Y. Hussaini, T.B. Gatski and T.L. Jackson, Vol. I, pp. 379–389, Kluwer Academic Publishers, 1994.

[18] J.J. QUIRK, *A Contribution to the Great Riemann Solver Debate*, Int. J. Numer. Methods Fluids **18**, 555–574, 1994.

[19] J.J. QUIRK, T.L. JACKSON & A.K KAPILA, *Numerical Study of the Evolution of a Compressive Pulse in an Exploding Atmosphere*. In *Transition, Turbulence and Combustion*, edited by M.Y. Hussaini, T.B. Gatski and T.L. Jackson, Vol. II, pp. 313–329, Kluwer Academic Publishers, 1994.

[20] J.J. QUIRK, *A Cartesian Grid Approach with Hierarchical Refinement for Compressible Flows.* In *Computational Fluid Dynamics '94*, Invited Lectures and Special Technological Sessions of the Second European Computational Fluid Dynamics Conference, edited by S. Wagner, J. Périaux and E.H. Hirschel, Wiley, pp. 200–209, 1994.

[21] J.J. QUIRK, *An Introduction to Amrita*, in preparation.

[22] M. RICHARDS & C. WHITBY-STREVENS, *BCPL – the Language and its Compiler*, Cambridge University Press, 1980.

[23] G. STRANG, *On the construction and comparison of finite-difference schemes*, SIAM J. Num. Anal. **5**, 506–517, 1968.

[24] R.A. STREHLOW, R.E. MAURER & S. RAJAN, *Transverse Waves in Detonations: I. Spacing in the Hydrogen-Oxygen System*, AIAA Journal **7**, 323–328, 1969.

[25] B. STROUSTRUP, *The C++ Programming Language* (2nd ed.), Addison-Wesley, 1991.

[26] SUN MICROSYSTEMS, *Frequently Asked Questions – JAVA Security*, http://www.javasoft.com/sfaq/index.html, 1997.

[27] SUN MICROSYSTEMS, *Security: Safe-Tcl*, http://www.sun.com/960710/cover/tcl-safe.html, 1997.

[28] L. WALL & R.L. SCHWARTZ, *Programming Perl*, O'Reilly & Associates, 1991.

notes prepared for the VKI 29th CFD lecture series
23-27 February, 1998
lecture 2 of 3

Amr_sol: Design Principles and Practice

James J. Quirk
Graduate Aeronautical Laboratories
California Institute of Technology
Pasadena, CA 91125, USA.
jjq@galcit.caltech.edu

Abstract

This second lecture describes how you can use *Amrita* to explore some of the issues which shaped the design and construction of the `plugin` *Amr_sol*. The aim is to look beyond the bare-bone algorithmic details to allow you to build up a first-hand understanding of how the high-resolution, shock-refraction simulations presented in lecture three are produced. This knowledge is needed to be able to separate physical-fact from numerical-fancy when determining just how far the simulations can be trusted. In that regard, although *Amr_sol* has propagated well beyond the development stage to become a reliable investigative tool, there remains much room for improvement. Therefore this lecture will also indicate ways that *Amrita* could be used to orchestrate the required algorithmic improvements in a systematic fashion.

1 Introduction

Despite the impressive number-crunching power of massively parallel computers, it is worth illustrating that brute force computations of phenomena which contain disparate physical scales are ill-conceived. Consider the following example taken from the study of detonation waves.

1.1 Disparate Physical Scales

The usefulness of solid explosives stems from their ability to convert chemical energy very rapidly into heat energy via the propagation of a detonation wave consisting of a reaction zone coupled to a strong shock front (see §E.3)[1]. When a detonation propagates through an explosive material, the material is compressed by the lead shock front and the resultant rise in temperature behind the shock triggers a chemical reaction which releases large amounts of energy in the form of heat. This localized heat release provides motive force for the detonation front to propagate further into the unburnt material and a balance can be reached whereby a given explosive supports a nominally steady speed of detonation propagation. This speed of propagation is significant in that it characterizes the performance of the explosive.

Traditionally, detonation speeds in solid explosives are found from experiment. A cylindrical charge – known as a rate-stick – is ignited at one end, and the propagation speed – which can

[1] A good solid explosive converts energy at a rate $\approx 10^{10}$ watts/cm^2, thus a wave front 20 m square would operate at a power level equal to all the power the earth receives from the sun[11].

reach as high as 9,000 m/s – is measured at the other end, with the assumption that the length of the stick is sufficient to allow the detonation to reach its nominally steady speed.

The direct numerical simulation[2] of a rate-stick test represents a formidable computational challenge. Since the chemical reaction drives the detonation wave, the simulation must be able to resolve the narrow fire-region in the reaction zone where the bulk of the heat is released (see p. 68). Results with reduced reaction models suggest that at the very least 10 mesh cells are needed to capture the fire-region accurately[8]. For certain types of solid explosive the pertinent length scale may be as small as 0.02 mm[3], in which case the mesh spacing within the reaction zone would need to be no larger than 0.002 mm. Given that a rate-stick is of order 100 mm in length and 25 mm in diameter, some 3.13×10^8 cells would be required for an axisymmetric flow calculation on a uniform mesh. Moreover, from the point of view of numerical accuracy, because of the non-linearities involved, it is unlikely that the detonation front could be propagated by more than one mesh cell per time step. Consequently, it would take some 5×10^4 time steps for the detonation to travel the full length of the rate-stick. Therefore the total workload for the simulation would be of order 1.56×10^{13} cell updates.

Such a calculation would be absurdly uneconomic. A 1 Gflop computer might be capable of 10^6 mesh updates per second[4], in which case the calculation would take 181 days to run. Clearly, to make such a simulation viable, something other than a more powerful computer is required[5]. Hence the need for adaptive mesh refinement[6].

1.2 Adaptive Mesh Refinement

Adaptive mesh refinement (AMR) schemes(see [17, 33]) attempt to reduce computing costs by dynamically matching the local resolution of the computational grid to the requirements of an evolving flow solution. Thus very fine mesh cells are restricted to those regions where they are needed, and elsewhere the computational grid is kept relatively coarse. Such a strategy can dramatically reduce the computational effort required to simulate phenomena containing disparate physical scales. In the above rate-stick example, if the fine mesh cells were restricted to the vicinity of the fire region, only about 6.25×10^4 cells would be needed resulting in the order of 3.13×10^9 cell updates. Therefore, whereas the uniform mesh simulation might take 181 days to run, the adaptive mesh simulation would take just 52 minutes[7]. Because the potential savings are so large: any AMR scheme is better than none. Consequently, the computational literature is littered with examples of one-off, problem-specific mesh refinement strategies.

Superficially, the one-off approach appears attractive, because the development costs are considerably less than those for a general purpose AMR scheme. In practice, however, the development costs of a general scheme can be recouped across a wide range of projects, and

[2]This is used in the sense of reproducing the full nuances of the physical system, as opposed to merely predicting one or two global quantities. However, the discussion presented here is restricted to macroscopic scales. A Quantum Chemist, for example, would be concerned with reaction mechanisms measured in pico-seconds.

[3]The reaction zone could be as wide as 1mm, but most of this would be taken up by an induction region.

[4]A single cell-update using realistic chemistry would require far more than 1,000 floating point operations and so this assumption is optimistically high.

[5]Notionally, a tera flop computer would reduce the time to under a day, but the practical turnaround time would still most likely take several days, if not weeks, given the number of jobs which vie for a large, centralized computing resource.

[6]Similar arguments can be put forward (say by a Quantum Chemist, see footnote 2) for the need to supplement AMR simulations with improved reaction modelling and analysis.

[7]Given the approximations made, these times should not be taken literally. Nevertheless, the conclusion stands: a calculation with local mesh refinement can – if the physical scales are disparate enough – be up to three orders of magnitude cheaper than an equivalent uniform mesh calculation.

over time the cost per project becomes negligible. On the other hand, with the one-off approach the effective costs accumulate with each passing project and can become unexpectedly large over time. Moreover, since one-off schemes rarely reach maturity, they tend to be needlessly expensive to run. Therefore, taken overall, there is little merit in pursuing a one-off approach.

Even amongst general purpose AMR schemes, there remains an element of "horses for courses," because an algorithm has to strike a balance between the desirable and the practicable. Therefore a method, say, which was designed to provide the cheapest medium-accuracy solution to a steady flow problem might not be competitive when it comes to producing the most accurate solution to a time-dependent problem, and vice versa. Consequently some care should be taken in choosing the most appropriate form of mesh refinement, for a given application, before embarking on what might be an arduous exercise in software development.

In 1988 a series of circumstances[8] led me to adopt a form of block-structured, AMR algorithm first proposed by Berger and co-workers[4, 6, 5], and the mesh refinement guts of *Amr_sol* were written shortly thereafter.

The bare-bone algorithmic details which underpin *Amr_sol* are sufficiently well documented elsewhere[21, 25], that here only a brief overview is presented in Appendix A[9]. Instead, this lecture aims to engender some discussion as to the strengths and weaknesses of various refinement strategies as applied to investigations of time-dependent, shock wave phenomena. The aim is not to promote one scheme over another, but to reveal specific issues which shaped the design of *Amr_sol*. To help place this discussion in the right context, three – hands on – numerical simulations are presented which were inspired by a series of experiments performed by Takayama *et al.*[32]. The experiments were done to classify the canonical reflection processes which occur when a planar shock wave interacts with a double wedge. The numerical simulations serve a number of purposes, not least of which is that they provide templates for you to construct your own investigations. To help you understand the construction of the provided scripts, Appendices B – G describe the specialist *Amrita* keywords used to drive *Amr_sol*. These should be read in conjunction with the *Amrita* primer given in lecture 1.

Here is the complete road map for the lecture:

Road Map

[8] As is often the case with a Ph.D. thesis, external influences led to Hobson's choice.

[9] Given *Amrita*'s document preparation skills, you should not be too surprised to learn that Appendix A can be reproduced using the four line script:

```
LatexHead
LatexPlugin plugin=amr_sol
LatexTail
Latex
```

although the precise formatting depends on the LaTeX page size, margin width etc.

2 Shock Double-Wedge Interactions

To obtain the *Amrita* scripts needed to follow this Section, type:

```
unix-prompt>amrcp vki/ramp.mailit
unix-prompt>amrita ramp.mailit
unix-prompt>cd ramp
```

then run:

```
unix-prompt>amrita -a schematic do.one_off.ramp
unix-prompt>amrps ps/ramp_schematic.ps
```

to produce Figure 1.

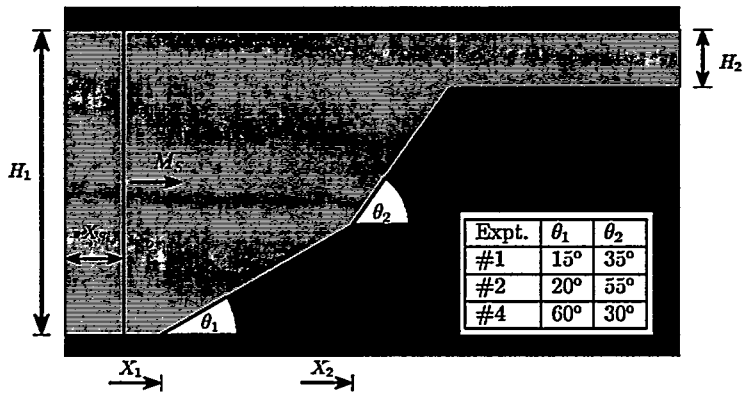


Figure 1: Double-wedge configurations used to simulate the experiments of Takayama *et al.*[32].

The main purpose of this Section is to observe the variations in the shock reflection process, for the three sets of $\{\theta_1, \theta_2\}$ shown above, so as to provide a solid introduction to the discussion in §3. However, along the way, some pointers are also given on how best to construct an *Amrita* investigation. First-off, it is worth noting that Figure 1 is drawn using:

```
proc RampSchematic
   parse file studies/schematic
   RampProblem
   ... draw domain
   ... draw shock
   ... draw table of experiments
   ... label theta1 and theta2
   ... label X1
   ... label X2
   ... label Xs
   ... label H1
   ... label H2
end proc
```

which relies on the actual `RampProblem` procedure used to set up the simulations. Thus the schematic is guaranteed to be a faithful reflection of events. The file *studies/schematic* contains *Amrita* script to fix: M_S, X_S, θ_1, θ_2, $X1$, $X2$, $H1$ and $H2$, and fulfills the same purpose as the actual *studies*: *TK1*, *TK2* and *TK4*. Using a separate *schematic* `study`, in this fashion, opens up the possibility of introducing a domain re-scaling should the aspect ratio of the true domain not lend itself to a schematic. Similarly, the pseudo-study *help* provides the user with a set of help instructions and *clean* removes all derived files to return *ramp* to its original pristine condition.

2.1 File Management

The *ramp.mailit* employs the following directory structure:

- *ramp*
  This directory contains the driver scripts needed to perform generic ramp-investigations, e.g. *do.one_off.ramp* and *do.ramp.study*.

- *ramp/lib*
  This directory contains the *Amrita* procedures needed to perform generic ramp-investigations, e.g. *InitRampResources.amr*, *BasicRampGrid.amr*, *RampProblem.amr*.

- *ramp/code*
  This directory is created by `InitRampResources` and contains the compiled code needed for generic ramp-investigations, e.g. *body_roe* and *ramp*. The *body_roe* `solver` is fashioned by *BCG* and the *ramp* `grid` is fashioned by `BasicRampGrid`.

- *ramp/studies*
  This directory contains the *Amrita* script needed to set the scope of a specific investigation or `study`, e.g. *TK1*, *TK2* and *TK4*.

- *ramp/results/*`study`
  This directory contains the output from a specific `study`.

- *ramp/ps*
  This directory contains miscellaneous *PostScript* output such as the file *ramp_schematic.ps*.

- *ramp/logs/*`study`
  This directory contains any diagnostics output by a specific `study`.

- *ramp/help*
  This directory contains the *HTML* instructions produced by running:

  unix-prompt> **`amrita -a help do.one_off.ramp`**

as does the *la.mailit* on p. 47.

The precise details of the above directory structure are relatively unimportant, the observation is that repeatable investigations stem from good file management, and so the *ramp* application maintains distinct sets of like-minded files. This mundane observation transcends the mathematical complexity of the target application, but is all too easily over-looked in the rush to produce numerical results. *Amrita* commands which output files: `flowout`, `plotfile`, `printfile`, `logfile` etc., automatically create directory paths as needed by their arguments. For instance, the procedure `SolverMontage`, from lecture 1, contains the line:

```
plotfile $amrita:latex::PS/$solver/Ms$Ms/schlieren.ps
```

to arrange *PostScript* plots indexed by the flow solver used to simulate a specific Mach number. The procedure `OutputResults` from *la.mailit* uses a more convoluted index:

```
set la::results = \
    results/$la::study/$la::solver/$la::profile/$la::cfl
printfile $la::results/numerical
```

but the end purpose is the same.

2.2 RampProblem

The procedure `RampProblem` is short enough to be listed here in full:

```
proc RampProblem {
     Ms            =  2          # Mach number of incident shock
     Xs            =  5          # initial shock position
     Xf                          # final   shock position
     X1            = 25          # foot  of first  wedge
     THETA1       #= rad(15)     # angle of first  wedge
     X2            = 40          # foot  of second wedge
     THETA2       #= rad(35)     # angle of second wedge
     H1            = 80          # inflow  duct height
     H2            = 10          # outflow duct height
     npatches      = 4           # number of patches in G0
     lmax          = 1           # number of grid levels
     r             = 4           # refinement ratio
   } <-> ramp::

   def Domain
      do n=1,$npatches
         patch <+, 1,w40,h80>
      end do
      export ramp::{X1,X2,H1,H2,THETA1,THETA2}
      grid code/ramp
   end def

   W'quiescent  ::= <RHO=1,U=0,V=0,P=1>
   ShockWave Ms=$Ms, state1=quiescent,\
                     state2=post_shock

   def BoundaryConditions
      Nbdy domain: reflect
      Sbdy domain: reflect
      Ebdy domain: extrapolate
      Wbdy domain: prescribe   W'post_shock
   end def

   def SolutionField
      setfield W'quiescent
      setfield W'post_shock X[]<$Xs
   end def
   makefield

   def MeshAdaption
      adaption  on
      lmax      $lmax
      r         $r
   end def

   def RefinementCriteria
      DensityGradient
   end def

   do l=1,$lmax
      adapt
      makefield
   end do

   ...  compute tf
end proc
```

`RampProblem` is essentially a template for all you need provide to set up an arbitrary problem[10] for *Amr_sol* to solve. Therefore, although lecture 1 may have given the impression that *Amrita* requires you to become a heavy-duty programmer, this is not the case. Naturally, the more effort you put in, the greater the return[11].

The `def` blocks: `EquationSet`[12], `Domain`, `BoundaryConditions`, `SolutionField`, `MeshAdaption` and `RefinementCriteria`, are described in Appendices B-G. If you recall from the *Amrita* primer, `def` blocks act as interlocks which allow *Amrita* to maintain some semblance of control over the order in which a simulation is set up. They also introduce the specialist commands needed to get the job done. For instance, `Domain` must be supplied before `BoundaryConditions`, and `patch <+, 1, w40, h80>` lays down a mesh patch 40 cells wide and 80 cells high. The + signifies that the `patch` should be tacked on to the end of the previous patch (details are given in §C).

At this early stage, you should not be too worried about the CFD details of a boundary-condition such as `reflect`, because ultimately you can provide your own interpretation. Therefore *Amrita* scripts should be read at face value, on the understanding that *someone, somewhere* has provided the correct, number-crunching code. In time, with an appropriate amount of learning effort, this *someone* could be yourself and so there is no need to feel you relinquish basic intellectual control over a simulation by electing to make use of *Amrita*[13].

2.3 Experiment #1

Without further ado, you can type:

```
unix-prompt>amrita -a TK1 do.one_off.ramp
```

to produce an *Mpeg* animation of the first experiment which can be viewed using[14]:

```
unix-prompt>netscape results/TK1/ramp.mpg
```

The simulation takes around 15 minutes to run on an SGI Indigo2 machine (195 Mhz Mips R10000 processor) with 384 Mbytes of memory, a large percentage of which is simply taken up with writing the individual frames of the animation to the directory *results/TK1/jpg*. A restart file is also written to the directory *results/TK1/io*.

Amrita views animations as working diagnostics and provides machinery to allow them to be produced routinely[15]. However, for these printed notes, this script was run:

```
unix-prompt>amrita -a TK1 do.vki.ramp
```

to produce Figure 2.

---

[10] This is used strictly in the sense of the problems that *Amr_sol* is designed to tackle, e.g. 2D, time-dependent, compressible flows.

[11] But to repeat *Amrita*'s golden rule: the more trouble the systems-level programmer goes to, the easier programming-life becomes for the applications specialist. Therefore, if you find yourself writing contorted *Amrita* scripts, you could always argue the need for a new language feature to make your programming-life easier.

[12] Here this is buried inside the library procedure `EulerEquations`.

[13] A distinction can be made between "using *Amrita*" and "making use of *Amrita*." The former implies passive acceptance of anything the system provides, the latter implies acceptance of the system as a labour saving device with the realization that you can, when necessary, stamp your authority on proceedings.

[14] *Amrita* provides an *Mpeg* encoder but not a viewer and so here it is assumed your web browser is able to play *.mpg* files.

[15] Try dissecting the procedures `MakeRampAnimation`, `SaveRampImage` and `EncodeMpeg`.

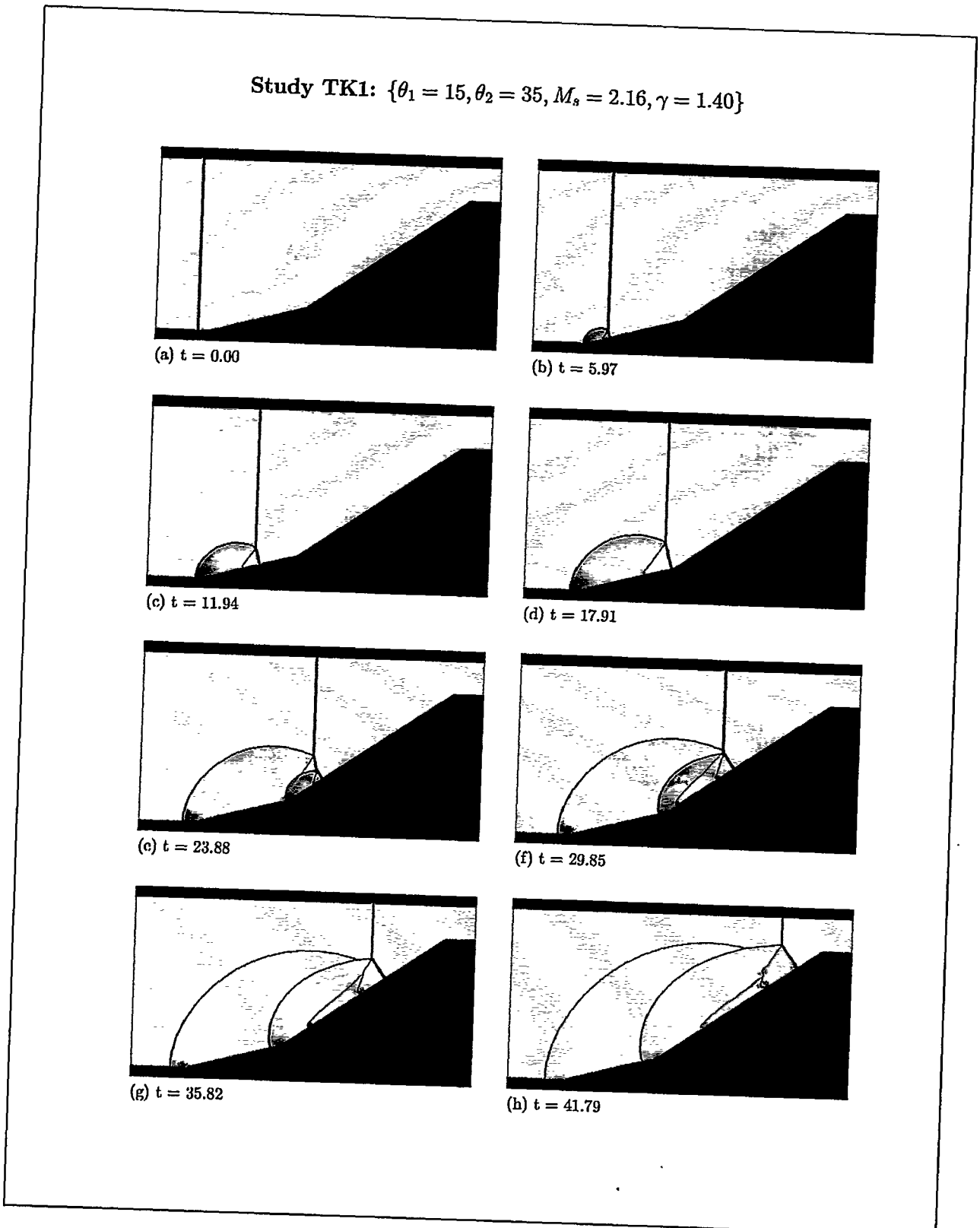


Figure 2: Page output by the *PostScript* file *ramp/results/TK1/ps/montage.ps.* At early times, frames (a) and (b), there is single Mach reflection (SMR) of the incident wave from the first ramp(see [3] for an introduction to shock reflexions). At intermediate times, frames (c) and (d), the Mach stem from this primary reflection interacts with the second wedge giving rise to a secondary reflection which is also of type SMR. At late times, frames (e) to (h), the secondary reflection interferes with the primary reflection. The snapshots were produced using the `SchlierenImage` procedure from lecture 1.

Given the time it takes to run, this first simulation falls firmly in the category of "cheap and cheerful," but it is still no less demanding to craft than the higher resolution results shown in Figure 3, which here for expediency was obtained using:

```
amrcp  vki/paste.tk1
amrita paste_tk1
amrps  ps/tk1.comparison.ps
```

```
plugin amr_sol
postscript on
...  latex title and captions
...  set page locations
set experiment = http://www.amrita-cfd.com/vki/TK1/experiment.ps.gz
set simulation = http://www.amrita-cfd.com/vki/TK1/simulation.ps.gz
paste $experiment in box  $e::xoff,$e::yoff,$e::width,?
paste $simulation in box  $s::xoff,$s::yoff,$s::width,?
```

As discussed in lecture 1, the comparison between numerics and experiment is not a mathematical exercise. Here the numerical and experimental interferograms are in close agreement, at least to the eye. But, because an interferogram provides quantitative values of the density field, it can be argued that there is also a reasonable quantitative agreement between simulation and experiment. Nevertheless, there are clear discrepancies on the small scale. For instance, in the experiment the base of the primary reflected shock has a small lambda foot due to its interaction with the boundary layer on the bottom wall of the shock tube (see bottom-left corner of image). This feature is missing in the numerical image since the simulation assumed that the flow was inviscid. Adding viscous terms to this type of simulation can be done (e.g. [14]), but the following scaling argument suggests that the grid needed to resolve the relevant viscous length-scale would not be cost effective for the small improvements it would bring.

The pertinent viscous length scale to resolve, δ_v, is of the order $\sqrt{\nu t_v}$ where ν is the kinematic viscosity of the fluid and t_v is the time vorticity has to diffuse from its point of origin. Taking ν to be $0.15\text{cm}^2/\text{sec}$[1] and t_v to be – on average[16] – $50\mu s$ gives a δ_v of just over $10\mu m$. But a typical shock-capturing scheme might need 10 cells to resolve a feature at this length scale and so the pertinent mesh spacing would be around $1\mu m$. The finest mesh spacing used for the simulation in Figure 3 was approximately $100\mu m$. Therefore a 100 fold reduction in mesh spacing would be needed in viscous dominated regions. Unfortunately, the cost of an unsteady, two-dimensional simulation, at least for a uniform mesh, increases eight-fold every time the mesh spacing is halved. The analogous increase in cost for an AMR scheme is more difficult to predict, because it is highly problem dependent. Here, because of the manner in which the flow is integrated, the increase is likely to be closer to 8 than the optimum – but unobtainable scaling – of unity[17]. Hence the above assertion that a viscous simulation is not cost effective.

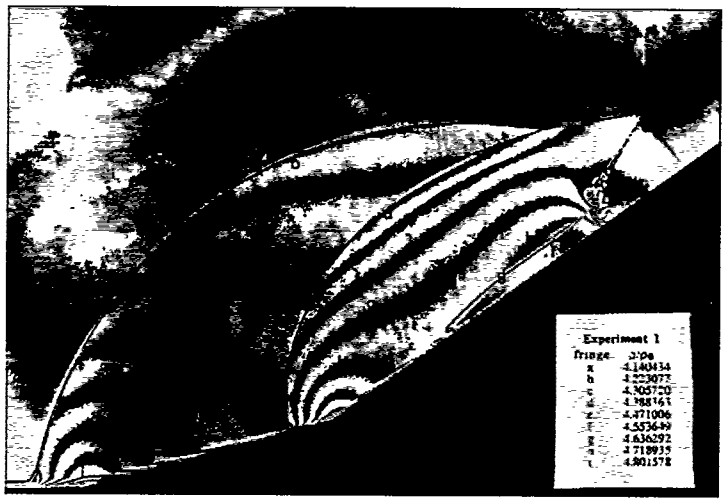
The scaling figures presented here are pessimistic, but the thrust of the argument remains true even when the figures are re-jigged to give the optimistic prediction of a 10 fold decrease in spacing. An engineering calculation, using a highly stretched mesh near the solid boundaries, would probably be sufficient to pick out the lambda shock, but would do nothing to improve the resolution of the roll-up of the contact surface. Interestingly, because of its extra dissipation, the low resolution simulation (Figure 2) gives a better prediction of the contact surface than the high-resolution simulation (Figure 3) with its exaggerated Kelvin-Helmholtz instability. Consequently, as done with the Schardin experiment in lecture 1, a viscous investigation would have to run a full-blown sensitivity study to ensure any observed improvements were down to improved numerics and not just a fortuity of grid resolution.

