Geometry 2.0
User's Manual[1]

Edwin Jiménez

[1]Revised May 2, 2005.

Contents

Chapter 1

Getting Started

In this chapter we will see how to install Geometry 2.0[1] and discuss a typical application of the program.

1.1 Installation

The following steps install Geometry 2.0 in a UNIX environment. As of the writing of this document Geometry 2.0 has not been installed on a Windows operating system; future versions of this manual will include the necessary modifications to install Geometry 2.0 on a Windows environment.

The following steps install Geometry 2.0:

1. Create a new directory named `geometry2.0`.
2. Put `geometry20.tar.gz` in that directory.
3. Unzip this file (in the `geometry2.0` directory) with the command `gunzip geometry20.tar.gz`.
4. `tar -xf geometry20.tar`.
5. `cd iamrlib/run2d` (or `cd iamrlib/run3d`).
6. Edit the `GNUmakefile` (set `DEBUG, COMP, USE_MPI, GCC_MAJOR_VERSION`) Note: If using `g++` or `mpiCC, GCC_MAJOR_VERSION` should be either 2 or 3 depending on what you get when you type `g++ -dumpversion`.
7. Type `make`. This should produce the executable `amr2d.Linux.g++.f77.ex` (or `amr3d.Linux.g++.f77.ex`). Set `DEBUG=TRUE` in the `GNUmakefile` to get a different executable filename.

[1] Geometry 2.0 is a program that is constantly evolving, and therefore this manual is constantly updated to reflect these changes. The user is welcome to periodically download the latest incarnation of this document from Dr.Mark Sussman's website `http://www.math.fsu.edu/~sussman`.

1.2 Running the Program

To get a general idea of how Geometry is used, consider the following problem. Given a two-dimensional gas bubble rising in a liquid (imagine a beaker filled with a liquid and a gas bubble just below the liquid's surface) and let's say we want to use Geometry to simulate the fluid interaction. First, we have to determine the pertinent physical parameters such as density (of the gas and liquid), surface tension, viscosity, etc. Other important factors we must consider are the coordinate system to be used, dimensions of the domain, boundary conditions, as well as parameters relevant to the numerical solution of the problem. These numerical parameters may include inputs used in adaptive mesh refinement (AMR), conjugate gradient methods, block factors in grid generation, etc. We will discuss these parameters in detail in subsequent chapters.

Once we have determined the parameters relevant to our problem, we store this information in an *inputs file*. Although an inputs file contains most of the information necessary to simulate a problem, an inputs file alone is insufficient for Geometry to run a given problem as there are modifications that must be made to the program before it is ready to run a simulation.

Continuing our rising bubble example, suppose we have determined all the necessary parameters that define our problem. We will save this information in the inputs file `inputs.bubrise` (this inputs file is included in Appendix A). In addition to writing an inputs file, most applications will require the user to make changes to various files. For instance, a problem might require alterations to be made to an equation of state and this cannot be accomplished through an inputs file; the equation must be programmed directly in the Geometry code. Once we have made all the necessary additions we run the program[2] (from `geometry2.0/iamrlib/run2d` in this case) with the command

```
amr2d.Linux.g++.f77.ex inputs.bubrise
```

When you successfully run the program you will receive a series of output windows showing the evolution of the bubble rising to the surface. Several output options are available, e.g., you may choose to have one screen display the fluid flow field and another a pressure diagram. The various output options available will be discussed in detail in chapter 3.

1.3 Saving Your Output

Geometry generates several folders and files of output each time it's run, however, everything is erased when you rerun the program. With simulations often running for hours and sometimes days, it would be disturbing to find your work gone because you decided to run a new problem. To prevent this from happening, it is advisable to send all output to a subdirectory under `run2d`. For example, suppose we want to send the output of the rising bubble problem discussed in the previous section to a subdirectory named `bubbleout`. From the `bubbleout` directory we would execute the program as follows

```
../amr2d.Linux.g++.f77.ex ../inputs.bubrise
```

This sends all output folders and files to `bubbleout`.

[2] To stop the program in mid-execution use `Ctrl+C`.

Chapter 2

Inputs Files

The goal of Geometry 2.0 is to accurately simulate a wide range of fluid phenomena. One of its major attributes is its ability to simulate multi-phase flows with high density ratios (see for example [MS97]). Although setting up a simulation is generally not as straight forward as with some off-the-shelf programs, Geometry 2.0 rewards the patient user with results that employ numerical techniques that are at the cutting edge of today's computational physics research.

The parameters that define a problem are read by Geometry via an *inputs file.* This file must include all of the important physical, numerical, and output parameters. Several inputs files can be found in the `run2d` and `run3d` directories; these inputs files model a diversity of fluid problems ranging from a drop impinging on a surface to an overturning wave. These inputs files can be invaluable in assisting the user to learn to write his own inputs files. In some instances it may only be necessary to modify an existing inputs file, however, most problems will undoubtedly require the user to write his own inputs file which, in turn, will entail additional modifications to Geometry before it is ready to run a simulation.

Typically the same parameters[1] are included in all inputs files although some parameters may not be used in a particular problem; this practice is mainly done for consistency. The parameters are grouped as follows: Inputs to the *main program*, inputs to the *adaptive mesh refinement (AMR) class*, inputs to the *physics class*, *boundary condition (BC) flags*, *run-time timing statistics* and *2-D contour plotting parameters*[2]. The following sections explain the first five input classes. Chapter 3 will be devoted to plotting.

2.1 Inputs to the Main Program

This section includes parameters that control the maximum time step taken as well as the stopping time which dictates how long our simulation runs. Once the more delicate parameters have been set, this section allows the user to quickly modify these frequently-used variables.

1. `max_step` Maximum time step (a nonnegative integer).
2. `stop_time` When running a simulation, the program runs until `stop_time` is reached.
3. `thickness` Unused.

[1] As Geometry evolved, some parameters became obsolete. For convention we have chosen not to delete these parameters from the inputs files; they will simply be labeled *unused.*

[2] Contour plotting is available only in 2D.

2.2 Inputs to AMR Class

Most of the inputs in this section pertain to the adaptive mesh refinement (AMR) class although here we shall also set our coordinate system, various tolerances, and choose whether or not to use a conjugate gradient method.

The parameters below use the following prefix convention: `mg` *multigrid*, `cg` *conjugate gradient*, `ns` *Navier-Stokes*, `Lp` *Linear operator*.

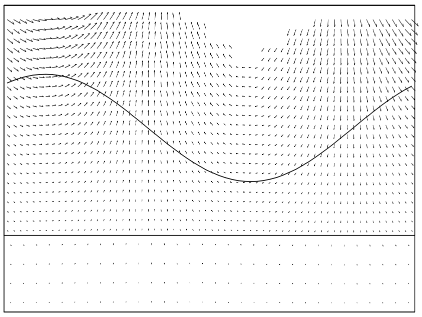

Figure 2.1: `inputs.overturn` image. Contour plot with `grid_eff` = 1.0 and `blocking_factor` = 8.

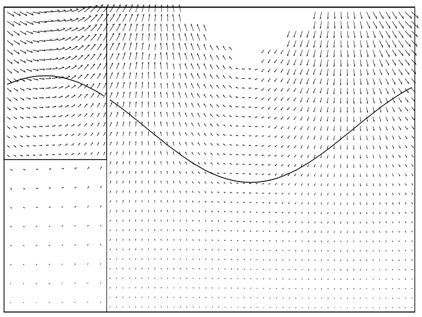

Figure 2.2: `inputs.overturn` image. Contour plot with `grid_eff` = 1.0 and `blocking_factor` = 16.

