Developer's Guide for GADGET2 Modifications

David Riethmiller

August 9, 2013

Contents

1 Introduction

This is a guide that describes only the modifications made to the public TreeSPH code GADGET2 (Springel, 2005), available along with its User Guide from the Gadget2 website `http://www.mpa-garching.mpg.de/gadget/` It is assumed that the user has already read both the journal article and the user guide for the public code.

2 Installation Notes

As outlined in the main Gadget2 User's Guide, it is necessary to install GSL, FFTW, and an MPI library. After spending several days of trial-and-error compilations to discover the correct compiler flags, I wrote down the successful commands into the following pseudo-scripts. These commands are specific to Coma. "%> " indicates the terminal command prompt. These compilations should be done in the order given, as they build on dependencies.

2.1 Compiling GNU Architectures

Compile MPICH2:

```
%> cd /Volumes/SecondHD/rieth/mpich2_gnu/mpich2-1.4.1p1/
%> ./configure --prefix=/Volumes/SecondHD/rieth/mpich2_gnu CC=/usr/local/bin/gcc
F77=/usr/local/bin/gfortran
%> make
%> make install
```

Compile GSL:

```
%> cd /Volumes/SecondHD/rieth/gsl_gnu/gsl-1.9
%> ./configure --prefix=/Volumes/SecondHD/rieth/gsl_gnu CC=/usr/local/bin/gcc
F77=/usr/local/bin/gfortran
%> make
%> make install
```

Compile FFTW:

```
%> cd /Volumes/SecondHD/rieth/fftw_gnu/fftw-2.1.5
%> ./configure --prefix=/Volumes/SecondHD/rieth/fftw_gnu --enable-mpi --enable-type-prefix
CC=/usr/local/bin/gcc F77=/usr/local/bin/gfortran
MPICC=/Volumes/SecondHD/rieth/mpich2_gnu/bin/mpicc
%> make
%> make install
```

2.2 Compiling Intel Architectures

The Intel versions are better for use on a supercomputer, since Intel runs faster than GNU. However, the Intel builds are much less forgiving during compilation. I found that a script execution was much cleaner than terminal commands.

Compile MPICH2:

```
#! /bin/sh

CDIR=/opt/intel/
FDIR=/opt/intel/fc/10.1.008/

sh ${CDIR}/bin/iccvars.sh ia32
sh ${CDIR}/bin/compilervars.sh ia32
sh ${FDIR}/bin/ifortvars.sh

./configure \
    --prefix=/Volumes/SecondHD/rieth/mpich2_intel \
    CC=icc \
    CFLAGS="-m32 -I${CDIR}/include" \
    CXX=icpc \
    CXXFLAGS="-m32 -I${CDIR}/include" \
    FC=ifort \
    FCFLAGS="-m32 -I${FDIR}/include" \
    F77=ifort \
    FFLAGS="-m32 -I${FDIR}/include" \
    LDFLAGS="-L${CDIR}/lib -L${FDIR}/lib" \
    2>&1 | tee c.txt

make
make install
```

Compile GSL:

```
#! /bin/sh

CDIR=/opt/intel/
FDIR=/opt/intel/fc/10.1.008/

sh ${CDIR}/bin/iccvars.sh ia32
sh ${CDIR}/bin/compilervars.sh ia32
sh ${FDIR}/bin/ifortvars.sh

./configure \
    --prefix=/Volumes/SecondHD/rieth/gsl_intel \
    CC=icc \
    CFLAGS="-m32 -I${CDIR}/include" \
    CXX=icpc \
    CXXFLAGS="-m32 -I${CDIR}/include" \
    FC=ifort \
    FCFLAGS="-m32 -I${FDIR}/include" \
    F77=ifort \
    FFLAGS="-m32 -I${FDIR}/include" \
    LDFLAGS="-L${CDIR}/lib -L${FDIR}/lib" \
    2>&1 | tee c.txt

make
make install
```

