StarCrash: A Parallel Smoothed Particle Hydrodynamics (SPH) Code for Calculating Stellar Interactions

Version 1.0

Joshua Faber

Department of Physics and Astronomy, Northwestern University

jfaber@northwestern.edu

James C. Lombardi, Jr.

Department of Physics and Astronomy, Vassar College

Frederic Rasio

Department of Physics and Astronomy, Northwestern University

Contents

1 Introduction **4**

2 Installation Instructions **5**

3 Numerical Methods **7**

3.1 SPH: Smoothed Particle Hydrodynamics 7

3.1.1 SPH History and Basics 7

3.1.2 The SPH Density: Kernels, Sums, and Special Tricks 7

3.1.3 Parallelization of the Hydro Routines 9

3.1.4 Derivatives in SPH 9

3.1.5 Evolving Quantities in Time 10

3.2 Neighbor Lists 11

3.2.1 Neighbor Lists: Parallel Issues and Particles Off the Grid 12

3.3 The Gravity Solver: an FFT-based Approach 12

3.3.1 FFT Convolution 12

3.3.2 Built-in Gravity Grid Options . . . 14
3.3.3 Gravity Grid Parallelization . . . 14
3.3.4 The FFTW . . . 15
3.4 Relaxation . . . 16
3.4.1 Single Stars . . . 16
3.4.2 Binary Stars . . . 17
3.4.3 Finishing Relaxation . . . 17
3.5 Artificial Viscosity . . . 18
3.5.1 Monaghan . . . 18
3.5.2 Hernquist and Katz . . . 19
3.5.3 Balsara . . . 19
3.5.4 Parallelization of the AV Routines . . . 20
3.6 Gravitational Radiation Reaction . . . 20

4 Running a Calculation: Initial Routines 22
4.1 Automatically Generated Input Files: testinput . . . 22
4.2 Single Stars . . . 23
4.2.1 Equal-mass Particles . . . 23
4.2.2 Equally-spaced Particles . . . 23
4.3 Corotating Binaries . . . 24
4.3.1 Mass Ratio q=1.0: Equal-mass Particles . . . 25
4.3.2 Mass Ratio q=1.0: Equally-spaced Particles . . . 25
4.3.3 Mass Ratio q=1.0: Previously Calculated Single Star Models . . . 25
4.3.4 Mass Ratio q<1.0: Equal-mass Particles . . . 25
4.3.5 Mass Ratio q<1.0: Equally-spaced Particles . . . 26
4.3.6 Mass Ratio q<1.0: Previously Calculated Single Star Models . . . 26
4.4 Irrotational Binaries . . . 27
4.4.1 Equal-mass Binaries . . . 28

4.4.2 Unequal-mass Binaries 28

4.5 Hyperbolic Collisions 29

4.5.1 Equal-mass Hyperbolic Binaries 30

4.5.2 Unequal-mass Hyperbolic Binaries 30

4.6 Restarting From a Previous Run 31

5 File Input/Output **32**

5.1 Input Files 32

5.1.1 Key System Parameters: sph.input 32

5.1.2 Starting New Runs: sph.init 33

5.1.3 Restarting From a Previous Run: restart.sph 34

5.1.4 Using a Single-star Configuration in a Binary: pos.sph 35

5.1.5 Defining Triaxial Ellipsoids for Irrotational Binaries: axes1.sph and axes2.sph 35

5.2 Output Files 35

5.2.1 restart.sph and outxxx.sph 35

5.2.2 energy.sph, biout.sph, gwdata.sph, com.sph, and spin.sph 36

5.2.3 Subroutine densplot 37

6 Common Block Variables and User-defined Parameters **38**

6.1 Common Block Parameters in spha.h 38

6.2 Other Code Parameters 38

6.3 Common Block Variables 39

7 Subroutines **42**

8 References to Cite and Legal Information **56**

8.1 Gnu General Public License 56

9 References **62**

1. Introduction

Hello, and welcome to the first public release of the StarCrash SPH-based parallel hydro code, developed and maintained by Joshua Faber, James Lombardi, and Fred Rasio. We hope this instruction manual will allow you to make the most effective use of our code.

In the sections below, you will find installation instructions, a description of the various parts of the code, starting with the theory behind SPH, continuing on to the techniques we have added to our code involving both parallelization and various physical effects, and concluding with a guide explaining how to use the code properly, as well as to add your own routines.

While this code is being freely distributed, we ask that you please give appropriate credit to those who have spent a great deal of time developing it. If you are going to publish work done with this code, please see Sec. 8 for the proper references to cite, as well as the license agreement we are using, the rather ubiquitous Gnu General Public License (GPL).

2. Installation Instructions

We have prepared the release version of the code to work on as wide an array of systems as possible. This version assumes that your computer system is a parallel system running MPI, the Message Passing Interface. Several versions of MPI are in common use, and we have successfully compiled our code against many of them.

For our code to run properly with self-gravity, you will need to install one freely available software library. It is the FFTW (Fastest Fourier Transform in the West) library, written by Mateo Frigo and Steven Johnson, available for download at `http://www.fftw.org`, and described in Frigo & Johnson (1997, 1998). It is, to the best of our knowledge, the most widely used free, MPI-parallelized, multi-dimensional FFT code available, and forms the central part of our gravity solver. Our code is currently compiled against Version 2.1.5 of the FFTW, and will not work properly by default with older releases. See the notes below, however, for the steps to follow in this case, specifically release 2.0 of the FFTW. We are currently investigating the use of our code with the newly released FFTW version 3.0, and will update this documentation when and if we decide to write it into our code. Currently, FFTW 3.0 does not support MPI-based parallelization, but this may change in the future.

The FFTW `configure` routine does not compile the MPI-based routines by default. Make sure to run `configure` with the option `--enable-mpi`. You may need to set change your system flags to get FFTW to compile properly with the MPI-based routines included. See their documentation for details.

Once you have installed the FFTW, you are ready to install the SPH code. To do so, just follow these simple instructions, and everything should work smoothly.

1. First, unpack the tarball file, `starcrash.tar.gz`, into the directory which will hold all the files.
2. Next, you will need to edit the `makefile`. Set the variable `MPID` to the `/include/` subdirectory of the MPI distribution you will be using. Set `FFTDIR` to the main directory of the FFTW, which should have subdirectories named `/rfftw/` and `/mpi/`. Please note if you are using an older version of the FFTW (version 2.0.x), it may have a slightly different subdirectory structure. If you believe this to be the case, try using the file `makefile.old`. Edit the variables `FC` and `CC` with the name of your preferred Fortran and C Compilers, and set the other variables to match your preferred compilation flags.
3. Now, make the executables. The file which runs the StarCrash code is named `sphagr`. The routine `testinput`, described in Sec. 4.1, will help you generate the input files needed to run the code. The routines `b2pos` and `b2pos2` help you convert our binary output files into ASCII files which can be read into other runs, and are easily converted into general-purpose I/O tools for use in data analysis.

Hopefully, your installation should now be complete. Enjoy the StarCrash code!