[16] The effective t_v varies across the flowfield: the value near the incident shock is significantly less than that near the foot of the ramp.

[17] Strictly, this observation is anecdotal until it is backed up by an explicit test.

Study TK1

(a) Experimental Interferogram, courtesy of Prof. Takayama



(b) Numerical Interferogram

Figure 3: Page output by *paste_tk1*.

2.4 Experiment #2

The second simulation can be run, as before, by typing:

```
unix-prompt>amrita -a TK2 do.one_off.ramp
```

to produce an animation, to be viewed using:

```
unix-prompt>netscape results/TK2/ramp.mpg
```

or by typing:

```
unix-prompt>amrita -a TK1 do.vki.ramp
```

to produce Figure 4. Similarly this script:

```
plugin amr_sol
postscript on
...  latex title and captions
...  set page locations
set experiment = http://www.amrita-cfd.com/vki/TK2/experiment.ps.gz
set simulation = http://www.amrita-cfd.com/vki/TK2/simulation.ps.gz
paste $experiment in box  $e::xoff,$e::yoff,$e::width,?
paste $simulation in box  $s::xoff,$s::yoff,$s::width,?
```

```
amrcp  vki/paste.tk2
amrita paste_tk2
amrps  ps/tk2.comparison.ps
```

can be used to obtain the comparison between numerics and experiment shown in Figure 5.

As with experiment #1, the two interferograms are in reasonable agreement, but the tie-up is noticeably poorer than before. Again the discrepancies are due to the lack of physical viscosity in the flow model. For instance, in the experimental image there is a recirculation zone at the apex of the first ramp, and the base of the secondary reflected shock has a lambda foot due to its interaction with the boundary layer on the wedge. But these features cannot be reproduced by an inviscid simulation. The shock-boundary layer interactions are now stronger than in Experiment #1 and have had quite a pronounced affect on the curvature with which both the primary and secondary reflected shocks run in to the wall. Consequently there would be some justification for switching to a viscous simulation for this experiment.

2.5 Experiment #4

The last simulation is the sequence can be run in the same manner as the other two to produce Figures 6 and 7. The two interferograms are again in fair agreement, except for those regions where viscous effects are expected to be important[18]. Namely, the vortex core near the convex corner, and the foot of the reflected shock where it interacts with the boundary layer on the wall of the shock tube. This interaction affects the curvature of the reflected shock and would seem to account for the difference in the curvature of the fringes between the computational and experimental interferograms. However, the tie-up is sufficiently good that, as in Experiment #1, it is not clear that a viscous simulation would be worth the extra effort involved.

[18] On closer inspection, however, it is clear that the numerical results are for an earlier time than the experiment. The two plots are scaled using the the distance from the incident shock to the foot of the second wedge, which is why these two reference points line up. But because the simulation was effectively stopped too early, there is a significant discrepancy in the position of the foot of the first-ramp. Such "deliberate mistakes" are not uncommon in the CFD literature and make it harder for the discriminating reader to draw an independent conclusion as to the quality of the presented results. Such errors provide one practical reason why *Amrita* goes to the trouble of providing the means to automate document preparation. Once an error is spotted, it can be easily remedied and so does not have to remain a permanent source of confusion.

Study TK2: $\{\theta_1 = 20, \theta_2 = 55, M_s = 2.16, \gamma = 1.40\}$

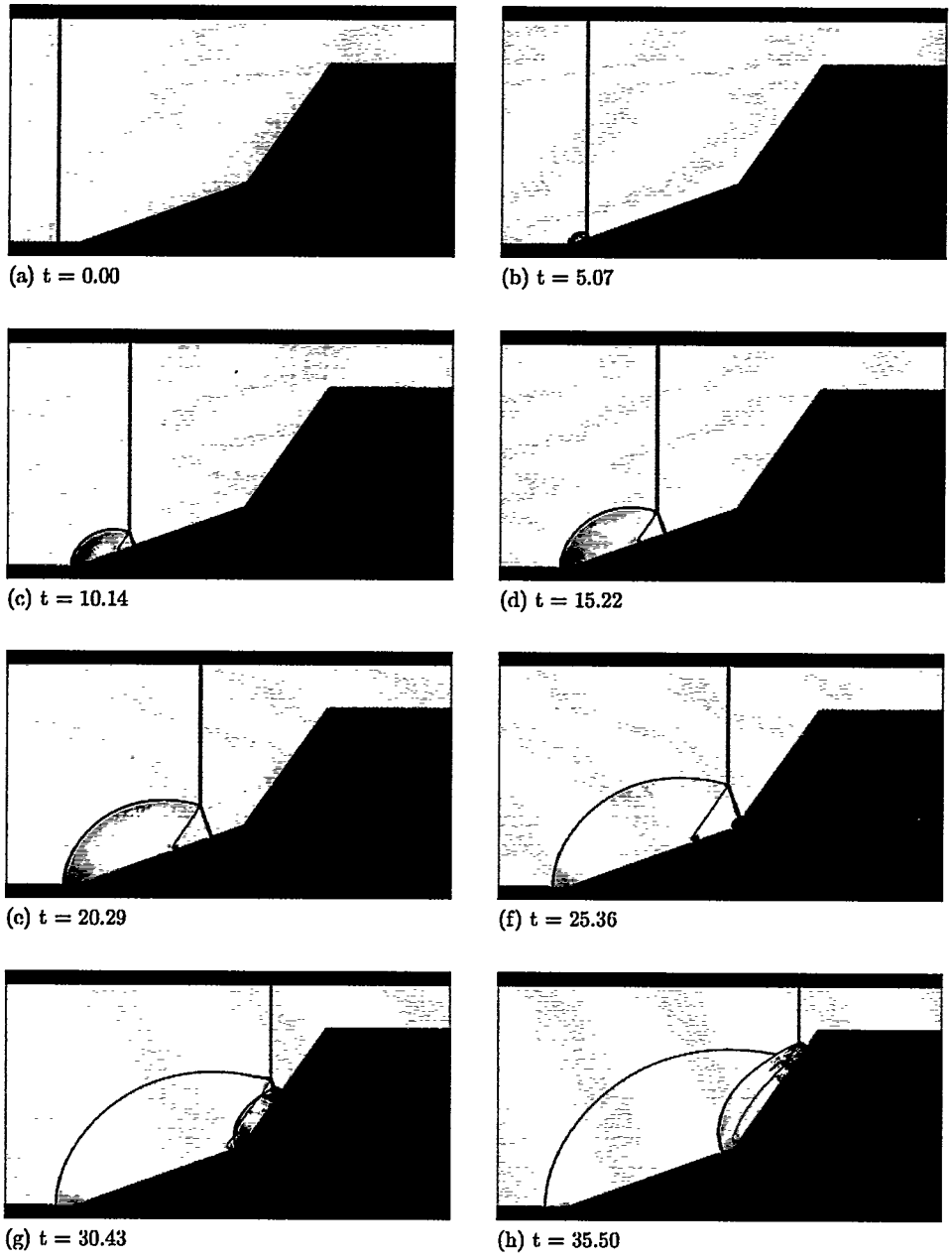
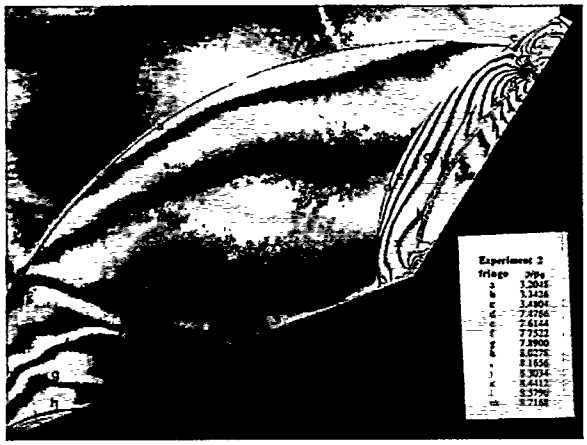


Figure 4: Page output by the *PostScript* file *ramp/results/TK2/ps/montage.ps*. At early times, frames (a) and (b), there is SMR of the incident wave from the first ramp as in Experiment #1. However, at intermediate times, frames (c) and (d), the reflection of the Mach stem is now complex Mach reflection (CMR) rather than SMR. At late times, frames (e) to (h), the secondary reflection again interferes with the primary reflection.

Study TK2

(a) Experimental Interferogram, courtesy of Prof. Takayama



(b) Numerical Interferogram

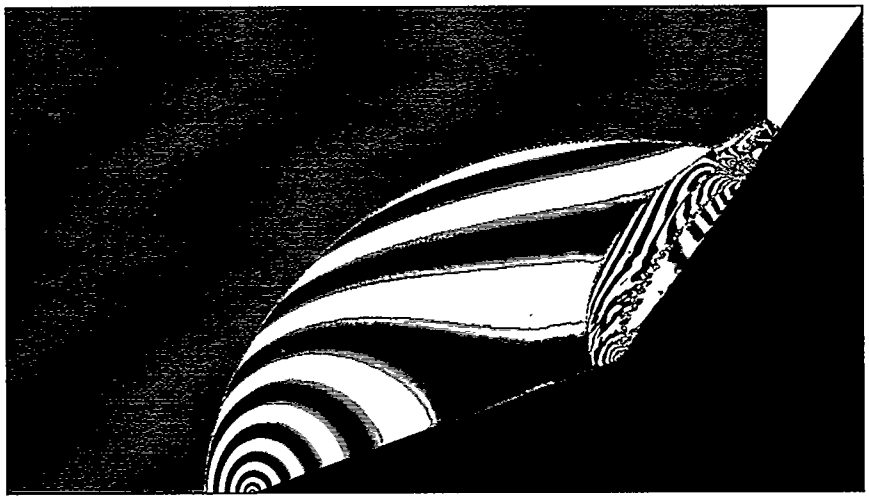
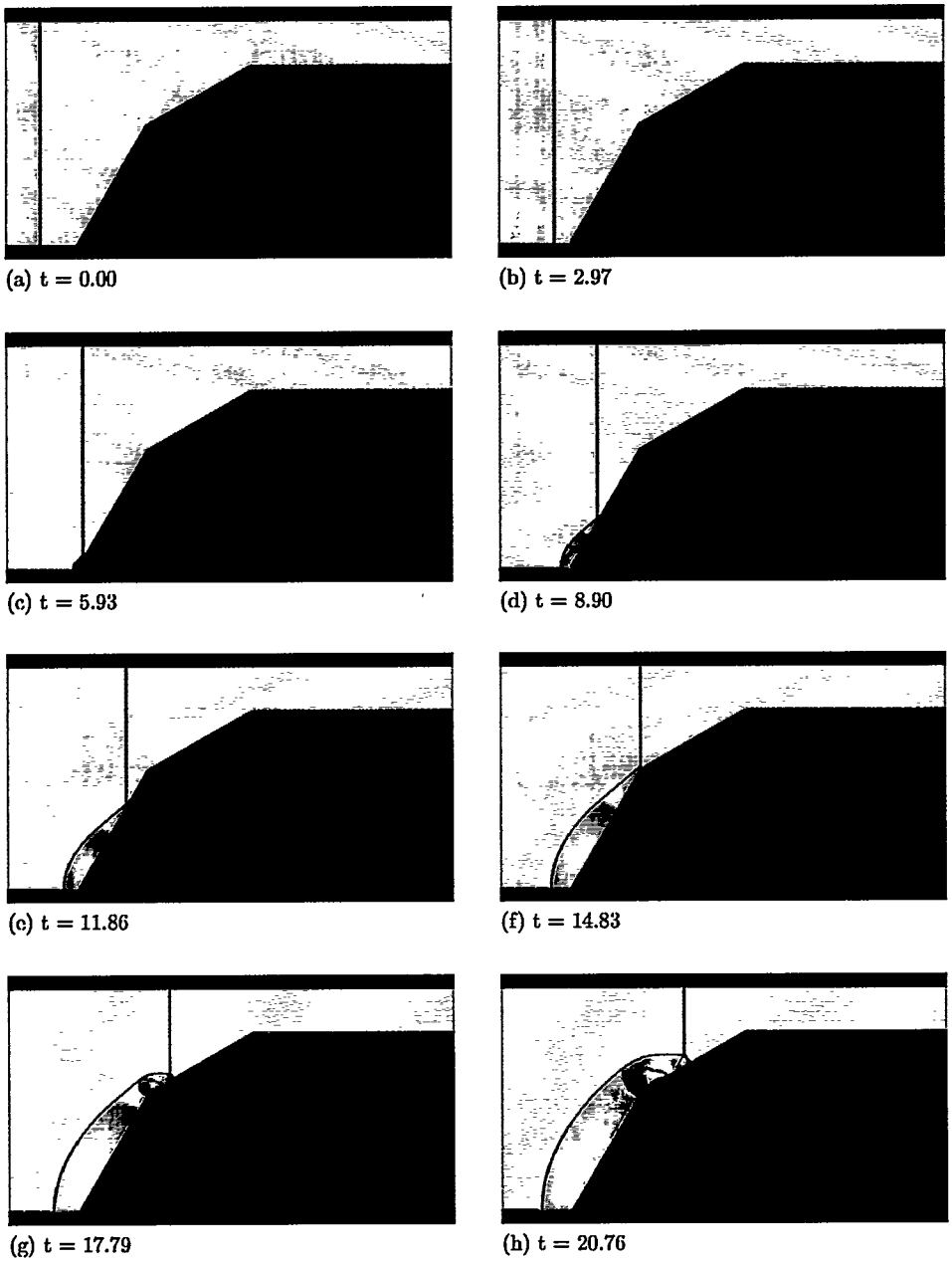


Figure 5: Page output by *paste_tk2*.

Study TK4: $\{\theta_1 = 60, \theta_2 = 30, M_s = 2.16, \gamma = 1.40\}$



(a) t = 0.00 (b) t = 2.97

(c) t = 5.93 (d) t = 8.90

(e) t = 11.86 (f) t = 14.83

(g) t = 17.79 (h) t = 20.76

Figure 6: Page output by the *PostScript* file *ramp/results/TK2/ps/montage.ps* At early times, frames (b) to (c), the slope of the first wedge is sufficient that there is regular reflection (RR) and not SMR as in the other two experiments. At late times, frames (d) to (h), the incident shock diffracts around the convex corner formed by the two wedges.

Study TK4

(a) Experimental Interferogram, courtesy of Prof. Takayama

(b) Numerical Interferogram

Figure 7: Page output by *paste_tk4*.

3 Discussion

The following discussion is restricted to specific observations concerning the development of mesh refinement methods for simulating unsteady shock wave phenomena; descriptions for the underlying algorithmic techniques are available elsewhere, e.g. [20, 33]. The observations are mainly based on simple physical arguments and programming common sense, but are no less useful because of it[19].

3.1 Temporal Refinement

Many mesh refinement schemes give the impression of having been designed solely to minimize the number of grid cells that are required to compute a solution of a given resolution or accuracy. This design philosophy is based on the notion that the effort required to integrate a discretized flow solution decreases as the number of grid cells decreases. But the following example demonstrates that the number of grid cells can have surprisingly little bearing on the cost of performing a time-dependent simulation and so this particular design philosophy is flawed[20].

Consider the propagation of a shock down a uniform mesh of N cells, each of width Δx. If a uniform time step is chosen such that the Courant number based on the speed of the shock is one (hence the shock traverses one cell per time step), it will take N integrations of N cells for the shock to pass through the domain, for a total of N^2 cell updates. Now halve one cell in the grid such that there are $N-1$ cells of width Δx and two of width $\Delta x/2$. Again, if a uniform time step is used to propagate the shock through this domain, without violating the CFL condition it will take $2N$ integrations of $N+1$ cells to propagate the shock through the domain, for a total of $2N^2+2N$ cell updates. Therefore, although but a single cell has been added to the grid the cost of the simulation has more than doubled. Consequently, for time-dependent problems it is desirable to refine in time as well as space[21]. Here, using temporal refinement, the two small cells would be integrated $2N$ times and the other $N-1$ cells would be integrated N times as in the uniform mesh case, for a total of N^2+3N cell updates. Thus, for N reasonably large, the cost of the refinement becomes negligible. As an alternative to temporal refinement one could conceivably opt for an integration scheme which was stable for large Courant numbers, but for highly non-linear problems the loss in temporal accuracy, associated with large time steps, would probably prove unacceptable.

A temporal refinement strategy is easily incorporated into hierarchical refinement schemes such as those based on quad-trees (e.g. [7]) or embedded patches (e.g. [5, 21] since it is possible to avoid ever having to interpolate across discontinuities[21]. However, a temporal refinement strategy seems ill-suited to refinement schemes based on unstructured triangular meshes (as typified by[16]), at least when combined with a shock-capturing methodology, since one cannot avoid having to perform awkward non-linear interpolations at discontinuities. Such interpolations are unlikely to satisfy a shock-capturing scheme's unique smeared shock profile and so would result in spurious oscillations[21]. One convenient way around this difficulty would be to employ an integration scheme based on floating shock-fitting[18, 34] rather than shock-capturing. Then there would be no smeared discontinuities and the cause of the problem disappears. This strategy illustrates an important design principle of mesh refinement methods: it is often better to work around difficulties than to attempt to effect a cure.

[19] In many regards, such observations are more likely to stand the test of time than overly sophisticated arguments. As a case in point, §3.2 – which first appeared in[21] – is perennially useful in dispelling qualms that AMR methods inevitably introduce spurious numerical vorticity: they do so, only when operated incorrectly.

[20] This comment, and the others which follow, are only accurate in the context of time-dependent, compressible flow simulations. Even then, however, you are advised to recall the *Dumas* quote from lecture 1.

3.2 Fine-Coarse Boundaries I

A number of techniques have been devised to lessen the spurious reflections which occur when a numerical shock wave crosses a grid discontinuity (e.g. [28]). In practice, the performance of such remedial procedures is problem dependent; provided that the shock waves are not too strong, and the grid discontinuities are not too severe, then satisfactory results can be obtained, otherwise the "cures" are found wanting. The following thought experiment, taken from [21], suggests that preventative measures are a better design principle than curative measures:

> "Consider the composite grid formed from abutting two uniform rectangular meshes, one mesh being r times finer than the other, and suppose a planar shock wave is allowed to propagate in a direction which runs parallel to this join. All things being well, one would expect the flow to remain one-dimensional. But, if r is large then it is difficult to see how such a two-dimensional simulation could maintain, indefinitely, a one-dimensional flow. For a given shock-capturing scheme the numerical representation of a shock is self-similar with mesh spacing. Therefore, the shock wave on the coarse mesh would be much wider than that on the fine mesh. Consequently, at the foot of the shock there would be a pressure gradient which acts across the grid discontinuity from the coarse mesh to the fine mesh, and at the head of the shock there would be a pressure gradient which acts in the opposite direction. Thus, the grid discontinuity would cause the supposedly planar shock wave to act as a vorticity generator! Even if the rate of production were small, the accumulation would be relentless. So, sooner or later the two-dimensional numerical solution would differ markedly from the expected one-dimensional solution."

This script, which outputs Figure 8, can be used to test the validity of the thought experiment:

```
amrcp   vki/fc.1
amrita  fine_coarse_bdy
cd      ps/roe/split
amrps   phase1.ps
amrps   phase9.ps
```

```
set vki = $amrita::AMRITA/examples/vki
autopath +$vki/lecture2/lib
EulerEquations
plugin amr_sol
set flux  = roe
set plate = yes
if(!&amrso("code/$flux")) then
   BasicCodeGenerator {
      solver = $flux
      scheme = first-order'operator-split
      flux   = bcg/$flux
   }
endif
FineCoarseBoundary Ms=10,splitter_plate=$plate
solver  code/$flux
logfile logs/$flux
postscript on
do phase=1,10
   march 30 steps with cfl=0.8
   flowout  io/$flux/split/phase$phase
   plotfile ps/$flux/split/phase$phase.ps
      PlotShock
end do
```

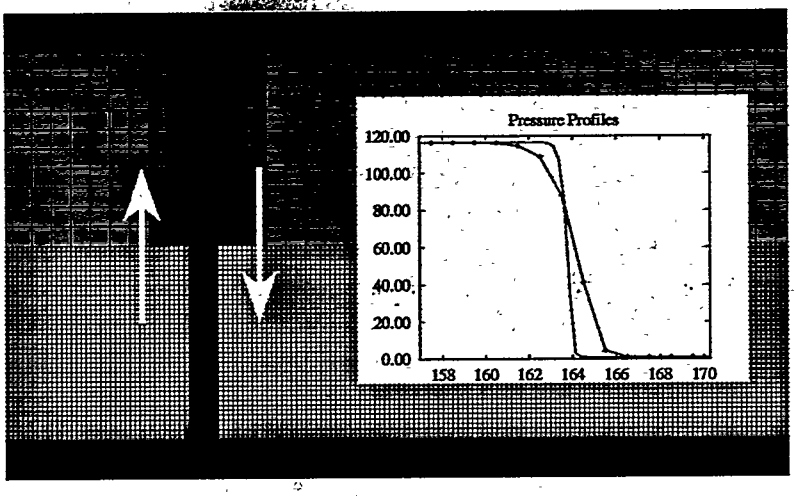


Figure 8: A grid discontinuity can cause a planar shock to act as a vorticity generator. Although here, a solid plate is used to decouple the coarse grid solution from the fine grid solution so as to suppress the generation of vorticity.

The simulation shown in Figure 8 uses a splitter plate (i.e. a solid wall) to decouple the coarse grid solution from the fine grid solution so as to prevent the generation of vorticity. But when the plate is removed:

```
unix-prompt>amrcp vki/fc.2
unix-prompt>amrita fine_coarse_bdy
```

the shock structure rapidly breaks down in the manner shown in Figure 9 and the thought experiment is vindicated. But events, since the test problem was formulated in 1991, indicate that this is not the entire story. The breakdown observed here is particularly bad, because the Roe linearization predicts wave speeds which are too high along the centreline of the duct leading to the shock protrusion[21]. If the test is re-run with `flux = godunov`, the results are better but far from perfect (see Figure 10), and yet a further improvement can be obtained by switching to `flux = efm` (see Figure 11). However, before these results add more fuel to the "Great Riemann Solver Debate"[22], it should be pointed out that a ten line fix can be added to the `roe` flux so as to produce Figure 12. Nevertheless, the quality of results is less than acceptable for simulations of reactive flows, since small peturbations can be amplified chemically to the point where they dominate proceedings[24].

Given the stylized setting of the present split-grid test, it is not inconceivable that some form of "intelligent" interpolation, which took special account of both the local pressure gradient and the local gradient in mesh spacing – knowing the setup of the problem – could prevent the generation of spurious vorticity[22]. But such a "cure" would not generalise. Therefore the practical solution is not to argue about the choice of flux, or interpolation function, but to recognize that the test is artificial: it is more natural to orchestrate the mesh adaption in such a way that a shock wave is not given the opportunity to cross a grid discontinuity[21]. When this is done:

```
unix-prompt>amrcp vki/fc.1e
unix-prompt>amrita fine_coarse_bdy
```

even `flux = roe` can produce the pristine results shown in Figure 13.

[21]Examples of the patterns of data which cause Roe's scheme to give poor results are given in the next section.
[22]This is used in the complete sense: a reduction in spurious vorticity is good, but not good enough.

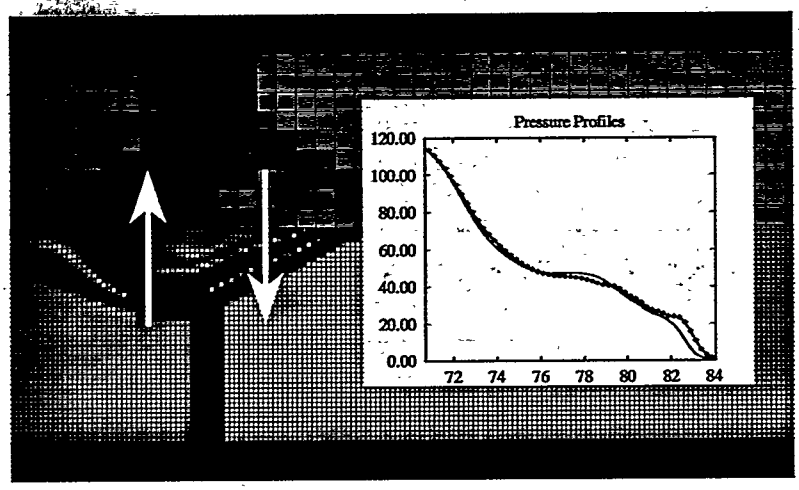


Figure 9: *fine_coarse_bdy* results for `flux = roe`.

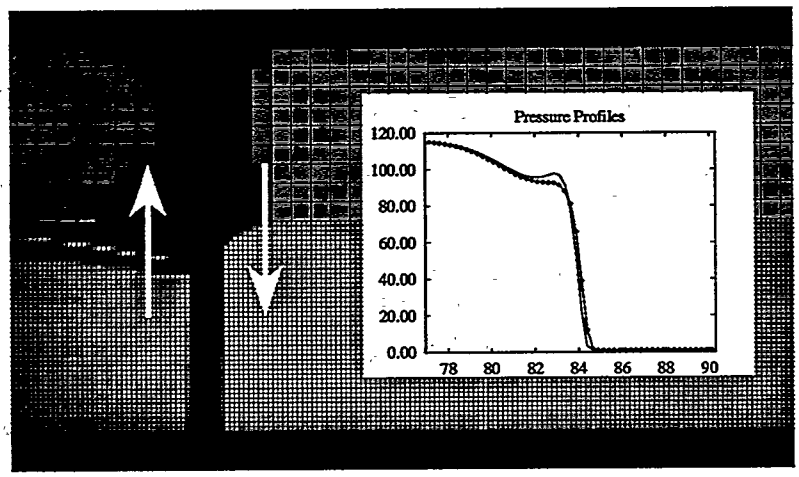


Figure 10: *fine_coarse_bdy* results for `flux = godunov`.