Compile FFTW:

```
#! /bin/sh

./configure \
    --prefix=/Volumes/SecondHD/rieth/fftw_intel \
    --enable-mpi \
    --enable-type-prefix \
    CC="/opt/intel/bin/icc -no-gcc" \
    --disable-f77 \
    --disable-fc \
    MPICC=/Volumes/SecondHD/rieth/mpich2_intel/bin/mpicc \
    2>&1 | tee c.txt

make
make install
```

and then edit Makefile:

```
    MPILIBS = -L/Volumes/SecondHD/rieth/mpich2_intel/lib -lampe -llmpe -lmpe \
    -lmpe_nompi -lmpe_nompi_null -lmpe_null -lmpich -lmpichxx -lmpl -lopa -lpmpich \
    -ltmpe
```

2.3 Compiling Gadget

Once `Makefile` reflects the correct build and library locations, the GADGET2 code is simply compiled by entering

```
%> make
```

in the code directory.

3 Utilities

Before we start into the actual use of the code, it is useful to review some of the utility routines, many of which are duplicated from the gasICs code.

3.1 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).

3.2 invertfunction

Contained within: `invertfunction.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

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

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

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

Then we define a fraction μ such that

$$\mu = \frac{y_{out} - y_{in}[i]}{y_{in}[i+1] - y_{in}[i]} \tag{3}$$

which makes the value of `xout`

$$x_{out} = (1-\mu)x_{in}[i] + \mu x_{in}[i+1] \tag{4}$$

This linear interpolation checks for monotonicity (see section §3.4) 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.

3.3 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).

3.4 is_mono

Contained within: `invertfunction.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 1.

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

Table 1: Monotonic Check Results.

3.5 killwithsegfault

Contained within: `endrun.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.

4 The Parameter File

4.1 Sample Parameter File

This is a sample parameter file that is read into the Gadget2 code. Additions which are not included in the default public code are highlighted in red.

```
%  Relevant files

InitCondFile            statpot_gas.gad
OutputDir               .
CoolingFile             mzero.cie
StatPotFile             StatPotTable+BH+stars.dat
EnergyFile              energy.txt
InfoFile                info.txt
TimingsFile             timings.txt
CpuFile                 cpu.txt
RestartFile             restart
SnapshotFileBase        snapshot
OutputListFilename      parameterfiles/output_list.txt

% CPU time-limit

TimeLimitCPU      345600  % = 4 days
ResubmitOn        0
ResubmitCommand   my-scriptfile

% Code options

ICFormat                2
SnapFormat              2
ComovingIntegrationOn   0

TypeOfTimestepCriterion  0
OutputListOn             0
PeriodicBoundariesOn     0
AdiabaticSqueeze         0  % Deform spherical config into triaxial via adiabatic squeeze.
do_cooling               2  % Execute cooling curve.
hold_particles_fixed     0  % Do not allow Gadget to update particle positions.
do_static_potential      2  % Apply a static potential to gas particles.
read_poly_params         0  % Read in the values of npoly and Cpoly.
grow_BH                  0  % Place a seed black hole at simulation center
do_feedback              0  % Execute energy feedback mechanism.
do_thermal_feed          0  % Do thermal feedback from black hole
```

```
do_mech_feed             0  % Do mechanical (momentum) feedback from BH.
do_settle                1  % Do acoustic settling run
do_stellarFB             0  % Add gas mass loss from evolving stars
SendText                 0  % Send a text message to phone # in main.c when complete.

%  Characteristics of run

TimeBegin              0.0          % Begin of the simulation
TimeMax                600.0        % End of the simulation

Omega0                 0
OmegaLambda            0
OmegaBaryon            0
HubbleParam            1.0
BoxSize                0

MBH0                        1.e6   % in solar masses!

% Output frequency

TimeBetSnapshot        1.0
TimeOfFirstSnapshot    0

CpuTimeBetRestartFile     36000.0    ; here in seconds
TimeBetStatistics         0.05

NumFilesPerSnapshot       1
NumFilesWrittenInParallel 1

% Accuracy of time integration

ErrTolIntAccuracy     0.025

CourantFac            0.15

MaxSizeTimestep       0.1
MinSizeTimestep       0.00

% Tree algorithm, force accuracy, domain update frequency

ErrTolTheta                   0.5
TypeOfOpeningCriterion        1
ErrTolForceAcc                1.e-2

TreeDomainUpdateFrequency    0.1

%  Further parameters of SPH

DesNumNgb              64
MaxNumNgbDeviation     2
ArtBulkViscConst       0.8
```

```
InitGasTemp            1.e7          % always ignored if set to 0, applied iff u=0 in ICs
MinGasTemp             0

% Memory allocation

PartAllocFactor       1.5
TreeAllocFactor       3.0

BufferSize            25          % in MByte

% System of units

UnitLength_in_cm         3.085678e21        %  1.0 kpc
UnitMass_in_g            1.989e43           %  1.0e10 solar masses
UnitVelocity_in_cm_per_s 2.0735e7           %  207 km/sec
GravityConstantInternal  0

% Softening lengths

MinGasHsmlFractional        0.01

SofteningGas       0.01
SofteningHalo      0
SofteningDisk      0
SofteningBulge     0
SofteningStars     0.01
SofteningBndry     0

SofteningGasMaxPhys       0.01
SofteningHaloMaxPhys      0
SofteningDiskMaxPhys      0
SofteningBulgeMaxPhys     0
SofteningStarsMaxPhys     0.01
SofteningBndryMaxPhys     0

MaxRMSDisplacementFac 0.2
```