3. Numerical Methods

3.1. SPH: Smoothed Particle Hydrodynamics

3.1.1. SPH History and Basics

The basic Smoothed Particle Hydrodynamics (SPH) method was created by Lucy (1977) and Gingold & Monaghan (1977) in order to study fission in rotating stars. It has since been used to study, among other astrophysical topics, large scale structure in the universe, galaxy formation, star formation, supernovae, solar system formation, tidal disruption of stars by massive black holes, and stellar collisions; see Rasio & Shapiro (1992), Monaghan (1992), and Rasio & Lombardi (1999) for a more complete list of references. Our particular code has been used primarily in the study of stellar collisions and mergers, including merging compact object binaries (Rasio & Shapiro 1992, 1994, 1995), collisions involving main sequence stars (Rasio & Shapiro 1991; Lai et al. 1993a; Sills et al. 1997, 2001), blue-straggler formation (Lombardi et al. 1995, 1996, 2002), and most recently post-Newtonian (PN) and relativistic studies of binary neutron star (NS) systems (Faber & Rasio 2000; Faber et al. 2001; Faber & Rasio 2002). A post-Newtonian code is in preparation for public release, and will form the core of version 2.0 of this code.

Because of its Lagrangian nature, SPH presents some clear advantages over more traditional grid-based methods for calculations of stellar interactions. Most importantly, fluid advection, even for stars with a sharply defined surface such as NS, is accomplished without difficulty in SPH, since the particles simply follow their trajectories in the flow. In contrast, to track accurately the orbital motion of two stars across a large 3D grid can be quite tricky, and the stellar surfaces then require a special treatment (to avoid "bleeding"). SPH is also very computationally efficient, since it concentrates the numerical elements (particles) where the fluid is at all times, not wasting any resources on emty regions of space. For this reason, with given computational resources, SPH provides higher averaged spatial resolution than grid-based calculations, although Godunov-type schemes such as PPM typically provide better resolution of shock fronts (this is certainly not a decisive advantage for binary coalescence calculations, where no strong shocks ever develop). SPH also makes it easy to track the hydrodynamic ejection of matter to large distances from the central dense regions. Sophisticated nested-grid algorithms are necessary to accomplish the same with grid-based methods.

3.1.2. The SPH Density: Kernels, Sums, and Special Tricks

A good description of the particular SPH algorithm implemented in our code can be found in Rasio & Shapiro (1992). We summarize here the salient features of the method. In what follows, the letters i and j refer to hydrodynamical quantities defined for each particle, and

The key advance of SPH is to associate with each particle a "smoothing length" h, representing the finite spatial extent of the particle, which can differ in value for separate particles, as well as vary in time. The smoothing length is used in calculating all hydrodynamic terms, since rather than viewing the density of the particle as a spatial delta-function, we represent the spatial density distribution of particle "i" by a kernel function, such that

$$\rho_i(\vec{r}) = m_i W(|\vec{r} - \vec{r}_i|, h), \tag{3-1}$$

where $\vec{r}_i$ is the actual position of particle i, and W is a kernel function.

For the method to work, it is necessary that this kernel function have compact support, and the kernel function be at least singly differentiable. In practice, we define the kernel function W such that it drops to zero at a radius equal to 2 smoothing lengths, and we choose a function which is actually C^2. The choice we use is defined as follows

$$W(r,h) = \frac{1}{\pi h^3} \begin{cases} 1 - \frac{3}{2}\left(\frac{r}{h}\right)^2 + \frac{3}{4}\left(\frac{r}{h}\right)^3, & 0 \le \frac{r}{h} < 1, \\ \frac{1}{4}\left[2 - \left(\frac{r}{h}\right)\right]^3, & 1 \le \frac{r}{h} < 2, \\ 0, & \frac{r}{h} \ge 2. \end{cases} \tag{3-2}$$

To save time, we calculate this function at the beginning of a run and store it in tabulated form.

As the final results of the SPH evolution calculation should not depend on the prescription for choosing the smoothing length h, we are free to choose a convenient expression. In practice, we let h evolve over time such that the particle overlaps a fixed number of neighbors, a parameter set prior to starting the calculation.

By using this formulation, our realization of a fluid configuration into particles results in a well-defined, continuous and differentiable density field which can be evaluated at every point in space. The density at an arbitrary point is given in terms of a sum over all neighboring particles which overlap the point, denoted here by "j", as:

$$\rho(\vec{r}) = \sum_j m_j W(|\vec{r} - \vec{r}_j|, h_j). \tag{3-3}$$

The SPH method allows us to calculate the density at each particle position following the formula above, but in practice we alter the formula slightly, by taking the average smoothing kernel function computed for the particle and its neighbor, finding

$$\begin{aligned} \rho_i &= \sum_j m_j \frac{W(|\vec{r}_i - \vec{r}_j|, h_i) + W(|\vec{r}_i - \vec{r}_j|, h_j)}{2} \\ &\equiv \sum_j m_j W_{ij}, \end{aligned} \tag{3-4}$$

where we define the quantity W_{ij} to be the averaged smoothing kernel function. Furthermore, we modify even this equation slightly during computations, in order to ensure momentum conservation. It turns out that the property of "neighborhood" is not exactly reflexive, i.e., particle j can be a neighbor to particle i, but particle i is not a neighbor to particle j, since the two particles have different smoothing lengths. Because of this, we implement a nice trick when computing the density according to the the formula above. If particle j is a neighbor to particle i, we calculate the first half of W_{ij}, i.e. the term involving h_i, and add it to the sum for both particle i and j. After all, this value is just the second half of W_{ij} for particle j. If particle i is in turn a neighbor to particle j, we pick up the second half of the density expression involving h_j later. Although this may sound complicated, we have tested it numerous times, and it does in the end produce the right answer. If you will, you can think of it as a way to enforce the balances demanded by Newton's laws on a numerical level. In the comments to the code, this method is referred to as a "gather-scatter" technique.

3.1.3. Parallelization of the Hydro Routines

As this is a parallel code, we divide up this task among processors in a rather straightforward way. In everything which follows, "`NPROCS`" represents the number of processors used in the calculation. First, we break up the N particles into `NPROCS` subsets, each with approximately the same number of particles. As an example, for a simulation with 10,000 particles and four processors, we define the lower and upper limits "`n_lower`" and "`n_upper`" for each MPI process such that the rank 0 process has values of 1 and 2500, respectively, rank 1 has 2501 and 5000, etc. Each processor is responsible for the piece of the total particle summation defined by `n_lower` $\leq i \leq$ `n_upper`. Note that since there is no restriction on particle j, our "gather-scatter" method can contribute density contributions to particles within the processor's particle range as well as to particles outside of the range, defeating efforts to use HPF-style "`DO INDEPENDENT`" formulations. After the partial sums are calculated, we use a "`MPI_ALLREDUCE`" command to total up the partial sums and redistribute all of the values to all of the processors.

The routine we use to calculate the ranges of particles fow which each processor is responsible is "`GRAVQUANT`", which also calculates the sectioning of the 3-d gravity for input into our FFT algorithm, discussed below in Sec. 3.3.3. It is ABSOLUTELY necessary to call `GRAVQUANT` when starting the SPH code, without exceptions. See Sec. 4 on initial conditions for more on this topic.