`amr.blocking_factor` Block factor in grid generation. Grid size has to be a multiple of `blocking_factor`. See Figures 2.1 and 2.2.

`amr.check_file` Root name of checkpoint file, e.g., `chk`.

`amr.check_int` The number of timesteps between checkpoints, e.g., 50. Setting `amr.check_int=50` and `amr.check_file=chk` results in the generation of a series of files `chk0000, chk0050, chk0100`, etc. These files can be useful when we would like to *restart* a simulation from a given timestep (an not necessarily the initial timestep). Suppose we stopped the computation at the 120th timestep, then the file `chk0100` (the last checkpoint file generated) allows us to restart the computation by simply setting `amr.restart=chk0100`. *Note:* When using `check_per` instead of `check_int`, then set `change_max` $= 1.0 \cdot 10^{15}$.

`amr.grid_eff` A number between 0 and 1. A smaller number results in fewer boxes.

`amr.grid_log` Name of grid logging file, for example `grdlog`.

`amr.max_grid_size` Maximum grid size; typical values are 64 or 128.

`amr.max_level` 0 when using a single grid (no AMR), otherwise set to 1 or 2 for 2 and 3 levels respectively.

`amr.n_cell` Number of cells, written in the form $x \quad y$. For instance, 32 16.

`amr.n_cycle` Always set to 1 1 1 1 (number of fine level time steps per coarse level time step).

`amr.n_error_buf` Number of buffer cells in error estimate. For example, 2 2 2 2 2.

`amr.plot_file` Root name of plot file.

`amr.plot_int` The number of timesteps between plots, e.g. 200. *Note:* When using `plot_per` instead of `plot_int`, then set `change_max` $= 1.0 \cdot 10^{15}$.

`amr.ref_ratio` Refinement ratio. A typical value is 2 2 2 2 2.

`amr.regrid_int` How often to regrid. Typical value 1.

`amr.restart` Name of checkpoint file used when restarting a computation, e.g., `chk0220`.

`cg.maxiter` Maximum number of iterations.

`cg.verbose` Set 2 for maximum verbosity

`diffuse.max_order` Always leave as 2.

`geometry.coord_sys` Sets the coordinate system. 0 for *cartesian* coordinates and 1 for *cylindrical* coordinates. *Note: This should always be set to 0 in 3D.*

`geometry.is_periodic` This input is of the form $x \quad y$ where x and y can be either 0 or 1. If $x \quad y = 0 \quad 1$, then the problem is periodic in the y coordinate. For a 3-D problem `geometry.is_periodic` $= 0 \quad 1 \quad 1$ means the problem is periodic in y and z (but not in x). In each case the boundary conditions must be modified accordingly. For instance, in a 2-D problem if $x \quad y = 1 \quad 0$ then we must modify the *low* and *hi* boundary condition parameters `ns.lo_bc` and `ns.hi_bc` to, say,

`ns.lo_bc` $= 0 \quad 2$
`ns.hi_bc` $= 0 \quad 2$.

These BC parameters are of the form $x_{lo} \quad y_{lo}$ for `ns.lo_bc` (similarly for `ns.hi_bc`) and for the example above we have set *interior Dirichlet* boundary conditions for the x coordinate and *outflow* conditions for the y coordinate. Boundary conditions will be discussed in more detail in section 2.4.

`geometry.prob_lo`	Sets the lower left problem coordinate.
`geometry.prob_hi`	Sets the upper right problem coordinate.
`godunov.printMinMax`	Unused.
`Lp.harmavg`	Sets the *harmonic average*. This quantity must be set to 0.
`Lp.v`	Unused.
`mg.bot_atol`	This parameter should be at least 2 orders of magnitude less than regular tolerance.
`mg.maxiter`	Maximum number of iterations.
`mg.nu_f`	Number of iterations on bottom solver (viscous solve).
`mg.nu_0`	Set to 1 for *v-cycle* and 2 for *w-cycle*. Typically set to 1.
`mg.rtol_b`	Relative bottom tolerance $\lvert\frac{E}{E_0}\rvert < \epsilon$, set to -1 to skip.
`mg.usecg`	Set to 1 to use conjugate gradient and set to 0 for no CG.

`mg.verbose`	This input can be set to 1 (on) or 0 (off). When turned on, the initial error is displayed on screen as follows:
	`Bottom Smoother:  Initial error (error0) = 0.123456789`
	`Bottom Smoother:  Iteration 1 error/error0 = 0.03191976`

`ns.be_cn_theta`	The parameter θ in $u^{n+1} = u^n + \theta\Delta t\Delta u^{n+1} + (1-\theta)\Delta t\Delta u^n$. Here $\theta \geq 1/2$, use $\theta = 1$ if non-Newtonian.
`ns.bubble_pressure`	Initial bubble pressure.
`ns.dencav`	`dencav`$=\rho($`pcav`$)$.
`ns.denchopp`	If `is_compressible=1`, then set `denchopp=0.9`
`ns.djwl`	See `ns.gammavapor`.
`ns.ejwl`	See `ns.gammavapor`.
`ns.gammavapor`	Exponent in the equation of state $p(V) = \texttt{djwl}e^{-r_1\texttt{jwl}V/V_0} + +\texttt{ejwl}e^{-r_2\texttt{jwl}V/V_0} + C\left(\frac{V_0}{V}\right)^{\texttt{gammavapor}}$. Here $C = \texttt{bubble\_pressure} - \texttt{djwl}e^{-r_1\texttt{jwl}} - \texttt{ejwl}e^{-r_2\texttt{jwl}}$.
`ns.hydrostatic_pressure`	Hydrostatic pressure.
`ns.is_compressible`	1 for a compressible fluid; 0 for incompressible.

`ns.is_twophase` — If `is_twophase=1`, then the pressure varies in the vapor.
If `is_twophase=0`, then the pressure is spatially constant.

`ns.jwl` — See `ns.gammavapor`.

`ns.pcav` — Compressible liquid cavitation pressure (sound speed = 0 in cavitated region).

`ns.pchopp` — Set this quantity to 0.

`ns.repeat_iter`

`ns.rk_theta` — Unused.

`ns.shock_capture` — Always 0.

`ns.shock_eta` — Always 1.

`ns.shock_timestep` — If `shock_timestep=1`, then $\Delta t < \frac{\Delta x}{\max\{|u|\}+|c|}$.
If `shock_timestep=0`, then $\Delta t < \frac{\Delta x}{\max |u|}$.

`ns.soundchopp`

`ns.visual_hi` — Higher cell coordinate. A typical value is 512 512.
Warning: These are *not* physical units.

`ns.visual_lo` — Number of cells. A typical value is 0 0.
Warning: These are *not* physical units.

`ns.visual_option` — This parameter controls the *visual files* format options available.
See chapter 3 section 3 for more information.

`ns.visual_side_max` — 512.

2.3 Inputs to Physics Class

With the exception of pressure, which will be measured in *bars* in compressible flow problems, the user may use any units he chooses as long as he is consistent.

