Developer's Guide for gas ICs

David Riethmiller

August 6, 2013

Contents

1 Introduction

The gasICs code is written in C, designed to be executed in parallel, and intended to assemble a distribution of SPH gas particles within an assumed static gravitational potential. This gravitational potential may have up to three components: 1) a mandatory dark matter halo, assumed to be an NFW distribution, 2) an optional stellar distribution, selected from NFW, Jaffe, or Hernquist shapes, and 3) optional contribution from a supermassive black hole. The gas density distribution is obtained via a polytropic map from the total gravitational potential, as per Equation 16.

The result of the code is an ASCII file containing gas initial condition data (see Section §7), and an input file compatible with Volker Springel's GADGET2 TreeSPH code (Springel, 2005).

2 The Parameter File

2.1 Sample File

```
% Parameters for Polytropic Gas

OutputDir    .              % Output Directory

Npart        50000.          % Total number of desired gas particles

DMrmax       100.           % Maximum radial extent of dark matter halo in kpc
DMmass       138.           % Dark matter mass, in units of 1e10 Solar Masses

star_rmax    50.            % Maximum radial extent of stellar profile, in kpc
star_mass    100.           % Stellar mass, in units of 1e10 Solar Masses
star_profile 3              % Stellar distribution. 0:none 1:NFW 2:Jaffe 3:Hernquist

grmax        30.            % Maximum radial extent of gas profile in kpc
gMtot        0.1            % Total gas mass, in units of 1e10 Solar Masses

MBH          1.e8           % Mass of seed black hole to be added to simulated potential

npoly        1.5            % Polytropic index
adindex      1.6667         % Adiabatic index

Ascale       0.8            % Ellipsoidal axis ratios
Bscale       0.9            %    (relative to each other)
Cscale       1.0

rot_percentage  0.00        % Percentage (1.0) of potential used for rotation

G            1.0003         % Grav constant
TINY_NUMBER  1.e-12         % A small number
SendText     0              % Send a text message when complete
PhoneNumber  0123456789@vtext.com
```

2.2 Explanation of Parameters

`OutPutDir .`
The output directory, where the code will write all of its output files. A period indicates, "the current directory."

`Npart 50000`
The number of gas particles to attempt. The actual number of particles created will be this number minus mod the number of processors used (Ntask). If Npart / Ntask is an integer, then Npart will result as requested.

`DMrmax 100.`
The maximum radial extent of the dark matter halo, in kpc. If the system is triaxial, this is the maximum extent of the semi-major axis.

`DMmass 138.`
The total dark matter mass, in units of 10^{10} solar masses.

`star_rmax 50.`
Maximum radial extent of the stellar profile, in kpc. If the system is triaxial, this is the maximum extent of the semi-major axis.

`star_mass 100.`
The total stellar mass, in units of 10^{10} solar masses.

`star_profile 3`
The type of stellar profile to create (and then modify, if triaxial). Enter 0 for no stellar profile, 1 for an NFW stellar profile, 2 for a Jaffe stellar profile, or 3 for a Hernquist stellar profile.

`grmax 30.`
Maximum radial extent of the gas profile, in kpc. If the system is triaxial, this is the maximum extent of the semi-major axis.

`gMtot 0.1`
The total gas mass, in units of 10^{10} solar masses.

`MBH 1.e8`
Mass of the seed black hole to be added to the simulated potential, in units of solar masses. Note that a black hole particle is not added to the simulation here; only the effects of its potential are considered.

`npoly 1.5`
The polytropic index.

`adindex 1.6667`
The adiabatic index.

`Ascale 0.8, Bscale 0.9, Cscale 1.0`
The ellipsoidal axis ratios, *relative to each other.* $A_{\rm scale} \leq B_{\rm scale} \leq C_{\rm scale}$.

`rot_percentage 0.00`
: This parameter is antiquated and should always be set to zero. At one point, rotational velocities were built into this initial conditions code, such that some small percentage of the total gravitational potential was used to support rotational velocity, while the remaining percentage went into pressure support. When the code was adapted to create ellipsoidal distributions instead of purely spherical ones, this functionality was disabled, although the routine that reads the parameter file still looks for this line. (The next developer of this code might spend some time working to remove this option entirely.)

`G 1.0003`
: The gravitational constant.

`TINY_NUMBER 1.e-12`
: A very small number.

`SendText 0`
: Send an SMS text message when the code has finished, to the phone number listed below. 0 for "do not send," 1 for "send."