3.1.4. Derivatives in SPH

For other hydrodynamic quantities q, defined for every particle q_i, can be calculated at arbitrary points in space by a corresponding equation

$$q_i = \sum_j m_j \frac{q_j}{\rho_j} W_{ij}, \tag{3-5}$$

using the gather-scatter method as described above. We find that essentially any extensive quantity can be calculated as a density wieghted sum. For intensive quantities, simply multiply by the density in the SPH sum, and then divide by the density. For instance, the SPH averaged velocity at a particle position can be given as

$$v_i = \frac{(\rho v)_i}{\rho_i} = \frac{1}{\rho_i}\sum_j m_j v_j W_{ij}. \tag{3-6}$$

Perhaps more importantly, the SPH method allows us to calculate not just hydro quantities, but their spatial derivatives as well, such as in the Newtonian force law $F_{hydro} = \nabla P/\rho$. We define the derivative of a hydro quantity "q" by using the chain rule, finding that

$$\nabla q_i = \sum_j m_j \frac{q_j}{\rho_j} \nabla W_{ij}, \tag{3-7}$$

where all terms in the expression, most importantly the derivative, are well-defined. As an example of this approach in action, the standard Newtonian fluid acceleration is calculated by the SPH code as

$$\begin{aligned} \vec{a}_{hydro} = \frac{1}{\rho}\nabla P &= \nabla(P/\rho) + \frac{P}{\rho^2}\nabla\rho \\ &= \sum_j m_j \frac{P_j}{\rho_j^2}\nabla W_{ij} + \frac{P_i}{\rho_i^2}\sum_j m_j \nabla W_{ij} \\ &= \sum_j m_j \left(\frac{P_j}{\rho_j^2} + \frac{P_i}{\rho_i^2}\right)\nabla W_{ij}. \end{aligned} \tag{3-8}$$

The code currently assumes a polytropic form for the equation of state, i.e. $P_i = A_i \rho_i^\Gamma$, where Γ is a constant, and A_i, our entropic variable, is constant unless an artificial viscosity prescription is used. For more on this, see Sec. 3.5.

3.1.5. Evolving Quantities in Time

The equations of motion for SPH particles are relatively simple, since particles have well-defined positions, velocities, and accelerations. We calculate a timestep which satisfies the Courant stability conditions, picking it to be sufficiently small compared to the typical

- Sound crossing time for a particle, h_i/c_s, where the speed of sound c_s is given by $c_s^2 \equiv (\partial P/\partial\rho)_A$
- Accelerative timescale for a particle, $\sqrt{(h_i/a_i)}$.

These steps help to ensure accuracy, but we do slightly better. To increase our numerical accuracy, we use a second-order "leapfrog" approach during each timestep. At the beginning of a run, we advance the velocities half a timestep. In each succeeding iteration, we first advance the particles' positions half a timestep using the velocity value already calculated for mid-timestep, calculate the rate of change of the entropy at the mid-timestep if AV is being used, and then advance the particles to the end of the step. We go back, advance the velocities half an iteration with the old values of the acceleration, recalculate the accelerations (which are weakly velocity-dependent), and advance the velocities a full time-step from the old values. Essentially, this method reproduces the second order Runge-Kutta approach, while consistently updating all the other things that make the code work. Please note that we adjust everything back to a consistent reference time when producing output files.

3.2. Neighbor Lists

The sums used in the SPH expressions above are dependent on each particle's kernel function, which in turn depends on each particle's smoothing length. How do we determine what it should be, and how it will evolve over time? Our method works by defining an optimal number of neighbors for each particle. This value is set by the user in the input files (see Sec. 5.1.1) by the parameter `NNOPT`. The kernel function of the particle, a sphere with radius equal to 2 smoothing lengths, ideally will overlap precisely this number of other particles. In practice, we use a relaxation technique, such that a every iteration of the code, the smoothing length is set equal to the arithmetic mean of the previous value and the power law estimate of the new value, thus

$$h_{new} = \frac{1}{2} h_{old} \times \left[1 + \left(\frac{N_{opt}}{N_N} \right)^{\frac{1}{3}} \right]. \tag{3-9}$$

In many cases, it is advisable to place limits on the minimum and maximum smoothing length allowed in the calculation. In practice, we have found that a minimum value is generally unnecessary, and setting a fixed floor could lead to particles in high-density regions reaching the maximum number of neighbors allowed in memory before all overlapping particles are reached. Setting a maximum is often important, since smoothing lengths grow dramatically when particles find themselves alone, and in this limit the SPH density kernel inherently does a poor job of describing the local density field. Typically, setting the maximum smoothing length to be comparable to the stellar radius is appropriate. The maximum and minimum smoothing lengths are set with the parameters "`HMAX`" and "`HMIN`", respectively. Setting `HMIN` < 0 tells the code there is no fixed minimum for the smoothing length.

To determine which particles neighbor each other, we use a linked list/head-of-cell method. First, we define both an integer vector with one entry for every particle and a 3-dimensional integer grid around the matter, whose size is set by the parameters `NCMX, NCMY`, and `NCMZ` in the file `nelist.f`. The physical extent represented by each grid cell in each direction is calculated by taking the average smoothing length for all particles, and multiplying by a factor of 1.3 (this just works faster). Once the grid dimensions are defined, we run a loop over all the particles in the simulation, calculating the grid cell in which each one lies. If the particle is the *first* one to fall in its grid cell, we enter its number into the grid. If there are already one or more particles in the grid cell, we copy the old particle's number to the vector in the new particle's slot, and enter the new particle's number in the grid cell. When we are done, the algorithm to find every particle in a given grid cell is simple: the number written in the grid cell, i_1, is the highest-numbered particle which lies in the cell. The i_1th element of the vector contains i_2, the next-highest numbered particle in the cell. By making our way through the vector, we recover all particles in the cell, until we reach an element in the vector containing zero, indicating that the cell has been accounted for completely.

Next, we go through the process of assembling the neighbor lists for each particle. For each particle, we loop over all grid cells which may lie within two smoothing lengths of the particle, and determine for every particle in the cell whether this is the case. For each particle meeting the criterion, we record its value in the array "`NNI`". Since all hydrodynamical interactions are parallelized as described in Sec. 3.1.3, we only record the neighbor lists spanning particles with `n_lower` $< i <$ `n_upper` on each processor. We also keep track of the number of neighbors each particle has, `NN`, but in a way such that every processor has access to the complete list for all particles.

3.2.1. Neighbor Lists: Parallel Issues and Particles Off the Grid

If the 3-d head-of-cell grid is not large enough in a particular dimension to contain all the particles, it centers the grid in that direction about zero, and prints out a warning of the form "`Warning!  NELIST: EXC true in ...`", listing the relevant dimension(s), as well as the number of grid cells which would have been required to fit all the particles and the current maximum.

In some cases, setting these grid dimensions to be symmetric around zero may be inappropriate for what you want to do, and you should change the algorithm accordingly.

The code can run perfectly well when particles fall outside the neighbor grids. If this happens, the code will produce a warning "`NELIST: WARNING !!!  NPOUT=`" listing the number of particles outside the grid. These particles will have no neighbors, and will not play a role in any hydrodynamic interactions. They will, however, be accounted for in gravitational interactions (see Sec. 3.3 for more on this).