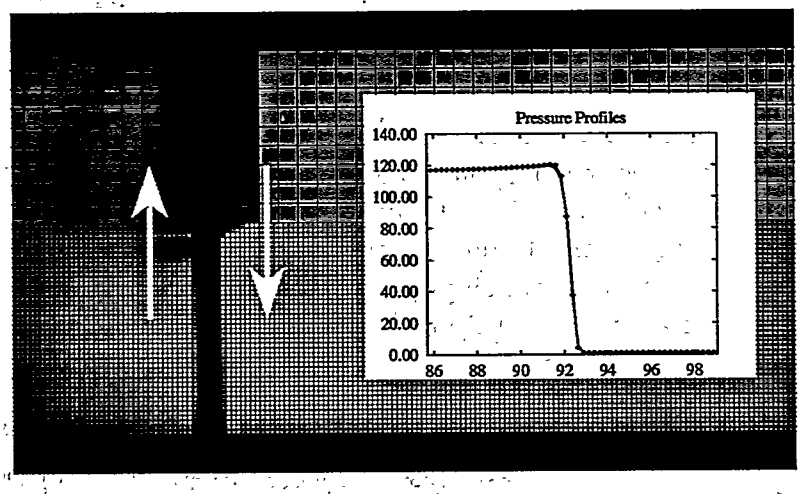


Figure 11: *fine_coarse_bdy* results for `flux = efm`.

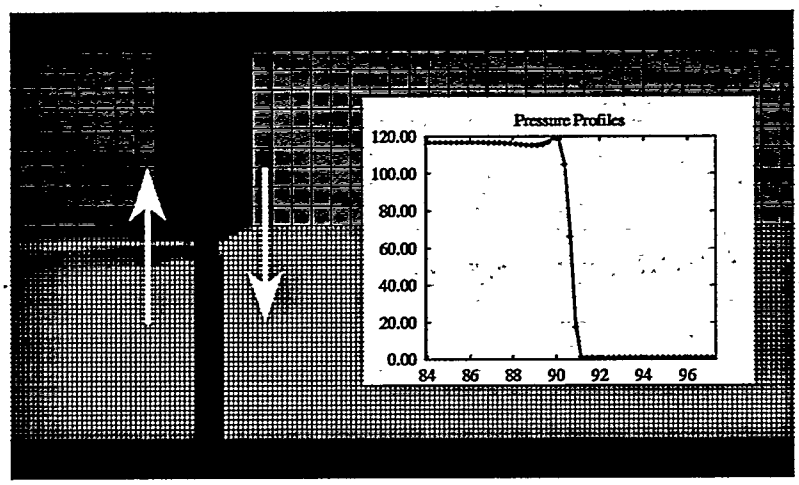


Figure 12: *fine_coarse_bdy* results for `flux = roemk2`.

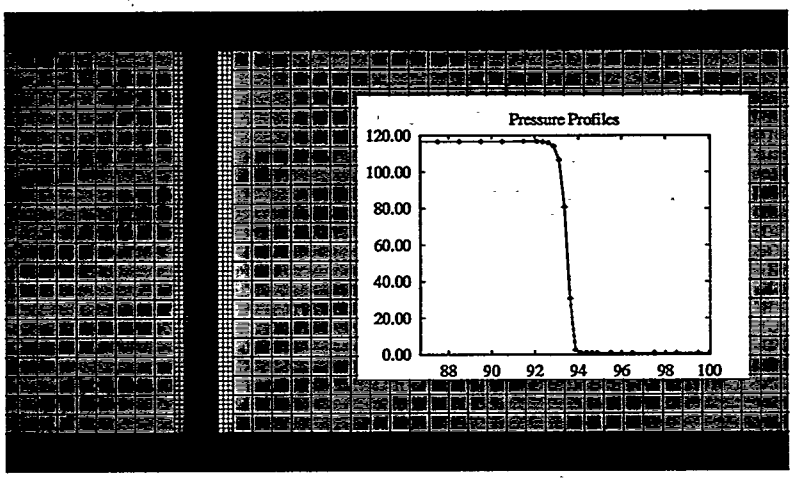


Figure 13: *fine_coarse_bdy* results for `flux = roe` when the grid is adapted to the shock.

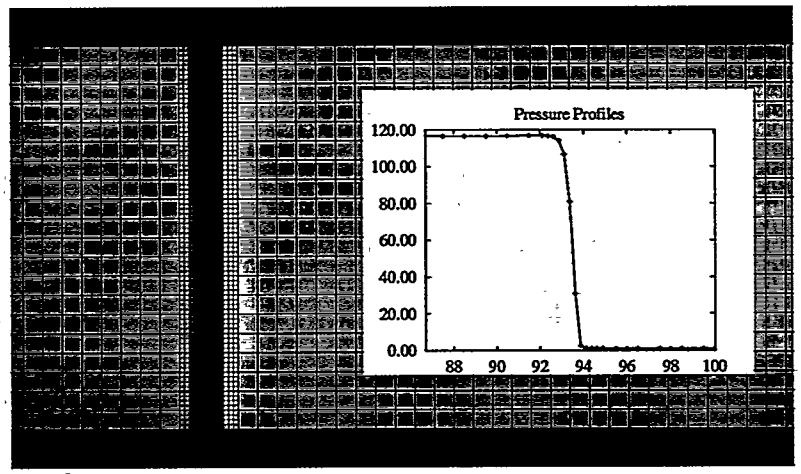


Figure 14: *fine_coarse_bdy* results for `flux = godunov` when the grid is adapted to the shock.

3.3 Fine-Coarse Boundaries II

When *Amr_sol* was first constructed it was deemed necessary to enforce strict conservation at a fine-coarse boundary by applying a fixup pass to the integration process which took strict account of the net difference in the cumulative flux seen by the fine and coarse grids[5, 21]. Subsequent experience shows that this fixup procedure is counterproductive.

First, to preserve monotonicity, it introduces a secondary time step restriction on the maximum allowable Courant number, ν_{max}, used to integrate the flow:

$$\nu_{max} \leq \sqrt{\frac{2}{r-1}}$$

which for refinement ratios $r > 3$ is more restrictive than the standard CFL condition (i.e. $\nu_{max} = 1$)[23], see [21] for details. Second, when running on a parallel machine, it introduces an extra layer of communication which can impact on performance[24]. Third, and in some ways most telling, the fixup procedure is merely a book-keeping mechanism which credits or debits the coarse grid solution to maintain conservation. As such, at least on a finite sized grid, it does not guarantee the consistency and convergence of results needed to ensure waves travel at the correct speed, which is the motivation for using the fixup in the first place[25]. Consequently, *Amr_sol* no longer employs the conservative-fixup pass reported in[21][26].

The following script demonstrates that given the manner in which *Amr_sol* adapts the grid (i.e. only smooth flow is allowed to cross a grid discontinuity), no special treatment need be applied at a fine-coarse boundary to ensure an embedded shock wave travels at the correct speed:

```
amrcp   vki/fc.2
amrita  shock_speed
cd      results/roe-1d
amrps   error.ps
```

```
set flux = roe
EulerEquations space=one-dimensional
plugin amr_sol
BasicCodeGenerator {
   solver = $flux-1d
   scheme = kappa-muscl'operator-split
   flux   = bcg/roe
}
CheckShockSpeed {
   solver     = $flux-1d
   lmax       = 4
   r          = 4
   MachNumbers = 1.2,1.5,2,3,4,5,10,20
}
```

Here, the procedure `CheckShockSpeed`:

[23] If a safety factor is applied to the CFL condition, as is common practice, then the cut off point in refinement ratio will be higher e.g. a CFL of 0.8 allows $r = 4$ to be used.

[24] The amount of data involved is relatively small, but it can have a detrimental affect on load balancing and so lead to unexpectedly large losses in performance.

[25] The next section provides a concrete example of why conservation should not be viewed as a panacea. Also, Whitham[36] provides a nice example, using the shallow water equations (SWE), which shows the difference between casting equations in conservation form (the SWE have an infinite number of such formulations) from the unique, weak-formulation needed to predict shock jumps correctly.

[26] But this does not imply that conservation has been relaxed elsewhere: it is still important for the integration of the flow solution held by a patch and for the interpolation operators which transfer the flow solution from old patches to new patches when the grid adapts.

```
proc CheckShockSpeed {
     solver
     MachNumbers
     Xs         = 28.00  # shock position
     lmax       =  0     # grid levels
     rI         =  2     # refinement ratio
     nG0        = 10     # patches in G0
     nphases    = 20
     nsteps     = 100
     cfl        = 0.8
     io        "= results/$solver
   } <-> S::
   ...  procedure definitions
   solver  code/$solver
   logfile logs/$solver
   ...  setup $io/error
   foreach Ms ($MachNumbers)
      SetupTest Ms=$Ms
      ...  setup $io/Ms$Ms.xt
      do phase=1,$nphases
         march $nsteps steps with cfl=$cfl
         along y=0 locate last RHO[]>$S::RHOt -> xs
         time -> t
         printf("%.8f %.8f\n",$xs,$t)
      end do
      set Xf    #= $Xs+$Ms*sym(C'quiescent)*$t
      set error #= ($Xf-$xs)/($Xf-$Xs)*100
      set error #= $error*1000
      ...  plot Ms results
   end foreach
   GraphError {
      odir      = results/$solver
      output    = error.ps
   }
end proc
```

marches the shock profile on G_{lmax} for 512,000 time steps [27]. The %error in shock location – scaled by a factor of 1000 – is shown in Figure 15. Here the largest recorded error occurs for the weakest shock and is only 10 parts in 10^5 and is comparable to the uncertainty in locating the shock position using the `along` command[28]. For a given Mach number, this uncertainty is of fixed size, therefore the recorded %error increases as the travel distance of the shock decreases.

The above highlights one important *Amr_sol* design principle: accountability. The underlying AMR algorithm does not have a concise recipe and is non-trivial to code. Therefore, it is essential to avoid excess baggage: if a component does not earn its keep, it is jettisoned. *Amrita* is designed to facilitate the testing of components in an automated, objective manner

[27] The solution on G_0 is integrated in 20 phases of 100 time steps, but folowing the time-stepping procedure in §A, with a fixed refinement ratio of 4, G_l is integrated 4^l times for every G_0 integration: $20 \times 100 \times 4^4 = 512,000$. If you have limited computing resources, you can run *shock_speed* with `lmax=2` and `nsteps=20` for a total of 6,400 time steps.

[28] The weakest shock is the most smeared out in terms of mesh cells and so has the greatest uncertainty in location which helps explain why it gave the largest recorded error.

to allow critical design decisions to be vouchsafed through masked scrutiny. Consequently, the decision to drop the conservative-fixup pass is not irreversible and the fixup would be resurrected should a sufficient number of *mailit* files emerge to support its reinstatement. However, criticisms must be made in the operational context shown by this script which outputs Figure 15:

```
amrcp  vki/fc.3
amrita fc_context
amrps  ps/fc_context.ps
```

```
EulerEquations
plugin amr_sol
flowin io/Corner5
autoscale
postscript on
plotfile ps/fc_context.ps
SchlierenImage
AmritaBlue
filled rectangle 0,0,30,40
ps>15 setlinewidth
m<1>
plot domain {G2}
```

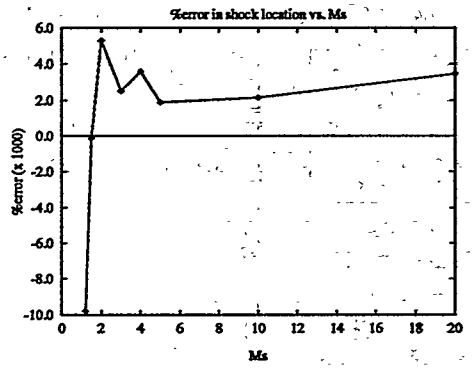
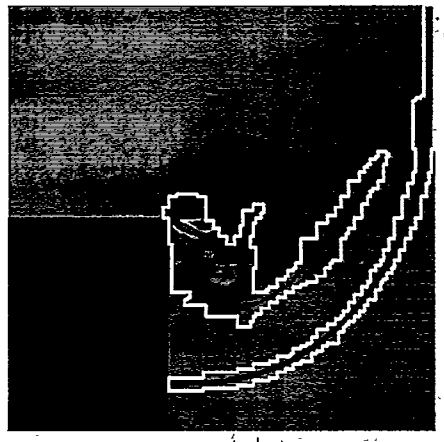


Figure 15: Graph of %error in shock location versus Mach number for a shock integrated 512,000 time steps. Note the %error is multiplied by a factor of 1000 to make the scale easier to read.



(a)

Figure 16: Output from *fc_context*. *Amr_sol* is designed to encase shocks using a seamless, collection of rectangular mesh patches, such that to all intents and purposes, the shock sees a uniform grid.

3.4 Flow Solvers

The early development of *Amr_sol* was plagued by a series of obscure numerical failings which afflict shock-capturing schemes[22]. In the end, these failings were tracked down not so much by analysis, but by searching out pathological patterns of flow data, in much the same way as tracking down a bug in a program. This *mailit* which outputs Figures 17 and 18 and can be used to hunt for such failings[29]:

```
unix-prompt>amrcp vki/st.mailit
unix-prompt>amrita st.mailit
```

In Figure 18, the density dip at $X = 50$ is similar in nature to the startup-error discussed in lecture 1. The initial conditions for this shock-tube problem are two impinging shock waves:

```
proc ShockShock Ms=3
   ... st::*
   W'quiescent  ::= <RHO=1,U=0,P=1>
   ShockWave Ms=$Ms, state1=quiescent,\
                     state2=post_shock
   W'left  ::= W'post_shock
   W'right ::= W'post_shock<U=-U'post_shock[]>
   ... st::notes
end proc
```

The exact solution consists of two shocks of equal strength which move away from one another leaving behind stationary fluid as they go. By design, Roe's scheme can recognize a single shock wave, but the linearization used cannot cope with two waves as here. Thus the estimate for the speed of the shocks is wrong by $O(1)$, leading to the error in density.

Using standard notation – see the two given references – because of the symmetry of the initial data:

$$\tilde{u} = \frac{\sqrt{\rho_l}u_l + \sqrt{\rho_r}u_r}{\sqrt{\rho_l} + \sqrt{\rho_r}} = 0$$

and so the shock speed for the first time step, $\tilde{\lambda} = \tilde{u} + \tilde{a}$, is given by:

$$\tilde{a} = \sqrt{\frac{\gamma p_r}{\rho_r} + \frac{\gamma - 1}{2}u_r^2} = 2.265$$

This over-predicts the exact speed:

$$\lambda = \frac{(\gamma+1)u_r + \sqrt{(\gamma+1)^2 u_r^2 + 16a_r^2}}{4} + u_r = 1.446$$

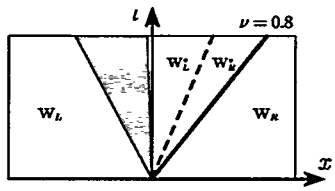
Therefore, in the shock frame of reference, the post-shock velocity is too high, indicating the shock is too weak. Hence the predicted density is lower than the exact solution. The numerical solution never recovers from this first step because the error appears on the contact wave, and since the flow velocity is everywhere zero behind the shock wave no dissipation is added[30] to damp out the error at the "wall" (i.e. line of symmetry). The `roemk2` flux used in §3.2 achieves its robustness by the expediency of artificially increasing the velocity of a stationary contact wave from 0 to ϵ to ensure that some dissipation is added to prevent pathologies developing in the flow solution. Thus `flux = roemk2` performs better than `flux = roe` for the `ShockShock` problem.

[29] These figures show just two pages from the voluminous output produced by the *mailit*. For convenience purposes, as with the *la.mailit*, the output is collated in the form of an *HTML* document.

[30] In Roe's scheme, dissipation is directly proportional to wave speed.

Shock Tube Test: "Sod's problem"

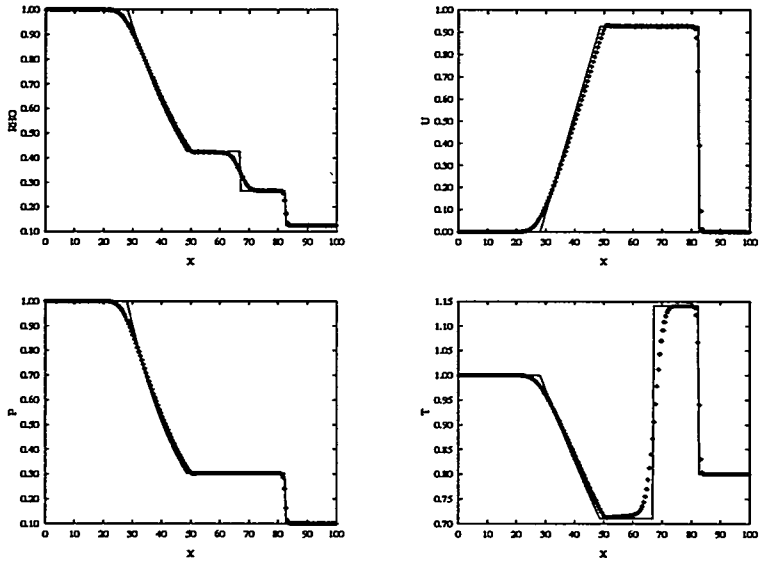
ρ_l	= 1	ρ_r	= 0.125
u_l	= 0	u_r	= 0
p_l	= 1	p_r	= 0.1
γ_l	= 1.4	γ_r	= 1.4



The classic shock-tube problem, first used by G. A. Sod (*J. Comput. Phys.* **27**, 1-31, 1978).

Scheme euler-code::1d-c-fo-os

Flux roe



References

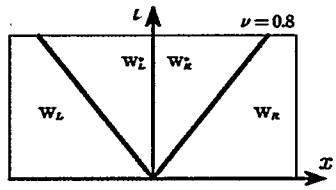
[1] ROE, P. L. 1981 Approximate Riemann Solvers, Parameter Vectors, and Difference Schemes. *J. Comput. Phys.* **43**, pp. 357–372.

[2] ROE, P. L. AND PIKE, J. 1984 Efficient Construction and Utilisation of Approximate Riemann Solutions. *Comput. Math. Appl. Sci. & Eng.* **VI**, eds. Glowinski, R. and Lions, J-L., pp. 499–518.

Figure 17: Page output by *shock_tube*.

Shock Tube Test: Shock-Shock

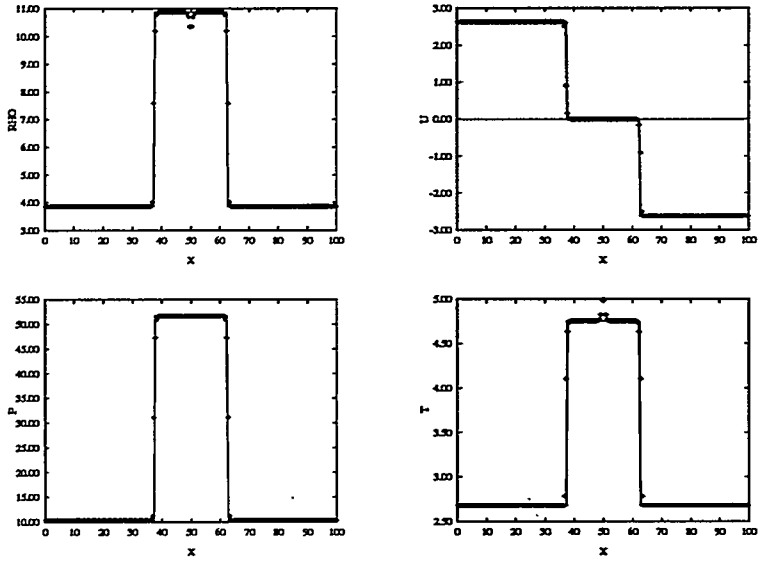
ρ_l	=	3.857	ρ_r	=	3.857
u_l	=	2.629	u_r	=	−2.629
p_l	=	10.33	p_r	=	10.33
γ_l	=	1.4	γ_r	=	1.4



Excessive "wall heating" can be produced by numerical shock-shock ineractions.

Scheme euler-code::1d-c-fo-os

Flux roe



References

[1] ROE, P. L. 1981 Approximate Riemann Solvers, Parameter Vectors, and Difference Schemes. *J. Comput. Phys.* 43, pp. 357–372.

[2] ROE, P. L. AND PIKE, J. 1984 Efficient Construction and Utilisation of Approximate Riemann Solutions. *Comput. Math. Appl. Sci. & Eng.* VI, eds. Glowinski, R. and Lions, J-L., pp. 499–518.

Figure 18: Page output by *shock_tube*.

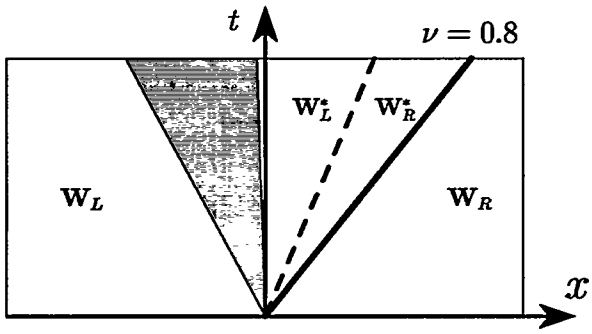


Figure 19: Solution to the Riemann problem $\{\mathbf{W}_L, \mathbf{W}_R\}$.

The following consideration of the Riemann problem shown in Figure 19 suggests that Roe's scheme is susceptible to problems near shear waves[21]. For the Euler equations, the generic solution to a Riemann problem $\{\mathbf{W}_L, \mathbf{W}_R\}$, where two semi-infinite states $\mathbf{W}_L = (\rho_l, u_l, v_l, p_l)^t$ and $\mathbf{W}_R = (\rho_r, u_r, v_r, p_r)^t$ are prescribed at $t = 0$, consists of three waves separating four regions. The two outer waves, are acoustic waves which can either be shocks or expansions that match the left- and right-states to a common pressure, p^*, and a common normal component of velocity, u^*. The inner wave is a contact surface, or slip line, which accounts for any differences in density, $\rho_r - \rho_l$, and shear velocity, $v_r - v_l$, between the left- and right- states. Now suppose that the prescribed tangential component of velocity v_r is replaced by $-v_r$, and call this new state $\mathbf{W}'_R$. The exact solution to this new Riemann problem $\{\mathbf{W}_L, \mathbf{W}'_R\}$ is the same as before, with the exception that $v_r^* = v_r' = -v_r$. But, as an inevitable consequence of the linearization process, for Roe's approximate Riemann solver the form of solution to these two problems will be very different. Specifically, the average:

$$\tilde{v} = \frac{\sqrt{\rho_l} v_l + \sqrt{\rho_r} v_r}{\sqrt{\rho_l} + \sqrt{\rho_r}}$$

must differ from $\tilde{v}'$ and therefore the acoustic wave speed:

$$\tilde{a} = \sqrt{(\gamma - 1)\tilde{h} - \frac{1}{2}\tilde{u}^2 - \frac{1}{2}\tilde{v}^2}$$

must differ from $\tilde{a}'$, and so on for the wave strengths, thus altering the entire solution. If a change in the prescribed shear velocity can result in a difference between the approximate solutions for the two Riemann problems that is not reflected in the two exact solutions, then the accuracy of the approximate solver is clearly sensitive to the prescribed data. Therefore, since v_r can be made arbitrarily large in relation to u_r, the error in the approximate solution can be made arbitrarily large.

As was done in §3.2, this next example is intended to cut short any unproductive Riemann-solver debate that may be generated by the above shear-wave revelation. Consider the initial conditions shown in Figure 20. At t^0 a shear wave is coincident with one mesh interface of a one-dimensional, finite-volume grid. Now suppose this discrete solution is advanced using a conservative discretization to produce the solution shown at t^1. Assuming the CFL condition is satisfied, the shear wave will fall short of the next grid interface[31] and so introduce a smeared cell containing the state (ρ^*, u^*, v^*, p^*).

[31]With a CFL of 1 the shear wave will reach the next interface, but in practice this special case is less likely to occur than a CFL<1.

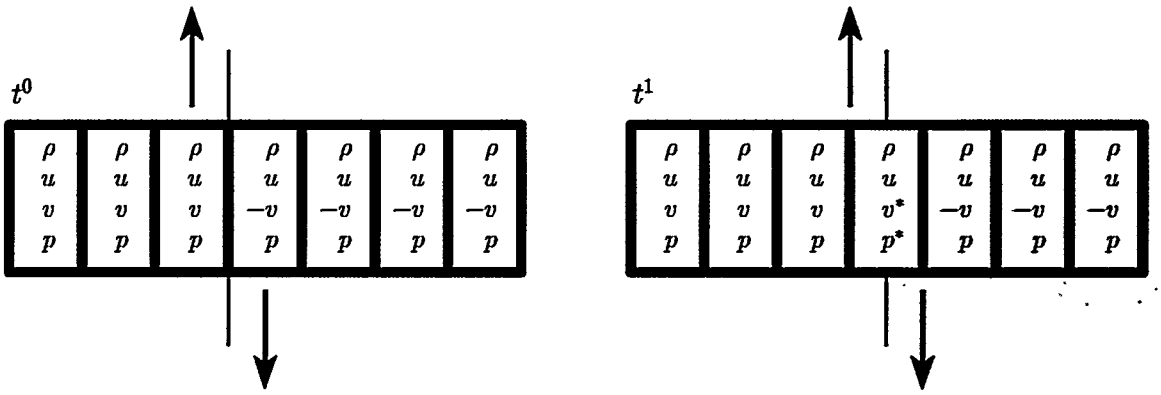


Figure 20: Schematic showing a shear-wave propagating along a one-dimensional, finite-volume grid.

Because a conservative discretization is assumed:

$$\begin{aligned} \rho^* &= \rho \\ \rho^* u^* &= \rho u \\ \frac{p^*}{\gamma-1} + \frac{1}{2}\rho u^{*2} + \frac{1}{2}\rho v^{*2} &= \frac{p}{\gamma-1} + \frac{1}{2}\rho u^2 + \frac{1}{2}\rho v^2 \end{aligned}$$

and so *regardless* of the choice of scheme:

$$p^* - p = \frac{1}{2}\rho(v^2 - v^{*2})$$

Now unless the scheme employs some form of sub-cell resolution, the shear must smear numerically, that is:

$$-v < v^* < v$$

and so $p^* > p$, *regardless* of the choice of flux function. The worst case, which is likely to be the practical case, is that the shear smears symmetrically leading to $v^* \approx 0$, giving:

$$p^* - p = \frac{1}{2}\rho v^2$$

Again, since v is a prescribed quantity, it can be made arbitrarily large resulting in an arbitrarily large, spurious pressure p^*.

The above might appear to be a contrived example, but the numerical difficulty arises whenever there is a component of the total energy which is passively advected with the flow (e.g. [8] or [26]). The solution options are: (i) relax conservation locally; (ii) employ sub-cell resolution to prevent the interface from smearing; (iii) use front-tracking in preference to shock-capturing; (iv) pre-smear troublesome interfaces. Solution (iv) is probably the most widespread, because it is the easiest to implement. Solution (i) is often done unwittingly (e.g. [37]) and so the problem never surfaces.

Whatever the preferred solution approach, the shear-wave pathology illustrates the dangers of blindly following the concept of conservation. In the context of *Amr_sol*, when performing high-resolution simulations, it pays to watch out for nuances in the flow solution which might have severe consequence further down the line. For this reason, it is important to perform mission-critical simulations using as many different, disparate numerical techniques as you can afford, as this facilitates the process of distinguishing numerical-fancy from physical-fact. For this reason alone, many of the algorithmic arguments put forward – concerning the relative merits of shock-capturing schemes – are academic in the worst sense of the word.