`anelastic` — Setting this parameter to 0 has the effect of no anelastic approximation.

`advbot` — Set to 0 (unused).

`adv_dir` — Direction of inflow velocity.

`adv_vel` — Inflow velocity.

`axis_dir`

`centerpressure` Unused.

`cfl` CFL number for a hyperbolic system. `error_buf`=1.5`cfl`; if `cfl`> 1 make sure `error_buf` is sufficiently large.

`change_max` Scale back initial time step, where $\Delta t^{n+1} \leq \texttt{change\_max} \cdot \Delta t^n$.

`denair` Air density (0.001).

`denfact` Unused.

`denspread` Density spread. This determines the thickness of the "narrow band." Thickness = `denspread` + 4.

`denvapor` Vapor density (0.001).

`denwater` Water density (1.0).

`do_radiation` Unused.

`dt_cutoff` Level 0 timestep below which programs halts. Typical value: $0.5 \cdot 10^{-5}$.

`dump_time` Unused.

`extendcurv` Unused (leave as 1).

`fixed_dt` Hardwire timestep.

`gravity` Body force.

`gravityangle` Gravity angle.

`ignition`

`includehoop` Unused (leave as 1).

`init_iter` Number of initial iterations to define pressure.

`init_shrink` Scale back initial time step. Typical value: 1.0.

`is_conservative` Conservative system (1=yes, 0=no).

`kinematicfixed` Unused.

`mac.mac_tol` Tolerance for mac projections.

`mac.mac_abs_tol` Absolute tolerance.

`mac.mac_order` 0 for pc, 1 for quad, and 2 for jbb `mac.use_cg_solve` Use conjugate gradient solve (1=yes, 0=no).

`probtype` When defining a new inputs file, the user must assign a number to each problem. This information is recorded in the `probtype` parameter and it is used as an identifier by the relevant subroutines. Currently there is no systematic way of numbering problems, however, the user is cautioned not to assign the same number to different problems.

`radblob` Radius (of the bubble, for example).

`rgasinlet`

`RUNGAKUTTA` Unused.

`stokes_mode` Set as 0. Note: If `stokes_mode=1`, the term $u \cdot \nabla u$ in $u_t + u \cdot \nabla u = -\frac{\nabla p}{\rho} + \nabla \cdot 2\mu D$ is ignored.

`sum_interval` Time steps between computing mass.

`tcenter`

`tension` Surface tension.

`tensorsolve` Unused (leave as 1).

`twall` Set to 0 (unused).

`usekluge` Unused (leave as 1).

`velfact` Set to 0.

`version_A` 1 = version A; 0 = version B.

`vinletgas` Set to 0 (unused).

`visc_abs_tol` Viscous absolute tolerance, e.g., $1.0 \cdot 10^{-7}$.

`visc_coef` The coefficient α in the equation $\rho(u_t + (u \cdot \nabla)u) = -\nabla p + \alpha\nabla \cdot 2\mu D + \sigma k \nabla H$, where $\mu = \mu_L H + \mu_G(1 - H)$ and $\rho = \rho_L H + \rho_G(1 - H)$. Here H is the Heaviside function, μ_L = `viscunburn`, μ_G = `viscvapor`, or `viscburn`, ρ_L = `denwater`, and ρ_G = `denvapor`, respectively.

`viscburn` Gas viscosity.

`visctimestep` Unused.

`viscunburn` Liquid viscosity.

`viscvapor` Vapor viscosity.

`vorterr`	Set to 0 for `inputs.bubrise`.
`xblob`	Center of bubble (for example).
`yblob`	Center of bubble (for example).
`zblob`	Center of bubble (for example).

2.4 Setting Boundary Conditions

The following parameters are responsible for setting the boundary conditions (BC); these have the following form

$$\begin{aligned} \texttt{ns.lo\_bc} &\quad x_{lo} \quad y_{lo} \quad z_{lo} \\ \texttt{ns.hi\_bc} &\quad x_{hi} \quad y_{hi} \quad z_{hi} \\ \texttt{ns.conserveden} &\quad value, \end{aligned}$$

where *value* can be either 0 or 1. The following boundary conditions or *flags* are available:

0 = `Interior`	3 = `Symmetric`
1 = `Inflow`	4 = `SlipWall`
2 = `Outflow`	5 = `NoSlipWall`

`0=Interior` is used for periodic BC. Thus, for example,

```
ns.lo_bc    0 2
ns.hi_bc    0 2
```

sets x_{lo} to an interior Dirichlet boundary condition, y_{lo} to an outflow condition, and x_{hi} to an interior Dirichlet boundary condition, and y_{hi} to an outflow condition, respectively.

2.5 Run-time Timing Statistics

The user can select the form of FAB output by setting `fab.format` to any of the choices below:

IEEE	Default (portable).
ASCII	This is slow.
NATIVE	Native binary form on machine.

Select single or double precision of FAB output data; default is whatever precision code is compiled with. `fab.precision` = FLOAT (or DOUBLE).

2.6 Defining Your Own Input Files

In this section we will examine several problems for which we will define inputs files. In addition to an inputs file, to run a simulation we generally need to make modifications to the actual Geometry program. Naturally, the changes made will depend on the particular problem; a new problem bearing a strong resemblance to a previous problem might require only that we make marginal changes to the existing code.

Even when a new problem has little in common with the problems included with your version of Geometry, and there is no Geometry guru around to help you, a close inspection of the inputs files can still be helpful. Imagine we are trying to code a problem where, say, compressible effects are important. We first search for a problem, however different it may seem from our own save for the fact that compressibility is important, and take note of the `probtype` parameter—suppose it is `probtype=45`. In a UNIX or Linux environment we can easily perform a search of what subroutines (and where they are) are associated with `probtype=45` by using the command

```
grep "probtype.eq.45" *.F
```

or

```
grep "probtype==45" *.cpp
```

to search `.F` (FORTRAN) and `.cpp` (C++) files, respectively. If we are dealing with a 2D problem, it may not only be necessary to search the `run2d` directory but also the `iamrlib` directory. Our search will help us narrow the list of subroutines that will potentially require modifications. The above example presents the user a worst case scenario; the aim of this document is to help the user avoid this type of trial-and-error tinkering.

2.6.1 Gas Bubble Bursting at a Free Surface

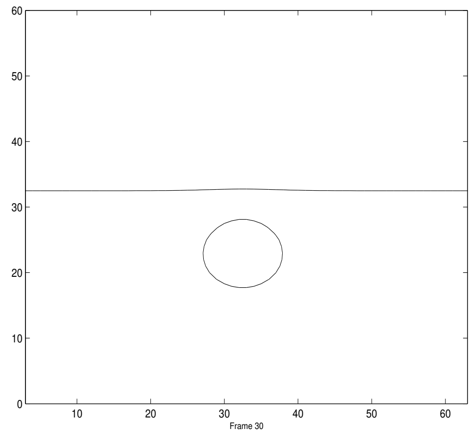

Figure 2.3: `inputs.bubrise` initial state.

The first problem we consider is that of a gas bubble rising to the free surface of a fluid. When the bubble breaks through the surface, large surface tension forces are produced which ultimately cause a jet of liquid to be ejected; this jet can subsequently break up into drops [MS97].