`PhoneNumber XXXXXXXXXX@vtext.com`
: **Please change this to YOUR phone number if you choose to use this option!** The format of the phone number is as follows:

  XXXXXXXXXX@SUFFIX

  where XXXXXXXXXX is the number, and the suffix depends on the carrier.

Carrier	Suffix
Verizon	vtext.com
AT&T	txt.att.net
Sprint PCS	messaging.sprintpcs.com
T-Mobile	tmomail.net

Table 1: Phone Number Suffix

3 Compilation

Compilation of the gasICs code requires an installation of MPI. Both the Wormhole and Coma machines are configured with mpich2, while the Ohio Supercomputing Center (OSC) uses mvapich2-gnu. The gasICs `Makefile` has presets for each of these machines; just uncomment the appropriate line. To install gasICs on a new machine, install an MPI distribution, and add a new `SYSTYPE` entry in the gasICs makefile, pointing the library entry to the appropriate MPI library.

The gasICs compilation results in two executables: `poly_ICs`, which is the main executable, and a supplemental executable called `gadcombine`. The latter entry is merely a compilation of the `gad_combine()` routine in the former entry, as it occasionally is useful as a stand-alone code.

Once the Makefile is correctly configured, type

```
make
```

at the command prompt to create the executables, or

```
make clean
```

to remove them.

4 Starting the Code

The gas initial conditions code is started by entering the command line sequence:

```
mpirun -n 4 /Directory/Where/You/Put/The/Code/poly_ICs paramfile.param
    > outfilename.dat &
```

where the option `-n 4` tells the code to execute on four processors, `paramfile.param` is the name of the parameter file (see Section §2.1), and the output is piped into a file called `outfilename.dat`.

The user may monitor the input to this dump file in real time by calling:

```
less outfilename.dat
```

and entering Shift-f. To freeze the output display, Control-c, and then to exit, q.

5 Utilities

Before we start into the actual pipeline of the code, it is useful to review some of the utility routines, contained within the file `utilities.c`.

5.1 singlPrintf

Contained within: `utilities.c`

```
singlPrintf("Hello world!");
```

This routine is called exactly like the function `printf()` included from `stdio.h`, except that this routine only prints out its argument on processor 0. Very useful when the user wants to print diagnostic information, but not allow *every* processor to print to the terminal. `singlePrintf()` uses `singlAutoflush`. I copied this functionality over from Steven Diehl's modifications to Chris Fryer's SNSPH code (Fryer et al., 2006).

5.2 calc_rhonorm

Contained within: `calculate.c`

We need to specify a normalization density for the dark matter and stellar density profiles.

```
double result = calc_rhonorm(double alpha, double beta, double
    gamma);
```

where `alpha, beta, gamma` are the parameters α, β, γ in the equation

$$\rho(r) = \frac{\rho_0}{(r/r_s)^\gamma [1 + (r/r_s)^\alpha]^{(\beta-\gamma)/\alpha}} \tag{1}$$

This routine essentially integrates Equation 1 backwards to obtain the total desired mass, thus setting $\rho(r)$. Note that this routine does assume spherical symmetry, which may later be broken.

5.3 shell_dV

Contained within: `calculate.c`

This is a utility that carves the volume into ellipsoidal shells using the ellipsoidal ratios given in the parameter file, and integrates whatever integrand is provided with the volume element dV.

```
double result = shell_dV(double *integrand, double *xarr, double
    *yarr, double *zarr, int num, int mode);
```

Here, `integrand` is the array of values to be integrated, `xarr, yarr, zarr` are the arrays of xyz locations, `num` is the number of elements in each of these arrays, and `mode` is a flag that specifies whether or not to output diagnostic information (0 for silent, 1 for verbose). The routine divides the volume into shells along logarithmically increasing segments of the semi major axis. Currently it is hard coded to use 10,000 shells.

5.4 invertfunction

Contained within: `utilities.c`

We need a program that will re-interpolate the values of a function at a set of newly defined points. In other words, given arrays `xin` and `yin,` and a new y-coordinate `yout`, we want to return the corresponding new x-coordinate `xout`.

There are two versions of basically the same routine, each of which executes a linear interpolation. The simpler version is called as:

```
double result = invertfunction_single(double *xin, double *yin,
    double yout, int numin, int verbose);
```

where `xin` and `yin` are the input arrays of $y(x)$ data each containing `numin` elements, `yout` is the y-coordinate of the new point, and `verbose` is an option that displays diagnostic output to the terminal screen. This routine returns the value of the interpolated

x-coordinate of the point corresponding to `yout`.