3.5 Grid Efficiency

To pick up from where §3.1 left-off, leaving aside the issue of temporal refinement, minimizing the number of grid cells will not automatically lead to an efficient method of refinement. Consider the case of an isolated discontinuity which runs oblique to the grid, as shown in Figure 21. It is clear that cellular quad-tree refinement (say [7]) is more efficient than embedded patch refinement (say [21]) in terms of the number of cells each method requires to tile the discontinuity. However, it also has the larger storage overheads per mesh cell of the two associated data structures. For inert shock wave simulations, which generally need only a small number of levels of refinement, the storage overheads from quad-tree refinement are easily tolerated, but this might not be the case, if the flow contained chemical reaction.

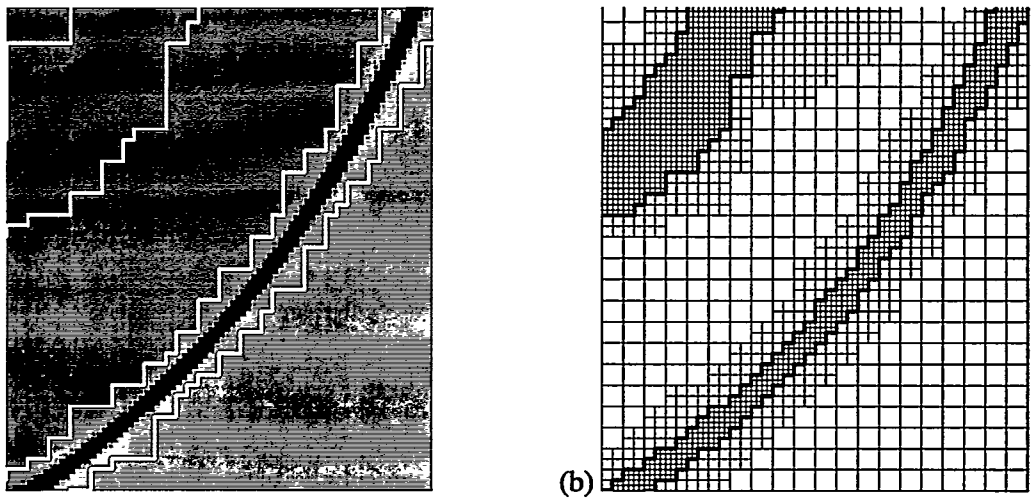


Figure 21: Output from *oblique_shock* (**`amrcp vki/fc.3z`**) showing a zoomed region of Figure 16. Plot (a) shows the `domain {G2}` (i.e. the outline of the finest grid tier) drawn in white. Plot (b) is a simulation (using very small patches) of how a cellular refinement scheme might be able to resolve the flow features; the equivalent `domain {G2}` is shown by the bold black lines. Clearly, compared to a cellular refinement scheme, there is room for improvement in the efficiency with which *Amr_sol* tiles a shock which runs oblique to the grid.

Instead of a shock, consider a detonation wave oblique to the grid, which in addition to a shock front has some internal structure (see §E.3), albeit on a very fine scale, which must be resolved and cannot be captured. In this instance, a wide swathe of cells would be needed to cover the reaction zone which might be ten or more levels of refinement down in the quad-tree, because of the disparateness between the width of the reaction zone and the distance over which the detonation wave needs to be propagated. Therefore, although the cells in the swathe are close to one another spatially, they could lie far apart in the grid data-structure. Not only would this impact on cache performance, and increase communication traffic in a parallel implementation of the scheme, but each cell would introduce a sizeable overhead due to the accumulation of pointers down to its level in the data structure. Consequently, embedded patch refinement might now prove to be more efficient, because its storage overheads would be that much lower and it would better preserve the proximity of cells within the reaction zone.

Adaptive mesh refinement algorithms, compared to classical numerical methods, entail writing sophisticated software. Therefore arguments, such as the one above, must be tempered by the realization that specific implementation details can make or break an algorithm in terms of its practical performance. In particular, the grid data structure needs to be well crafted. For

example, the data storage needs to be flexible enough to cope with dynamic allocation and deallocation as local refinement is added and removed, and data accesses have to be efficient so as not to impact on performance. Since it is all too easy to underestimate the level of commitment needed to write, test and debug a general purpose mesh refinement code, a newcomer would be well advised to take his or her own software skills in to account before choosing to code up any one particular method.

At times, the number of considerations appear legion, even when the application needs are fairly specific. For instance, given the results from §2, it would appear that *Amr_sol* is well suited to time-dependent simulations of shock wave phenomena. But suppose you were dissatisfied with the quality of the results shown in Figure 5 and wanted to perform a viscous simulation, it remains unclear just how well *Amr_sol* would perform.

In the past, it has been successfully used to perform viscous simulations of shock-boundary layer interactions[21], and so there is no reason to believe that it could not cope with a viscous simulation of study `TK2`. However, since viscous flow features tend to be anisotropic in nature, such a simulation would expose a weakness of the refinement scheme: it does not cope that well with anisotropic refinement. The method used[21] is essentially limited to features such as boundary layers which are affixed to solid surfaces. To refine a free shear layer which might happen to lie oblique to the mesh, *Amr_sol* would be forced to use isotropic refinement which would be needlessly expensive. This is an example where a change in the flow model can have a significant impact on the refinement efficiency, even though the application remains unchanged. Thus the correct choice of refinement strategy is never straightforward.

To complicate matters even further, interplays between the method of refinement and the method of flow integration cannot be ignored. For instance, a triangular unstructured mesh has the geometric flexibility to allow for efficient anisotropic refinement but a certain amount of care must still be taken to generate meshes which are suitable for viscous simulations[19]. Sometimes, depending on the application, it is necessary to compromise the refinement efficiency so as to avoid compromising the accuracy of the flow integration (or vice versa). Of course, the accuracy of a refinement scheme is, for the most part, ordained by the monitor functions which determine where refinement does or does not take place.

3.6 Refinement Criteria

As outlined in §G, *Amr_sol* employs heuristic monitor functions to determine where to refine[32]. For instance the double-wedge simulations use a combination of two monitor functions: density gradients locate shocks and a local comparison between density and pressure gradients locate contact discontinuities. Now there are numerous reasons why this type of heuristic approach is unsatisfactory, not least of which is that it introduces tunable parameters and so increases the experience factor needed to operate a refinement scheme reliably (§G.1). As Warren *et al.*[35] have shown, a poorly constructed heuristic monitor function can cause a mesh refinement scheme to home in on an incorrect solution[33]. But this can happen with any refinement function, heuristic or not, which provides estimates for the local error without also providing estimates for how the local error affects the global error i.e. every refinement function in common use. To a large extent the mesh refinement community has been lulled into a false sense of security by the general experience that local errors are usually benign. The test case discussed in [35] is a gentle reminder that small local errors can sometimes tip the balance and result in large global errors, but other more pathological examples are not difficult to find, especially where chemical reaction is involved.

Figure 22 (a) shows a trace of the pressure behind the lead shock front of a one-dimensional detonation wave, driven by the 3-step chain-branching reaction given in §E.3, see [30] for details. By normal standards, this computation would be thought well resolved, since 160 mesh points cover the so-called reaction half-length (giving some 256,000 cells over the time period shown), whereas contemporary simulations have ten or less points in the reaction half-length. However, when the simulation is repeated with the grid spacing halved, the dynamic behaviour of the detonation wave alters dramatically, see Figure 22 (b). At first glance, Figure 22 (b) appears to be from the coarser computation, since it looks more dissipative in that a two mode pulsation is decaying to a single mode pulsation. But in fact it is the extra dissipation in Figure 22 (a) that sustains a spurious two mode pulsation, whereas the correct behaviour should be that of a two mode pulsation with a time-attractor limit cycle[30], i.e. Figure 22 (b). Interestingly, as observed in §E.3, the difference in behaviour arises not from an error in resolving the detonation shock-front, but from a failure to resolve a seemingly innocuous part of the reaction zone which is smooth.

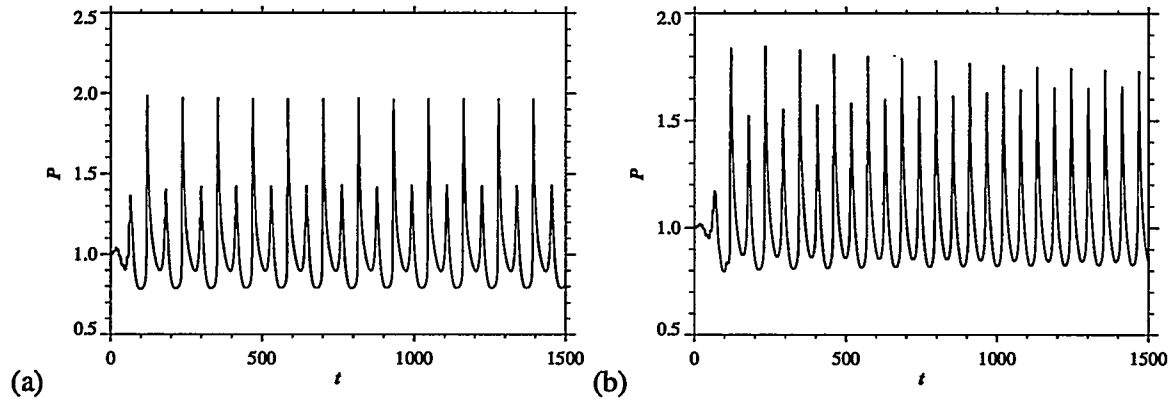


Figure 22: Variation in the computed pressure history trace for a galloping detonation driven by a 3-step chain-branching reaction model[30]: (a) 160 pts/$L_{\frac{1}{2}}$; (b) 320 pts/$L_{\frac{1}{2}}$.

[32]Coarsening takes place naturally by choosing not to refine and so involves no additional criteria[21].

[33]It is worth noting that an undivided-difference when used by *Amr_sol* has an implicit constant length scale, and so does not suffer from the precise problem discussed in [35] for unstructured triangular meshes.

Clearly there is much room for improvement in the current crop of criteria used to control refinement. However, any attempts at devising rigorous mathematically based refinement criteria should not ignore the operation of the underlying grid adaption algorithm. For example, in detonation simulations it can be necessary to adapt the grid tens of thousands of times[25] and so the method of determining where to refine must be reasonably cheap so as not to cripple the performance of the simulation. Also, the physical scales involved are so disparate they preclude the luxury of periodically comparing the solution computed with refinement against that computed on a uniform mesh of the same high resolution, as is effectively done in[13], because of the unrealistically large amount of storage involved.

For practical purposes the lack of a fool-proof refinement criteria does not undermine the usefulness of adaptive mesh refinement schemes for investigating shock wave phenomena, but it does complicate matters. The practical solution, when starting a new investigation, is to perform a sensitivity study to see how the computed results vary with, amongst other things, the effective resolution of the computational grid as controlled by the chosen refinement criteria. The aim is to tool-up to a position where a reliable simulation can be produced. This approach is not just restricted to simulations which use mesh refinement. General experience shows that past performances are no real guide as to how a specific numerical scheme will fair on a new problem. Therefore, the results from any new CFD simulation, regardless of the solution method involved, should be viewed with a healthy degree of scepticism until the results have been shown to be reliable.

For serious investigations the cost of tooling is generally spread over a parameter study and so is not excessive. The only drawback is that the results from grid sensitivity studies are rarely conclusive. Many shock wave phenomena exhibit physical instabilities and so the notion of a grid converged solution is not always clear, or even appropriate since the flow model might preclude the possibility of having a sensible solution in the limit of the mesh spacing going to zero. For example, in [23] results are presented for the vortex sheet produced by a shock wave diffracting over a knife edge. These results show that an inviscid simulation can reproduce the correct physical behaviour and yet provide no limiting solution, because the numerical dissipation which controls the fine scale structure of the vortex sheet, in the absence of physical viscosity, never bottoms out as the grid is refined. On the other hand, in some simulations of detonation phenomena, it is clear that it not practical to reach a fully converged solution, either because the physical scales are too disparate for the available computing resources, or the physical behaviour of the system is non-deterministic in that variations in discretization errors, no matter how small, lead to significant variations in dynamical behaviour.

The majority of CFD simulations are performed with the aim of producing quantitative answers to well understood problems, in which case the above vagaries are unacceptable. In contrast, *Amr_sol* is typically used a qualitative diagnostic in an attempt to fathom behaviour which is not known, and so a certain amount of subjectivity cannot be avoided. And this often involves using *Amr_sol* to perform simulations which are more detailed than would otherwise be possible. Consequently, no attempt is made in this discussion to "sell" the method in terms of how efficiently it was able to compute the double wedge problems. While this might be viewed as contrary, any results which could be presented would have little practical value. Moreover, by comparison to other work[25] the present simulations are so cheap as to be almost inconsequential. It should also be appreciated that the cost of performing a time-dependent simulation can pale into insignificance when compared to the time taken to decipher the results, and so to bandy performance figures loses sight of the fact that *Amr_sol* has progressed well beyond the development stage and is used as an everyday tool.

4 Closing Comments

To close, it is clear that many theoretical aspects of adaptive mesh refinement algorithms require further investigation, e.g. the rigorous control of errors via well founded refinement criteria, or when running on a parallel machine, the performance bounds on load balancing strategies. But the present theoretical shortcomings of *Amr_sol* do not undermine its usefulness as an investigative tool. Moreover, it is worth noting that the algorithm requires little mathematical respectability of its own, because it is designed to subsume the stability characteristics of application specific, patch-integrators.

If the next generation of mesh refinement algorithms are to offer substantial improvements over existing methods – to this author at least – it seems essential that common ground be found between theoreticians and practitioners. Thus, in the case of *Amr_sol*, *Amrita* was constructed to facilitate third-party contributions which might help reduce the current heuristic elements to more acceptable levels.

Acknowledgements

This work was supported by Los Alamos National Laboratory – subcontract 319AP0016-3L under DOE Contract W-7405-ENG-36. I am grateful to Prof. K. Takayama for providing me with the experimental interferograms shown in Figures 3, 5 and 7, and I am happy to acknowledge the efforts of Dr. H. Babinsky in this matter.

A `plugin` *Amr_sol*

The `plugin` *Amr_sol* is based upon a general purpose Adaptive Mesh Refinement (AMR) algorithm for integrating systems of hyperbolic partial differential equations. This algorithm attempts to reduce the costs of a simulation by matching the local resolution of the computational grid to the local requirements of the solution being sought. For example, in simulations of gas dynamic flows, a fine mesh would be used only in the vicinity of shock waves and other flow discontinuities, leaving a coarse mesh elsewhere. Although the savings which accrue from this technique are entirely problem dependent, they can be every bit as attractive as those gained from using expensive parallel computers (savings of more than five hundred-fold have been obtained for simulations of detonation phenomena, Quirk 1996). The foundations of the present AMR algorithm lie with the works of Berger & Oliger (1984) and Berger & Colella (1989), but the derivative outlined here is due to Quirk (1991, 1996). Contemporaneous AMR work is listed in the references.

A.1 Overview of AMR Algorithm

The AMR algorithm employs a hierarchical grid system. In the following, the term "mesh" refers to a single topologically rectangular patch of cells and the term "grid" refers to a collection of such patches. At the bottom of the hierarchy a set of coarse mesh patches delineates the computational domain. These patches form the grid G_0 and they are restricted such that there is continuity of grid lines between neighbouring patches. This domain may be refined locally by embedding finer mesh patches into the coarse grid G_0. These embedded patches form the next grid in the hierarchy, G_1. Each embedded patch is effectively formed by subdividing the coarse cells of the patches that it overlaps. The choice for the refinement ratio is arbitrary, but it must be the same for all the embedded patches. Thus, by construction, the grid G_1 also has continuity of grid lines. This process of adding grid tiers to effect local refinement may be repeated as often as desired, see Figure 23.

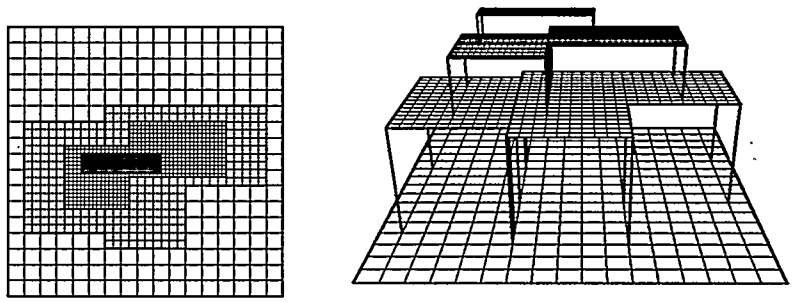


Figure 23: *Amr_sol* employs a hierarchical grid system.

From stability considerations, many numerical schemes have a restriction on the size of time step that may be used to integrate a system of equations. The finer the mesh, the smaller the allowable time step. Consequently, the AMR algorithm refines in time as well as space. More but smaller time steps are taken on fine grids than on coarse grids in a fashion which ensures that the rate at which waves move relative to the mesh (the Courant number) is comparable for all grid levels. This avoids the undesirable situation where coarse grids are integrated at very small Courant numbers given the time step set by the finest grid's stability constraints: some schemes (e.g. Lax-Wendroff) give poor accuracy for small Courant numbers.

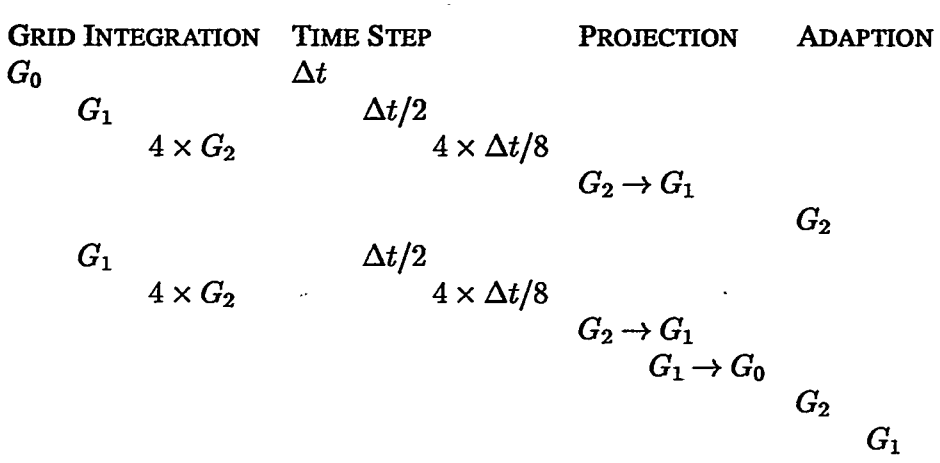


Figure 24: Grid operations are recursively interleaved (to be read from top to bottom).

The field solution on each grid is retained even in regions of grid overlap and so all grid levels in the hierarchy coexist. The order of integration is always from coarse to fine since it is necessary to interpolate a coarse grid solution in both time and space to provide boundary conditions for its overlying fine grid. The various integrations at the different grid levels are recursively interleaved to minimize the span over which any temporal interpolation need take place. Periodically, for consistency purposes, it is necessary to project a fine grid solution on to its underlying coarse grid. Figure 24 shows the sequence of integration steps and back projections for a three level grid $\{G_0, G_1, G_2\}$ with refinement ratios of 2 and 4.

The integration of an individual grid is extremely simple in concept. Each mesh is surrounded by borders of ghost cells. Prior to integrating a grid, the ghost cells for every mesh patch in the grid are primed with data which is consistent with the various boundary conditions that have to be met. Each mesh patch is then integrated independently by an application dependent, black-box integrator that never actually sees a mesh boundary. In principle any cell-centred scheme developed for a single topologically rectangular mesh could form the basis for the integration process.

In general it is necessary to adapt the computational grid to the changes in the evolving flow solution and so the grid structure is dynamic in nature. Monitor functions based on the local solution are used to determine automatically where refinement needs to take place to resolve small scale phenomena (Quirk 1991). For a simple example, Figure 25 shows several snapshots taken from the simulation of a shock wave diffracting around a corner. Each snapshot shows the outlines of the mesh patches which go to make the finest grid. This grid clearly conforms to the main features of the flow, namely the diffracted shock front and the vortex located at the apex of the corner (van Dyke 1982). Although the changes in grid structure shown here are dramatic, many adaptions have taken place between each frame (the mesh patches appear small, but each patch actually contains several hundred cells).

A large number of small grid movements occurs because the adaption process dovetails with the integration process, see Figure 24. Observe that the adaption always proceeds from fine to coarse so as to ensure that there is never a drop of more than one grid level at the edge of a fine grid to the underlying coarse grid. A grid adaption essentially produces a new set of mesh patches which must be primed with data from the old set of patches before the integration process can proceed. Where a new patch partially overlaps an old patch of the same grid level, for the region of overlap, data may be simply shovelled from the old patch to the new patch. In regions of no such overlap, the required field solution is found by interpolation from the

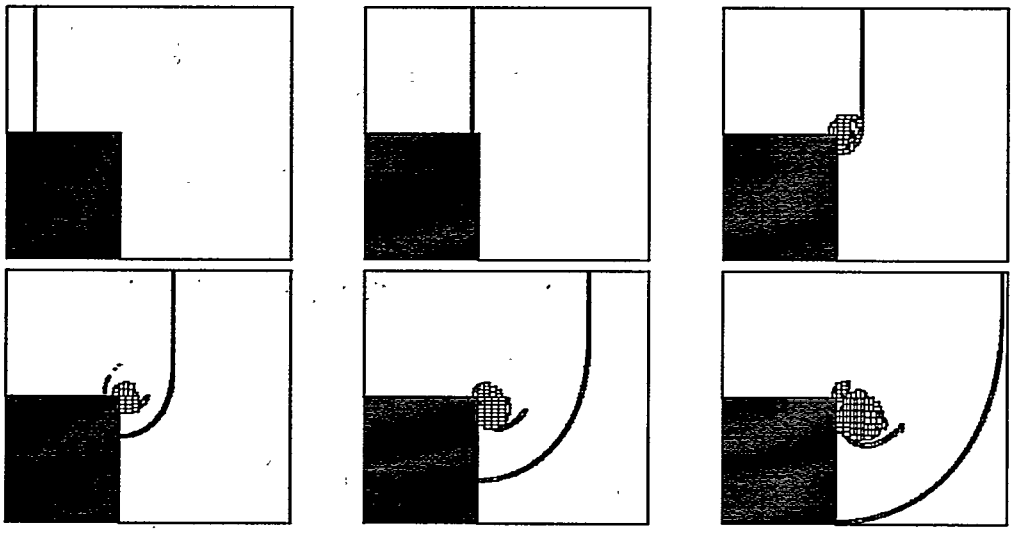


Figure 25: *Amr_sol* employs a dynamic grid system.

underlying coarse grid solution.

In a typical application the finest grid will contain several hundred mesh patches. Consequently the mesh patch is a sufficiently fine unit of data for efficient parallelism. The parallel version of *Amr_sol* (Quirk 1996) is implemented using a Single Program Multiple Data (SPMD) model. Each processing node executes the basic serial algorithm (Quirk 1991) in isolation from all other nodes, except that at a few key points messages are sent between the nodes to supply information that an individual node deems to be missing, that is off-processor. For example, during the integration of a grid, the only point at which a processor needs to know about other processors is during the priming of the ghost cells. Whereas in a serial computation all data fetches are from memory, for a parallel computation some are from memory and some necessitate receiving a message from another processor. Each time the grid adapts, the algorithm generates a schedule of tasks that have to be performed so as to prime correctly the ghost cells of a given grid. If running in parallel, this schedule is parsed to produce a schedule of those tasks that necessitate off-processor fetches. At which point, individual processors can exchange subsets of their fetch schedules, as appropriate, so that every node can construct a schedule of messages that it must send out at some later date. Thus, the priming process is carried out in two phases. First, all the local data fetches are performed as for the serial case. Second, each node sends out the data that has been requested of it. The node then waits for those data items it has requested. For each incoming message it can readily determine from its own schedules what to do with the off-processor data, and so the order in which messages arrive is unimportant. The adaption process and the back projection of the field solution between grid levels also necessitate sizable amounts of communication, these are handled in a similar fashion to the priming of the ghost cells.

The problem of load balancing the AMR algorithm rests on determining the best distribution of the new patches amongst the processing nodes before the new field solution is interpolated from the old field solution. Currently, this is done using heuristic procedures which bear strong similarities to classical "bin packing" algorithms (e.g. Graham 1969) with the added complication that they must account for the communication costs of data transfer between nodes.

References

[1] BERGER, M. J. & OLIGER, J. 1984 Adaptive mesh refinement for hyperbolic partial differential equations. *J. Comp. Phys.* **53**, 482–512.

[2] BERGER, M. J. & COLELLA, P. 1989 Local adaptive mesh refinement for shock hydrodynamics. *J. Comp. Phys.* **82**, 67–84.

[3] GRAHAM, R. L. 1969 Bounds on certain multiprocessing anomalies. *SIAM J. Appl. Math.* **17**, 416–429.

[4] QUIRK, J. J. 1991 An adaptive mesh refinement algorithm for computational shock hydrodynamics. PhD Thesis, Cranfield Institute of Technology, U.K.

[5] QUIRK, J. J. 1996 A parallel adaptive mesh refinement algorithm for computational shock hydrodynamics. *Appl. Numer. Math.* **20**, pp. 427–453.

[6] VAN DYKE, M. 1982 *An album of fluid motion.* Parabolic Press, p. 148.

Contemporaneous AMR Work[34]

[7] ARNEY, D.C. & FLAHERTY, J.E. 1989 An adaptive local mesh refinement method for time-dependent partial differential equations. *Appl. Numer. Math.* **5**, pp. 257–274.

[8] ARNEY, D.C. & FLAHERTY, J.E. 1990 An adaptive mesh-moving and local refinement method for time-dependent partial differential equations. *ACM Trans. Math.* **16**, pp. 48–71.

[9] BELL, J.B., BERGER, M.J., SALTZMAN, J.S., WELCOME M. 1994 Three-dimensional adaptive mesh refinement for hyperbolic conservation laws. *SIAM J. Sci. Comput.* bf 15, p. 127.

[10] BELL, J.B., COLELLA, P., TRANGENSTEIN, J.A. & WELCOME, M. 1987 Adaptive methods for high Mach number reacting flow. AIAA Paper 87-1168.

[11] BERGER, M.J. & SALTZMAN, J.S. 1994 AMR on the CM-2. *Appl. Num. Maths.* **14**, pp. 239–253.

[12] CLARKE, J.F., KARNI, S., QUIRK, J.J., ROE, P.L., SIMMONDS, L.G. & TORO, E.F. 1993 Numerical Computation of 2-dimensional unsteady detonation-waves in high-energy solids. *J. Comput. Phys.* **106**, pp. 215–233.

[13] COLELLA, P. & HENDERSON, L.F. 1990 The von Neumann paradox for the diffraction of weak shock-waves. *J. Fluid Mech.* **213**, pp. 71–94.

[14] FISCHER, J. 1993 Selbstadaptive, lokale Netzverfeinerungen für die numerische Simulation kompressibler, reibungsbehafteter Strömungen. Ph.D. thesis, Institut für Aerodynamik und Gasdynamik, Universität Stuttgart.

[34] This list is restricted to block-structured, adaptive mesh refinement schemes. Please send additions to *librarian@amrita-cfd.com*.