Suppose we have an air bubble in a beaker of water. Our first task is to define a distance function (this is our level set function[3]) that describes the initial state of our system. We require that our distance

[3]We normally set our level set function to be a distance function.

function ϕ be positive in the water and negative inside the bubble and in the air above the surface (see figure 2.3). It can be shown that

$$\phi(x,y) = \begin{cases} \phi_1 = \sqrt{(x - \texttt{xblob})^2 + (y - \texttt{yblob})^2} - \texttt{radblob} & \phi_1 < \phi_2 \\ \phi_2 = 1 - (y - \texttt{yblob}) & \text{otherwise,} \end{cases} \tag{2.1}$$

i.e., $\phi = \min\{\phi_1, \phi_2\}$. Here (`xblob,yblob`) is the center of the bubble and `radblob` is its radius.

We will start by writing a suitable inputs file which we will call `inputs.bubrise`. Having done this, we proceed to create and modify the necessary subroutines to allow Geometry to simulate our problem.

Inputs to Main Program. We set our maximum computing time to `stop_time`= 20.0 and our maximum time step to `max_step` = 999999.

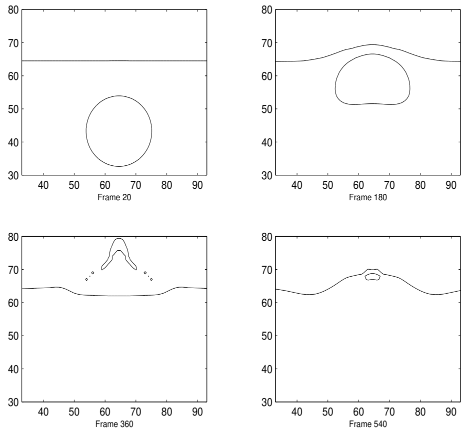

Figure 2.4: `inputs.bubrise` images.

Inputs to AMR Class. In this problem it is more natural to use cylindrical coordinates. For efficiency, we compute only 'half' of the problem (symmetry about the y-axis) and we take the problem to be periodic in x. Thus, we set the following parameters as follows:

`geometry.coord_sys`	1
`geometry.prob_lo`	0.0 0.0
`geometry.prob_hi`	3.0 6.0
`geometry.is_periodic`	1 0

Inputs to Physics Class. Here we set the water density $\rho_w = 1.0$ and the air density to $\rho_a = 0.001$. The body force in this case is gravity which has been set to $g = -9.80$. There are other important variables in this section and the user is encouraged to read the `inputs.bubrise` file found in Appendix A as well as reference [MS97] for a more detailed explanation of the physical parameters used in this computation.

Problem Dependent Inputs. Since we require solid boundaries on three sides of the beaker and a free boundary (the water surface) we obtain the following *slip-wall* and *symmetry* boundary conditions.

`ns.lo_bc`	3 4
`ns.hi_bc`	4 4.

Recall that when defining a new inputs file the user must assign a positive integer to each problem. This information is recorded in the `ns.probtype` parameter. In our case, `ns.probtype` = 49. Currently there is no systematic way of numbering problems so the user is cautioned not to assign the same number to different problems.

Contour Plots. In this example we have opted to display the output (the flow field) on PostScript files. Note that since the problem is symmetric about the vertical axis, only 'half' of the problem will be plotted by the program. More information about contour plots can be found in Section 3.2.

Now that we have identified our distance (level set) function and written an appropriate inputs file, we must make a few additions to `PROB_2D.F`. First, we must add our level set function to the subroutine `vapordist` where liquid-gas distance functions are defined. This can be done as follows.

```
    if (probtype.eq.49) then
       dist=sqrt((x-xblob)**2+(y-yblob)**2) - radblob
       dist1= 1 - (y-yblob)
       if (dist1.lt.dist) then
        dist=dist1
    endif
```

The subroutine `FORT_INITVELOCITY` is responsible for initializing the problem velocity. Using the `probtype` parameter this subroutine calls the corresponding initializing subroutine. Hence, we must first write a bubble initializing subroutine and add it to the `PROB_2D.F` file–we'll call it `initbubrise`. The subroutine can be written as follows.

```
   subroutine initbubrise(level,time,lo,hi,
     &                 vel,DIMS(vel),
     &                     dx,xlo,xhi)
      IMPLICIT NONE

      integer    level
      integer    lo(SDIM), hi(SDIM)
      integer    DIMDEC(vel)

      REAL_T     time, dx(SDIM)
      REAL_T     xlo(SDIM), xhi(SDIM)
      REAL_T     vel(DIMV(vel),SDIM)


c     :::: local variables
      integer i, j
      REAL_T  x, y, xtemp,ytemp
      REAL_T  hx, hy
      REAL_T  rho, dist, dist1,c, rhobub
      REAL_T  x_vel, y_vel
      REAL_T  rho1,rho2,hval
      integer i1,j1,numsub
      REAL_T rhoh,fnumsub
      REAL_T xl1,xl2,yl1,yl2,xmin,ymin,height,lenarc,theta
      REAL_T x1,y1,x2,y2,area1,area2,area3,area4


#include "probdata.H"
```

```
      print *,"levelrz in initbubrise is ",levelrz

      hx = dx(1)
      hy = dx(2)

      if (adv_dir .eq. 1) then
         x_vel = adv_vel
         y_vel = zero
      else if (adv_dir .eq. 2) then
         x_vel = zero
         y_vel = adv_vel
      else if (adv_dir.eq.3) then
         x_vel = adv_vel
         y_vel = adv_vel
      else
         write(6,*) "initbubrise: adv_dir = ",adv_dir
         stop
      endif
      if (adv_vel.ne.zero) then
       print *,"adv_dir,adv_vel ",adv_dir,adv_vel
      endif

         do j = lo(2), hi(2)
            y = xlo(2) + hy*(float(j-lo(2)) + half)
        do i = lo(1), hi(1)
               x = xlo(1) + hx*(float(i-lo(1)) + half)
           vel(i,j,1) = x_vel
           vel(i,j,2) = y_vel
        enddo
         enddo

      return
      end
```

Now we can call this subroutine from `FORT_INITVELOCITY`. We must also add the fragment `probtype.eq.49` to the following block in the `selectdist` subroutine.

```
     ...
      else if ((probtype.eq.36).or.(probtype.eq.37).or.
     &         (probtype.eq.39).or.(probtype.eq.41).or.
     &         (probtype.eq.42).or.(probtype.eq.43).or.
     &         (probtype.eq.44).or.(probtype.eq.45).or.
     &         (probtype.eq.46).or.(probtype.eq.49).or.
     &         (probtype.eq.61)) then
       dist=99999.0
     ...
```

Geometry is now ready to run our first simulation (see Figure 2.4).

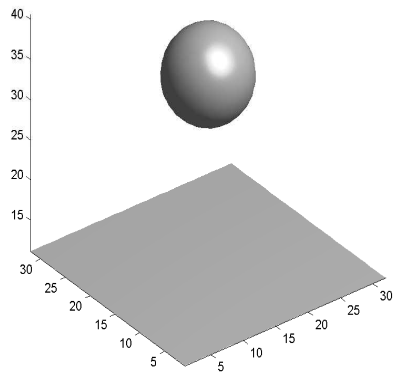

Figure 2.5: `inputs.milkdrop` image just before the drop impinges on the surface.