3.3. The Gravity Solver: an FFT-based Approach

3.3.1. FFT Convolution

There are SPH techniques which can be used to solve for the gravitational force on each particle, but we have found it easiest to use FFT convolution to calculate the gravitational potential and force for each particle. Note that according to Newton's law, the gravitational potential Φ satisfies an equation

$$\Phi(\vec{r}) = \int \rho(\vec{r}') \frac{d^3\vec{r}'}{|\vec{r}-\vec{r}'|} = \sum_i \frac{m_i}{|\vec{r}-\vec{r}_i|}. \tag{3-10}$$

The direct sum requires N^2 operations, and is unfeasible to implement. The integral, on the other hand, is in the form of a convolution, and we find

$$\Phi(r) = \mathrm{FFT}^{-1} \times \left[\mathrm{FFT}(\rho) * \mathrm{FFT}\left(\frac{1}{r}\right)\right]. \tag{3-11}$$

To use FFT convolution techniques, we have to import the data to a 3-dimensional grid. Setting the boundaries for this grid is a topic worthy of some explanation, and can have significant effects on both the performance and accuracy of the code. Essentially, if the grid covers too large a spatial volume, the matter becomes concentrated in fewer cells, and we lose the sharpness of our solution for the gravitational force and potential. If the grid is too small, though, particles can be lost off the edges, which in some cases can be a problem. Resizing the grid allows for flexibility, but requires doing a third FFT (that of $1/r$) every timestep in addition to that of the matter and the inverse FFT of the potential, adding up to 50% to the time required to compute the gravitational field. If the grid spacing is known to be constant, we can calculate the FFT of $1/r$, save the result, and reuse it over and over again. In all cases, the dimensions of the grid are fixed in advance, by the parameter "`NGRAV`" in the header file, representing the size of the grid devoted to the matter. The actual size of the grid is $\texttt{NNGRAV}^3 \equiv (2 \times \texttt{NGRAV})^3$, since convolution techniques require double the size in every dimension to properly calculate the potential, a process known as "zero-padding". If you will, the convolution has to account for the fact that one particle can be located across the grid all the way to the right, or all the way to the left, so the range of values of $\vec{x}_i - \vec{x}_j$ spans a length twice that of the size of the grid. It will come as no surprise that the FFT works dramatically better if `NGRAV` is chosen to be a power of two, or at least contains nothing but small prime factors. The FFTW can handle multiples of 5 as well as other small primes, but at a huge loss of computational speed.

When calculating the density at each grid vertex, we use a "cloud-in-cell" method, whereby the mass contributions from any given particle are attributed to the eight vertices surrounding by means of a weighting function which contains the product of three terms, each of which are linear in a dimension, with the total weighting normalized to unity. If a particle is respectively $25\%, 40\%$, and 70% of the way across the cell from the lower front left corner in the three dimensions, we attribute $0.75 \times 0.6 \times 0.3$ of the particle's mass to that corner.

After calculating the gravitational potential at each vertex, we finite difference to find the gravitational force at each vertex. The potential and gravitational force are interpolated for each particle using the "cloud-in-cell" technique described above, with the same weightings used on the way out as were used on the way in.

Particles not on the grid are flagged, and a warning will be printed saying "`QGRAV: NPOUT=... AMIN=`", where `NPOUT` is the number of particles off the grid, and `AMIN` is the total mass of particles located within the grid. For particles off the grid, gravitational terms are calculated between them and particles on the grid (but not among particles off the grid) via a simple monopole approximation.

3.3.2. Built-in Gravity Grid Options

We have several built-in options for the code, to cover a variety of situations. They are defined in the input files by the parameter "`NGR`", as well as the parameters "`XGRMAX, XGRMIN, XGRLIM,` etc." They work as follows:

- `NGR` $= 0$: No self-gravity at all
- `NGR` $= 1$: Fixed gravity grid, running from `XGRMIN` $< x <$ `XGRMAX`, `YGRMIN` $< y <$ `YGRMAX`, and `ZGRMIN` $< z <$ `ZGRMAX`.
- `NGR` $= 2$: The gravity grid will be adjusted every timestep to cover *all* of the particles, no matter how far from the origin or each other.
- `NGR` $= 21$: Initially like `NGR` $= 2$, but after a certain time "`TGR`" has passed, switch to `NGR` $= 1$, with $-$`XGRLIM` $< x <$ `XGRLIM`, $-$`YGRLIM` $< y <$ `YGRLIM`, and $-$`ZGRLIM` $< z <$ `ZGRLIM`.
- $90 \leq$ `NGR` ≤ 99: Like `NGR` $= 1$ at first, with $-$`XGRLIM` $< x <$ `XGRLIM`, $-$`YGRLIM` $< y <$ `YGRLIM`, and $-$`ZGRLIM` $< z <$ `ZGRLIM`. Sets the grid boundaries to contain `NGR`% of the total mass of the matter in a given dimension once a particle leaves the range $-$`XGRLIM` $< x <$ `XGRLIM`, etc.
- $290 \leq$ `NGR` ≤ 299: Like `NGR` $= 2$ until *any* particle leaves the box defined by $-$`XGRLIM` $< x <$ `XGRLIM`, $-$`YGRLIM` $< y <$ `YGRLIM`, and $-$`ZGRLIM` $< z <$ `ZGRLIM`. From that point onwards, `NGR` is changed to a new value (`NGR` $- 200$).
- `NGR` $= 398$: Like `NGR` $= 98$, but the grid cannot expand indefinitely. The boundaries are constrained such that `XGRMIN` $\geq -2 \times$ `XGRLIM`, `XGRMAX` $\leq 2 \times$ `XGRLIM`, etc.

3.3.3. Gravity Grid Parallelization

Whereas we tend to divide up our list of particles among processors to parallelize hydro routines, we are forced to divide up our gravity grids among processors spatially. For its parallel FFT routines, the FFTW requires that all grids be divided equally along the first dimension of the grid, but it is VITAL to note that the FFTW is written in C. Since C and Fortran use opposite notations for indexing vectors, C stepping along the last index and Fortran along the first, this means that in the Fortran part of the code we must break up the grids passed to the FFTW in the z-direction. Thus, all grids in the gravity routine have dimensions `NNGRAV`x`NNGRAV`x`NNP`, where `NNP` $=$ `NNGRAV/NPROCS`. This is straightforward when inputting values onto the grids, but harder when outputting values back to the particles. We keep careful track of the z-coordinate of vertices whose values contribute to the potential or force of a given particle, only summing those that lie on a given processor, before using an `MPI_ALLREDUCE` to gather the sums properly.

The sectioning of the 3-d grids sent to the parallel 3-d FFT algorithm is computed in the subroutine "`GRAVQUANT`", which is invoked from all of the initialization subroutines. See Sec. 4 on initial conditions for more on the topic.

3.3.4. The FFTW

While the FFTW webpage contains extensive documentation on how it works and why it is so flexible, we feel it is appropriate to comment on the way in which we implement it.

The FFTW, like many numerical FFT codes, requires two different types of calls. First, the parameters are passed to it so it can properly allocate memory. Second, the actual FFT is computed.

During the first iteration of the code, we invoke the subroutine "`makeplans`", which proceeds to call out all the plans we will need later. Since these are stored as pointers in the C-language part of the code, they are treated by Fortran as if they are integers. Attempting to display the plans in the Fortran code will yield you nothing more than a memory address.

Next, the Fortran code calls out the routines "`rcfft`" for the real-to-complex, 3-d forward FFTs, and "`crfft`" for the complex-to-real, 3-D inverse FFTs. These routines, written by us based on templates found within the FFT release, actually do several important steps.