4.2 Explanation of Parameters (Except the Code Options)

A brief description of the added elements in the parameter file is provided here. We leave the more complicated *Code Options* for Section §5.

`CoolingFile` `mzero.cie`

The name of the file containing information related to the Sutherland and Dopita gas cooling curves (Sutherland & Dopita, 1993), any of which can be downloaded from Ralph Sutherland's website, `http://www.mso.anu.edu.au/~ralph/data/cool/` . Then simply edit the file to strip out the column headings, replacing them with the number of lines in the file.

`StatPotFile` `StatPotTable+BH+stars.dat`

The name of the static potential file created by the gasICs code during its `dark_matter()` routine. See the gasICs User Guide, Section 6.2. This file contains columns for x, y, z locations, and the potential at each of those locations.

`MBH0 1.e6`

If `grow_BH` is set, this is the mass, in solar masses, of the seed black hole that is created.

5 Code Options

5.1 Adiabatic Squeeze

If `AdiabaticSqueeze` is set to 1 in the parameter file, we apply a viscous drag force as described in Widrow (2008), which in turn draws from Holley-Bockelmann et al. (2001). The point is to "squeeze" the system slowly, first in one direction, then in another, with the intent to deform a spherical system into a triaxial one. The routine `squeeze()` is called from `accel.c` and written in `squeeze.c`.

5.2 Cooling

Before we discuss gas cooling, we should note that Gadget tracks entropy as opposed to internal energy per unit mass. As such, every time we want to modify the temperature or energy, we must convert in and out of entropy. This is relevant in all cooling options, and in the feedback procedures described later. Entropy S is converted as follows:

$$S = \frac{u(\gamma - 1)}{\rho^{\gamma - 1}} \tag{5}$$

where u is the internal energy per unit mass, ρ is the local gas density, and γ is the adiabatic index.

In all methods of cooling used here, the general idea is first to compute the cooling rate Λ, and then subtract the amount of energy appropriate for the timestep dt:

$$\frac{du}{dt} = \Lambda \frac{n^2}{\rho} \tag{6}$$

where n is the number density of ions, and we assume completely ionized Hydrogen.