2.6.2 3D Milk Drop Experiment

Next we will prepare a 3D problem: A milk drop impinging on a pool of milk. We see that the set up of this problem, as far as the distance function is concerned, is similar to our rising bubble example. It can be shown that in this case the distance function is $\phi(x, y, z) = \max\{-\phi_1, -\phi_2\}$, where

$$\begin{cases} \phi_1 = \sqrt{(x - \texttt{xblob})^2 + (y - \texttt{yblob})^2 + (z - \texttt{zblob})^2} - \texttt{radblob} & \phi_1 < \phi_2 \\ \phi_2 = 1 - (z - \texttt{zblob}) & \text{otherwise,} \end{cases} \tag{2.2}$$

for a drop with center at (`xblob,yblob,zblob`) and radius `radblob`. Having determined the distance function, we now proceed exactly as in the previous example. After preparing the inputs file (`inputs.milkdrop`), we need to modify three subroutines `FORT_INITVELOCITY, vapordist,` and `selectdist` (the 3D counterparts of the same subroutines we modified in the previous example) found in `PROB_3D.F` to initialize the problem.

The velocity initializing subroutine that must be added to the `PROB_3D.F` file can be found in Appendix B. Remember that in `FORT_INITVELOCITY` we simply call the velocity initializing subroutine and `vapordist` and `selectdist` deal with our distance function. Notice that so far we have dealt with problems where compressibility has been largely neglected. In the next problem we will see that compressible effects are essential.

2.6.3 Oscillating Water Column

The following example is taken from [BK02]. We consider a closed 1D tube with impermeable boundaries at the left and right and with initial solution as sketched in Figure 2.6. Starting from $t = 0$, the air on the right is compressed by the water and the air on the left expands. Hence, a pressure difference is built up across the column of water, with a consequent deceleration of the latter's flow to the right, followed by stagnation, and then an acceleration and flow to the left. This leads to a reverse pressure gradient across the water, which redirects the flow from left to right again, and so on. The water column starts to oscillate.

Once we have written the appropriate inputs file (we will call it `inputs.watercol` with `probtype=47`), we make the usual modifications to the program, i.e., the distance function, velocity initializing subroutine, except that now we must also concern ourselves with setting up the pressure and density.

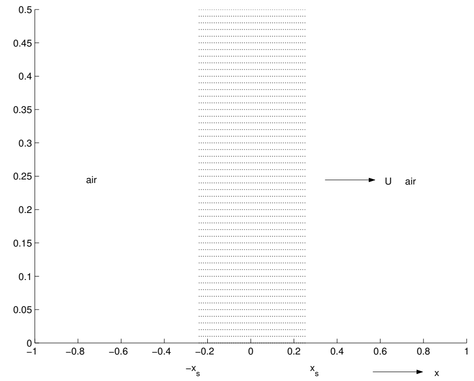

Figure 2.6: Initial condition: Shut off tube with column of water (center) between two columns of air, all three columns flowing to the right at constant speed U and pressure P.

The subroutines `get_explosion` and `get_vapor_pressure` are responsible for computing the pressure in each air column. We set the water density, in `FORT_INITDENSITY`, `density(i,j)=one.` The pressure characteristics are similar to those in `probtype.eq.43` (an implosion problem) thus we add only the line `.or.probtype.eq.47` in the conditional statement in `get_explosion` to select the same pressure coefficients.

We must add an integer parameter `iflag` to the `get_vapor_pressure` subroutine since we must now be able to determine whether we are computing the pressure on the left (`iflag=1`) side of the water column or on the right (`iflag=2`). In addition, we have to keep track of the vapor masses on the left and on the right, we deal with this issure in the subroutine `FORT_BPRESSURE`, found in `LEVELSET_2D.F`.

2.6.4 Implosion of An Axisymmetric Vertical Tube

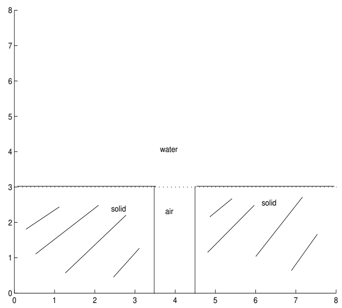

Figure 2.7: Water mass the instant before imploding into a vertical tube filled with air.

In this section we consider the implosion of a water mass, at high pressure, into a vertical tube filled with air. Since we wish to take advantage of the symmetry of the problem, we will place the center of the tube along the y-axis. The tube has radius `radblob`$= 1$ and height `yblob`$= 5$; we set `xblob`$=0$. The water mass sits above the line $y =$`yblob`, on top of the air tube if $x <$`radblob` and on a solid surface if $x >$`radblob`. The water is initially at about 53 atm and the air at 1 atm. This information is recorded in `ns.hydrostatic_pressure=52.723` in `inputs.tubeimplode` (with `probtype.eq.48`).

Once we've written the inputs file, we will need to modify several files before the program is able to execute the problem.

In LEVELSET_2D.F, the fragment probtype.eq.48 (in the if-then statement...) must be added to the get_explosion subroutine. Next, we add an if-then statement to get_vapor_pressure which, along with the get_explosion subroutine, computes the pressure.

In MACOPERATOR_2D.F, the fragment probtype.eq.48 must be added to the functions taitden, taitpres, sound_speed, and hydropres. These subroutines take care of the equations of state (of γ-law type), and since we are dealing with air and water, we need only make calls and/or minor modifications. Had we been dealing with more exotic chemicals, with more specialized equations of state, we would have required to program the appropriate subroutines ourselves.

Next we add the following fragment

```
// tube implosion
} else if (probtype==48) {
if (levelcomp==LevelGas) {
    grav=0.0;
    return 0;
} else {return 1; }
```

to NavierStokes3.cpp.

We must add the distance function and a velocity initialization subroutine to PROB_2D.F. For this problem we will in fact have *two* distance functions, one separating the air and the water, and the other separating the fluids (water and air) with the solid. The distance function that separates air and water must be added to the vapordist subroutine as follows

```
...
      else if (probtype.eq.48) then
       xmin=-xblob-half*xblob
       xmax=xblob+half*xblob
       ymin=-yblob
       ymax=yblob
       call squaredist(x,y,xmin,xmax,ymin,ymax,dist)
...
```

The distance function separating the fluids from the solid must be added to the soliddist subroutine as follows

```
...
      else if (probtype.eq.48) then
       xlarge=1.0e+10
       call squaredist(x,y,xblob,xlarge,-yblob,yblob,dist)
...
```

Note the calls made to the squaredist subroutine; this subroutine automatically generates a distance function that is positive in the solid and negative in the fluid. This routine defines a rectangle with lower coordinates at (xmin,ymin) and upper coordinates at (xmax,ymax).

Having written the velocity initialization subroutine, which we will call inittubeimp, we simply call it from FORT_INITVELOCITY (found in PROB_2D.F) with the lines

```
...
else if (probtype.eq.48) then
call inittubeimp(level,time,lo,hi,
     &               vel,DIMS(vel),
     &               dx,xlo,xhi)
...
```

By now it should be evident that because of the vast variety of fluid mechanics problems, there is usually no standard way of writing an initializing subroutine, a level set function, an equation of state, etc. In the future, more intricate examples will be added; these will hopefully help the user acquire greater dexterity with Geometry 2.0.