For the forward FFTs, we first run a 2-d real-to-complex transform on the dimensions which are stored completely within each processor. This step requires no communication whatsoever between processors. Next, we call on an MPI-based transpose routine found within the FFTW code, though not within the libraries. This transpose switches two of the dimensions of the array, so the untransformed dimension now lies within a processor, rather than across all of them. Finally, we run a 1-dimensional complex-to-complex transform on this last dimension. The resultant complex data structure is left in the transposed form, since the transpose is undone by the inverse FFT.

The inverse FFT works in essentially the opposite order. We inverse FFT the second dimension within each processor, transpose the grid back to the original form, and run a 2-d complex-to-real transform within each processor on the remaining dimensions.

It is extremely important to note that the authors of the FFTW have changed the format of the transpose routines between the previous releases and the current one. If you must use an older version of the FFTW (version 2.0.x), please replace our file "`fftw_f77.c`" with "`fftw_f77.c.old`". The only difference is in the calling sequence of variables, the calculation is performed in the same way for either case.

It is VITAL to get the subroutine name syntax right when calling a routine in C from a routine in Fortran. We have encountered problems in the past with any C-based subroutine which has an underscore character in its name (i.e. `make_plans` instead of `makeplans`). Consider yourself warned.

3.4. Relaxation

A common problem with SPH codes is the fact that the initial configuration of particles you lay down will be to some extent out of equilibrium. In some case, the resulting oscillations of the matter can lead to spurious results. As a result, we recommend that whenever it is possible, you use our routines to relax a material configuration before computing a dynamical run. We currently have two such routines, for both single-stars and corotating binaries. We discuss each in turn, but the idea behind each is essentially the same. For equilibrium situations where the matter is fixed in a given reference frame, you calculate forces by the standard techniques, but add a drag term to the accelerations, such that

$$\frac{d\vec{v}}{dt} = \vec{a} - \frac{\vec{v}}{t_{relax}}. \tag{3-12}$$

The relaxation time, t_{relax} is input to the code as a parameter `TRELAX`, and should correspond roughly with the dynamical timescale of the system, given by $t_d \equiv 1/\sqrt{G\rho} \approx \sqrt{R^3/GM}$, in order to ensure nearly critical damping.

In all cases, we turn off the dissipative effects of both artificial viscosity (Sec. 3.5) and radiation reaction (Sec. 3.6) during relaxation.

3.4.1. Single Stars

In many cases, the first thing to do with the SPH code is to calculate a single-star model, which can be used later either by itself or in a multi-body system. The SPH code contains two subroutines which create single stars. One uses equal-mass SPH particles, which are laid down Monte-Carlo style, based on a weighting function calculated for the density profile you choose. The other sets up a hexagonal close-packed array of particles, with constant grid spacing, but with particles of varying mass. The latter approach gives better resolution for the outer edge of each star, but can be vulnerable to spurious mass segregation within collision remnants, as heavier particles fall into the regions with the largest gravitational potential. For either case, we recommend you relax the model by setting the parameter `NRELAX`=1, yielding a damping equation exactly like the one above. Typically, you should calculate for several relaxation times before treating a configuration as relaxed.

To ensure that the configuration computed during relaxation will work properly during later calculations, we make sure to adjust the center of mass of the matter during each timestep so that it is located at the origin.

3.4.2. Binary Stars

The SPH code has routines to create binaries with both irrotational and corotating velocity profiles. For the former case, there is no fixed reference frame in which the material is stationary, and we have not yet implemented a way to relax the matter configuration exactly. For ways of calculating these configurations, see, e.g., Gourgoulhon et al. (2001); Taniguchi et al. (2001); Taniguchi & Gourgoulhon (2002), and references therein.

On the other hand, binary stars in corotation, also known as synchronized binaries, can be relaxed, since they are static in a frame which rotates with the same angular frequency as the matter. For this case, called by setting the parameter `NRELAX`=2, the code adds in a false centrifugal force, to handle the rotation of the frame, yielding a force law for quantities in the corotating frame

$$\frac{d\vec{v}}{dt} = \vec{a} + \Omega^2\vec{r} - \frac{\vec{v}}{t_{relax}}, \tag{3-13}$$

where we calculate the corotating frame's angular velocity by taking the average of the net forces on each component of the binary divided by the distance from the origin to the center of mass of each component. Note that this routine works for a given binary separation, which is given by the input parameter `SEP0`. After each timestep, we shift the positions of both stars very slightly to keep the center of mass separation at the proper value, with the system center of mass at the origin. It is possible to edit the code to produce a relaxed sequence with a smoothly varying separation, by defining a new separation variable and replacing references to `SEP0` in the relaxation subroutines. See the file `sepscan.f` for an example of this.

We have subroutines that can take either a relaxed single-star model and place it in a binary, or create both stars from scratch, with either equal-mass or equally-spaced particles, like before. The code can also handle binaries for which the stars have different masses. Note that in this case, if you wish to use models of relaxed single stars, you will have to calculate each separately.

3.4.3. Finishing Relaxation

It is possible to start a dynamical run after relaxation finishes without starting the code. The parameter "`TRELOFF`" gives the time at which relaxation finishes, and if this is less than the time at which the run itself finishes, "`TF`", the code will switch to a dynamical mode, turning on the effects of AV and radiation reaction if the user indicated they should be used.

For a single star configuration the process is extremely simple. All velocities are set to zero and the run is relaunched. For the corotating binary case, All velocities are set such that

$$\vec{v} = \vec{\Omega} \times \vec{r}, \tag{3-14}$$

where $\vec{\Omega}$ is taken to point in the z-direction. Note that this approach eliminates all spurious velocities which had yet to damp out.

3.5. Artificial Viscosity

As we mentioned before, so long as the evolution of matter is completely adiabatic, the entropic variable A_i, found from the polytropic equation of state $P_i = A_i\rho_i^\Gamma$, remains constant. For most physical systems, however, this is a poor assumption. Several different techniques have been developed to deal with this situation, by adding an "artificial viscosity", or "AV", which damps the velocity of converging matter flows while heating up the material (thus increasing A_i). Our code includes three such prescriptions, all of which are described in great detail in Lombardi et al. (1999). We recommend giving this article a thorough perusal before using the AV schemes.

No matter which AV scheme is used, the effect on various quantities is determined from the laws of thermodynamics. Essentially, for an SPH artificial viscosity term Π_{ij}, if the viscous acceleration of the matter is given by

$$\vec{a}_{AV} = \sum_j m_j \Pi_{ij} \nabla W_{ij}, \tag{3-15}$$

then the entropy equation will read

$$\frac{dA_i}{dt} = \frac{\Gamma - 1}{2\rho_i^{\Gamma-1}} \sum_j m_j \Pi_{ij} (\vec{v}_i - \vec{v}_j) \cdot \nabla W_{ij}. \tag{3-16}$$