[15] HENTSCHEL, R. & HIRSCHEL, E. H., 1994 *Self adaptive flow computations on structured grids.* In *Computational Fluid Dynamics '94*, Proceedings of the Second European Computational Fluid Dynamics Conference, edited by S. Wagner, J. Périaux and E.H. Hirschel, Wiley, pp. 242–249.

[16] HENDERSON, L.F., COLELLA, P. & PUCKETT, E.G. 1991 On the refraction of shock-waves at a slow fast gas interface. *J. Fluid Mech.* **224**, p. 1.

[17] HORNUNG, R.D. & TRANGENSTEIN, J.A. 1997 Adaptive mesh refinement and multi-level iteration for flow in porous-media. *J. Comput. Phys.* **136**, pp. 522–545.

[18] PEMBER, R.B., BELL, J.B., COLELLA, P., CRUTCHFIELD, W.Y. & WELCOME, M.L. 1995 An adaptive Cartesian grid method for unsteady compressible glow in irregular regions. *J. Comput. Phys.* **120**, pp. 278–304.

[19] PUCKETT, E.G. & SALTZMAN, J.S. 1992 A 3D adaptive mesh refinement algorithm for multimaterial gas-dynamics. *Physica D* **60**, pp. 84–93.

[20] QUIRK, J.J. & KARNI, S. 1997 On the dynamics of a shock-bubble interaction. *J. Fluid Mech.* **318**, pp. 129–163.

[21] SHORT, M. & QUIRK, J.J. 1997 On the nonlinear stability and detonability limit of a detonation-wave for a model 3-step chain-branching reaction. *J. Fluid Mech.* **339**, pp. 89–119.

[22] UPHOFF, U., HANEL, D., ROTH, P. Numerical Modelling of detonation structure in 2-phase flows. *Shock Waves* **v6**, pp. 17-20.

[23] VERWER, J.G. & TROMPERT, R.A. 1991 *Local uniform grid refinement for time-dependent partial differential equations.* Centrum voor Wiskunde en Informatica, Report NM-R9105.

B def EquationSet

Amr_sol provides a `def EquationSet` block to allow you to define mappings which dictate how physical quantities – such as density, pressure and velocity – are written to, and extracted from, the computational grid. The strict mathematical interpretation of the `EquationSet`, however, is left up to the `solver` and so you retain complete control over its formulation. Given pre-defined routines such as these:

```
amrcp  vki/eqns.1
amrita invoke_eqns
```

```
PotentialFlowEquations
LinearAdvectionEquation
BurgersEquation
ShallowWaterEquations
EulerEquations
IsentropicEquations
IsothermalEquations
NavierStokesEquations
ReactiveEulerEquations
RelativisticEulerEquations
```

you may well feel you have no need to program down at the level described in this Appendix. If this is the case, you should at least skim through the following material to gain an appreciation of what happens when an `EquationSet` is invoked. If nothing else, this will show that *Amrita* is built to last.

B.1 The EulerEquations

The mathematical formulation for `EulerEquations` can be obtained using this script:

```
amrcp  vki/eqns.2
amrita show_euler_eqns
cd     latex_files
amrps  amrita.ps
```

```
EulerEquations {
   space    = 2D
   symmetry = slab
}
set thisfile = file $amr_sol:EquationSet::file
grab::info BCG:[Latex::Document] from $thisfile
LatexHead
   parse token Latex::Document
LatexTail
Latex
```

which conveniently provides the background information (see Figure 26) needed to follow the rest of this section.

Amr_sol views an `EquationSet` solely in terms of how data must be shovelled to and from the discrete solution vector, W, stored for an isolated cell, in an isolated mesh patch. For instance, to be able to write field data, this command:

```
setfield <RHO=1,U=1,V=0,P=1>
```

needs to know how to pack the quantities ρ, u, v and p together to form W. Similarly, to be able to return the minimum and maximum pressures, this command:

```
minmax P[] -> Pmin, Pmax
```

needs to know how to unpack p from W.

1 Two-dimensional Euler Equations (slab symmetry)

Notation:

$x-y$	Cartesian coordinates
t	time
ρ	density
u	x-component of velocity
v	y-component of velocity
p	pressure
E	total energy
γ	ratio of specific heats
c	sound speed

$$\mathbf{W} = \begin{pmatrix} \rho \\ \rho u \\ \rho v \\ E \end{pmatrix}$$ conservative solution vector

$$\mathbf{F} = \begin{pmatrix} \rho u \\ \rho u^2 + p \\ \rho uv \\ (E+p)u \end{pmatrix}$$ flux in x-direction

$$\mathbf{G} = \begin{pmatrix} \rho v \\ \rho vu \\ \rho v^2 + p \\ (E+p)v \end{pmatrix}$$ flux in y-direction

Formulation:

$$\frac{\partial \mathbf{W}}{\partial t} + \frac{\partial \mathbf{F}}{\partial x} + \frac{\partial \mathbf{G}}{\partial y} = 0$$

Perfect gas model:

$$p = (\gamma - 1)(E - \frac{1}{2}\rho u^2 - \frac{1}{2}\rho v^2) \qquad \Longrightarrow \qquad c = \sqrt{\frac{\gamma p}{\rho}}$$

Predefined Functions:

X[]	$::= x$	RHO[]	$::= \rho$	C[]	$::= c$
Y[]	$::= y$	U[]	$::= u$	E[]	$::= E$
t[]	$::= t$	V[]	$::= v$		
		P[]	$::= p$		
		GAMMA[]	$::= \gamma$		

Figure 26: Page output by the script *show_euler_eqns*. If curious, try: (i) re-running the script with `space` set to `1D` and `symmetry` set to `cylindrical` or `spherical`; (ii) adding the line `echo $Latex::Document` to see what `grab::info` returns; (iii) viewing `$thisfile` with *amrgi* to locate the `fold::info` blocks which contain the LaTeX typesetting information; (iv) re-running the script with another `EquationSet`, e.g. `ReactiveEulerEquations`.

The required mapping information is furnished using a `def EquationSet` block. The `EulerEquations` employs three such blocks to cover `1D`, `2D` and `3D` flows[35]:

```
proc EulerEquations {
     space    = two-dimensional
     symmetry {slab|cylindrical|spherical} = slab
     gamma    = 1.4
   }
   switch on $space
      case 1D:one-dimensional:
         ...  1D EquationSet
      case 2D:two-dimensional:
         ...   2D EquationSet
      case 3D:three-dimensional:
         ...   3D EquationSet
      default:
         error  '$space\' space unknown by EulerEquations!
   end switch
end proc
...  BCG documentation
```

Below is the two-dimensional version of the `EquationSet`[36]:

```
fold::amrita'space2D {  2D EquationSet
   def EquationSet
      name      $amrita::proc0
      note      equations are cast in conservation form
      space     two-dimensional
      symmetry  $symmetry
      neqns     4
      notation  RHO,U,V,P,GAMMA
      notation  C,E
      problem specific    GAMMA
      def SolutionVector
          require RHO,U,V,P,GAMMA
          hint precompute P
          W[1] ::= RHO[]
          W[2] ::= RHO[]*U[]
          W[3] ::= RHO[]*V[]
          W[4] ::= P[]/(GAMMA[]-1.0)+0.5*RHO[]*(U[]**2+V[]**2)
          specify GAMMA::= $gamma
          RHO  ::= W[1]
          U    ::= W[2]/W[1]
          V    ::= W[3]/W[1]
          P    ::= (GAMMA[]-1)*(W[4]-0.5*(W[2]**2+W[3]**2)/W[1])
      end def
      C ::= sqrt(GAMMA[]*P[]/RHO[])
      E ::= P[]/(GAMMA[]-1)+0.5*RHO[]*(U[]**2+V[]**2)
   end def
}
```

[35]At the time of writing, *Amr_sol* cannot compute three-dimensional flows, but it can nonetheless be taught a `three-dimensional EquationSet`.

[36]`space2D` is simply a retrieval-name given to the program-fold and has no physical significance.

The first command specifies a name with which to label the `EquationSet`. The string token `amrita::proc0` expands to the name of the last procedure entered by *Amrita*, and so here yields `EulerEquations`. This programming trick ensures an `EquationSet` is named after the procedure which created it and is used by all the pre-supplied routines[37].

The second command, provides a simple reminder (i.e. `note`) that the `EquationSet` employs a conservative solution vector $\mathbf{W} = (\rho, \rho u, \rho v, E)^t$, as opposed to the primitive variable vector $(\rho, u, v, p)^t$, or some other set of variables.

The third command, indicates the `EquationSet` is for `two-dimensional space`, as opposed to `one-dimensional space` or `three-dimensional space`. The abbreviations `1D`, `2D` and `3D` may also be used to indicate the desired `space`.

The fourth command is used to indicate the `symmetry` implied by the `EquationSet`. By default `$symmetry` has the value `slab` and so the `EquationSet` is considered strictly two-dimensional. But if `cylindrical` symmetry were specified, as in:

```
EulerEquations symmetry=cylindrical
```

the `EquationSet` would be considered axisymmetric[38].

The fifth command, `neqns`, is slightly misleading in that it specifies the number of components in the solution vector and not the total number of equations in the system, which explains why here it is given the value four and not five.

The two `notation` commands provide a list of the quantities permitted in the formulation of the solution vector. One or more of these quantities may be tagged `problem specific` to indicate that they fix the `EquationSet` for a specific problem and so must be input somewhere along the line by the user. Here only γ is identified as being problem specific, but as shown below this does not mean that γ need be a constant. The commands `notation` and `problem specific` may be repeated as often as needed and so there is no need to cram long lists on to one line. The `notation`: x, y, t and $\mathbf{W}$ is predefined[27] and so does not need to be declared explicitly. Sometimes, however, it is useful to employ spatial notation which is more meaningful than the default x and y. For instance, to use r and z with an axisymmetric set of equations, you could use either:

```
notation R,Z          or      notation R,Z
R ::= Y[]                     R ::= X[]
Z ::= X[]                     Z ::= Y[]
```

depending on the physical orientation of the grid.

Given the above preliminaries, the functional form of W is defined using template expressions[39] inside a `def SolutionVector` block. The `require` command identifies a subset of the `notation` quantities to provide a checklist with which to trap careless errors where a state cannot be defined because key information is missing. For instance, given:

```
EulerEquations
W'quiescent ::= <RHO=1>
```

```
amrcp  vki/eqns.3
amrita missing_data
```

[37]To see what other system tokens *Amrita* keeps track off, type:

```
unix-prompt>amrita -c
amrita>Show tokens=amrita*
```

[38]Not that this has any affect on the rest of the `EquationSet` block, because the chosen `symmetry` is simply a directive to be interpreted by other parts of *Amrita* as they see fit, e.g. `BasicCodeGenerator` (recall Appendix B from lecture 1).

[39]Expression templates were described in §2.4 of lecture 1.

Amrita responds:

```
Error at line 2 of file missing_data:
state is missing: P,U,V!

Line 2 is:
W'quiescent ::= <RHO=1>

error near:
end of line
```

The `hint` directive is an optimization which instructs *Amr_sol* to pre-compute pressure whenever it expects to do a large amount of interpreted computation[40] involving the solution vector. As such, it is non-essential to the operation of `EulerEquations`.

The first four template expressions in the `SolutionVector` block define how the primitive variables ρ, u, v and p are to be mapped to **W**. The second four expressions define the inverse mapping. The keyword `specify` which precedes the definition of γ is syntactically redundant and could be omitted, nevertheless it helps emphasize that `GAMMA` is `problem specific` and so is not entirely useless. The significance of this fact are two fold: (i) provided it is independent of **W**[41], a `problem specific` quantity does not need to be specified when a state is defined as it can default to the value used in the definition of the `EquationSet` (which reveals why γ was not listed as missing in the above script); (ii) unlike plain `notation` quantities, which are considered cast in stone, `problem specific` quantities can be changed by the user. Consequently, this next script bombs out on the third line rather than the second:

```
EulerEquations
specify GAMMA ::= X[]
RHO ::= X[]
```

```
amrcp  vki/eqns.4
amrita reserved_notation
```

with the error message:

```
Error at line 3 of file reserved_notation:
'RHO' is reserved by the current 'EquationSet'!

Line 3 is:
RHO ::= X[]

error near:
RHO ::= X[]
```

Note, however, that `specify` is mandatory when updating a `problem specific` quantity.

The two remaining `notation` quantities (`C` and `E`) are not critical to the definition of the `EquationSet` and are provided merely as a convenience, which is why they are declared separately from the other quantities and also defined outside the `SolutionVector` block.

[40]Template expressions are interpreted at run-time and so execute more slowly than compiled code. With *Amr_sol* the cost of decoding a template expression is borne by a mesh patch and not a single mesh cell, and so the overhead can be tolerated for lightweight-tasks such as flagging for refinement. The `hint` directive helps reduce the overhead still further. Nevertheless, for heavy-duty work *Amr_sol* always falls back to compiled code.

[41]Here, γ could be made temperature dependent. However, although `EulerEquations` is happy to cope with a variable γ, the flow `solver` may not be so obliging.

B.1.1 Thermodynamic States

Once an `EquationSet` has been defined, you are free to specify thermodynamic states which are automatically checked for consistency. For instance, the left- and right-states in the ubiquitous Sod's problem[31] could be specified using:

```
EulerEquations space=one-dimensional
W'left_state  ::= <RHO=1.0,  U=0,P=1.0>
W'right_state ::= <RHO=0.125,U=0,P=0.1>
```

```
amrcp  vki/state.1
amrita set_sod_states
```

and later used as parameters to a command such as `setfield`:

```
setfield W'left_state   X[] < $diaphragm
setfield W'right_state  X[] >=$diaphragm
```

which is described in §E.

Internally, states are stored as template expressions, which explains the use of `::=` rather than a plain `=`. The `W'` part alerts *Amrita*[42] that a state is to be defined (or used), and the accompanying state-label (here `left_state` and `right_state`) enables the interpreter to distinguish one state from another. If you experiment with this two line script:

```
EulerEquations
W'mystate  ::= <RHO=1,U=0,V=0,P=1,GAMMA=1.4>
```

```
amrcp  vki/state.2
amrita set_mystate
```

you will find that the state-label can be set to any string made up from the characters {`A-Z`, `a-z`, `0-9` and `_`}, provided the string starts with a letter[43]. Also, the order in which the quantities `RHO`, `U`, `V`, `P` and `GAMMA` are supplied is unimportant. You might also like to check *Amrita*'s response when you deliberately mistype a required variable e.g. type ***Rho*** instead of ***RHO***. As usual, the system goes to some lengths to trap any careless errors you might make.

Amrita allows simple state assignments of the form:

```
W'state2 ::= W'state1
```

But compound expressions such as:

```
W'state2 ::= 2*W'state1
```

and:

```
W'state3 ::= W'state1* W'state2
```

are ruled out, because of the thermodynamic implications of allowing states to be naïvely manipulated as numbers. Similarly, the individual expressions of a state must evaluate to a real result, although as in `PotentialFlowEquations`, they can involve complex arithmetic en route to a real result. On the other hand, it is perfectly natural to define a new state in terms of a previously defined state[44]. For example, try adding this line:

```
W'new ::= W'mystate<RHO*=mul,U+=add,V-=sub,P/=div,GAMMA=num>
```

to the *set_mystate* script, and experiment by replacing *mul*, *add*, *sub*, *div* with expressions of your choice. The operators `*=`, `/=`, `+=` and `-=` work as in *C*.

[42]Strictly speaking, the machinery used to parse a state is provided by *Amr_sol*, but because the relevant *Perl* is sucked into *Amrita*, the distinction is moot.

[43]With *Amrita* v1.38, states are viewed as global quantities and cannot be given namespaced labels: in effect `W'` acts as a `state::` namespace. On the other hand, template expressions, like string tokens, can be given an explicit namespace. This distinction is historical and will likely be removed in later *Amrita* releases.

[44]Recall the definition of `ShockWave` in §2.4 of lecture 1.

Amrita allows a state's constituent template-expressions to be accessed individually. For instance, this script from the end of §2.4 in lecture 1:

```
EulerEquations
plugin amr_sol
W'one ::= <RHO=1,U=0,V=0,P=1,GAMMA=1.4>
ShockWave state1=one,state2=two,Ms=2
exprA ::= P'two[]
W'one ::= <RHO=1,U=0,V=0,P=1,GAMMA=X[]>
ShockWave state1=one,state2=two,Ms=2
exprB ::= P'two[]
export exprA[],exprB[]
```

```
amrcp  vki/sym.1
amrita -debug run_export
amrgi  debug.isl
```

accesses the pressure for state `two` using `P'two[]`. You could similarly use: `RHO'two[]`, `U'two[]`, `V'two[]` and `GAMMA'two[]` to access the other available expressions.

Here it is instructive to observe the difference between `exprA` and `exprB`[45]:

```
amrita:export::expr {
   str exprA
   expr {
      1
      n 4.5
   }
}
```

```
amrita:export::expr {
   str exprB
   expr {
      1
      n 1
      n 2
      v 500
      o 0 0
      b 15
      n 2
      b 15
      n 2
      b 15
      v 500
      o 0 0
      n 1
      m 13
      m 13
      v 500
      o 0 0
      n 1
      b 12
      b 16
      b 15
   }
}
```

Although both templates where constructed in the same fashion, `exprA` involved only constants and so could be mangled down to a single number, but `exprB` involved the system variable `X[]` and so is left as a postfix version of $\frac{2\gamma M_S^2-(\gamma-1)}{\gamma+1}p_1$, with $M_S = 2$, $\gamma =$ `X[]` and $p_1 = 1$.

Because `exprA` is a constant, it can be assigned to an *Amrita* string token using:

```
set token #= sym(exprA[])
```

but `sym` is currently unable to convert a variable template such as `exprB`.

[45] An `expr` fold consists of a series of low-level op-codes which employ postfix arithmetic to leave one or more results on an expression stack. Here `exprA` pushes one number on to the stack, and `exprB` performs a series of operations to leave one number on the stack. For instance, `b 15` is a binary operator which takes two numbers off the stack, multiplies them together, then pushes the result back on to the stack. Analogously: `b 16` performs division; `b 12` performs addition; `m 13` performs subtraction. The opcode `v 500` with the offset `o 0 0` pushes `X[]` on to the stack. This internal `expr` format is scheduled to be revamped and so the fact `exprB` contains a redundant multiplication by 1 is of no great concern.

B.2 The `LinearAdvectionEquation`

Dropping down the mathematical scale, the `EquationSet` for the linear-advection equation[15]:

$$\frac{\partial U}{\partial t} + a\frac{\partial U}{\partial x} = 0 \qquad \text{with} \qquad a > 0$$

follows the exact same form as that used by the `EulerEquations`[46]:

```
proc LinearAdvectionEquation {
     space   = one-dimensional
     a [0:?] = 1.0
   }
   switch on $space
      case 1D:one-dimensional:
         def EquationSet
            name       $amrita::proc0
            space      one-dimensional
            neqns      1
            notation   U,A
            problem specific A
            def SolutionVector
                require U,A
                W[1] ::= U[]
                U    ::= W[1]
                specify A ::= $a
            end def
         end def
      default:
         error  '$space\' space unknown by LinearAdvectionEquation!
   end switch
end proc
...  BCG documemtation
```

Amrita is designed to operate more or less independently of the mathematical complexity of the target application. Consequently the *Amrita* programming skills acquired by working with the `LinearAdvectionEquation` are directly transferable to projects which employ full systems of partial-differential equations. Because of this, instead of diving head long into your chosen application, you should first serve a short apprenticeship dissecting the linear-advection investigation obtained by typing:

```
unix-prompt>amrcp Chp7/1a.mailit
```

This may seem a retrograde step, but it will speed progress in the long run. To quote from the *HTML* help page which is unpacked when the *mailit* is first run:

> CFD algorithms are often designed by considering *model problems*. The insight gained from studying the model problem is then *extrapolated* to find a successful solution procedure to some target application which itself might be too difficult to analyse or so expensive to compute it precludes a trial and error solution approach.

The same is true for learning to use *Amrita* efficiently. Learn new programming constructs using model scripts, because the turnaround time for your chosen application is likely too high for you to stumble through writing the required code by trial and error.

[46] Internally, `symmetry` defaults to `slab` and so need not be given.

B.3 The `FractalFactory`

The *fractal.mailit* from lecture 1 (Appendix C) contains a routine `FractalFactory`:

```
proc FractalFactory {
     set      = mandelbrot
     nmax     = 255
     startup "= ArraySizes NGIxJ=800000
   } <-> fractal::
   ...  compile fractal_factory
   def EquationSet
      name      FractalFactory
      space     two-dimensional
      neqns     1
      notation  xo,yo
      problem specific xo,yo
      def SolutionVector
         require xo,yo
         Z    ::= {X[]+xo[],Y[]+yo[]}
         W[1] ::= fn(fractal_factory::$set,$nmax,Re(Z[]),Im(Z[]))
         specify xo  ::= 0
         specify yo  ::= 0
      end def
   end def
   W'fractal       ::= <xo=0,yo=0>
   fractal::count ::= W[1]/$nmax
   parse token fractal::startup
end proc
```

which shows how to construct an `EquationSet` when the `SolutionVector` mapping is not in closed-form. The expression template:

```
W[1] ::= fn(fractal_factory::$set,$nmax,Re(Z[]),Im(Z[]))
```

uses an *Amrita* `fn()` hook to call a routine `$set` (i.e. `mandelbrot`) from a shared-object package *fractal_factory*, which is produced by the program-fold.

When called[47], the routine `mandelbrot`:

```
      AMRDBL FUNCTION MANDELBROT(PAR)
      AMRDBL PAR(0:*)
      AMRCPX Z,C
      AMRINT N,NMAX
      NMAX = PAR(1)
      C    = CMPLX(PAR(2),PAR(3))
      Z    = (0,0)
      N    = 0
  100 CONTINUE
         Z = Z*Z+C
         IF((ABS(Z).GE.2).OR.(N.GE.NMAX)) THEN
            MANDELBROT = N
            RETURN
         ENDIF
         N = N+1
      GOTO 100
      RETURN
      END
```

[47]The `setfield` command in the *Amrita* routine `DrawFractal` implicitly sweeps over the grid, calling `mandelbrot` for each mesh cell as it goes.

is passed an array of four *AMRDBL* numbers: `PAR(0)` contains the number of parameters in the `fn()` call; `PAR(1)` contains the value `$nmax`; `PAR(2)` contains the current value of `Re(Z[])`; `PAR(3)` contains the current value of `Im(Z[])`. The body of the `FUNCTION` performs its business and then returns a result in the normal *Fortran* fashion.

Although – in the context of these lecture notes – the *fractal.mailit* may appear flippant the *Amrita* programming construct outlined above transfers directly to genuine fluids applications. For instance, this `EquationSet` was written to drive a two-phase (solid-gas) code used to investigate deflagration-to-detonation transition in damaged energetic materials[2]:

```
proc SeptemberEquations {
     space    = one-dimensional
     code     = september::
     startup "= $code'get_tokens
   } -> SEPT::
   switch on $space
      case 1D:one-dimensional:
         def EquationSet
            name  $amrita::proc0
            space one-dimensional
            neqns 7
            notation RHOs,Us,PHIs,Ps,Ts
            notation RHOg,Ug,Pg,Tg
            def SolutionVector
               require RHOs,Us,PHIs,Ps,Ts
               require RHOg,Ug,Pg,Tg
               W[1] ::= RHOs[]
               W[2] ::= RHOs[]*Us[]
               W[3] ::= RHOs[]*PHIs[]
               W[4] ::= fn($code'fn_rhoets,RHOs[],Us[],Ps[],PHIs[])
               W[5] ::= RHOg[]
               W[6] ::= RHOg[]*Ug[]
               W[7] ::= fn($code'fn_rhoetg,RHOg[],Ug[],Pg[],Tg[],PHIs[])
               RHOs ::= W[1]
               Us   ::= W[2]/W[1]
               PHIs ::= W[3]/W[1]
               PHIg ::= 1-PHIs[]
               Ps   ::= fn($code'fn_ps,W[1],W[2],W[3],W[4],W[5],W[6],W[7])
               Ts   ::= fn($code'fn_ts,W[1],W[2],W[3],W[4],W[5],W[6],W[7])
               RHOg ::= W[5]
               Ug   ::= W[6]/W[5]
               Pg   ::= fn($code'fn_pg,W[1],W[2],W[3],W[4],W[5],W[6],W[7])
               Tg   ::= fn($code'fn_tg,W[1],W[2],W[3],W[4],W[5],W[6],W[7])
            end def
         end def
      default:
         error  '$space\' space unknown by SeptemberEquations!
   end switch
end proc
```

The tie-up to `FractalFactory` is self-evident.

B.4 Keywords

The earlier observation that you should learn new programming constructs using model examples, and not your target application, cannot be emphasized strongly enough. The turnaround time for most CFD simulations is sufficiently long that it inhibits the development of good programming style. To employ a hackneyed, but nonetheless appropriate adage – practice makes perfect. Consequently, *Amrita* programming skills are best honed using short, targeted scripts. For instance, this script will list all the specialist keywords which can be used inside a `def EquationSet` block:

```
plugin amr_sol
keywords amr_sol:EquationSet*
```

```
amrcp  vki/key.1
amrita show_EquationSet
```

and forms the basis of the system routine:

```
proc HtmlKeywords search=*
   set amrita:html::file = $amrita::junkdir/$amrita::jobno.html
   HtmlHead
   HtmlSearchBanner banner=keywords: $search
   keywords $search -> keywords
   if(token(keywords)) then
      foreach keyword ($keywords) split on /\n/
         HtmlKeyword keyword=$keyword
      end foreach
   endif
   HtmlTail
   Netscape
end proc
```

which is activated when you type:

```
unix-prompt>amrita -c
amrita>plugin amr_sol
amrita>Show keywords=amr_sol:EquationSet*
```

Similarly, this script lists the specialist keywords described in Appendices C-G:

```
plugin amr_sol
foreach defblock (Domain,                \
                  BoundaryConditions,    \
                  SolutionField,         \
                  MeshAdaption,          \
                  RefinementCriteria)
   keywords amr_sol:$defblock* -> list
   echo $list
end foreach
```

```
amrcp  vki/key.2
amrita show_keywords
```

Although in practice you would obtain the information, in the form of an *HTML* document, by typing:

```
unix-prompt>amrita -c
amrita>plugin amr_sol
amrita>Show keywords=amr_sol*
```

C def Domain

With *Amr_sol*, after selecting an `EquationSet`, the first step in setting up a simulation is to define the computational `Domain`. This is done using a logical Cartesian space:

$$C_o = \{i \times j : i \in \mathcal{N}, j \in \mathcal{N}\}$$

where each coordinate pair (i, j) identifies a possible mesh cell[21, 25]. Specific domains are constructed by laying down rectangular patches of cells, each `patch` being fixed in terms of its lower-left and upper-right corners in C_o, and this information can be supplied explicitly as two pairs of coordinates or implicitly as one-coordinate pair plus a width and height. For example, the following would select the cells shown in Figure 27 (a):

```
proc CornerProblem
   def Domain
      lscale 1
      patch <1,11,w8, h10>
      patch <+, 1,w12,h20>
   end def
end proc
```

```
amrcp  Chp2/corner.1
amrita run_corner_mk1
amrps  ps/corner.ps
```

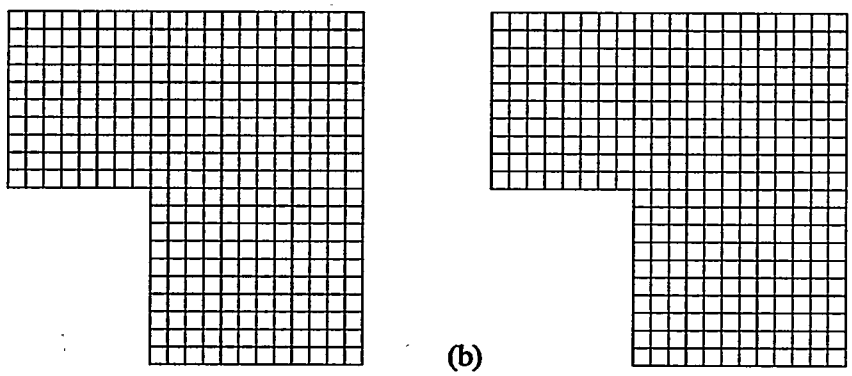


Figure 27: Computational domains: (a) 90° corner (full scale); (b) multiple cavity (quarter scale).