Chapter 3

Displaying Output

In this chapter we discuss the various output options available. Geometry has the ability to display plots on the screen as well as on PostScript files. In addition to on-screen images, when running a simulation Geometry also generates visual files that contain raw data–essentially text files–that give the user the versatility to use any plotting program capable of reading files. In section 3 we will learn to use visual files to make plots and movies with MatLab.

3.1 Displaying Output on the Screen

When executing the program two general types of plots are available: A velocity field plot[1] and a pressure field plot. The *visual files* Geometry generates contain information such as the velocity field, level set functions, pressure, viscosity, etc. Visual files are discussed in more detail in section 3. In the first two sections we will learn about the most basic output Geometry 2.0 is capable of generating.

At the end of each inputs file is a *contour plotting* (see Appendix B for a sample inputs file) section which gives the user the choice to display output on the screen and/or on a PostScript file. Selections are made via the `device` parameter. Below is a list of the contour plotting variables found at the end of every inputs file along with a brief explanation of their function.

`file_name` Root name of postscript file (e.g., `triple`)

`var_name` Name of thermodynamic variable to plot.

`device` 1=XWindow
2=PostScript
3=both

`freq` Interval between plots (set to -1 to turn off).

`n_cont` Number of contour lines per plot. Setting this variable to -1 shows the level set.
Setting it to -2 shows the level set and contours centered about the zero level set.

`sho_grd` 0=Do not show grid placement.
1=Show grid placement.
2=Show grid placement and overlay velocity vector plot.
3=Same as 2 except show Umac and not Ucell.

[1]Sometimes only a portion of the problem will be displayed if there are symmetries present.

`win_siz` Number of pixels in the maximum window direction.

`sval` It is possible to analyze only a slice of a problem, e.g., to observe the x velocity at a fixed value of y, say at $y = 0.25$ (see Figure 3.1). In this case we would set `sval` the *slice value* to 0.25. Note that if instead of sending the slice to the screen we wanted to generate a *ps* file of the slice we need only set `device=2.`

`sdir` Slice direction. This is the direction to be held constant. Set to 0 for x, and 1 for y.

`sstr` An identifier for each 'slice.' Suppose we wanted to make two slices of the fluid profile in a 3D problem at a particular instant and we wanted to distinguish between, say an xy-plane slice and a yz-plane slice, then we would set `sstr=xy` and `sstr=yz`. For example, at the tenth timestep we would have the files `triple00010xy` and `triple00010yz`.

A typical configuration is summarized in the table below. This example corresponds to the `inputs.overturn` file.

file_name	var_name	device	freq	n_cont	sho_grd	win_siz	sval	sdir	sstr
triple	triple	3	5	-1	2	400	0.25	1	y

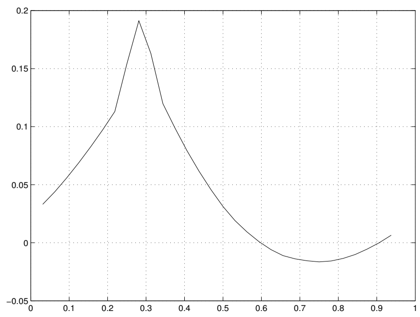

Figure 3.1: `inputs.overturn` image with `sval`=0.25, i.e., $y = 0.25$ fixed.

In this example the `device` variable instructs the program to output the fluid profile to an XWindow *and* a PostScript file. Since `freq`=5, the screen is refreshed every 5 iterations. `n_cont` has been set to -1 (see Figure 3.1). Setting `sho_grd=2` shows the grid placement and a velocity plot.

In this example we plot the x-velocity u versus x at $y = 0.25$ (we are considering only a slice of the problem), so we set `sval=0.25` [2]. `sdir=1` indicates the y direction is held constant (to hold x constant we set `sdir=0`).

3.2 2-D Contour Plotting: Sending Output to a PostScript File

Contour plotting is available for 2-D problems. Simply set the `device` variable to 2 to send the output to a PostScript file or 3 to output to *both*, the screen and a PostScript file. The table below summarizes a

[2]`sval` is measured with respect to 0 and *not* with respect to (x_{lo}, y_{lo}).

typical example where the velocity field is sent to a *ps* file and a pressure plot is displayed on the screen.

file_name	var_name	device	freq	n_cont	sho_grd	win_siz	sval	sdir	sstr
triple	triple	2	5	-1	2	400	-1	-1	0
presliquid	presliquid	1	10	-1	2	400	-1	-1	0

3.3 Visual Files

In addition to output being sent to the screen and/or a PostScript file, Geometry also generates a set of *visual files* (a typical one being `visual00010`) which contain columns of visual data the user may use to draw his own plots using a program such as MatLab or TecPlot. There are four output formats available and these are controlled by way of the `ns.visual_option` parameter. As is customary, we denote the level set function by ϕ. The tables below summarize the various 2D visual option formats available (the 3D case is the same save for the addition of a k and z_{vel} column).

ns.visual_option	i	j	x_{vel}^{liq}	y_{vel}^{liq}	ϕ_{gas}	ϕ_{vapor}	ϕ_{solid}	p	μ	$\lvert\dot{\gamma}\rvert$
0	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
1	✓	✓	–	–	✓	✓	✓	–	–	–
2	✓	✓	–	–	–	✓	✓	✓	–	–

Here p, μ, and $|\dot{\gamma}|$ denote the pressure, viscosity, and shear stress, respectively. The column output when using `ns.visual_option=3` differs slightly from the first three options.

ns.visual_option	i	j	x_{vel}^{liq}	y_{vel}^{liq}	x_{vel}^{vap}	y_{vel}^{vap}	ϕ_{vapor}
3	✓	✓	✓	✓	✓	✓	✓

The frequency of output of the visual files is determined by the `amr.plot_int` or `amr.plot_per` parameter. When using the former, visual files are generated at regular *step* intervals while the latter outputs visual files at regular *time* intervals. Recall that `ns.change_max` controls the maximum factor that the time step can grow from one step to another. For example, setting it to 1.1 means `dt_new` $\leq$ 1.1`dt_old`. The user must be careful when using `amr.plot_per` instead of `amr.plot_int` since the program may come to a crawl attempting to adjust to take the appropriate time step. To avoid this from happening, the user should set `ns.change_max` to a large value, say, 10^{13}.

3.3.1 Generating Plots Using MatLab

We can use the visual files in MatLab (or any other plotting program that can read the visual files) to generate plots. Suppose we want to render a 3D image of the milk drop example from Chapter 2. With the visual files at our disposal, we can easily write a MatLab file to accomplish this. First, to load the visual file (we'll plot frame 80) `visual00080` and then store its contents in `finedata`, we use the following MatLab commands.

```
finedata=load('visual00080');

D=size(finedata);

if (D(2)~=12)

 sprintf('invalid number of columns\n')
```

```
end
```

Next, (we have set `visual_option=0` in the inputs file; the `if` statement checks this) we extract our data and label it accordingly,

```
numrows=D(1);
rimax=0;
rjmax=0;
rkmax=0;
for I=1:numrows,
 if (finedata(I,1)>rimax)
  rimax=finedata(I,1);
 end
 if (finedata(I,2)>rjmax)
  rjmax=finedata(I,2);
 end
 if (finedata(I,3)>rkmax)
  rkmax=finedata(I,3);
 end
end
imax=round(rimax)+1;
jmax=round(rjmax)+1;
kmax=round(rkmax)+1;
x=1:1:imax;
y=1:1:jmax;
z=1:1:kmax;

U=zeros(imax,jmax,kmax);
V=zeros(imax,jmax,kmax);
W=zeros(imax,jmax,kmax);
PHI=zeros(imax,jmax,kmax);

for I=1:numrows,
 II=round(finedata(I,1))+1;
 JJ=round(finedata(I,2))+1;
 KK=round(finedata(I,3))+1;
```

```
 U(II,JJ,KK)=finedata(I,4);
 V(II,JJ,KK)=finedata(I,5);
 W(II,JJ,KK)=finedata(I,6);
 PHI(II,JJ,KK)=finedata(I,8);
end
```

With all the data prepared, we may now use the commands below to produce an image of the milk drop just after it impinges on the surface (see Figure 3.2).