Assuming that in the discreet order of the `yin` array, for the index i it needs to be true that

$$y_{in}[i] < y_{in}[i+1] \tag{2}$$

(in other words, `yin` is monotonically increasing). The `invertfunction` routines may call `reverse()` (see Section §5.8) to insure monotonic increasing. Then it locates the index for which it is true that

$$y_{in}[i] \leq y_{out} \leq y_{in}[i+1] \tag{3}$$

Then we define a fraction μ such that

$$\mu = \frac{y_{out} - y_{in}[i]}{y_{in}[i+1] - y_{in}[i]} \tag{4}$$

which makes the value of `xout`

$$x_{out} = (1-\mu)x_{in}[i] + \mu x_{in}[i+1] \tag{5}$$

This linear interpolation checks for monotonicity (see section §5.6) and stops the program execution if failed. It also looks to make sure the value of `yout` is within the table limits; if not, we perform a linear *extrapolation* rather than *interpolation*. Finally, it assumes that we only want a single set of coordinates for `(xout, yout)`.

The other function inversion routine performs this same computation on an entire array of data rather than for a single point:

```
double *result_array = invertfunction(double *x_in, double *y_in,
    double *y_out, int num_in, int num_out, int verbose);
```

where all the arguments are as before (except for the underscore characters, which are just cosmetic), and `num_out` is the size of the output array.

5.5 get_potential

Contained within: `quadinterp3d.c`

The first thing to note about this routine is that while it is called *get_potential*, it is actually used to perform an interpolation of **any function of three dependent variables**. In fact, we use it to interpolate the gravitational potential, the gas density, and even the gas distribution function.

`get_potential` functions much in a similar way to `invertfunction_single`, except that it takes three coordinates instead of one, and it performs the interpolation using a quadratic rather than a linear algorithm.

```
double result = get_potential(double *xtable, double *ytable,
    double *ztable, double *pottable, double xtarget, double
    ytarget, double ztarget, int numpart);
```

where `xtable, ytable, ztable`, and `pottable` are the input tables of x, y, z, and $f(x, y, z)$, respectively. `xtarget, ytarget, ztarget` are the cartesian coordinates of the location at which we want to interpolate $f(x, y, z)$, and `numpart` gives the number of elements in each of the input arrays.

`get_potential` is the last routine written in the file `quadinterp3d.c`, and all of its subroutines are also contained within the same file. Rather than repeat the text, I would refer the user to read through this code, which is very well commented. If more explanation is needed for the routines from Numerical Recipes, please see Press et al. (1992).

5.6 is_mono

Contained within: `utilities.c`

Determine if an array is monotonic:

```
int result = is_mono(double *array, int n, int verbose);
```

where `n` is the number of elements in `array`, and `verbose` prints diagnostic output to the terminal screen. The integer `result` will take the following values as shown in Table 2.

Result	Condition
0	Mono check has failed.
1	Monotonic increasing.
2	Monotonic decreasing.
3	Function is not monotonic.

Table 2: Monotonic Check Results.

5.7 killwithsegfault

Contained within: `utilities.c`

This routine causes an intentional segmentation fault crash in the code when something undesirable happens, so that the `gdb` gnu debugging utility can investigate what went wrong.

```
void killwithsegfault(char *this_file, int this_line);
```

This prints out to the terminal screen the file and line where the code stopped.

5.8 reverse

Contained within: `utilities.c`

We reverse the order of an input array. Used in `invertfunction()`:

```
double *reverse(double *arr, int n);
```

where `n` is the number of elements in the input array `arr`. Returns the re-ordered array.

5.9 reject

Contained within: `utilities.c`

This is the rejection algorithm that determines whether we accept or reject a given point into the density distribution. The rejection algorithm itself is relatively simple.

```
int reject(double *DF, double maxDF, int DF_num, double x, double
    y, double z);
```

where `DF` is the distribution function array passed in (see Section §**??**), `maxDF` is the maximum value of the DF, `DF_num` is the number of elements contained in the DF, and `[x,y,z]` is the cartesian coordinate location of a candidate SPH particle, to be tested for acceptance or rejection. The remainder of the algorithm works as follows:

1. We first check to see if the candidate particle location falls within the ellipsoid defined by `Ascale, Bscale, Cscale`, and `grmax` defined in the parameter file. If so, we continue, otherwise we reject.
2. We call `get_potential` (Section §5.5) to interpolate the distribution function at the candidate location.
3. We choose a random number between zero and `maxDF`.
4. If that random number is less than the interpolated value of the DF, then we accept. Otherwise, we reject.