There are two available options for radiative gas cooling. If the value of `do_cooling` in the parameter file is greater than zero (i.e. either 1 or 2), the code will look for the file specified by `CoolingFile`, and read in the data.

If `do_cooling` is set to 1, we completely ignore the cooling tables that we have just input, and instead apply a radiative cooling to the gas such that the energy removed is proportional to the square root of the current temperature. The math for this $T^{1/2}$ cooling curve is outlined in the supplementary documentation file `T_half.pdf`. This option is well-suited for executing a cooling test, as the theoretical cooling rate is known against the rate that Gadget computes.

The other cooling option is enabled by setting `do_cooling` to 2. It uses the Sutherland & Dopita tables (Sutherland & Dopita, 1993) to compute the cooling rate Λ, and employs the Exact Time Integration Scheme described in Townsend (2009). The important points of the math are summarized in the documentation file `Townsend.pdf`.

Additionally, if the parameter `do_settle` is enabled, cooling does not engage until some specified percentage of the simulation's finishing time has been reached. See Section §5.9.

The cooling routines are stored in the file `cooling.c`, which is well commented.

5.3 Holding Particles Fixed in Space

If we want to execute a simulation where the particles are not allowed to move, such as the nicknamed "can of gas cooling test" in which we measure the cooling rate of a fixed cloud of gas particles against a theoretical cooling rate, we need to disable the chunk of code that updates particle positions. That is, we allow the updates to be calculated, but not applied. This happens in `predict.c`, but also has implications in `accel.c` where an alternate density calculation is performed, and in `hydra.c` where some entropy considerations must be skipped. To enable this function, set `hold_particles_fixed` to 1 in the parameter file.

5.4 Implementing a Fixed Static Potential

Gadget's normal mode of operation is to have the gas represented by SPH particles, and the dark matter, stars, black hole, bulge, etc. represented by collisionless Nbody particles. We can remove the necessity of these latter particles by imposing a static gravitational potential on the gas particles via an extra component added into their accelerations.

If `do_statpot` is set to 1 in the parameter file, we read in the table specified by `StatPotFile`. Then, the gravitational potential is interpolated at the location of each particle in turn, and differentiated with respect to each direction to obtain the force law. As one might expect, this option is computationally expensive, and slows the code execution significantly.

The other option is to set `do_statpot` to 2. Instead of the `StatPotTable` file, we instead read in a file called `norm_params.dat`, also created by the gasICs code. This file contains:

1. The dark matter density normalization
2. The stellar density normalization
3. The type of stellar profile used (0: None, 1: NFW, 2: Jaffe, 3: Hernquist)
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 $C_{\rm poly}$

This list is identical to the one specified in the gasICs User Guide, section §7, and is sufficient to recreate an analytical representation of the entire gravitational potential at any specified location, using the same math as described in gasICs Section §6.2. This option is much more computationally efficient, provided that the potential is known in this manner.

Both options for the static potential implementation are written in the file `statpot.c`, which is well commented.

5.5 Creating a Seed Black Hole

If `grow_BH` is set to 1 in the parameter file, and this run is not restarted from a previous snapshot, then we're going to deposit a black hole particle (technically, a type-4 "star" particle by Gadget's definitions) of mass `MBH0` (specified in parameter file) at the center of the simulation. `run.c` will call `create_BH()`, stored in `feedback.c`. This routine creates the new particle at `[0,0,0]`, gives it zero velocity, and increments the total number of particles.

If we *are* restarting from a previous snapshot, then this option requires that we locate the black hole. `run.c` calls `find_restarted_BH()` (also from `feedback.c`), locates what it assumes is the only type-4 particle in the simulation, and identifies it as the black hole. **If you intend to add other star type-4 star particles, you will need to re-write this routine!** `run.c` also calls `center_BH()` (found just below `find_restarted_BH()`) and resets its position to the exact center if necessary.