The command `def Domain` instructs *Amrita* to start constructing a new domain and enables it to accept certain specialist commands such as `lscale`, which here sets the cell size to one. The first `patch` command selects a patch 8 cells wide by 10 cells high with its lower-left corner positioned at logical cell (1,11). The second `patch` command then places a patch immediately to the right of this first patch, because the i coordinate is specified implicitly using the notation +. Thus the lower-left corner of the second patch is positioned at (9,1) and its upper-right corner at (20,20). The following straightforward variation on a theme would produce the multiple-cavity grid shown in Figure 27 (b):

```
proc MultipleCavityProblem
   def Domain
      lscale 1
      do n=1,4
         patch <+,11,w8, h10>
         patch <+, 1,w12,h20>
      end do
   end def
end proc
```

```
amrcp  Chp2/mcavity.1
amrita run_mcavity
amrps  ps/mcavity.ps
```

C.1 Six Specifics

1. Entering a `def Domain` block wipes the computational slate clean, that is any previous flow solution is erased.

2. The commands:

   ```
   patch <1,11,w8,h10>    and    patch <1,11,8,20>
   ```

   are equivalent to one another.

3. Both these commands are invalid:

   ```
   patch <0,5,10,10>    and    patch <10,10,1,1>
   ```

   The first, because 0 falls outside the range of coordinates used to define patches. The second, because the lower-left coordinates are greater than the upper-right coordinates.

4. A '–' is treated as the inverse of '+', therefore:

   ```
   def Domain
      patch <1,11,w8 ,h10>
      patch <+,1 ,w12,h20>
   end def
   ```

   produces the same domain as both:

   ```
   def Domain
      patch <9,1 ,w12,h20>
      patch <-,11,w8 ,h10>
   end def
   ```

   and:

   ```
   def Domain
      patch <1,11,w20,h10>
      patch <9,- ,w12,h10>
   end def
   ```

   The script obtained with ***amrcp Chp2/verify.1*** can be used to verify this equivalence graphically.

5. To prevent user mishaps, *Amr_sol* precludes the input of overlapping patches. Internally, the algorithm could cope with overlapping patches, but on balance it is thought more user-friendly to exclude them.

6. Because of storage efficiency reasons, *Amr_sol* places two internal restrictions on the upper size of a mesh patch: (i) the longest side cannot be greater than 210 cells; (ii) the total number of mesh cells plus ghostcells cannot exceed 5500. Consequently this command generates an error:

   ```
   patch <1,1,100,100>
   ```

   and should be replaced by:

   ```
   patch <1, 1,w50,h50>
   patch <+, 1,w50,h50>
   patch <1,51,w50,h50>
   patch <+,51,w50,h50>
   ```

C.2 Curvilinear Geometry

Section D.6 in lecture 1 described how to produce a polar grid. This script:

```
amrcp  vki/nozzle.1
amrita run_nozzle
amrps  ps/nozzle.ps
```

```
... create code/nozzle
EulerEquations
plugin amr_sol
def Domain
   lscale 0.4
   patch <1,21,w25,h20>
   patch <+,21,w50,h20>
   patch <+,21,w25,h20>
   patch <1, 1,w25,h20>
   patch <+, 1,w50,h20>
   patch <+, 1,w25,h20>
   ...   export names of data files
   grid   code/nozzle
end def
... plot grid
```

uses the same basic grid-generation technique to produce the configuration shown in Figure 28, and was written for a simulation of a supersonic shear-layer experiment[10]. The grid quality is not the best that could be generated, but it does have the merit that the associated code is short enough to be dissected here in full.

The program fold:

```
fold::amrita'dat {  export names of data files
   set GEOMETRY = $amrita::AMRITA/examples/Chp6/GG
   set Nupper = $GEOMETRY/top.wall.nozzle
   set Supper = $GEOMETRY/top.wall.splitter
   set Nlower = $GEOMETRY/bottom.wall.nozzle
   set Slower = $GEOMETRY/bottom.wall.splitter
   set Xo     = -8.0
   export Xo,Nupper,Nlower,Supper,Slower
}
```

locates four data files which tabulate, in the form of $x-y$ data pairs, the geometry for the upper and lower walls of both the nozzle and the splitter-plate. For example, here are a few lines from the file *top.wall.nozzle*:

```
8.48000 15.43964
8.64000 15.32548
8.80000 15.21182
8.96000 15.09881
9.12000 14.98656
9.28000 14.87518
9.44000 14.76478
9.60000 14.65547
```

The locations of the geometry files, together with a reference position, `Xo`, are exported to *Amr_sol* so that they can be read by the *Fortran* code:

```
fold::print'src1 { write nozzle.src
   fold>file = nozzle.src
   #include "AMR_SOL/AMRITA"
         SUBROUTINE GEN_NOZZLE(GRD,NG,IM,JM,DX,X,Y,IW)
         AMRSTR*255 Nupper,Nlower,Supper,Slower
         AMRINT GRD,NG,IM,JM,IW
         AMRDBL DX,X(*),Y(*)
         CALL AMR::GET_TOKEN('AMRSTR::Nupper',Nupper)
         CALL AMR::GET_TOKEN('AMRSTR::Nlower',Nlower)
         CALL AMR::GET_TOKEN('AMRSTR::Supper',Supper)
         CALL AMR::GET_TOKEN('AMRSTR::Slower',Slower)
         IF(GRD.LE.3) THEN
            CALL GEN_PATCH(Supper,Nupper,NG,IM,JM,DX,X,Y,IW)
         ELSE
            CALL GEN_PATCH(Nlower,Slower,NG,IM,JM,DX,X,Y,IW)
         ENDIF
         RETURN
         END
}
```

This code acts as a driver for the routine:

```
fold::print'src2 {  write nozzle.src
   fold>col=7,file .= nozzle.src
   SUBROUTINE GEN_PATCH(TABDATA1,TABDATA2,NG,IM,JM,DX,X,Y,IW)
   AMRSTR*255 TABDATA1,TABDATA2
   AMRINT NG,IM,JM,IW
   AMRDBL X(amrVpatch(IM,JM,NG))
   AMRDBL Y(amrVpatch(IM,JM,NG))
   AMRDBL DX,AMR::INTERP
   AMRDBL Xo,Xs,Y1,Y2
   CALL AMR::GET_TOKEN('AMRDBL::Xo',Xo)
   DO I=1-NG,IM+NG+1
      Xs = (I+IW-2)*DX+Xo
      Y1 = AMR::INTERP(3,TABDATA1,Xs)
      Y2 = AMR::INTERP(3,TABDATA2,Xs)
      DO J=1-NG;JM+NG+1
         X(I,J) = Xs
         Y(I,J) = (J-1)*((Y2-Y1)/JM)+Y1
      END DO
   END DO
   RETURN
   END
}
```

which constructs the geometry for a single patch using calls to `AMR::INTERP`. This is a hook into *Amr_sol*'s internal machinery to perform the necessary interpolation[48] of a tabulated data-file to find – for a specified X station – the top-most (`Y2`) and bottom-most (`Y1`) points of a vertical grid line. The interior points are then found by sub-division with equal spacing.

[48]Here third-order Lagrange interpolation is used, but `AMR::INTERP` can be dynamically over-loaded to use other types of interpolation. Internally, `AMR::INTERP` is used to decode expression templates such as this one taken from the *run_cellular.mailit* from lecture 1 (also see §E.3):

```
W'znd ::= < RHO = interp($znd.RHO,Xd[]), U = interp($znd.U,Xd[]), V = 0,\
            P   = interp($znd.P  ,Xd[]), Z = interp($znd.Z ,Xd[]) >
```

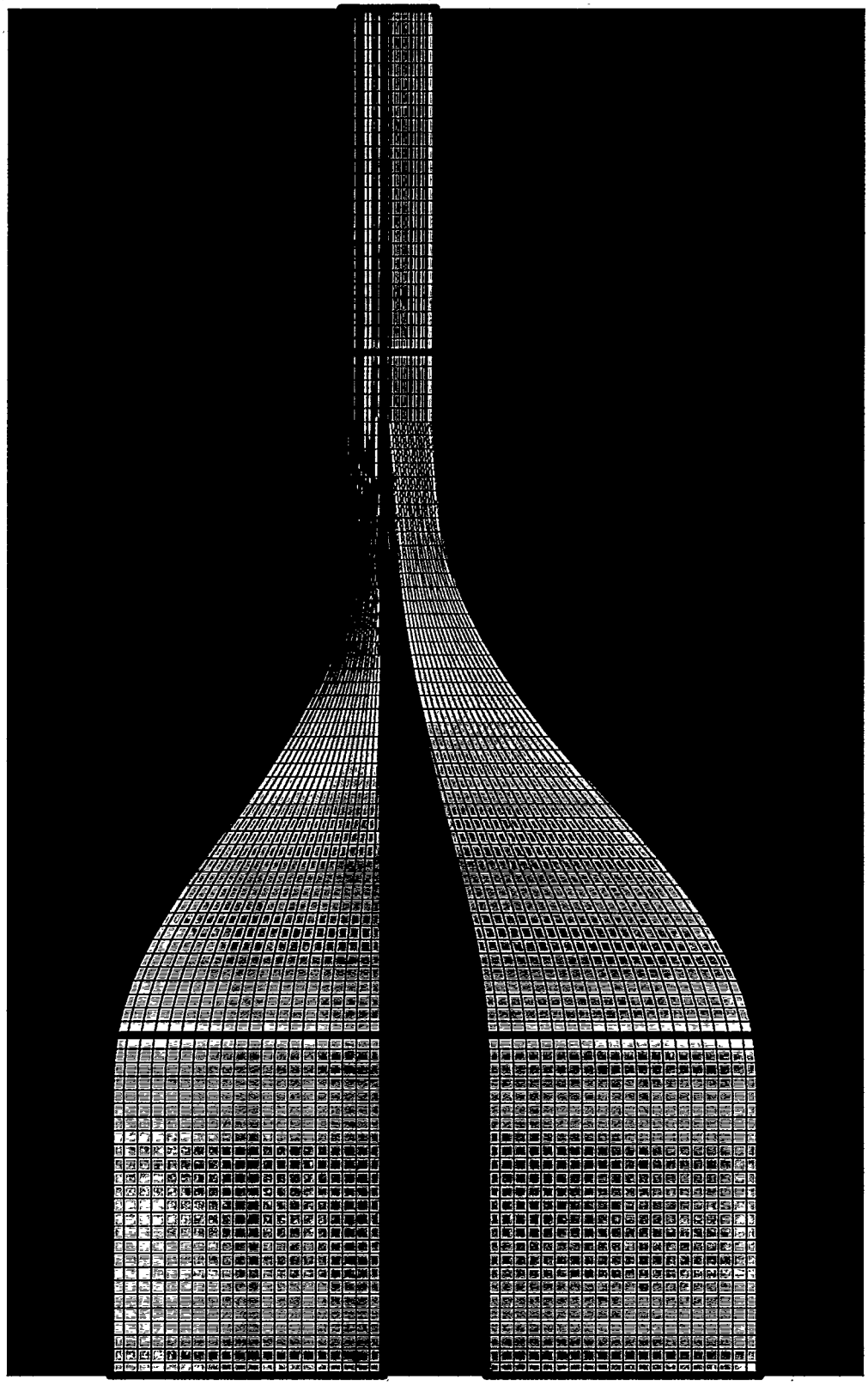


Figure 28: Multi-block grid generated using *run_nozzle*. Eventually, the work done by the routine GEN_PATCH will be abstracted down to a specialist *Amr_sol* keyword, but given the versatility provided by *Amrita*'s dynamic-linking mechanism, the upgrade is not deemed urgent.

D def BoundaryConditions

With *Amr_sol*, even after an `EquationSet` and a `Domain` has been specified, there is still insufficient information to run a simulation. Consequently, this script:

```
amrcp  Chp2/corner.2
amrita run_corner_mk2
```

```
proc CornerProblem
   def Domain
      lscale 1
      patch <1,11;w8, h10>
      patch <+, 1,w12,h20>
   end def
end proc

EulerEquations
plugin amr_sol
CornerProblem
march 150 steps with cfl=0.8
```

generates an error:

```
Error at line 12 of file run_corner:
cannot march: no BoundaryConditions!
cannot march: no SolutionField!
cannot march: no solver!


Line 12 is:
march 150 steps with cfl=0.8

error near:
150 steps with cfl=0.8
```

Amr_sol treats `def` blocks as interlocks which allow it to maintain some semblance of control on how a simulation is set up, without introducing draconian rules on what you can and cannot do. The simulation order:

1. `EquationSet`
2. `Domain`
3. `BoundaryConditions`
4. `SolutionField`
5. `MeshAdaption`
6. `RefinementCriteria`

allows *Amr_sol* to perform far more comprehensive consistency checks than would be possible with a free-for-all approach[49].

[49]The system allows for a certain reordering in that the last three `def` blocks may be repeated out of sequence once a problem has been set up. For instance, `def SolutionField` is used by a routine, `FireLaser`, from the *ramp.mailit* in §2 to add a perturbation to an existing flow field, and many scripts alter `RefinementCriteria` during the course of a simulation, or toggle `MeshAdaption` on and off. But this apart, the presented ordering is mandatory when starting a fresh simulation.

D.1 CornerSchematic

The schematic shown in Figure 29 is drawn using:

```
amrcp  Chp2/schematic.2
amrita run_schematic
amrps  ps/schematic.ps
```

```
proc CornerSchematic
   PlotDomain Twall=8
   DrawShock      xo=10,yo=40,dx=0.6,dy=40
   DrawRightArrow xo=12,yo=60,dx= 10,dy=0.6
   DrawMeasuringStrut x1=0,y1=45,x2=10,y2=45
   LatexLabel label=\$X_S\$,xo= 1,yo=46,height=6
   LatexLabel label=\$M_S\$,xo=15,yo=62,height=6
   LatexLabel label=\$A\$  ,xo=-5,yo=38,height=4
   LatexLabel label=\$B\$  ,xo=-5,yo=77,height=4
   LatexLabel label=\$C\$  ,xo=80,yo=77,height=4
   LatexLabel label=\$D\$  ,xo=80,yo=-2,height=4
   LatexLabel label=\$E\$  ,xo=30,yo= 0,height=4
   LatexLabel label=\$F\$  ,xo=30,yo=38,height=4
end proc

...  driver script
```

and comes in useful for describing the keywords used to specify `BoundaryConditions`.

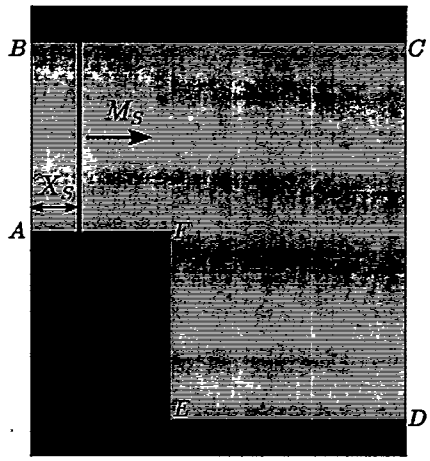


Figure 29: Schematic showing the initial conditions for a shock-diffraction problem. The backslashes are needed to prevent *Amrita* from acting on the $ symbols intended for LaTeX.

Such drawing capability should not come as a surprise, given that one of the purposes of *Amrita* is to generate standardized test output. Here the schematic was constructed entirely using standard library routines. Recall that an *HTML* listing for any of these routines can be found using the library procedure `Show`. For instance:

```
unix-prompt>amrita -c
amrita>Show procs=LatexLabel|Draw*|PlotDomain
```

would list all five procedures used to generate the schematic. Observe that the locations and sizes for the various text labels are supplied relative to the computational grid and not in physical page coordinates. Consequently the present code would also work for the multiple-cavity problem with no modifications whatsoever.

D.2 {N,S,E,W}bdy

This next version of the CornerProblem shown in Figure 29 contains a def BoundaryConditions block[50]:

```
amrcp  Chp2/corner.3
amrita run_corner_mk3
```

```
proc CornerProblem {
     Ms        =  1.25   # shock strength
   }

   def Domain
      lscale 1
      patch <1,41,w30,h40>
      patch <+, 1,w50,h80>
   end def

   W'quiescent  ::= <RHO=1,U=0,V=0,P=1>
   ShockWave Ms=$Ms, state1=quiescent,\
                     state2=post_shock

   def BoundaryConditions
      Nbdy domain: reflect
      Sbdy domain: reflect
      Ebdy domain: extrapolate
      Wbdy domain: prescribe   W'post_shock
      Sbdy along J=41 from I=1 to 30: reflect
      Wbdy along I=31 from J=1 to 40: reflect
   end def

end proc

EulerEquations
plugin amr_sol
CornerProblem
march 150 steps with cfl=0.8
```

so as to reduce the run-time error for *run_corner_mk3* to:

```
Error at line 29 of file run_corner:
cannot march: no SolutionField!
cannot march: no solver!


Line 29 is:
march 150 steps with cfl=0.8

error near:
150 steps with cfl=0.8
```

[50] In practice, for reasons given later, the above def BoundaryConditions block would be replaced by a second slightly terser version. Note that the resolution of the domain has been increased to a more respectable level than that in *run_corner_mk1* and *run_corner_mk2*. Also, pre- and post-shock states are now defined using the constructs described in §B.1.1, and the controlling Mach number is made a parameter, Ms, of CornerProblem and given a default value of 1.25. Syntactically, the definition of W'quiescent and the call to ShockWave could appear inside the def BoundaryConditions block, but W'quiescent is also needed by def SolutionField and so, on the basis of impartiality, is best placed outside the def block.

The command:

```
Nbdy domain: reflect
```

instructs *Amr_sol* to employ reflecting conditions[51] along any boundary-segment of the domain which lies on the northern edge of the logical bounding box which just encompasses the domain. Thus segment BC (in Figure 29) would be treated as a solid wall when the time comes to run the simulation. Similarly:

```
Sbdy domain: reflect
```

requests reflecting conditions for the boundary segment ED, but says nothing about the segment AF as it does not form part of the bounding box. Alternatively:

```
Ebdy domain: extrapolate
```

requests zeroth order extrapolation from the interior and results in the segment DC being treated as an outflow boundary.

Boundary segments which are not coincident with the domain's bounding box can be specified using C_o coordinates explicitly, as in:

```
Sbdy along J=41 from I=1 to 30: reflect
```

which fixes the segment AF, or:

```
Wbdy along I=31 from J=1 to 40: reflect
```

which fixes EF. However, such prescriptions would need to be changed each time the corner altered in cell resolution. Therefore a better prescription for segments AF and EF is:

```
Sbdy default: reflect
Wbdy default: reflect
```

which provides *Amr_sol* with a standing order to employ reflecting boundary conditions for those western and southern patch-boundaries not covered by an explicit instruction.

For segment AB, the explicit prescription:

```
Wbdy  domain: prescribe   W'post_shock
```

takes precedence over the default instruction:

```
Wbdy default: reflect
```

regardless of the order in which the two commands are posted.

In summary, the preferred way to prescribe boundary conditions for `CornerProblem` is:

```
def BoundaryConditions
   Nbdy  domain: reflect
   Ebdy  domain: extrapolate
   Wbdy  domain: prescribe   W'post_shock
   Sbdy default: reflect
   Wbdy default: reflect
end def
```

Again, the changes outlined here for `CornerProblem` are also appropriate for the `MultipleCavityProblem`, on p. 51, albeit an additional:

```
Ebdy default: reflect
```

would be needed to complete the specification of the boundary conditions (why?).

[51] The algorithmic details are given on the next page.

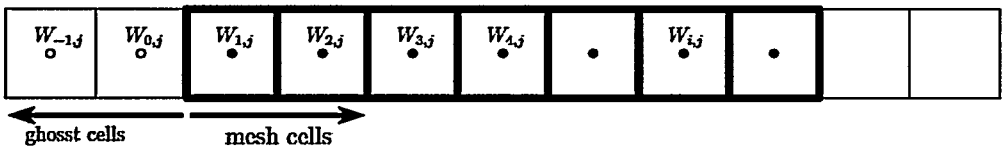
D.3 Time-Dependent Boundary Conditions

The subtleties of applying numerical boundary conditions preclude the possibility that a convenient set of pre-canned treatments can meet all needs[52]. Consequently, *Amr_sol* provides hooks which allow you to employ your own boundary condition code for when the built-in: `reflect`, `prescribe`, `noslip`, `extrapolate` and `periodic` prove deficient[27]. Such extensibility runs throughout *Amrita*'s design and affords tight control of the subtleties of a simulation. However, with some lateral thinking, expression templates can often do away with the need to add custom code. For instance, this script fragment (**`amrcp Chp2/CornerProblemMk2.amr`**):

```
fold::amrita'north { time-dependent boundary condition
   ... programmer notes
   Xs ::= $Xs+t[]*$Ms*sqrt(C'quiescent[])
   foreach q (RHO,U,V,P)
      $q'n ::= X[]<Xs[] ? $q\'post_shock[] : $q\'quiescent[]
   end foreach
   W'north ::= <RHO=RHOn[],U=Un[],V=Vn[],P=Pn[]>
   Nbdy  domain: prescribe W'north
}
```

provides a time-dependent boundary condition for side BC of the corner problem.

Internally, *Amr_sol* surrounds each mesh patch by NG rings of ghost cells[53] so that boundary conditions can be applied implicitly by priming the ghost cells with appropriate data[21]. If you examine the file *$AMRITA/src/amr_sol/bdy_lib.F*, you should be able to verify that for `EulerEquations`, with NG set to two, the following priming rules apply at a `Wbdy`[54]:



`extrapolate` $\Longrightarrow$ $\begin{pmatrix}\rho\\ \rho u\\ \rho v\\ E\end{pmatrix}_{-1,j} = \begin{pmatrix}\rho\\ \rho u\\ \rho v\\ E\end{pmatrix}_{0,j} = \begin{pmatrix}\rho\\ \rho u\\ \rho v\\ E\end{pmatrix}_{1,j}$

`reflect` $\Longrightarrow$ $\begin{pmatrix}\rho\\ \rho u\\ \rho v\\ E\end{pmatrix}_{-1,j} = \begin{pmatrix}\rho\\ -\rho u\\ \rho v\\ E\end{pmatrix}_{2,j}$ and $\begin{pmatrix}\rho\\ \rho u\\ \rho v\\ E\end{pmatrix}_{0,j} = \begin{pmatrix}\rho\\ -\rho u\\ \rho v\\ E\end{pmatrix}_{1,j}$

`noslip` $\Longrightarrow$ $\begin{pmatrix}\rho\\ \rho u\\ \rho v\\ E\end{pmatrix}_{-1,j} = \begin{pmatrix}\rho\\ -\rho u\\ -\rho v\\ E\end{pmatrix}_{2,j}$ and $\begin{pmatrix}\rho\\ \rho u\\ \rho v\\ E\end{pmatrix}_{0,j} = \begin{pmatrix}\rho\\ -\rho u\\ -\rho v\\ E\end{pmatrix}_{1,j}$

[52] Strictly speaking, the boundary condition used by `CornerProblem` for segment AB is over prescribed when the flow behind the shock is subsonic, as is the case for $M_S = 1.25$ (assuming $\gamma = 1.4$). Here, this transgression makes no real odds, but the same is not true for the nozzle calculation on p. 55.

[53] The call `ArraySizes NG=`*num* can be used to set the number of ghost cells. Large values of NG, say > 4, are not encouraged on efficiency grounds. Also, accuracy problems might arise, if NG is larger than the refinement ratio r.

[54] This figure can be generated by typing:

```
unix-prompt>amrcp Chp2/ghost.1
unix-prompt>amrita ghost_cells
```

E def SolutionField

Amr_sol allows initial flow conditions to be prescribed within a `def SolutionField` block using the `setfield` command. For instance, `CornerProblem` requires five lines be placed immediately after the `def BoundaryConditions` block[55]:

```
amrcp  Chp2/corner.4
amrita run_corner_mk4
```

```
proc CornerProblem {
     Ms        =  1.25   # shock strength
     Xs        = 28.00   # shock position
   }

   def Domain
      lscale 1
      patch <1,41,w30,h40>
      patch <+, 1,w50,h80>
   end def

   W'quiescent  ::= <RHO=1,U=0,V=0,P=1>
   ShockWave Ms=$Ms, state1=quiescent,\
                     state2=post_shock

   def BoundaryConditions
      Nbdy  domain: reflect
      Ebdy  domain: extrapolate
      Wbdy  domain: prescribe   W'post_shock
      Sbdy default: reflect
      Wbdy default: reflect
   end def

   def SolutionField
      setfield W'quiescent
      setfield W'post_shock X[]<$Xs
   end def
   makefield

end proc

EulerEquations
plugin amr_sol
CornerProblem
march 150 steps with cfl=0.8
```

so as to whittle the *run_corner_mk4* error down to:

```
Error at line 35 of file run_corner:
cannot march: no solver!


Line 35 is:
march 150 steps with cfl=0.8

error near:
150 steps with cfl=0.8
```

[55]The default shock position, `Xs`, is also added to the `CornerProblem` parameter block.

The first `setfield` command requests that the `quiescent` state, be used to set the solution vector for every cell in the current computational domain. The second `setfield` command employs a qualifier, `X[]<$Xs`, and so would only overwrite the `quiescent` field with the `post_shock` state for those cells whose centre-of-gravity, `X[]`, lies to the left of the shock position, `$Xs`. Strictly speaking, a `def SolutionField` block does nothing more than create a list of actions to follow and it is the `makefield` command which activates the actual process of updating the field solution.

This script, which outputs Figure 30:

```
... procedure definitions
NullEquationSet
plugin amr_sol
autoscale on 0,0,10,6
postscript on
plotfile ps/cell.ps
DrawAxes
DrawCell
AnnotateCell
```

```
amrcp  Chp3/cell.sch
amrita cell_schematic
amrps  ps/cell.ps
```

identifies a number of *Amr_sol*'s pre-defined expression templates, including `X[]`[56].