The routine returns 1 for acceptance, 0 for rejection.

5.10 compare_doubles

Contained within: `utilities.c`

Compare two doubles. Table 3 shows the integer values returned.

```
int compare_doubles(cont void *X, const void *Y);
```

Result	Condition
-1	$X < Y$
0	$X = Y$
1	$X > Y$

Table 3: Double Comparison Results

5.11 gas_diagnostics

Contained within: `utilities.c`

Read in the file `gas.dat` and print some diagnostic information to the terminal screen,

such as the total mass by integrating the density through various methods. *This routine is known to be buggy and temperamental. It suffers from a few conflicting data type issues that I never bothered to fix, since the diagnostic information has no bearing on the final result. You can fix it, comment out the line that calls it, or just ignore its output with minimal consequence.*

```
void gas_diagnostics(void);
```

6 Pipeline

The gasICs pipeline is run primarily from `main.c`. As opposed to the nbodyICs code, which is constructed modularly, most of the "action" happens from this main file. All of the prototype definitions are stored in `proto.h`, and all of the global variables are defined in `allvars.h` (and initialized in `allvars.c`).

The first items of interest in `main.c` are the parallel initialization calls:

```
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&NTask);
MPI_Comm_rank(MPI_COMM_WORLD,&ThisTask);
```

Here, we've defined `NTask` to be the number of processors used (input by the user at runtime), and `ThisTask` to be the "current" processor.

Next we do some book keeping, erasing all previous diagnostic output files if they exist, and starting a wall clock timer.

6.1 Reading the Parameter File

The next step in the code is to read in the parameter file, with obvious call:

```
read_parameter_file(ParameterFile);
```

which lives in `read_param.c`. This routine contains a list of parameters which it expects to find in the parameter file, whose name is specified by the user at runtime. Most of the parameters are read into a structure called `All`, defined in `allvars.h`, and so named because all processors have access to this structure.

If the user wants to add a new parameter to be read in and added to the `All` structure, say, a new double variable called `NewSample`, there are three places to edit:

1. Add a new entry to the parameter list in `read_params.c`:

   ```
   strcpy(tag[nt], "NewSample");
   addr[nt] = &All.NewSample;
   id[nt++] = DOUBLE;
   ```

2. Add a new variable to the `All` structure in `allvars.h`:

```
extern struct global_data_all_processes{
  [stuff]
  double NewSample;
  [stuff]
}
   All;
```

3. Add a new entry to the parameter file:

```
NewSample       4.0      % An example of a new double variable parameter
```

Then this new variable will be specified by the user in the parameter file, read in, and accessible to all processors as `All.NewSample`.

The code then returns to `main.c`, where we adjust the units of the specified black hole mass, and set the actual number of particles on each processor.

6.2 Dark Matter (or Gravitational Potential)

This routine actually sets the entire gravitational potential, not just the dark matter. A simple obvious call

```
void dark_matter(void);
```

places the code inside of the file `dark_matter.c`. We are going to define the total gravitational potential on a grid of 50,000 points (this is hard-coded), within a spherical volume with `All.grmax` as the maximum radius. While the actual points may be distributed randomly within a sphere, the value of the potential at those points will be ellipsoidally skewed by the axis ratios defined in the parameter file. In fact, we define

$$S = \sqrt{\left(x\frac{A}{A}\right)^2 + \left(y\frac{A}{B}\right)^2 + \left(z\frac{A}{C}\right)^2} \tag{6}$$

where $A \leq B \leq C$ are the ellipsoidal axis ratios. Note that for the spherical case, $A = B = C$, and S simply reduces to the radial coordinate r.