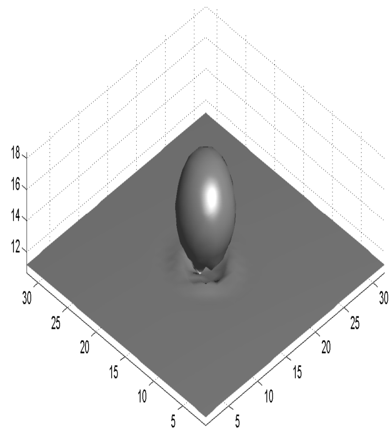

Figure 3.2: `inputs.milkdrop` MatLab image generated using a visual file.

```
p = patch(isosurface(x, y, z, PHI, 0));
isonormals(x,y,z,PHI, p);

set(p, 'FaceColor',[127/255 1 211/255], 'EdgeColor', 'none');
daspect([1, 1, 1])
view(3)
axis tight;
camlight; lighting phong;
```

3.3.2 Making Movies with MatLab

Once we are able to generate images using MatLab, it is not too difficult to make movies. In fact, let's continue the previous example and suppose that `amr.plot_int=10` so that we generate visual files in increments of 10 and say we output 20 visual files in total. Adding the commands[3]

[3]Here we've renamed the visual files `0` to `200` for easy access in the loop.

```
for k=1:21,                    %Begin loop...

  visualnum=num2str(10*(k-1));  %These are the visual files names

  finedata=load(visualnum);
```

just before

```
D=size(finedata);

if (D(2)~=12)

 sprintf('invalid number of columns\n')
```

in the previous example and then adding the lines

```
  MOV(k)=getframe;

end                            %...end loop

movie(MOV)
```

just after

```
camlight; lighting phong;
```

is enough to generate and play the movie[4] `MOV` (we're essentially running a loop and capturing the images in the matrix `MOV`; see the MatLab documentation for more information). To convert this movie to the popular `avi` format we need only add the command

```
movie2avi(MOV,'milkmovie')
```

This produces the movie `milkmovie.avi`.

3.3.3 Synchronizing Movie Frames With Timesteps

We can synchronize movie frames with timesteps as follows. Suppose we want to generate a movie of the bubble rising problem discussed in Chapter 2 with the timestep displayed as the movie plays. We will assume we are working under the `run2d/bubbleout` subdirectory (so we do not accidentally erase any data). First, we will redirect the output that is normally sent to the screen and send it to the file `run.out`. We can do this with the command

```
../amr2d.Linux.g++.f77.ex ../inputs.bubrise > run.out
```

If you were to open this file you would find information about various errors, mass conservation, solvers, AMR, etc. However, we only care about the time data that we will use in MatLab to synchronize with our movie frames. We extract the *step* data and the *time* data from the `run.out` file and send it to a file named `timedata` with the command

```
grep "TIME" run.out > timedata
```

In general, there is no reason why the timestep should be the same from one computational step to another. This is the reason why it would be incorrect to simply add a 'timer' that takes regular steps in

[4]This code was tested using MatLab R12.

the MatLab code and display it on our movie. When we open the `timedata` file it should look something like

```
STEP = 1 TIME = 0.0003357054085 DT = 0.0003357054085
STEP = 2 TIME = 0.000671410817 DT = 0.0003357054085
STEP = 3 TIME = 0.001007116226 DT = 0.0003357054085
STEP = 4 TIME = 0.001342821634 DT = 0.0003357054085
STEP = 5 TIME = 0.001678527043 DT = 0.0003357054085
...
```

Now we simply load this file in the same MatLab code that generates the movie and display a timer on each movie frame that outputs the contents of the `TIME` column. At this point we notice that it would be economic to simply write a *script* file that executes the previous commands automatically.