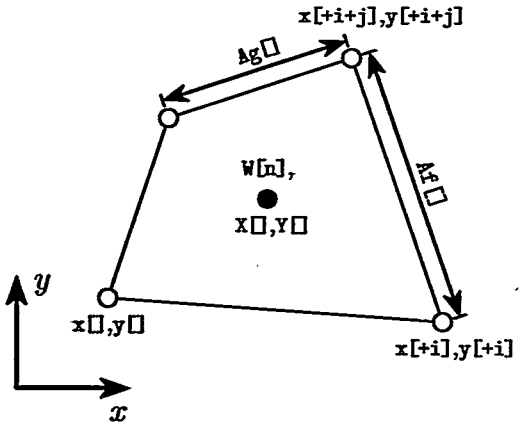


Figure 30: Schematic showing several of *Amr_sol*'s pre-defined expression templates.

Although this flow prescription:

```
def SolutionField
   setfield W'quiescent  X[] >= $Xs
   setfield W'post_shock X[] <  $Xs
end def
```

is nominally equivalent to the one used for `CornerProblem`, because of the vagaries of floating-point round-off errors, it does not guarantee that every mesh cell will receive data. In general, regardless of the choice of programming language, compound logical tests, whose members are nominally mutually exclusive, should always be cast in the form where one member acts as a catch-all to safeguard against unanticipated events.

[56]Two common templates missing form this Figure are: `t[]` which returns the current solution time and `Vol[]` which returns the volume of a cell.

E.1 Richtmeyer-Meshkov Problem

Arbitrarily complex initial conditions may be built-up by stringing multiple `setfield` commands together, each with their own separate qualifiers. But because *Amrita* uses thermodynamic states made up from expression templates, and not plain numbers, there is usually no need to employ more than a handful of `setfield` commands. For instance, the wavy interface shown in Figure 31, which might be required for a Richtmeyer-Meshkov problem[29], is created using just one `setfield` command.

```
amrcp  Chp2/RM.1
amrita run_RM
amrps  ps/RM.ps
```

```
...  preparatory script
def SolutionField
   RHOl      ::= 1
   RHOh      ::= 5
   interface::= 55+4*cos(Y[]/80*3*PI)
   wt        ::= (X[]-interface[])/5
   wt        ::= wt[]>1 ? 1 : (wt[]<-1 ? -1 : wt[])
   RM        ::= (RHOl[]+RHOh[])/2+wt[]*(RHOh[]-RHOl[])/2
   setfield <RHO=RM[],U=0,V=0,P=1>
end def
makefield
PlotDomain Twall=8
m<0>
plot RHO[] contours 10 levels
```

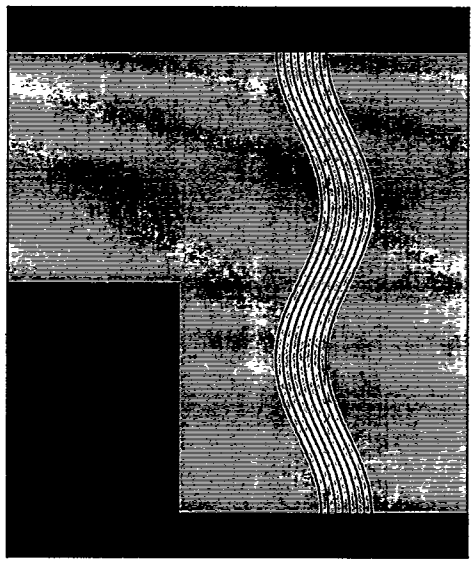


Figure 31: Smeared wavy interface between a light fluid on the left and a heavy fluid on the right.

Observe, the use of the ternary operator `a?b:c` in the second definition of `wt[]`: this operator works exactly as in *C*, i.e. if `a` is true, it returns `b`, otherwise `c` is returned. Consequently, `wt[]` is constrained to return values between -1 and $+1$ and can be used to determine whether a cell lies within the smeared interface ($wt = \phi$, $-1 < \phi < 1$), or to the left ($wt = -1$), or to the right ($wt = 1$) of the interface. Thus it is possible to define a single function `RM[]` which describes the entire density field. Also note the use of the system constant, `PI`, which *Amrita* provides gratis.

E.2 Inclined Measuring Gauge

This script:

```
... set autopath
EulerEquations
plugin amr_sol
postscript on
plotfile ps/gauge.ps
GaugeProblem
GaugeSchematic
```

```
amrcp  Chp2/gauge.1
amrita run_gauge
amrps  ps/gauge.ps
```

produces the schematic shown in Figure 32, and was written to describe a test problem in which a planar shock wave impinges on an inclined, rectangular measuring gauge of a heavier fluid. Although the geometry is straightforward, the appropriate construction of the required `SolutionField` necessitates a small amount of lateral thinking, and provides a nice counterpoint to mindless coding[57].

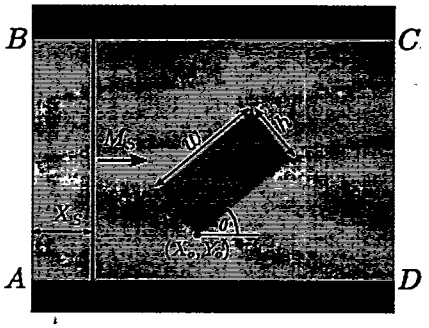


Figure 32: Schematic showing a planar shock about to impinge on an inclined, rectangular measuring gauge of heavier fluid.

The `quiescent` and `post_shock` states are dealt with as in `CornerProblem`. The trick for dealing with the inclined, measuring gauge is to define a new coordinate system aligned to the gauge using a complex expression template, `Z[]`, as shown here:

```
fold::amrita'lateral_thinking { SolutionField
   def SolutionField
      Z ::= {X[]-$Xo,Y[]-$Yo}*exp({0,-rad($theta)})
      in_gauge ::= (Re(Z[])>=0 && Re(Z[])<=$w) && \
                   (Im(Z[])>=0 && Im(Z[])<=$h)
      W'gauge  ::= W'quiescent<RHO*=$Dratio>
      setfield W'quiescent
      setfield W'post_shock X[]<$Xs
      setfield W'gauge      in_gauge[]
   end def
}
```

It is then a trivial matter to construct an expression template, `in_gauge[]`, to provide a boolean test for whether or not a cell falls inside the inclined gauge[58].

[57] If you find yourself writing contorted *Amrita* scripts, there is an evens chance that you are approaching the problem in the wrong manner. When this happens, step back from what you are doing and re-appraise your solution strategy. If you cannot find a better approach, and are convinced the fault lies with *Amrita*, please take the time to distil the contortion down to a clean *mailit* and log an official request for a language upgrade.

[58] The `setfield` command can circumvent complications such as a cell straddling the edge of the gauge by expeditious sub-sampling of the computational grid.

E.3 ZND Detonation Wave

The *run_cellular.mailit*, from §6 of lecture 1, contains a procedure `CellularProblem` which uses this `SolutionField`:

```
def SolutionField
   set znd = $io/$znd::header
   Xd     ::= X[]-$Xd
   W'znd  ::= <RHO= interp($znd.RHO,Xd[]),\
               U  = interp($znd.U  ,Xd[]),\
               V  = 0                    ,\
               P  = interp($znd.P  ,Xd[]),\
               Z  = interp($znd.Z  ,Xd[])>
   W'disturbance ::= <RHO=1.0,U=0,V=0,P=1.5,Z=1>
   extent -> xo,yo,dx,dy

   hot_spot ::= (abs(Y[]-$dy/2)<2) && (Xd[]>3) && (Xd[]<5)
   setfield W'znd
   setfield W'disturbance  hot_spot[]
end def
```

to prescribe a travelling ZND detonation wave[11], and is an example of how to prescribe non-analytic initial conditions.

The `interp` function[59] interpolates tabulated profiles of density, velocity, pressure and unburnt fuel through the detonation wave to return values for `setfield` to paint into the computational domain. The template, `Xd[]`, is a simple mapping to position the wave at the point `$Xd` in the computational domain, travelling from left to right.

If you are unfamiliar with the structure of a ZND wave, this script:

```
ReactiveEulerEquations {
   space = one-dimensional
   bcg   = yes
}
plugin amr_sol
set znd::gamma = 1.2
set znd::E     = 50.0
set znd::d     = 1
foreach Q (0.1,1,5,10,50,100)
   set znd::Q = $Q
   ZndProfile {
      io  = znd-1a/Q$Q
      doc = yes
   }
end foreach
```

```
amrcp   vki/znd.1
amrita  run_znd-1a
cd      znd-1a/Q50
amrps   profile.ps
cd      code/f77src
amrgi   znd-1a.F
```

outputs a number of pages similar to Figure 33 which depicts the detonation structure for a one-step Arrhenius reaction model. Details of the controlling parameters: heat release (Q), overdrive (d), activation energy (E) and ratio of specific heats (γ), can be found by running a suitably modified version of the script example on p. 40. The `bcg` parameter requests the production of the shared object *znd-1a* which is needed to compute the ZND profiles.

[59] Recall the discussion of `AMR::INTERP` on p. 54.

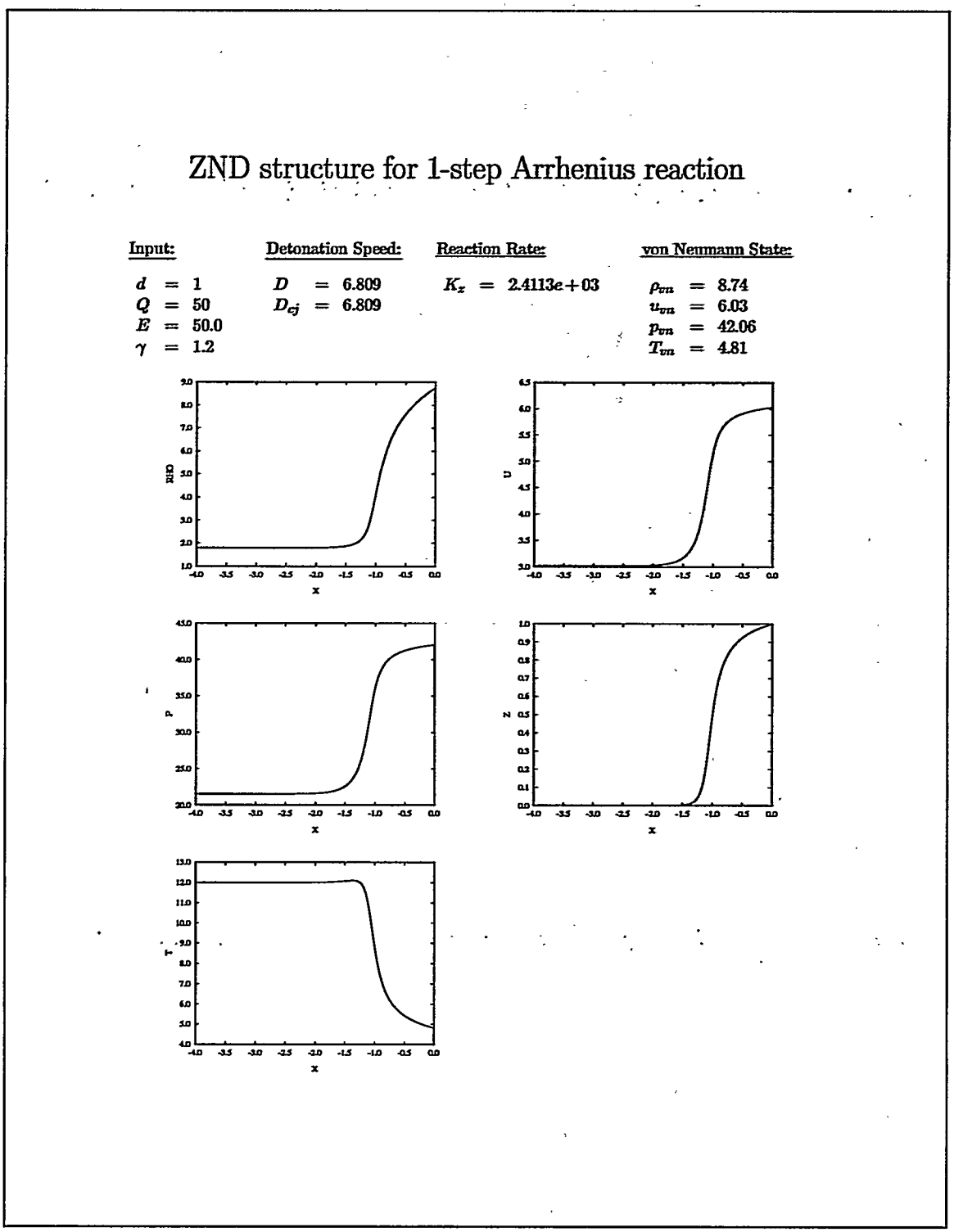


Figure 33: Output produced by *run_znd-1a*. The structure of the detonation is controlled by the four parameters $\{d, Q, E, \gamma\}$. The reaction rate, K_z, is a free parameter which is chosen such that $Z = \frac{1}{2}$ at $X = -1$. The *von Neumann* state refers to the point, $X = 0$, just behind the lead shock-front of the detonation. Note only half the shock jump in pressure is shown, i.e. `P'quiescent = 1.`

The one-step Arrhenius reaction model is attractive in that it is tractable to analysis, but it has several weaknesses: (i) it proves overly chaotic in the Chapman-Jouguet limit ($d = 1$); (ii) it has no mechanism for quenching; (iii) it does not mimic induction zones properly. On the other hand, simulations with full blown reaction-kinetics are orders of magnitude more expensive to compute and so prone to misinterpretation due to the much poorer grid resolution which can be afforded. This next script utilizes a three-step chain-branching reaction model which lies somewhere between these two extremes:

```
amrcp   vki/znd.2
amrita  run_znd-3cb
cd      znd-3cb
amrps   Tb.ps
cd      code/f77src
amrgi   znd-3cb.F
```

```
ReactiveEulerEquations {
   space = one-dimensional
   bcg   = yes
   model = 3cb
}
plugin amr_sol
set znd::gamma    = 1.2
set znd::Qf       = 3.0
set znd::Qd       = 0.0
set znd::Ei       = 20.0
set znd::Eb       = 8.0
set znd::Ti       = 3.0
set znd::Tb       = 0.8
set znd::d        = 1.2
set range = 0.8, 0.85, 0.9, 0.95
foreach Tb ($range)
   set znd::Tb = $Tb
   ZndProfile {
      io  = znd-3cb/Tb$Tb
      Xc  = 15
      RKi = 948.14
   }
end foreach
...  plot results
```

and helps place §1.1 in context.

Figure 34 demonstrates the change in detonation structure when just one of the eight controlling parameters is varied. Observe the increase in the induction length and the decrease in the radical spike as as T_B is increased. These variations hint at the observed, dramatic variations in the dynamical behaviour of the wave (see [30] for details). Here it is sufficient to note that the bulk of the heat release (i.e. the fire-region) occurs in the vicinity of the radical spike, and experience shows that failure to resolve this narrow, but smooth region, leads to completely erroneous results (see Figure 22). Consequently, even for this one reaction model, the mesh spacing needed to resolve the reaction zone properly is highly variable, and the common practice of choosing a fixed number of cells within the half-reaction length is inappropriate. Thus the reaction-width figures used in §1.1 were strictly chosen to reflect the care with which it is necessary to resolve the internal structure of a detonation wave, under *certain* circumstances. In *other* circumstances, the predicted behaviour may be fairly insensitive to grid resolution, allowing much coarser grids to be used. At a practical level, if it is to stand the test of time, a general purpose computing system must be designed with the pessimistic scenario in mind[60].

[60]Today's research problems, requiring cutting-edge numerical techniques, are tomorrow's homework assignments, covering everyday solution techniques.

3-Step Chain-Branching Reaction Model

The essential dynamics of chain-branching reactions can be represented by three main stages, initiation, chain-branching and chain-termination with the rates k_I, k_B and k_C respectively:

$$I: \quad F \quad \rightarrow \quad Y \quad \text{(Initiation);} \qquad \text{Rate} \quad k_I = \exp\left(\frac{1}{\epsilon_I}\left(\frac{1}{T_I} - \frac{1}{T}\right)\right)$$

$$B: \quad F+Y \quad \rightarrow \quad 2Y \quad \text{(Chain-branching);} \qquad \text{Rate} \quad k_B = \exp\left(\frac{1}{\epsilon_B}\left(\frac{1}{T_B} - \frac{1}{T}\right)\right)$$

$$C: \quad Y \quad \rightarrow \quad P \quad \text{(Chain-termination);} \qquad \text{Rate} \quad k_C = 1$$

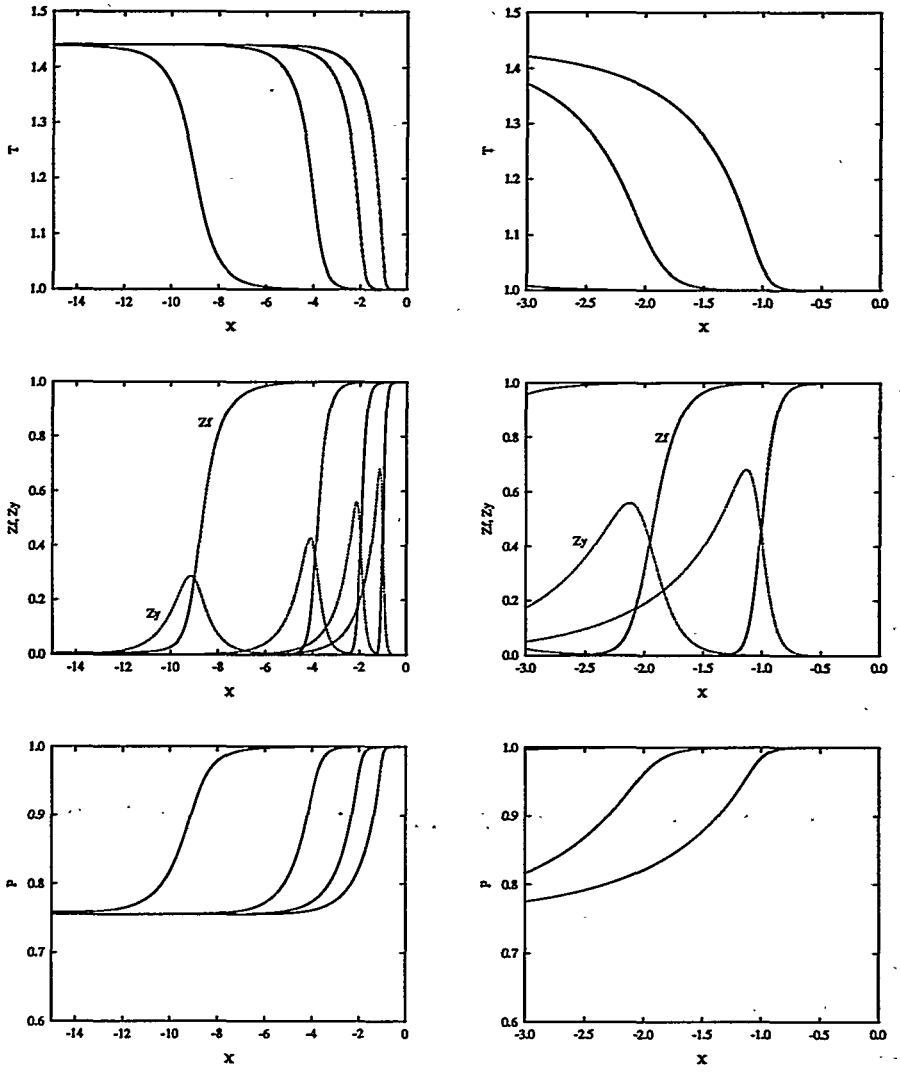


Figure 34: Output produced by *run_znd-3cb*. Note that the data has been scaled relative to the *von Neumann* state at $X = 0$, hence $P_{VN} = T_{VN} = 1$. The structure of the detonation is controlled by the eight parameters $\{d, Qf, Qd, Ei, Eb, Ti, Tb, \gamma\}$. Here, Tb is varied from 0.8 to 0.95 in steps of 0.05. As Tb increases, so the induction length increases and the peak in radical, Zy, decreases. For $Tb = 0.86$, one-dimensional simulations using a second-order TVD scheme needed 320 mesh points in the distance $X = 0$ to $Zf = \frac{1}{2}$ to reach a grid converged answer, and had to be propagated over 1500 half-reaction lengths in the process[30], i.e. to repeat this simulation using a uniform mesh would require 480,000 mesh cells. This statistic does not bode well for the viability of grid-resolved, multi-dimensional simulations with the three-step chain-branching reaction model.

F def MeshAdaption

With *Amr_sol*, once a `SolutionField` has been defined, a suitable `solver` can be loaded to integrate the prescribed flow forward in time. For instance, this *run_corner_mk5*:

```
... define CornerProblem
EulerEquations
plugin amr_sol
BasicCodeGenerator {
   solver = roe_fl
   scheme = flux-limited'operator-split
}
CornerProblem Ms=2, Xs=10
solver code/roe_fl
march 150 steps with cfl=0.8
... plot results
```

```
amrcp  Chp2/corner.5
amrita run_corner_mk5
cd     ps
amrps  coarse_grid.ps
amrps  coarse_flow.ps
```

invokes *BCG* to obtain a `solver` *roe_fl*, and marches `150 time steps` to produce the flow shown in Figure 35 (b). However, compared to the output of *my.script* from lecture 1, reproduced here in Figure 36, the flow is grossly under-resolved.

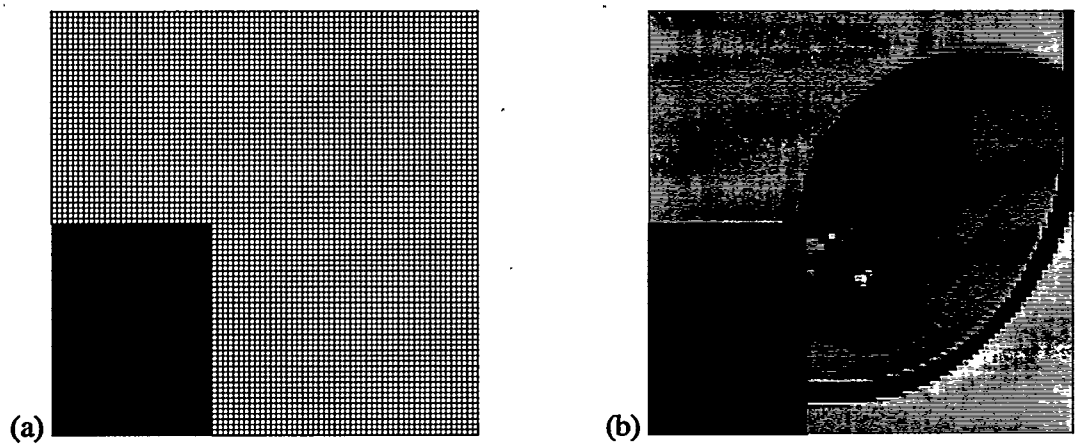


Figure 35: Output from *run_corner_mk5* (note the presence of the startup-errors).

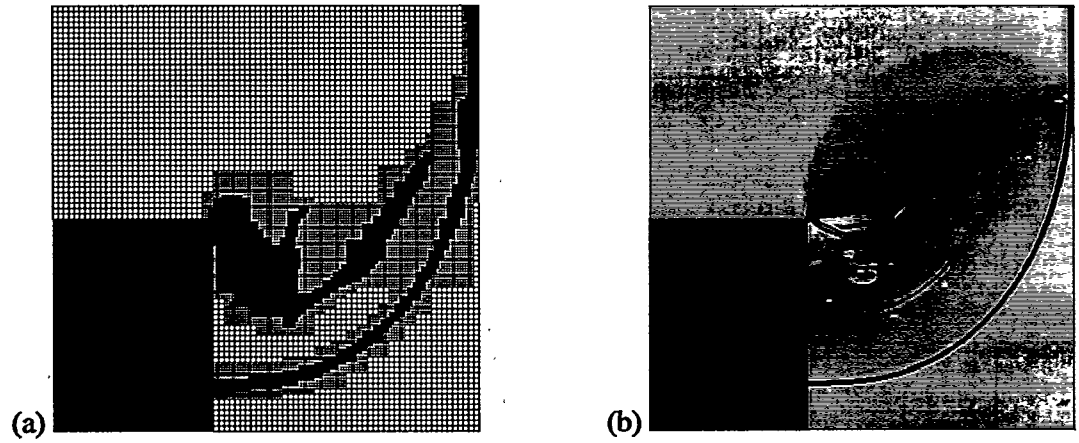


Figure 36: Output from *my.script* (reproduced from lecture 1).

F.1 Tiered Grid System

As explained in Appendix A, *Amr_sol* employs a tiered grid system:

$$G = \{G_0, G_1, \ldots, G_l, \ldots, G_{lmax}\}$$

in which the higher the grid tier, l, the smaller the mesh spacing. Figure 36 illustrates how this grid system allows the `plugin` to refine the computational domain locally to improve the resolution with which key flow features are captured.

Many *Amr_sol* commands can be given a grid qualifier to restrict their operation to a subset of the full grid, G. For instance, Figure 36 (a) was generated using:

```
plot grids
```

but could equally well have been produced using either:

```
plot grids {G}     or     plot grids {G0,G1,G2}
```

The qualifier, `{G}`, denotes the entire set of grid tiers, while `{G0,G1,G2}` denotes the coarsest three tiers. Sometimes it is more convenient to exclude a specific list of grids using the not operator '`!`'. For example, `{!G}` is the same as the empty set `{}` and is useful for turning a command off.

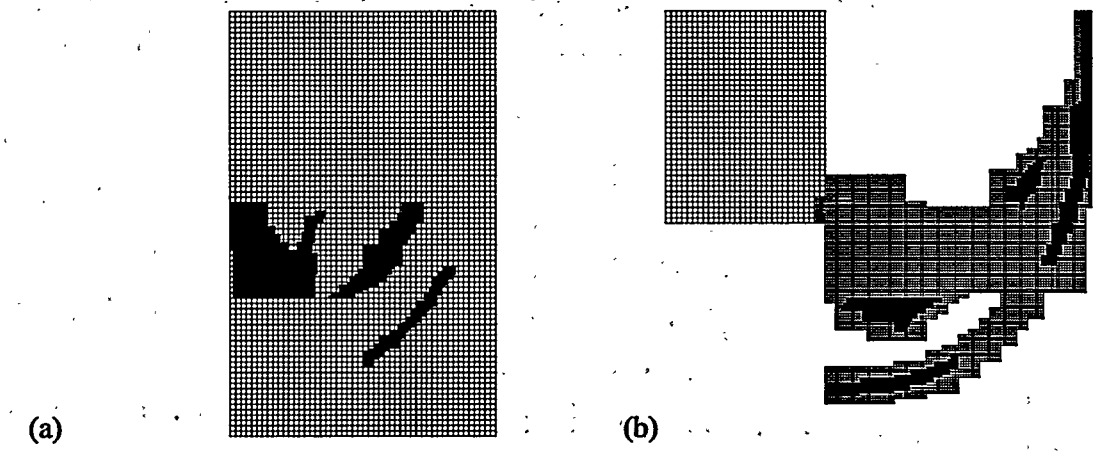
Recall from §C that G_0 consists of a set of logically rectangular patches anchored in a Cartesian space C_0, similarly the grid G_l consists of a set of patches anchored in a Cartesian space C_l. Thus *Amr_sol*'s grid-system may be viewed as a straightforward collection of patches which are labelled consecutively up through the grid tiers, using label 1 for the first patch of G_0. This enables a grid qualifier to be specified in terms of patch indices, say:

```
plot grids {2,60-120}
```

which requests a total of 62 patches be drawn, or a combination of grids and patches, such as:

```
plot grids {G1-G2,1,!60-120}
```

These two `plot` commands produce the non-intersecting subsets of G shown in Figure 37.