The dark matter halo is assumed to be an NFW profile (Navarro et al., 1996) , defined for spherical symmetry as

$$\Phi_{DM}(r) = \frac{-4\pi\rho_0 r_s^3 G}{r} \ln\left(\frac{r + r_s}{r_s}\right) \tag{7}$$

where r_s is the scale radius, equal to unity, and ρ_0 is the dark matter density normalization, obtained via the `calc_rhonorm` routine. We modify this spherically symmetric equation to use our skewed ellipsoidal scaling:

$$\Phi_{DM}(x, y, z) = \frac{-4\pi\rho_0 r_s^3 G}{S} \ln\left(\frac{S + r_s}{r_s}\right) \tag{8}$$

We then write out a file called `StatPotTable.dat`, containing a line with the number of dark matter points (50,000), a line with the dark matter density normalization, and then

columns of x, y, z, and $\Phi_{DM}(x, y, z)$. Note that this file contains the potential of *the dark matter component alone.*

Next we compute the contribution of stars to the potential. If the value of `All.star_profile` in the parameter file is greater than zero, we have the following possibilities for the stellar contribution (r_{max} is given in the parameter file for all cases, as is the total stellar mass M_*):

1. The stellar potential follows an NFW profile.

$$\rho_0 = M_* \left[4\pi r_s^3 \left(\ln \left(\frac{r_{max} + r_s}{r_s} \right) - \frac{r_{max}}{r_{max} + r_s} \right) \right]^{-1} \tag{9}$$

$$\Phi_*(x, y, z) = \frac{-4\pi r_s^3 \rho_0 G}{S} \ln \left(\frac{S + r_s}{r_s} \right) \tag{10}$$

2. The stellar potential follows a Jaffe profile (Jaffe, 1983).

$$\rho_0 = M_* \left[4\pi r_s^3 \left(\frac{r_{max}}{r_{max} + r_s} \right) \right]^{-1} \tag{11}$$

$$\Phi_*(x, y, z) = \frac{4\pi r_s^3 \rho_0 G}{r_s} \ln \left(\frac{S}{S + r_s} \right) \tag{12}$$

3. The stellar potential follows a Hernquist profile (Hernquist, 1990).

$$\rho_0 = M_* \left[2\pi r_s^3 \frac{r_{max}^2}{(r_{max} + r_s)^2} \right]^{-1} \tag{13}$$

$$\Phi_*(x, y, z) = \frac{-2\pi r_s^3 \rho_0 G}{S + r_s} \tag{14}$$

We then write out a file called `StatPotTable+stars.dat`, containing a line with the number of dark matter points (still 50,000), a line with the dark matter density normalization, and then columns of x, y, z, and $\Phi_{DM+*}(x, y, z)$. Note that this file contains the potential of *the dark matter component plus the stars.*

Finally, we compute the potential contribution from the assumed black hole of mass M_{BH}, using a softening length ϵ:

$$\Phi_{BH}(r) = -\frac{M_{BH} G}{r + \epsilon} \tag{15}$$

Note that despite our ellipsoidal dark matter and stellar potentials, the black hole is a point mass and thus constitutes a spherically symmetric potential by nature.

We then write out a file called either `StatPotTable+BH.dat` if there are no stars or `StatPotTable+BH+stars.dat` to include stars , containing a line with the number of dark matter points (still 50,000), a line with the dark matter density normalization, and then columns of x, y, z, and $\Phi_{Total}(x, y, z)$. Note that this file contains the potential of *the*

total gravitational potential.

We write out one additional file: `norm_params.dat`. The anatomy of this file is described in Section §7. Note that this file is not yet complete; we will append more data to it later in the code.

For the remainder of the code execution, variables which contain the abbreviation "DM" (for "dark matter") actually involve the total gravitational potential, as opposed to the dark matter component only.

6.3 Determining $C_{\rm poly}$ and $K_{\rm poly}$

At this point, we return to `main()` and define a subset of the DM grid which lie interior to the maximum extent of the gas, with variable prefix `inner_[whatever]`. We do this because we waste computing power when we interpolate the entire array of DM points, so it is advantageous to include only those grid points that will affect the gas distribution.

Of the potential values in the array `inner_DMpot`, the value that is the least negative (since gravitational potential is intrinsically negative) is set to $C_{\rm poly}$. As is evident from Equation 16, the purpose of $C_{\rm poly}$ is to shift the gravitational potential and drive it to zero at the "edge" of the gas distribution. However, because a zero potential maps into a zero density, and most SPH codes throw an error when they encounter a zero density value, we set $C_{\rm poly}$ to 80% of the edge potential, such that the effective potential becomes small but nonzero, as demonstrated in Figure 1.