3.5.1. Monaghan

The simplest form of the AV function is that of Monaghan (1989), which combines a linear bulk velocity with a quadratic von Neumann-Richtmeyer viscosity. It takes the form

$$\Pi_{ij} = \frac{-\alpha\mu_{ij}c_{ij} + \beta\mu_{ij}^2}{\rho_{ij}}, \tag{3-17}$$

where the speed of average sound speed is given by

$$c_{ij} \equiv \frac{c_i + c_j}{2}; \quad c_i \equiv \left(\frac{\partial P_i}{\partial \rho_i}\right)_{A_i}^{\frac{1}{2}} = \left(\frac{\Gamma P_i}{\rho_i}\right)^{\frac{1}{2}} = (\Gamma A_i \rho_i^{\Gamma-1})^{\frac{1}{2}}, \tag{3-18}$$

the average density is $\rho_{ij} \equiv (\rho_i + \rho_j)/2$, and μ_{ij} is a function which measures the velocity convergence, given by

$$\mu_{ij} = \begin{cases} \frac{(\vec{v}_i - \vec{v}_j)\cdot(\vec{r}_i - \vec{r}_j)}{h_{ij}(|\vec{r}_i - \vec{r}_j|^2/h_{ij}^2 + \eta^2)} & \text{if } (\vec{v}_i - \vec{v}_j)\cdot(\vec{r}_i - \vec{r}_j) < 0, \\ 0 & \text{if } (\vec{v}_i - \vec{v}_j)\cdot(\vec{r}_i - \vec{r}_j) \geq 0, \end{cases} \tag{3-19}$$

where $h_{ij} \equiv (h_i + h_j)/2$ is the average smoothing length. The parameters α, β, and η^2 are set in the input files as `ALPHA`, `BETA`, and `ETA2`. We recommend for most purposes $\alpha \approx 0.5$, $\beta \approx 1$, and $\eta^2 \approx 10^{-2}$, but see Lombardi et al. (1999) for more discussion.

3.5.2. Hernquist and Katz

Another form for the AV, introduced by Hernquist & Katz (1989) calculates Π_{ij} directly from the SPH estimate of the divergence of the velocity field:

$$\Pi_{ij} = \begin{cases} \frac{q_i}{\rho_i^2} + \frac{q_j}{\rho_j^2} & \text{if } (\vec{v}_i - \vec{v}_j) \cdot (\vec{r}_i - \vec{r}_j) < 0, \\ 0 & \text{if } (\vec{v}_i - \vec{v}_j) \cdot (\vec{r}_i - \vec{r}_j) \geq 0, \end{cases} \tag{3-20}$$

where

$$q_i = \begin{cases} \alpha \rho_i c_i h_i |\vec{\nabla} \cdot \vec{v}|_i + \beta \rho_i h_i^2 |\vec{\nabla} \cdot \vec{v}|_i^2 & \text{if } \left(\vec{\nabla} \cdot \vec{v}\right)_i < 0, \\ 0 & \text{if } \left(\vec{\nabla} \cdot \vec{v}\right)_i \geq 0, \end{cases} \tag{3-21}$$

and

$$(\vec{\nabla} \cdot \vec{v})_i = \frac{1}{\rho_i} \sum_j m_j (\vec{v}_j - \vec{v}_i) \cdot \vec{\nabla} W_{ij}. \tag{3-22}$$

We recommend, with the standard caveats, setting $\alpha \approx \beta \approx 0.5$.

3.5.3. Balsara

Finally, we include the AV form developed by Balsara (1995). He suggests

$$\Pi_{ij} = \left(\frac{p_i}{\rho_i^2} + \frac{p_j}{\rho_j^2} \right) \left(-\alpha \mu_{ij} + \beta \mu_{ij}^2 \right), \tag{3-23}$$

where

$$\mu_{ij} = \begin{cases} \frac{(\vec{v}_i - \vec{v}_j) \cdot (\vec{r}_i - \vec{r}_j)}{h_{ij} (|\vec{r}_i - \vec{r}_j|^2 / h_{ij}^2 + \eta^2)} \frac{f_i + f_j}{2 c_{ij}} & \text{if } (\vec{v}_i - \vec{v}_j) \cdot (\vec{r}_i - \vec{r}_j) < 0, \\ 0 & \text{if } (\vec{v}_i - \vec{v}_j) \cdot (\vec{r}_i - \vec{r}_j) \geq 0. \end{cases} \tag{3-24}$$

Here f_i is the form function for particle i, defined by

$$f_i = \frac{|\vec{\nabla} \cdot \vec{v}|_i}{|\vec{\nabla} \cdot \vec{v}|_i + |\vec{\nabla} \times \vec{v}|_i + \eta' c_i / h_i}, \tag{3-25}$$

where the factor $\eta' = 10^{-5}$ prevents numerical divergences, and

$$(\vec{\nabla} \times \vec{v})_i = \frac{1}{\rho_i} \sum_j m_j (\vec{v}_i - \vec{v}_j) \times \vec{\nabla} W_{ij}. \tag{3-26}$$

The function f_i acts as a switch, approaching unity in regions of strong compression ($|\vec{\nabla} \cdot \vec{v}|_i >> |\vec{\nabla} \times \vec{v}|_i$) and vanishing in regions of large vorticity ($|\vec{\nabla} \times \vec{v}|_i >> |\vec{\nabla} \cdot \vec{v}|_i$). Consequently, this AV has the advantage that it is suppressed in shear layers. In our code, we set $\eta' = 10^{-5}$, a choice which does not significantly affect our results. Note that since $(p_i/\rho_i^2 + p_j/\rho_j^2) \approx 2c_{ij}^2/(\Gamma \rho_{ij})$, Balsara's AV resembles Monaghan's AV when $|\vec{\nabla} \cdot \vec{v}|_i >> |\vec{\nabla} \times \vec{v}|_i$, provided one rescales Balsara's α and β in to be a factor of $\Gamma/2$ times the α and β in Monaghan's. We have taken this into account in the code, multiplying the AV term by a factor of $\Gamma/2$, so we recommend setting $\alpha \approx \beta \approx 1.0$ in the input files.

3.5.4. Parallelization of the AV Routines

The AV routines are parallelized in a similar way to the other hydro routines. Using the gather-scatter approach, we add half of the interaction between particles in turn to the AV force for each, breaking up the work by having each processor loop over a subset of the particles. Note that the gather-scatter technique used here guarantees that AV-related forces conserve both momentum and angular momentum up to machine precision.

An important fact to note about using an AV prescription is that it can greatly reduce the size of the timestep when a shock forms. In extreme cases, the timestep can become so small that the code effectively freezes. Be on the lookout for this, and, if necessary, edit the timestep routine until acceptable stepsizes result, but beware spurious results.

3.6. Gravitational Radiation Reaction

Our SPH code has been used extensively to study the properties of colliding neutron stars. In such systems, the general relativistic phenomenon of gravitational radiation losses can be very important. We include in our code a Newtonian-style treatment of radiation reaction, derived from the formalism of Blanchet et al. (1990), hereafter referred to as BDS, used by us (Faber & Rasio 2000; Faber et al. 2001; Faber & Rasio 2002) and other groups (Ruffert et al. 1996, 1997a,b; Ayal et al. 2001). It is consistent to lowest order, and more accurate than many slow-motion approaches, which are completely incapable of modeling the spatial dependence of the radiation reaction force.