(a) (b)

Figure 37: Grid plots produced using the script *run_gridlist* (**`amrcp vki/gridlist.1`**): (a) `plot grids {2,60-120}`, (b) `plot grids {G1-G2,1,!60-120}`; c.f. Figure 36 (a).

F.2 Activating Mesh Adaption

Amr_sol's mesh refinement machinery is controlled in two steps. The first step, which is discussed below, merely activates the machinery. The second step, which is described in the next section, defines the criteria by which the grid is adapted.

The script *run_corner_mk6*:

```
proc CornerProblem {
     Ms        =  1.25    # shock strength
     Xs        = 28.00    # shock position
     lmax      =  2       # grid levels
     r         =  2       # refinement ratio
   }

   def Domain
      lscale 1
      patch <1,41,w30,h40>
      patch <+, 1,w50,h80>
   end def

   W'quiescent  ::= <RHO=1,U=0,V=0,P=1>
   ShockWave Ms=$Ms, state1=quiescent,\
                     state2=post_shock

   def BoundaryConditions
      Nbdy  domain: reflect
      Ebdy  domain: extrapolate
      Wbdy  domain: prescribe   W'post_shock
      Sbdy default: reflect
      Wbdy default: reflect
   end def

   def SolutionField
      setfield W'quiescent
      setfield W'post_shock X[]<$Xs
   end def
   makefield

   def MeshAdaption
      adaption  on
      lmax      $lmax
      r         $r
   end def

end proc

EulerEquations
plugin amr_sol
CornerProblem Ms=2, Xs=10
solver code/roe_fl
march 150 steps with cfl=0.8
...  plot results
```

```
amrcp  Chp2/corner.6
amrita run_corner_mk6
cd     ps
amrps  coarse_grid.ps
amrps  coarse_flow.ps
```

turns mesh adaption on and requests a grid structure {G0,G1,G2} with a refinement ratio

of 2 in cell spacing:

$$\frac{\Delta l_0}{\Delta l_1} = \frac{\Delta l_1}{\Delta l_2} = 2$$

Thus 16 cells from G2 cover the same area as 1 cell from G1. However, the `MeshAdaption` does not have any effect until `RefinementCriteria` are selected, therefore the above script still produces the washed out results shown in Figure 35.

If desired, anisotropic refinement can be selected using:

```
fold::amrita'anisotropic {  MeshAdaption
   def MeshAdaption
      adaption  on
      lmax      $lmax
      rI        2*$r    {G1}
      rJ        1       {G1}
      rI        1       {G2}
      rJ        2*$r    {G2}
   end def
}
```

```
amrcp  Chp2/aniso.corner
amrita aniso_corner
amrps  ps/aniso_grid.ps
amrps  ps/aniso_flow.ps
```

to give nominally the same 16-fold increase in resolution. However, when such a simulation is run (see Figure 38), the `solver` generates a `NaN` shortly after the shock starts diffracting around the apex of the corner[61]. Internally the `solver` could likely be modified to overcome this robustness problem, but such a fixup would only delay matters until another set of contrived circumstances threw it out of kilter. *Amr_sol* traps a number of grid configurations which are unworkable[62], but the restrictions are kept to a minimum so as not to encroach on legitimate applications. For example, anisotropic refinement is needed for simulations of shock-boundary layer interactions[21], and so cannot be dismissed because of its poor showing here.

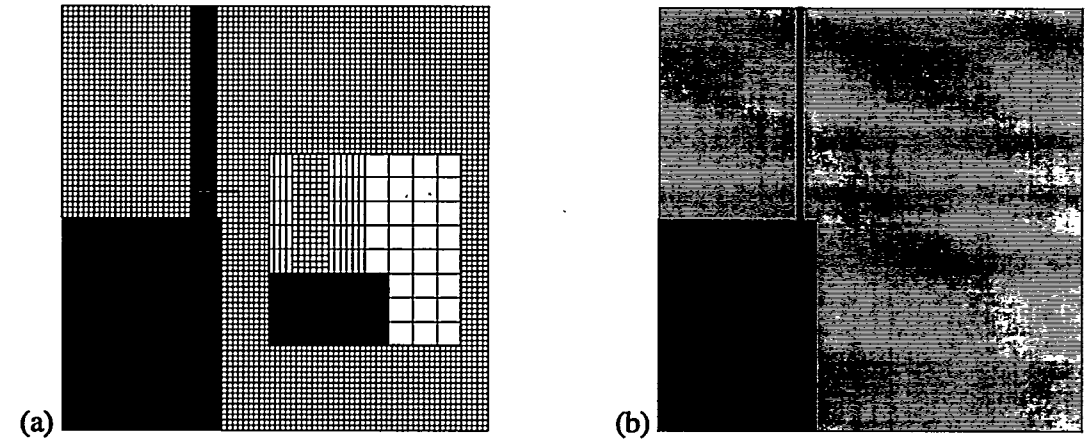


Figure 38: Output from *aniso_corner*. *Amr_sol* employs a fixed refinement ratio between any two grid levels, but the ratio can vary from one pair of levels to the next. Here, the use of anisotropic refinement causes the `solver` to generates a `NaN` shortly after the shock starts diffracting around the corner, and so the simulation is stopped after just 30 time steps.

[61]The `march` command stops calling the `solver` as soon as it detects a `NaN`.

[62]Internally *Amr_sol* can work with an arbitrary refinement ratio, but the commands: `r`, `rI` and `rJ`, restrict input to integers less than 10. This restriction was put in place after a student attempted to run two grid-levels of refinement, each with a refinement ratio of 100, i.e. one G_0 cell leading to 100 million G_2 cells.

G def RefinementCriteria

Lecture 1 outlined how *Amr_sol* can be taught heuristic `RefinementCriteria` to enable it to adapt its computational grid to an evolving flow solution. For `CornerProblem`, a call to the library routine `DensityGradient` is sufficient to instruct *Amr_sol* to refine the initial shock-position based on its associated discontinuity in density.

```
proc CornerProblem {
     Ms      =  1.25   # shock strength
     Xs      = 28.00   # shock position
     lmax    =  2      # grid levels
     r       =  2      # refinement ratio
   }

   def Domain
      lscale 1
      patch <1,41,w30,h40>
      patch <+, 1,w50,h80>
   end def

   W'quiescent  ::= <RHO=1,U=0,V=0,P=1>
   ShockWave Ms=$Ms, state1=quiescent,\
                     state2=post_shock

   def BoundaryConditions
      Nbdy  domain: reflect
      Ebdy  domain: extrapolate
      Wbdy  domain: prescribe   W'post_shock
      Sbdy default: reflect
      Wbdy default: reflect
   end def

   def SolutionField
      setfield W'quiescent
      setfield W'post_shock X[]<$Xs
   end def
   makefield

   def MeshAdaption
      adaption  on
      lmax      $lmax
      r         $r
   end def

   def RefinementCriteria
      DensityGradient
   end def
end proc

EulerEquations
plugin amr_sol
CornerProblem Ms=2, Xs=10
solver code/roe_fl
march 150 steps with cfl=0.8
...  plot results
```

```
amrcp   Chp2/corner.7
amrita  run_corner_mk7
cd      ps
amrps   poor_grid.ps
amrps   poor_flow.ps
```

But as Figure 39 shows, `DensityGradient`:

```
proc DensityGradient {
     Ms          = 2
     tolerance #= sprintf("%.4f",$Ms<2?0.1*($Ms/2)**2:0.1)
     grid        = {G}
   }
   setflags [ooo|oxx|ooo] abs(RHO[+i]-RHO[]) >($tolerance) $grid
   setflags [oxo|oxo|ooo] abs(RHO[+j]-RHO[]) >($tolerance) $grid
end proc
```

is insufficient for the simulation proper which contains a contact-surface, once the shock starts to diffract around the apex of the corner. For this reason, *my.script* employed:

```
def RefinementCriteria
   DensityGradient
   if($phase>1) ContactSurface
end def
```

If you recall from lecture 1, `ContactSurface` is not used for the first phase of the simulation so as to avoid flagging the start-up errors which occur when the prescribed shock smears to the actual profile supported by the `solver`.

The *mk7* version of `CornerProblem` introduces a second kind of start-up error in that grids G_1 and G_2 are not given explicit initial conditions. Instead, the procedure relies on *Amr_sol* interpolating the G_0 solution when it adds the extra grid tiers during the course of the flow integration. Fortunately, this start-up error is entirely avoidable using the script:

```
do 1=1,$lmax
   adapt
   makefield
end do
```

which explicitly invokes the adaption machinery to add one new grid tier, and then overwrites the interpolated solution with the prescribed `SolutionField`. Note, however, once the simulation is underway (i.e. `march` is called), the required sequencing of grid adaptions is too involved to be left under casual-user control, and so is fully automated.

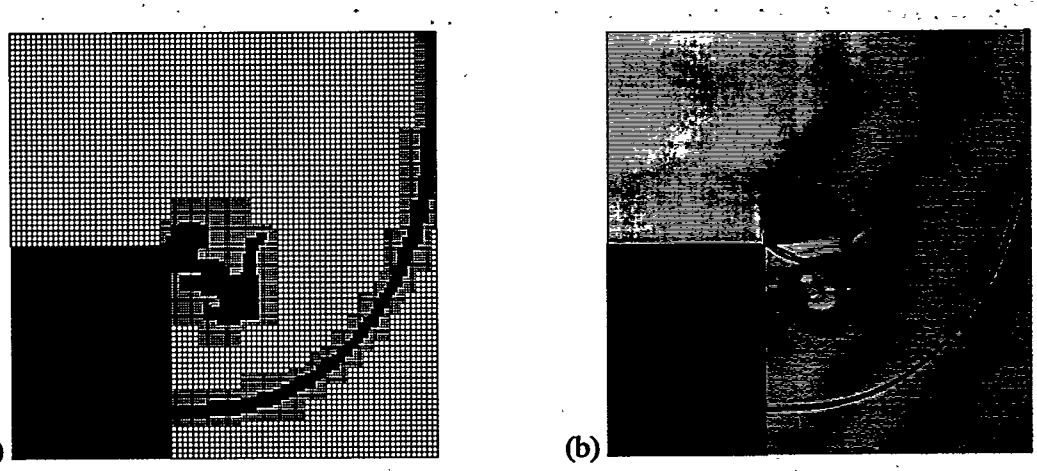


Figure 39: Output from *run_corner_mk7*. The refinement criteria set by `DensityGradient` is insufficient to keep track of the contact surface which separates fluid induced into motion by the planar, incident shock-front from fluid induced into motion by the curved, diffracted shock-front.

G.1 Tunable Parameters

In the absence of solid theoretical criteria[63] mesh refinement algorithms tend to employ heuristic monitor functions to determine where local refinement should take place to reduce error bounds to acceptable limits[20]. For instance, the following is useful for determining whether two neighbouring cells, which provide a pair of left and right states for their common interface, lie in the vicinity of a numerically smeared contact surface:

$$\frac{|\rho_r - \rho_l|}{\rho_r + \rho_l} > \frac{|p_r - p_l|}{p_r + p_l} \quad \cap \quad \frac{|\rho_r - \rho_l|}{\rho_r + \rho_l} > \text{round off.}$$

This test can be constructed using the command `setflags` and squirreled away into an *Amrita* procedure for general use:

```
proc ContactSurface tol=0.002, grid={G}
   set test = fRHO[]>fP[] && fRHO[]>$tol
   fP   ::= abs(P[+i]-P[])/(P[+i]+P[])
   fRHO ::= abs(RHO[+i]-RHO[])/(RHO[+i]+RHO[])
   setflags [ooo|oxx|ooo] $test $grid
   fP   ::= abs(P[+j]-P[])/(P[+j]+P[])
   fRHO ::= abs(RHO[+j]-RHO[])/(RHO[+j]+RHO[])
   setflags [oxo|oxo|ooo] $test $grid
end proc
```

The `setflag` command provides *Amr_sol* with a `$test` to apply to a mesh cell in a `$grid` to determine if the cell, or any of its neighbours, should be refined when the time comes to adapt the grid tier above it[64]. Thus the line:

```
setflags [ooo|oxx|ooo] $test $grid
```

requests that whenever the test:

```
fRHO[]>fP[] && fRHO[]>$tol
```

evaluates as true (that is non-zero) for a cell (i, j), in `{G}`, then the cells identified by an `x` in the flag mask `[ooo|oxx|ooo]` need refining. This flag mask is centred on (i, j) and is laid out by rows:

o	o	o
o	x	x
o	o	o

[63] A classical technique such as Richardson extrapolation[12] is strictly invalid when: (i) the numerical solution is non-smooth and so not well represented by a Taylor series, e.g. a smeared shock-wave; (ii) the formal order of accuracy of the integration scheme is not known or varies from point to point, e.g. TVD shock-capturing schemes. The given reference also stresses that the technique should be "used with caution and discrimination." This sound advice applies to any means of error estimation, including the ones presented here. Using *Amrita*'s dynamic linking mechanism, a `setflags` command can make a `fn` call to a user-supplied routine to compute any appropriate error estimate, and so the construct should be able to exploit improved techniques, as and when they are developed.

[64] The grid G_l is moved by examining the solution on G_{l-1}. This is done because: (i) it reduces the operation count, as there are far fewer cells in G_{l-1} than G_l; (ii) smeared discontinuities are steepened when back projected and so are best detected on G_{l-1} even though the grid is coarser than G_l. The subtleties of this second point, which runs against the normal grain of accuracy arguments, are detailed in[21].

and so the targeted cells are (i,j) and $(i+1,j)$. Alternatively, the mask `[oxo|xoo|oxo]`:

o	x	o
x	o	o
o	o	x

targets $(i,j+1)$, $(i-1,j)$ and $(i+1,j-1)$.

Although `setflags` can be given a grid qualifier to determine which subset of the computational grid will be tested for refinement, because of the way the adaption process is orchestrated, explicit patch indices are excluded. Therefore, for `CornerProblem` you could write:

```
setflags [ooo|oxo|ooo] 1 {G0}
```

to request that G_0 be completely refined, but the nominally equivalent:

```
setflags [ooo|oxo|ooo] 1 {1,2}
```

generates an error.

To facilitate the development of complex monitor functions, where multiple `setflags` commands must be strung together, `plot flags` identifies those grid cells which would be flagged for refinement, given the current `RefinementCriteria`. An example of its use has already been shown in lecture 1. This script, which reads the output from *my.script*, illustrates the principal weakness of heuristic refinement criteria, namely their inevitable reliance on tunable parameters:

```
amrcp   vki/tol.1
amrita  vary_tol1
cd      flags
amrps   vary_tol1.ps
```

```
EulerEquations
plugin amr_sol
postscript on
flowin io/Corner5
autoscale
LatexHead pagesize=problem-sheet,dir=flags,file=vary_tol1.tex
...  latex title
LatexNupFig iup=3,jup=4
do n=1,12
   def RefinementCriteria
      set tol  #= sprintf("%.2f",$n*0.01)
      DensityGradient tolerance=$tol
   end def
   plotfile flags/ps/flags$n.ps
      PlotDomain
      m<0>
      plot flags {G1}
   plotfile
   LatexNupFig {
      file    = ps/flags$n.ps
      caption = tol=$tol
      width   = 5cm
   }
end do
LatexTail
Latex
```

Figure 40: Page output by *vary_tol1*. Regardless of how small the tolerance is made, `DensityGradient` does not do a good job of picking out the contact-surface. Consequently, choosing a tolerance based on an analysis of the distribution of the density gradient (e.g. [9]), although mathematically more rigorous than chosing one by experience, will be no more successful. If the contact surface is deemed to be an important feature of the flow, then a change in refinement criteria is called for, see Figure 41.

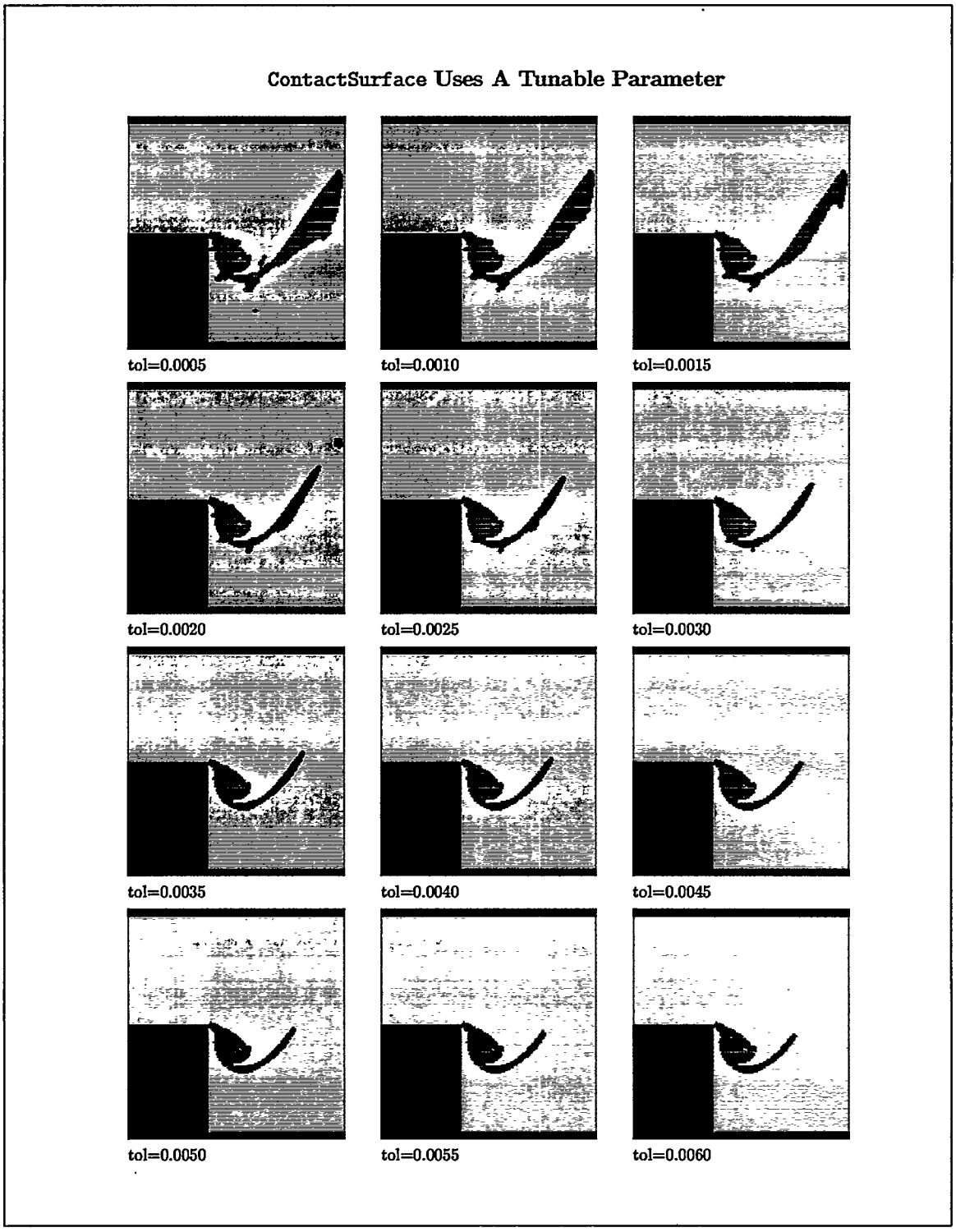


Figure 41: Page output by *vary_tol2*. The `ContactSurface` criteria is more adept at picking out the vortex core and slip line than `DensityGradient`, see Figure 40. Also, experience shows it requires less re-tuning between problems. This is not unexpected because the tolerance acts primarily as a noise filter for round-off errors and so does not play as active a selection role as the threshold used in `DensityGradient`.

References

[1] G.K. BATCHELOR, *An Introduction to Fluid Dynamics*, Cambridge University Press, 1967.

[2] J.B BDZIL & S.F. SON, *Engineering Models of Deflagration-to-Detonation Transition*, Los Alamos National Laboratory Report LA-12794-MS, 1995.

[3] G. BEN-DOR, *Shock Wave Reflection Phenomena*, Springer, 1991.

[4] M.J. BERGER, *Adaptive Mesh Refinement for Hyperbolic Partial Differential Equations*, Ph.D. thesis, Computer Science Dept., Stanford University, 1982.

[5] M.J. BERGER & P. COLELLA, *Local Adaptive Mesh Refinement for Shock Hydrodynamics*, J. Comput. Phys., **82** 67–84, 1989.

[6] M.J. BERGER & J. OLIGER, *Adaptive Mesh Refinement for Hyperbolic Partial Differential Equations*, J. Comput. Phys. **53**, 482–512, 1984.

[7] Y.-L. CHIANG, B. VAN LEER & K.G. POWELL, *Simulation of Unsteady Inviscid Flow on an Adaptively Refined Cartesian Grid*, AIAA Paper 92-0443, 1992.

[8] J.F. CLARKE, S. KARNI, J.J. QUIRK, P.L. ROE, L.G. SIMMONDS & E.F. TORO, *Numerical Computation of 2-Dimensional Unsteady Detonation-Waves in High-Energy Solids*, J. Comput. Phys. **106**, 215–233, 1993.

[9] J.F. DANNENHOFFER III, *Grid Adaptation for Complex Two-Dimensional Transonic Flows*, Sc.D. thesis, Massachusetts Institute of Technology, 1987.

[10] J.L. HALL, P.E. DIMOTAKIS, H. ROSEMAN, *Experiments in Nonreacting Compressible Shear Layers*, AIAA J. **31**, 2247-2254, 1993.

[11] W. FICKETT & W. DAVIS, *Detonation*, University of California Press, Berkeley, 1979.

[12] C-E. FRÖBERG, *Introduction to Numerical Analysis*, Addison-Wesley, 1965.

[13] A. HARTEN, *Multiresolution Algorithms for the Numerical Solution of Hyperbolic Conservation Laws*, UCLA CAM Report 93-03, 1993.

[14] L.F. HENDERSON, W.Y. CRUTCHFIELD & R.J. VIRGONA, *The effects of thermal-conductivity and viscosity of argon on shock-waves diffracting over rigid ramps*, J. Fluid Mech. **331**, 1–36, 1997.

[15] R.J. LEVEQUE, *Numerical Methods for Conservation Laws*, Birkhäuser, 1992.

[16] R. LÖHNER, K. MORGAN & O. ZIENKIEWICZ, *Adaptive Grid Refinement for the Compressible Euler Equations*. In *Accuracy Estimates and Adaptivity for Finite Elements*, Wiley, 1994.

[17] NASA CP-3316, *ICASE/LaRC Workshop on Adaptive Grid Methods*, Edited by J.C. South Jr., J.L. Thomas & J. Van Rosendale, 1995.

[18] M. PARASCHIVOIU, J.-Y. TRÉPANIER, M. REGGIO & R. CAMARERO, *A Conservative Dynamic Discontinuity Tracking Algorithm for the Euler Equations*, AIAA Paper 94-0081, 1994.

[19] S. PIRZADEH, *Unstructured Viscous Grid Generation by the Advancing-Layers Method*, AIAA J. **32**, 17–19, 1994.

[20] K.G. POWELL, P.L. ROE & J.J. QUIRK, *Adaptive-Mesh Algorithms for Computational Fluid Dynamics.* In *Algorithmic Trends in Computational Fluid Dynamics,* edited by M. Y. Hussaini, A. Kumar & M. D. Salas., 303–337, Springer, 1993.

[21] J.J. QUIRK, *An Adaptive Grid Algorithm for Computational Shock Hydrodynamics*, Ph.D. thesis, College of Aeronautics, Cranfield Institute of technology, 1991.

[22] J.J. QUIRK, *A Contribution to the Great Riemann Solver Debate*, Int. J. Numer. Methods Fluids **18**, 555–574, 1994.

[23] J.J. QUIRK, *A Cartesian Grid Approach with Hierarchical Refinement for Compressible Flows.* In *Computational Fluid Dynamics '94*, Invited Lectures and Special Technological Sessions of the Second European Computational Fluid Dynamics Conference, edited by S. Wagner, J. Périaux and E.H. Hirschel, Wiley, pp. 200–209, 1994.

[24] J.J. QUIRK, T.L. JACKSON & A.K KAPILA, *Numerical Study of the Evolution of a Compressive Pulse in an Exploding Atmosphere.* In *Transition, Turbulence and Combustion*, edited by M.Y. Hussaini, T.B. Gatski and T.L. Jackson, Vol. II, pp. 313–329, Kluwer Academic Publishers, 1994.

[25] J.J. QUIRK, *A Parallel Adaptive Grid Algorithm for Computational Shock Hydrodynamics*, Appl. Numer. Math.. **20**, 427–453, 1996.

[26] J.J. QUIRK & S. KARNI *On the Dynamics of a Shock-Bubble Interaction.* J. Fluid Mech. **318**, 129–163, 1997.

[27] J.J. QUIRK, *An Introduction to Amrita*, in preparation.

[28] M.M. RAI, *A Conservative Treatment of Zonal Boundaries for Euler Equation Calculations*, J. Comput Phys. **62**, 472–503, 1986.

[29] V. RUPERT, 1992 *Shock-Interface Interaction: Current Research on the Richtmyer-Meshkov Problem.* In *Shock Waves*, Proceedings of the 18th Intl. Symp. on Shock Waves, held at Sendai, Japan 1991 (ed. K. Takayama), 83–94, Springer, 1991.

[30] M. SHORT & J.J. QUIRK, On the Nonlinear Stability and Detonability Limit of a Detonation-Wave for a Model 3-Step Chain-Branching Reaction, *J. Fluid Mech.* **339**, 89–119, 1997.

[31] G. SOD, *A Survey of Several Finite Difference Methods for Systems of Nonlinear Hyperbolic Conservation Laws*, J. Comput. Phys. **27**, 1–31, 1978.

[32] K. TAKAYAMA, O. ONODERA & G. BEN-DOR, *Holographic Interferometric Study of Shock Transition over Wedges*, SPIE **491**, 976-983, 1984.

[33] J.F. THOMPSON & N.P. WEATHERILL, *Aspects of Numerical Grid Generation: Current Science and Art*, AIAA Paper 93-3539, 1993.

[34] J. Van Rosendale, *Floating Shock Fitting via Lagrangian Adaptive Meshes*, ICASE Report No. 94-89, 1994.

[35] G. WARREN, W.K. ANDERSON, J. THOMAS & S. KRIST, *Grid Convergence for Adaptive Methods*, AIAA Paper 91-1592, 1991.

[36] G. WHITHAM, *Linear and Nonlinear Waves*, Wiley-Interscience, 1974.

[37] S. XU, T. ASLAM & D.S. STEWART, *High Resolution Numerical Simulations of Ideal and Non-Ideal Compressible Reacting Flows with Embedded Internal Boundaries.*,