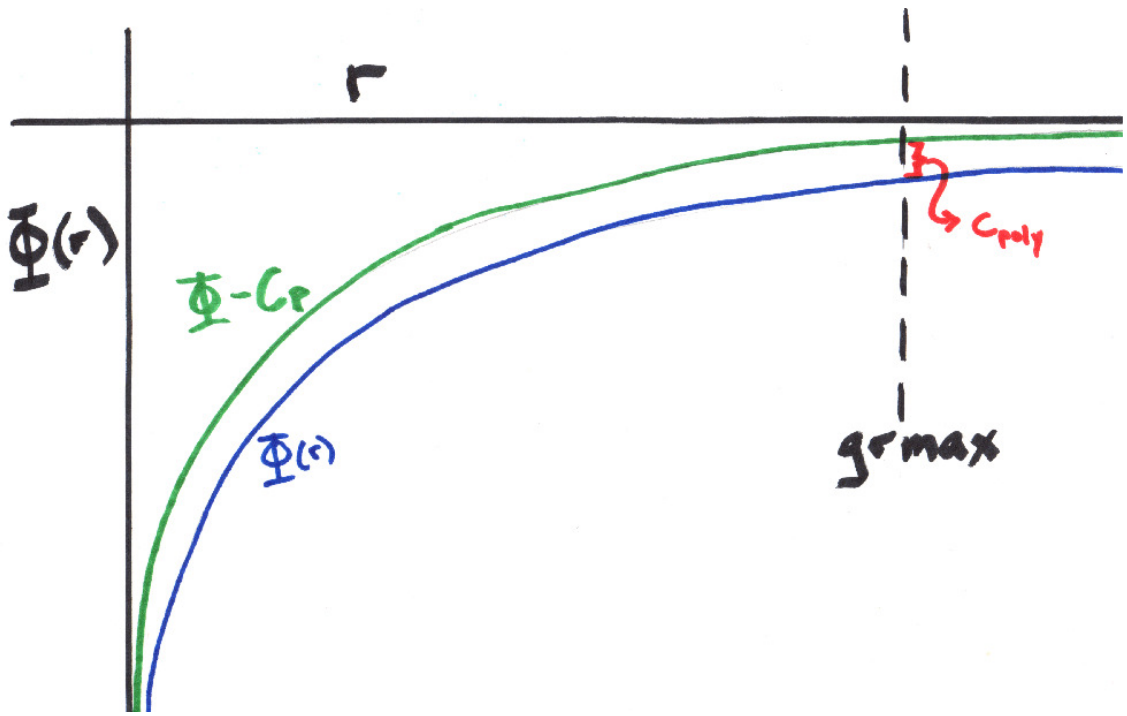

Figure 1: Effective potential shifted by $C_{\rm poly}$.

Once $C_{\rm poly}$ has been determined (and $n_{\rm poly}$ is set by the user in the parameter file), we move on to computing $K_{\rm poly}$, which is indirectly related to the total mass of the gas distribution. The routine `calc_kpoly()` is called (found in `calculate.c`), and basically inverts the density integral, knowing the desired total mass, to recover $K_{\rm poly}$. The user may consult `Calc_Kpoly.pdf` for the math involved in this routine.

We then return again to `main()`.

6.4 Gas Density and Distribution Function

At this point, we are ready to map the total gravitational potential into a gas density distribution, and we compute this map for every point on the inner_DM grid:

$$\rho(x,y,z) = \left[\frac{-1}{K_{\rm poly}(n_{\rm poly}+1)}\left(\Phi(x,y,z) - C_{\rm poly}\right)\right]^{n_{\rm poly}} \tag{16}$$

We then execute a sanity check by using the `shell_dV` routine to integrate the density over ellipsoidal volumetric shells, expecting to recover the total gas mass. If the system is perfectly spherical, we also execute a radial integral to the same effect.

The distribution function we compute is equal to the mass density profile divided by the total mass; essentially, we use the number density as the distribution function DF.

$$f(x,y,z) = \rho(x,y,z)/M_{\rm gas} \tag{17}$$

We do a sanity check to make sure the integral of the DF makes sense (we don't require $\int f(x,y,z)dV$ to equal unity, only to yield a finite value), and then write out a diagnostic file called `distfunc.dat`, with columns for the x, y, z locations, the mass density, the total gravitational potential, and the distribution function.

6.5 Parallel Particle Placement

Up until this point, we've instructed all processors to execute the same commands. Now we exploit the parallel capabilities of the code, dividing the work of accepting/rejecting particle placements among all processors. (*Note: this section of the code is also very well commented; in some cases it may be equally sufficient just to read through main.c as to read this user guide.*)

Each processor initializes xyz arrays of size `Nlocal`, the *local* number of particles. (That is, each processor will hold the total number of particles specified in the parameter file, divided by the number of processors). Each processor also opens up a temporary output file called `procDumpN`, where N is the number of the processor (i.e., processor zero opens up `procDump0`, processor one opens up `procDump1`, etc.). If these procDump files already exist, each processor opens its own respective file, and reads in xyz particle positions; we then continue to choose random xyz positions and accept or reject them via the `reject()` routine (Section §5.9) until the desired number of particles is reached. If accepted, each particle position is immediately appended to its processor's procDump file. This way, if the code crashes during this process, it is not necessary to start over completely. The procDump files are deleted only when the code has completely finished.