To include radiation reaction, we make several changes to the code. The BDS form for the radiation reaction form is given by

$$\vec{F}_{reac} = m_i \vec{a}_{reac} = m_i \frac{1}{c^5} \nabla U_5, \tag{3-27}$$

where c is the speed of light in code units, and the relativistic potential U_5 is given by

$$U_5 = \frac{2}{5} G \left(R - Q_{ab}^{[3]} x^a \frac{\partial \Phi}{\partial x^b} \right), \tag{3-28}$$

where G is the gravitational constant, set equal to 1.0 in our code by definition. Taking the derivative of the radiation reaction potential requires a second derivative of the standard gravitational potential Φ, which we compute by finite differencing during a modified version of the gravity solver. Here, a and b are spatial indices to be summed over in a standard tensor product, i.e. they run from 1 to 3. The third time derivative of the traceless quadrupole moment, $Q_{ab}^{[3]}$, is given by taking the time derivative of the second derivative, which itself can be expressed, as shown in Rasio & Shapiro (1992), in SPH terms as

$$Q_{ab}^{[3]} = \frac{d}{dt} Q_{ab}^{[2]} = \frac{d}{dt} \mathrm{STF} \left[\sum_j 2 m_j (v_j^a v_j^b + x_j^a a_{grav}^b) \right]. \tag{3-29}$$

Note that this quantity is a scalar. "STF" indicates to symmetrize and subtract away the trace of the tensor. The gravitational acceleration on a particle is denoted a_{grav}. The quantity R which appears in the radiation reaction potential is a quantity whose value is given by solving a Poisson equation of the form

$$\nabla^2 R = -4\pi Q^{[3]}_{ab} x^a \frac{\partial \rho}{\partial x^b}. \tag{3-30}$$

To solve this equation numerically, we finite difference the density field on the grid directly to get the gradient of the density, rather than import directly to the grid the gradient which can be calculated using SPH techniques. We have found that this approach leads to a smoother and more realistic source term distribution

Besides the computation of the radiation reaction force, we also have to adjust very slightly our velocity equation. The specific momentum $\vec{v}$, representing the quantity whose derivative with respect to time is given by the hydrodynamic, AV, gravitational, and radiation reaction forces, is not the same quantity in the BDS formalism as the velocity $\vec{u} \equiv d\vec{x}/dt$, as it is in Newtonian physics. Instead, we find

$$u^a = v^a + \frac{4}{5c^5} Q^{[3]}_{ab} v^b. \tag{3-31}$$

Unfortunately, in order to keep our code simple, the notation we use does not match up with that used in the BDS paper. What we call "$\vec{u}$", they call "$\vec{v}$", and what we call "$\vec{v}$", they call "$\vec{w}$". Please be careful if you feel the need to edit these routines.

To include radiation reaction in a run, you must set the input parameter `NGRAVRAD`=1. Compute the speed of light in the code units, since it is often not unity, and use this to set the parameter "`SOL`".

4. Running a Calculation: Initial Routines

All calculation using the SPH code are run using the executable "`sphagr`", but you will need different input files to tell the code to do different things. We start with a description of routines used to start a new calculation from scratch. Note that this covers the case of using the output of a single-star run as an initial condition for a binary run. We discuss in Sec. 4.6 how to pick up a run exactly where it left off.

First, and most importantly, you must decide how many stars you want in the calculation. We have routines that can set up a single star, or two stars in an irrotational binary, a corotating binary, or a hyperbolic orbit. Once you know what you want, you can edit the "`sph.init`" and "`sph.input`" files accordingly, or better yet, run the "`testinput`" utility we have written. This program queries the user about all the option which go into the input files, creating all that need to be created.

Some input routines require other files as well, and we discuss this below. For more on file I/O, please see Sec. 5.

It is not difficult to write your own initialization subroutine, and we encourage you to do so if ours do not meet your needs. You have complete freedom to do so, but be aware you will need to include ALL of the following things:

- Read in the run parameters from `sph.input`, as well as any other input files you need.
- Set the positions, velocities, smoothing lengths, masses, and entropic variables of all the particles, these are the common block variables `X, Y, Z, VX, VY, VZ, AM, HP, A`.
- Set the number of particles `N` to the correct value, since it is needed for the next step.
- Call the subroutine `GRAVQUANT`, which sets the proper limits for parallel computation
- Call the subroutine `LFSTART`, which initializes the leapfrog algorithm (see Sec. 3.1.5).

4.1. Automatically Generated Input Files: testinput

To ease the process of starting a run, we have created a routine called `testinput` which automatically generates all the input files you will need for a run, based on your answers to a succession of simple, straightforward questions. We recommend you try it out a few times before trying to do all of the work yourself.

4.2. Single Stars

In theory, there are an infinite number of ways to set up a star in SPH. We offer built in routines to cover two important cases. First, a run where all SPH particles are given the same mass. In order to do this, we need to use a variable interparticle spacing to get the correct density profile. Alternately, we can use evenly-spaced particles of varying mass to reproduce a given density profile. It is easiest to do this by setting up a cubic lattice, but unfortunately, such configurations are unstable against deformations, and thus take a longer time to relax. Instead, we use a hexagonal close-packed lattice, the configuration found, among other places, in the carbon atoms in a diamond, since these lattices are absolutely stable.

For all single star models, we recommend you relax your configuration long enough for all the vibration in the lattice to damp away significantly. Typically, the stellar model you compute will end up slightly smaller than the radius you requested. This is to be expected, since the extent of the density distribution extends two smoothing lengths beyond the edge of the particles. As a rule of thumb, don't worry too much abut small deviations from what you expect in the very outer edges of a star, since SPH cannot interpolate the density profile at an edge exactly (although it still does a better job than grid based codes).

4.2.1. Equal-mass Particles

To construct a single star with equal-mass particles, you can use the "`setup1em`" subroutine, which is called by setting the three letter code in `sph.init` to "`1em`".

This routine first constructs a polytropic fit to the density profile of the star, with the following parameters supplied in `sph.input`: Stellar mass "`AMNS`", stellar radius "`RNS`", and adiabatic index $\Gamma =$"`GAM`". It then will lay down the exact number of particles you set by the parameter "`N`" in `sph.input`, by means of a Monte Carlo technique, weighted by the radial density profile. Since this routine is extremely fast, and need never be repeated during a run, we have not used any Monte Carlo tricks to decrease the rejection rate (if you don't know what this means, it will not hurt you). Typically, the configurations which result will be rather unrelaxed, with non-trivial density perturbations. To relax the model, we recommend setting "`NRELAX`"=1 and "`TRELAX`"$\approx \sqrt{\texttt{RNS}^3/\texttt{AMNS}}$ in `sph.input`, with the time until relaxation finishes, "`TRELOFF`", set to a much larger value.

4.2.2. Equally-spaced Particles

To construct a single star with equally-spaced particles, you can use the "`setup1es`" subroutine, which is called by setting the three letter code in `sph.init` to "`1es`".

This routine first constructs a polytropic fit to the density profile of the star, with the following parameters supplied in `sph.input`: Stellar mass "`AMNS`", stellar radius "`RNS`", and adiabatic index $\Gamma =$"`GAM`". It then will lay down a number of particles similar to the number you set by the parameter "`N`" in `sph.input`, in a hexagonal-close packed lattice. Since it is only feasible to estimate the proper lattice spacing, we have set the code to create a number of particles slightly smaller than the requested number, to ensure no memory overflows will occur. Typically, the configurations which result will be somewhat unrelaxed, with small density perturbations. To relax the model, we recommend setting "`NRELAX`"=1 and "`TRELAX`"$\approx \sqrt{\texttt{RNS}^3/\texttt{AMNS}}$ in `sph.input`, with the time until relaxation finishes, "`TRELOFF`", set to a much larger value.