5.6 Enabling Feedback

If `do_feedback` is set to 1 in the parameter file, then `evole_BH()` is called (stored in `feedback.c`). This is likely the most complicated modification to the Gadget code. It is based on the the procedure outlined in Choi et al. (2012) (which in turn draws from Springel et al. (2005)), and although the routine is well commented, I will summarize some of the steps here. (However, the user is urged to read this paper before continuing.)

- The first step is to find the black hole particle - not in physical space, but which processor holds it. This is determined by tracking the BH particle's index, stored as `All.BH_index`. The code cycles through all particles on the local processor (*remember this is in PARALLEL, so all commands are executed on all processors*) until it finds an ID that matches the BH index. The processor on which this happens is announced to all other processors as `BHproc`. We also announce the current black hole mass and particle timestep (since all particles have individual timesteps).
- The next step is to calculate several kernel-weighted commodities, such as density and sound speed, at the black hole location. Note that there is no actual *SPH particle* at this location to carry the density tag; we simply want to know what the gas density should be at that location. We begin by computing the kernel w_j with respect to the origin for every particle j, using Gadget's kernel definition:

$$w_j = \frac{8}{\pi h^3} \begin{cases} 1 - 6\left(\frac{r}{h}\right)^2 + 6\left(\frac{r}{h}\right)^3 & \text{if } 0 \leq \frac{r}{h} \leq \frac{1}{2} \\ 2\left(1 - \frac{r}{h}\right)^3 & \text{if } \frac{1}{2} < \frac{r}{h} \leq 1 \\ 0 & \text{if } \frac{r}{h} > 1 \end{cases} \tag{7}$$

where r and h are the radial position and smoothing length of particle j, respectively.

- Then the gas density at the black hole's location is given by

$$\rho_{\mathrm{BH}} = \sum_j m_j w_j \tag{8}$$

where mj is the mass of particle j. And the sound speed c_s is given by

$$c_s = \frac{\sum_j \sqrt{\gamma P_j / \rho_j} w_j}{\sum_j w_j} \tag{9}$$

where γ is the adiabatic index and P_j and ρ_j are the pressure and density of each SPH particle j, respectively. Of course, an MPI call is needed to compute these sums across all processors.

- The black hole growth rate is computed as

$$\dot{M}_{\mathrm{BH}} = \frac{4\pi\alpha G^2 M_{\mathrm{BH}}^2 \rho_{BH}}{\left(c_s^2 + v^2\right)^{3/2}} \tag{10}$$

where α is a dimensionless parameter related to the simulation resolution (i.e. number of particles), G is the gravitational constant, and v is the speed of the black hole with respect to the gas cloud. Since we are holding the black hole fixed, for our purposes $v = 0$. We limit the black hole growth rate by the Eddington rate:

$$\dot{M}_{\mathrm{edd}} = \frac{4\pi G M_{\mathrm{BH}} m_p}{\epsilon_r \sigma_T c} \tag{11}$$

Here m_p is the proton mass, σ_T is the Thomson cross-section and ϵ_r is the radiative efficiency, assumed to be a fixed value. We define $\dot{M}_{in}$ to assume the lesser of $\dot{M}_{\mathrm{BH}}$ and $\dot{M}_{\mathrm{edd}}$.

- Next we calculate the probability p_j that a particle is accreted onto the black hole.

$$p_j = \frac{w_j \dot{M}_{in} \Delta t}{\rho_{BH}} \tag{12}$$

where Δt is the black hole particle's timestep. If a random number chosen between zero and one is less than p_j, then particle j is tagged for accretion, by changing its particle type to -1.

- The next block of code computes mechanical feedback, see Section §5.7.

- The next block of code computes thermal feedback, see Section §5.8.

- Next, we use the computed growth rate to update the black hole mass

$$M_{BH} = M_{BH} + \dot{M}_{\rm in}\Delta t \tag{13}$$

  and append some diagnostic information to the file `BH_evolve.dat`.