Each processor's local list of `[x,y,z]` particle positions is gathered and concatenated into a global list of `[allx, ally, allz]` positions, accessible to all processors.

6.6 Assigning Fluid Attributes

Now that we have the accepted particle positions, we can start to assign fluid attributes at each location. We declare this list of attributes: position `[usex, usey, usez]`, smoothing length `h`, particle mass `m`, pressure `Pr`, internal energy per unit mass `u`, particle velocity `[vx,vy,vz]`[1], particle acceleration `[ax,ay,az]`, gravitational potential `pot`, and gas density `gas_rho`.

In a similar process to the procDump file creation above, each processor opens its own gasfileN file (where again, N is the processor number). For every particle on the local processor, each of the fluid attributes is computed as follows:

1. Gravitational Potential (`pot`): we use the `get_potential()` function to interpolate the value of potential at the particle's location.
2. Density (`gas_rho`): we use Equation 16 to map the gravitational potential into density.
3. Velocity (`vx, vy, vz`): velocities are set to zero.
4. Acceleration (`ax, ay, az`): accelerations are set to zero.
5. Smoothing Length (`h`):
$$h = \rho_{\text{gas}}^{-1/3} \tag{18}$$
6. Particle Mass (`m`):
$$m = M_{\text{gas}}/N_{\text{particles}} \tag{19}$$
7. Pressure (`Pr`):
$$P = K_{\text{poly}}\rho_{\text{gas}}^{1+1/n_{\text{poly}}} \tag{20}$$
8. Internal Energy per Unit Mass (`u`):
$$u = \frac{P}{\rho_{\text{gas}}(\gamma - 1)} \tag{21}$$
where γ is the adiabatic index, specified in the parameter file.

Each set of fluid attributes, in addition to the particle locations, are immediately appended to the processor's gasfileN after they are computed.

At this point, we are finished with the parallel capabilities of the code, and shift to doing all of our work on processor 0.

All of the individual gasfiles are read in, starting with `gasfile0` and ending with `gasfileN`, where `N` is the number of processors minus one (since numbering starts at zero). The information from each gasfile is concatenated into a file called `gas.dat` (see Section §7 for

[1] As the rotational velocity component of this code is currently antiquated, we set all of the particle velocities to zero.

a detailed description of this file's anatomy).

The actual meat of the work is now finished. We call `gas_diagnostics()` to examine the newly created file (see Section §5.11 and the notes therein).

Finally, we call the routine `gadcombine()`, which also has its own standalone executable. This routine takes the `gas.dat` file, as well as a file called `nbody.dat`[2] containing Nbody particle information (if it exists in the same directory) and combines them into an initial conditions file usable in the SPH code GADGET2. This GADGET2 file will have the "snapshot 2" format described in Volker Springel's user guide, downloadable from: `http://www.mpa-garching.mpg.de/gadget` . The `gadcombine()` routine is a modification of the sample read/write routine that is also downloadable from the same URL, in the bundled Gadget code.

6.7 Cleaning Up

Now we append the values of $C_{\rm poly}$, $n_{\rm poly}$, and $K_{\rm poly}$ into the file `norm_params.dat`, which was created during Section §6.2.

We stop the wall clock timer, compute the OSC resource units computed (only relevant if running at Ohio Supercomputing Center), print this information to the terminal screen, and send it in a text message if so requested in the parameter file.

We delete all the temporary procDumpN and gasfileN files, and free all the dynamically-allocated memory arrays.

7 End Result

The end product of the gasICs code consists of two files concerning the particle data: an ASCII file called `gas.dat`, and a GADGET2 initial conditions snapshot called `statpot_gas.gad`.

`gas.dat` has the following anatomy:

1. A line of data specifying the number of particles, the value of $K_{\rm poly}$, the value of $n_{\rm poly}$, and the value of $C_{\rm poly}$.
2. Columns of data specifying particle positions x, y, z, particle velocities v_x, v_y, v_z, particle mass m, particle internal energy per unit mass u, mass density $\rho_{\rm gas}$, smoothing length h, gravitational potential Φ, and particle acceleration a_x, a_y, a_z.