Appendix A

Sample Inputs File

Below is the inputs file `inputs.bubrise`.

```
# ------------------  INPUTS TO MAIN PROGRAM  -------------------
# CHANGES FROM PAST VERSION:
# smg.useCG becomes mg.usecg, no more smg.eps
# no more raster stuff (hdf), add mac.mac_abs_tol
# blob.* now becomes ns.*

max_step  =  999999   # maximum timestep
#max_step  =  2    # maximum timestep
stop_time =  20.0  # maximum problem time
#stop_time = 2.0 #testing...
#thickness = 0.1

# ------------------  INPUTS TO CLASS AMR ---------------------
# set up for bubble
geometry.coord_sys      = 1        # 0 => cart, 1 => RZ
geometry.prob_lo   =  0.0 0.0
geometry.prob_hi   =  3.0 6.0
#geometry.is_periodic = 1 0

# multigrid class
```

```
mg.verbose = 0
cg.verbose = 0
# set above to 2 for maximum verbosity
mg.nu_f = 40
mg.nu_0 = 1   # 1 - v-cycle 2 - w-cycle
mg.maxiter = 60
mg.maxiter = 100
cg.maxiter = 9000
mg.usecg=0
mg.bot_atol = 1.0e-9
mg.rtol_b = -0.01
Lp.v = 0
Lp.harmavg = 0
ns.be_cn_theta=1.0
ns.rk_theta=1.0
diffuse.max_order=2
godunov.printMinMax=0

ns.pcav=220.2726
ns.dencav=0.001
ns.pchopp=220.2726
ns.denchopp=0.001
ns.soundchopp=1.0

ns.hydrostatic_pressure=10.0
ns.visual_option=2
ns.visual_lo = 0 0
ns.visual_hi = 512 512
ns.visual_side_max = 512

ns.is_twophase=1
ns.is_compressible=0
ns.shock_timestep=0
```

```
ns.shock_capture=0
ns.shock_eta=1.0

amr.n_cell    = 32 64
amr.max_level =  0
ns.repeat_iter = 1
# 0- 1 level 1- 2 levels  2- 3 levels
amr.ref_ratio       = 2 2 2 2 2    # refinement ratio
amr.regrid_int      = 1       # how often to regrid
amr.n_error_buf     = 2 2 2 2 2    # number of buffer cells in error est
amr.grid_eff        = 0.7   # what constitutes an efficient grid
# above was .55 (smaller=> less boxes)
amr.blocking_factor = 8       # block factor in grid generation
amr.check_int       = 50     # number of timesteps between checkpoints
amr.check_file      = chk     # root name of checkpoint file
amr.plot_int        = 200
amr.plot_file       = plt
amr.grid_log        = grdlog  # name of grid logging file
amr.max_grid_size   = 128
#amr.restart         = chk0050
amr.n_cycle          = 1 1 1 1
#amr.trace   =1

# ------------------  INPUTS TO PHYSICS CLASS -------------------
ns.is_conservative=0
ns.dt_cutoff      = 0.000005  # level 0 timestep below which we halt

mac.mac_tol        = 1.0e-7  # tolerence for mac projections
mac.mac_order      = 1        # interp type: 0 => pc 1 => quad 2 => jbb
mac.mac_abs_tol    = 1.0e-7
mac.use_cg_solve   = 1

ns.cfl            = 0.5      # cfl number for hyperbolic system
```

```
ns.init_shrink    = 1.0      # scale back initial timestep
ns.change_max     = 1.1      # scale back initial timestep
ns.visc_coef      = 0.0      # coef of viscosity
ns.visc_abs_tol   = 1.0e-7
ns.init_iter      = 2      # number of init iters to def pressure
ns.gravity        = -9.80    # body force  (gravity in MKS units)
ns.gravityangle = 0.0
ns.tension        = 72.8
ns.ignition       = -30000.0
ns.anelastic      = 0        # 0 => no anelastic approx
#ns.fixed_dt      = 0.02     # hardwire dt
ns.sum_interval   = 1        # timesteps between computing mass
ns.version_A      = 1        # 1 => ver A, 0 => ver B
ns.kinematicfixed = 0
ns.includehoop = 1
ns.tensorsolve = 1
ns.extendcurv = 1
ns.usekluge = 0
ns.do_radiation   = 0        # 1 => do discrete-ordinates radiation
ns.dump_time = -1.2
ns.RUNGAKUTTA = 0
ns.centerpressure=1
ns.visctimestep=1
ns.stokes_mode=0

ns.axis_dir=0
ns.vorterr=999999.0
ns.rgasinlet=1.57
ns.vinletgas=0.0
ns.twall=0.1
ns.advbot=0.0
ns.adv_vel=0.0
ns.adv_dir=1
```

```
ns.viscunburn=1.0

ns.viscburn=0.01    #Was 0.01

ns.viscvapor=0.01


ns.tcenter=-1.0

ns.denspread=1.0

ns.denwater=1.0

ns.denair=0.001

ns.denvapor=0.001

ns.xblob=0.0

ns.yblob=2.0

ns.zblob=0.0

ns.radblob=0.5

ns.denfact=1.0

ns.velfact=0.0

ns.probtype=49


#ns.mem_debug = 1

#ns.v = 1

#ns.d = 1



# ----------------  PROBLEM DEPENDENT INPUTS

ns.lo_bc          = 3 4

ns.hi_bc          = 4 4

ns.conserveden    = 1


# >>>>>>>>>>>>>  BC FLAGS <<<<<<<<<<<<<<<<

# 0 = Interior           3 = Symmetry

# 1 = Inflow             4 = SlipWall

# 2 = Outflow            5 = NoSlipWall


# turn any of these on to generate run-time timing stats
```

```
RunStats.statvar = godunov_box level_project sync_project

# select form of FAB output: default is IEEE
#        ASCII  (this is very slow)
#    NATIVE (native binary form on machine)
#        IEEE   (default, this is portable)
fab.format =  IEEE

# select single or double precision of FAB output data
#        default is whatever precision code is compiled with.
#fab.precision = FLOAT     # output in FLOAT or DOUBLE
fab.precision = DOUBLE    # output in FLOAT or DOUBLE

# ---------------------------------------------------------------------
# -----       CONTOUR PLOTTING ONLY AVAILABLE IN 2-D           -------
# ---------------------------------------------------------------------
# uncomment the next line to set a default level for contour plotting
# contour.level = 1

# These variables control interactive contour plotting on UNIX systems
# file_name   = root name of postscript file (will be appended with ".ps")
# var_name    = name of thermodynamic variable to plot
# device      = 1  => XWINDOW, 2 = POSTSCRIPT, 3 = both
# freq        = intervals between plots (-1 = off)
# n_cont      = number of contour lines per plot
# sho_grd     = 0 => don't show grid placement, 1 => show grid placement
#               2 => show grid placement and overlay velocity vector plot
#               3 => same as 2 except show Umac and not Ucell
# win_siz     = number of pixels in max window direction

#          file_name  var_name  device  freq  n_cont  sho_grd  win_siz  sval  sdir  sstr
contour.verbose = 1
contour.plot = triple triple         1    1       1      2      400      -1       -1    0
```

```
contour.plot = presliquid presliquid -1    1       20      2       400       -1        -1    0
```

Appendix B

Sample Velocity Initializing Subroutine

Below is the subroutine that initializes the velocity field for the 3D milk drop example (`inputs.milkdrop`) from section 2.6.2.

```
      subroutine initmilkdrop(level,time,lo,hi,
     &                      vel,DIMS(vel),
     &                      dx,xlo,xhi)
      IMPLICIT NONE

      integer    level
      integer    lo(SDIM), hi(SDIM)
      integer    DIMDEC(vel)

      REAL_T     time, dx(SDIM)
      REAL_T     xlo(SDIM), xhi(SDIM)
      REAL_T     vel(DIMV(vel),SDIM)

c     :::: local variables
      integer i, j, k
      REAL_T  x, y, z, xtemp,ytemp,ztemp
      REAL_T  hx, hy, hz
```

```
      REAL_T  rho, dist, dist1,c, rhobub
      REAL_T  x_vel, y_vel, z_vel
      REAL_T  rho1,rho2,hval
      integer i1,j1,k1,numsub
      REAL_T rhoh,fnumsub
      REAL_T xl1,xl2,yl1,yl2,zl1,zl2,xmin,ymin,zmin,height,lenarc,theta
      REAL_T x1,y1,z1,x2,y2,z2,area1,area2,area3,area4

#include "probdata.H"

      print *,"levelrz in initmilkdrop is ",levelrz

      hx = dx(1)
      hy = dx(2)
      hz = dx(3)

      if (adv_dir .eq. 1) then
         x_vel = adv_vel
         y_vel = zero
         z_vel = zero
      else if (adv_dir .eq. 2) then
         x_vel = zero
         y_vel = adv_vel
         z_vel = zero
      else if (adv_dir.eq.3) then
         x_vel = zero
         y_vel = zero
         z_vel = adv_vel
      else
         write(6,*) "initmilkdrop: adv_dir = ",adv_dir
         stop
      endif
      if (adv_vel.ne.zero) then
```

```
 print *,"adv_dir,adv_vel ",adv_dir,adv_vel
endif

do k = lo(3), hi(3)
   z = xlo(3) + hz*(float(k-lo(3)) + half)
   do j = lo(2), hi(2)
      y = xlo(2) + hy*(float(j-lo(2)) + half)
      do i = lo(1), hi(1)
         x = xlo(1) + hx*(float(i-lo(1)) + half)
         vel(i,j,k,1) = x_vel
         vel(i,j,k,2) = y_vel
         vel(i,j,k,3) = z_vel
      enddo
   enddo
enddo

return
end
```

Bibliography

[BK02] E.H. van Brummelen B. van Leer B. Koren, M.R. Lewis, *Riemann-problem and level-set approaches for homentropic two-fluid flow computations*, Journal of Computational Physics **181** (2002), 654–674.

[MS97] P. Smereka M. Sussman, *Axisymmetric free boundary problems*, Journal of Fluid Mechanics **341** (1997), 269–294.