- Finally, we count the number of particles tagged for accretion, and call `accrete_tagged_particles()` (written later in `feedback.c`), which functions essentially by storing the particles in temporary arrays, and then rewriting them in order by particle type from 0 to 5. Thus, particles of type -1 are not included in the new list.

Note that unless thermal feedback or mechanical feedback is enabled (see below), the only type of "feedback" that occurs is that the black hole grows in mass, and particles are removed from the simulation.

5.7 Mechanical Feedback

If both `do_mech_feed` and `do_feedback` are set to 1 in the parameter file, the mechanical feedback mechanism described in Choi et al. (2012) Section §2.2.2 engages. We follow their description nearly exactly; thus the user is referred to this paper to understand the physics. This part of the code (written inside `feedback.c`) is also very well commented. The only deviation from the Choi procedure is that while we do implement their momentum-sharing scheme, we do not deposit residual thermal energy into the particles.

5.8 Thermal Feedback

If `do_thermal_feed` and `do_feedback` are both set to 1 in the parameter file, this block of the code inside `feedback.c` engages. It follows Choi et al. (2012) Section §2.2.1, with only a few minor differences: instead of distributing the thermal energy to only the nearest neighbors, we distribute to all particles for which the kernel is nonzero (which usually encompasses only the nearest neighbors anyway). We also have to convert between entropy and internal energy per unit mass, via a conversion similar to Equation 5.

5.9 Executing a Settling Run

If `do_settle` is set to 1 in the parameter file, we're going to execute an acoustic settling simulation. This means that only the basic physics is turned on (no feedback or black hole growth). The black hole is not even present; its gravitational influence is merely added into the potential. The point of this simulation is to damp out the Poisson noise generated by Gadget's reconstruction of our initial conditions.

Additionally, if the option `do_cooling` is set in the parameter file, the gas cooling rate is gradually ramped up to its full value starting at some percentage of the simulation's end time. This percentage is hard-coded near the beginning of `cooling.c`.

5.10 Stellar Feedback

If `do_stellarFB` is set to 1 in the parameter file, the routines within `stellar_FB.c` engage, after a set amount of time.

At the time of this writing, this functionality is still in development, and does not work correctly.[1] I will describe briefly what is *supposed* to happen with the code, leaving it to the next developer to adjust it to ensure proper behavior.

The routine `stellar_FB()`, appropriately stored in the file `stellar_FB.c`, is designed to estimate the amount of stellar mass that should be returned to the system from planetary nebulae and supernova during a timestep, then spawn new gas particles containing this mass, and then re-order the array of particles appropriate to Gadget standards. The term "feedback" here is a bit of a misnomer, as no heating or energy injection occurs; all that happens is that new gas particles are added to the system.

The mathematical logic flows as follows.

- From Ciotti et al. (1991), we can estimate the expected stellar mass return rate in an elliptical galaxy as

$$\dot{M}_*(t) \approx 1.5 \times \frac{L_B}{10^{11} L_{B\odot}} t_{15}^{-1.3} M_\odot \mathrm{yr}^{-1} \tag{14}$$

  where L_B is the galaxy's total blue luminosity, $L_{B\odot}$ is the blue solar luminosity, and t_{15} is the simulation age of the elliptical galaxy, in units of 15 Gyrs (valid in the range of 0.5 to 15 Gyrs). Papers more recent than 1991 (as late as 2007) all say something to the effect of, "We've got a more complicated version of this, but for basic approximations, this will work."

- Next we sum Choi et al. (2012) Equation 26 to estimate the total xray luminosity of the galaxy:

$$L_x = \sum_i 1.2 \times 10^{-24} \left(\frac{m_i}{\mu m_p}\right) \left(\frac{\rho_i}{\mu m_p}\right) \left(\frac{k_B T_i}{\mathrm{keV}}\right)^{1/2} \mathrm{erg\ s}^{-1} \tag{15}$$

  where ρ_i, m_i, and T_i are the density, mass, and temperature of particle i respectively, m_p is the proton mass, and μ is the mean molecular weight.