The `statpot_gas.gad` file has the same format as a GADGET2 type-2 snapshot, and contains the gas particle information as well as the nbody particle information if the corresponding `nbody.dat` file was present in the same directory at runtime.

This code also outputs several files related to the static potential used: various versions of `StatPotTable.dat` (described in Section §6.2), as well as `norm_params.dat`, whose construction is as follows:

1. The dark matter density normalization

[2] refer to the nbodyICs users guide

2. The stellar density normalization
3. The type of stellar profile used [0-4]
4. A value for the stellar velocity dispersion (zero)
5. The maximum radial extent of the stellar profile
6. Ellipsoidal axis ratio A
7. Ellipsoidal axis ratio B
8. Ellipsoidal axis ratio C
9. The mass of the black hole
10. The value of $K_{\rm poly}$
11. The value of $n_{\rm poly}$
12. The value of $C_{\rm poly}$

Both the `StatPotTable.dat` files and the `norm_params.dat` file have options for use in GADGET2.

8 List of Files

Essential program files that comprise this code:

`Makefile`	`dydx.c`	`quadinterp3d.c`
`allvars.c`	`gadcombine.c`	`quadinterp3d.h`
`allvars.h`	`main.c`	`read_param.c`
`calculate.c`	`nrutil.c`	`utilities.c`
`comb_defs.h`	`nrutil.h`	
`dark_matter.c`	`proto.h`	

Non-Essential utilities:

`cleanICs.sh` - remove all files resulting from the execution of the code

`gadcom_main.c` - a standalone wrapper for `gadcombine()`.

`simplifyICs.sh` - remove only the diagnostic files, and keeping only the results

Documentation:

`Calc_Kpoly.pdf`

`Gas_User_Guide.pdf`

9 Appendix A: Things That Can Go Wrong

If you are not editing the code itself, there's not much that can really go wrong. Here are some things that I can think of, mostly related to what is set in the parameter file:

- Make sure that the ellipsoidal axis ratios preserve the relationship $A \leq B \leq C$.
- Every configuration I've created has the dark matter halo extend well beyond the gas cloud, i.e. `DMrmax > star_rmax`. I'm not sure what consequences may occur if this relationship is inverted.
- If there is to be no stellar distribution, it is wise to set both `star_profile` and `star_mass` to zero.
- If you're not working on Wormhole, Coma, or OSC, it may be tricky to set up or install the MPI libraries and include them in the Makefile; asking for help may be smart.
- Avoid invoking more processors than are available on your machine, as this will slow down the code execution.
- Obviously, make sure you have write permission to the directory in which you are working, as the code dumps and removes temporary files.

If you *are* editing the code, here are a few things to remember:

- You are coding in PARALLEL. Unless you enclose a command within a block such as

  ```
  if (ThisTask == 0){
      printf("Hello, world!");
  }
  ```

  that command is executed on ALL processors. Note, however, that the `singlPrintf()` call described in Section §5.1 accomplishes exactly the same task as that printed here.
- It is worth Googling `MPI_Allgather()` and `MPI_Barrier()`, and understanding their functionality in detail before attempting any changes to the parallel calls.
- If the code hangs, it is likely that one of the MPI calls is preventing further progress. This happens when one of the processors is unable to reach that line, and all other processors are waiting for it.
- The GNU Debugger `gdb` is a useful tool, but it only runs in serial mode (one processor only). If there's a problem in serial mode, the same problem will appear in parallel mode. However, problems with MPI calls may not show up in serial mode.
- I have not provided detailed descriptions for the routines that come from Numerical Recipes. For these routines, which are commented as such, I refer the user to Press et al. (1992).

10 Bibliography

Fryer, C. L., Rockefeller, G., & Warren, M. S. 2006, Astrophys. J., 643, 292

Hernquist, L. 1990, \apj, 356, 359

Jaffe, W. 1983, Mon. Not. R. Astron. Soc., 202, 995

Navarro, J. F., Frenk, C. S., & White, S. D. M. 1996, \apj, 462, 563

Press, W. H., Teukolsky, S. A., Vetterling, W. T., & Flannery, B. P. 1992, Numerical recipes in C (2nd ed.): the art of scientific computing (New York, NY, USA: Cambridge University Press)

Springel, V. 2005, Mon. Not. R. Astron. Soc., 364, 1105