- We can use the $L_x - L_B$ scaling relation to estimate the galaxy's blue luminosity. From Kim & Fabbiano (2013), we find

$$L_{x,\mathrm{gas}} \sim L_B^{5\pm1} \tag{16}$$

  and choosing a reference point on their Figure 2 such that $L_{x,\mathrm{gas}} = 10^{39}$ erg/s corresponds to $L_B = 10^{44}$ erg/s, we find the relation

$$L_B = \left(\frac{L_x}{10^{39}}\right)^{1/(5\pm1)} 10^{44} \tag{17}$$

- Then we have all of the elements necessary to plug into Equation 14, obtaining $\dot{M}_*(t)$, which we multiply by the time step to yield the amount of stellar mass returned during each step.

[1] When the routine engages at its pre-specified time, the addition of new particles somehow leads to the calculation of a negative time step, which then produces erratic temporal behavior.

- We spawn N new gas particles at random locations within the gas ellipsoid, where N is the total stellar mass returned divided by the average gas particle mass. (We want to maintain the particle masses, since depositing the stellar mass return into either significantly larger or significantly smaller mass particles will create an unphysical imbalance.)
- Finally, we reorder the particle arrays such that all the gas particles come first in memory, followed by the other particle types.

5.11 Sending a Text Message

Send an SMS text message when the code has finished, to the phone number listed in `main.c`. **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 2: Phone Number Suffix

Set `SendText` to 0 in the parameter file for "do not send," 1 for "send."

6 List of Added Files

Essential program files that are not included in the public Gadget download:

```
cooling.c                nrutil.h                 statpot.c
dydx.c                   num_integration.c
feedback.c               quadinterp3d.c           stellar_FB.c
invertfunction.c         quadinterp3d.h
nrutil.c                 squeeze.c                utilities.c
```

Documentation:

```
Gadget_Mods_User_Guide.pdf
T_half.pdf
Townsend.pdf
```

7 Appendix A: Things That Can Go Wrong

This list is very similar to the gasICs Appendix A. If you're not editing the code, and something crashes, you've most likely set something wrong in the parameter file.

- I recommend copying and pasting Section §4.1 into a file, and then changing the options as needed. This ensures that you're accounting for all parameters that need to be read.
- 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 §3.1 accomplishes exactly the same task as that printed here.
- It is worth Googling `MPI_Allgather()`, `MPI_Bcast()`, 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).
- Remember that the stellar feedback routine is in development, and is not expected to work correctly. If you set `do_stellarFB` to 1 in the parameter file, you should expect erratic behavior when the routine engages.
- The Gadget user list archive discussions are very useful for Gadget developers. Even if you don't sign up and contribute to the discussions, the solutions to many of your problems may already have been addressed in the archive.
  `http://www.mpa-garching.mpg.de/gadget/gadget-list/`

8 Bibliography

Choi, E., Ostriker, J. P., Naab, T., & Johansson, P. H. 2012, Astrophys. J., 754, 125

Ciotti, L., D'Ercole, A., Pellegrini, S., & Renzini, A. 1991, Astrophys. J., 376, 380

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

Holley-Bockelmann, K., Mihos, J. C., Sigurdsson, S., & Hernquist, L. 2001, Astrophys. J., 549, 862

Kim, D.-W., & Fabbiano, G. 2013, ArXiv e-prints

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

Springel, V., Di Matteo, T., & Hernquist, L. 2005, Mon. Not. R. Astron. Soc., 361, 776

Sutherland, R. S., & Dopita, M. A. 1993, Astrophys. J. Suppl., 88, 253

Townsend, R. H. D. 2009, Astrophys. J. Suppl., 181, 391

Widrow, L. M. 2008, Astrophys. J., 679, 1232