4.3. Corotating Binaries

Since we can relax corotating binary configurations, it is not necessary to compute a relaxed single star model first. In some cases, however, such as a system near the point of Roche lobe overflow, it is not always a bad idea. To allow for maximum flexibility, we have built-in routines which create binary systems from scratch with initially equally spaced particles in each star of variable mass, from scratch with equal-mass particles in each star laid down Monte Carlo style, and from previously calculated single star models. In all cases, please consult the notes on single-star configurations for more information on these techniques.

The initial binary separation is set by the user through the parameter "`SEP0`". It is possible during relaxation to have the binary separation be a function of time, with best results found by keeping the distance fixed for some amount of time until a relaxed configuration is achieved, then scanning slowly inwards in a quasi-equilibrium way. If you wish to employ a method like this, please look at the file named "`sepscan.f`", set the parameters as you wish, and use it to replace the "`CMADJ`" subroutine in `advance.f`.

In addition to equal mass binaries, i.e. those with mass ratio $q \equiv M_2/M_1 = 1.0$, the code can create from scratch unequal mass binaries with $q < 1.0$. For such systems, the user is free to set the mass ratio of the system using the parameter "`QDAR`". The code will calculate the radius of the secondary star assuming the same equation of state (pressure-density relation) holds exactly for both stars. The power-law mass-radius relation is given by

$$\frac{R_2}{R_1} = \left(\frac{M_2}{M_1}\right)^{\frac{\Gamma-2}{3\Gamma-4}} = q^{\frac{\Gamma-2}{3\Gamma-4}}. \tag{4-1}$$

It is no problem for the code to calculate binary configurations which have the same adiabatic index Γ, but different values for the entropic variable A_i, but you will need to create each star in turn as a single star model, and run the routine "`setup2qr`", which creates a binary from previous models.

4.3.1. Mass Ratio q=1.0: Equal-mass Particles

To construct a $q = 1.0$ corotating binary configuration with equal-mass particles in both stars, you can use the "`setup2cm`" subroutine, which is called by setting the three letter code in `sph.init` to "`2cm`". The code lays down each star in an analogous manner to the equal-mass single star routines, and the same instructions about the relaxation timescale and shutoff time apply.

Note that the parameter "`N`" used to initialize the run refers to the number of particles in EACH star, the total number of particles will be `2N`.

4.3.2. Mass Ratio q=1.0: Equally-spaced Particles

To construct a $q = 1.0$ corotating binary configuration with equally-spaced particles in both stars, you can use the "`setup2cs`" subroutine, which is called by setting the three letter code in `sph.init` to "`2cs`". The code lays down each star in an analogous manner to the equally-spaced single star routines, and the same instructions about the relaxation timescale and shutoff time apply.

Note that the parameter "`N`" used to initialize the run refers to the approximate number of particles in EACH star, the total number of particles will be approximately `2N`.

4.3.3. Mass Ratio q=1.0: Previously Calculated Single Star Models

To construct a $q = 1.0$ corotating binary configuration based on a previously computed single star model, you can use the "`setup2cr`" subroutine, which is called by setting the three letter code in `sph.init` to "`2cr`".

The particle positions and other important parameters are read in from a file called "`pos.sph`", which you will need to create. We have supplied the code to make an executable called "`b2pos`", which will write this file for you. Based on responses to two questions, it can convert binary output files, in the form "`out012.sph`" (or any other 3-digit number) or "`restart.sph`" into the ASCII `pos.sph` file. For more on file I/O, please see Sec. 5.

Note that the parameter "`N`" used to initialize the run is essentially ignored, since the number of particles is set by the configuration read in from the files.

4.3.4. Mass Ratio q<1.0: Equal-mass Particles

To construct a corotating binary configuration for stars of unequal mass, but with equal-mass SPH particles in both stars, you can use the "`setup2qm`" subroutine, which is called by setting the three letter code in `sph.init` to "`2qm`". The code lays down each star in an analogous manner to the equal-mass single star routines, and the same instructions about the relaxation timescale and shutoff time apply. The mass ratio is determined by the parameter `QDAR` $= M_2/M_1$.

Note that the parameter "`N`" used to initialize the run refers to the number of particles in the primary star, which we typically assume to be the larger star, although no attempt is made to enforce this in the code. The number of particles in the secondary is given by $\texttt{N}_2 = \mathrm{INT}(\texttt{QDAR} \times \texttt{N})$. As always make sure the parameter "`NMAX`" in the `spha.h` header file is as large or larger than the total number of particles needed!

4.3.5. Mass Ratio q<1.0: Equally-spaced Particles

To construct a corotating binary configuration for stars of unequal mass, but with equally-spaced SPH particles in both stars, you can use the "`setup2qs`" subroutine, which is called by setting the three letter code in `sph.init` to "`2qs`". The code lays down each star in an analogous manner to the equally-spaced single star routines, and the same instructions about the relaxation timescale and shutoff time apply. The mass ratio is determined by the parameter `QDAR` $= M_2/M_1$.

Note that the parameter "`N`" used to initialize the run refers to the approximate number of particles in the primary star, which we typically assume to be the larger star, although no attempt is made to enforce this in the code. The number of particles in the secondary is given approximately by the ratio of the volumes of the two stars, since the code currently using the same grid spacing for BOTH stars. As always make sure the parameter "`NMAX`" in the `spha.h` header file is as large or larger than the total number of particles needed! There is nothing stopping you from changing the grid spacing of the second star, should you wish, and we are unaware of any negative effects this could cause.

4.3.6. Mass Ratio q<1.0: Previously Calculated Single Star Models

To construct a corotating binary configuration for stars of unequal mass based on previously computed single star models, you can use the "`setup2qr`" subroutine, which is called by setting the three letter code in `sph.init` to "`2qr`".

For such configurations, you will need to construct models of both stars separately. Note that we do not require the stars to have different masses, it is fine to run different models for stars with the same mass. It is also acceptable to use stars with different values of the entropic variable A_i, and this is actually required if you want to deviate from the polytropic mass-radius relation. It is REQUIRED, however, that the stars have the same value of the adiabatic index Γ, since this is treated by the code as a single scalar defined for all of the particles. For stars with different values of Γ, you will need to make it a vector within the common blocks, and carefully edit every occurrence of it in the code to reflect the change.

The particle positions and other important parameters are read in from files called "`pos.sph`" and "`pos2.sph`", which you will need to create. We have supplied the code to make executables called "`b2pos`" and "`b2pos2`", which will write these files for you. Based on responses to two questions, they will convert binary output files, in the form "`out012.sph`" (or any other 3-digit number) or "`restart.sph`" into the ASCII `pos.sph` and `pos2.sph` files, respectively. In fact, they only differ in the name of the file to which they write their data. For more on file I/O, please see Sec. 5.

4.4. Irrotational Binaries

For irrotational NS, i.e. those with no net vorticity in the inertial frame, there is no simple way to relax the matter configuration. Instead, we highly recommend that you first compute relaxed single-star models, and then place them in a binary. We allow you to deform the spherical stars into triaxial ellipsoids, by specifying the semi-major axes in each of the three principal directions. Values for these axis lengths can be found for a number of different EOS in series of papers by Lai et al. (1993b,c, 1994a,b,c) and a paper by Lombardi et al. (1997). We recommend perusing them thoroughly before continuing onward. Please note that by our definition, "irrotational" does not necessarily mean that every point on the star has the same velocity in the inertial frame. Rather, it refers only to a lack of vorticity. From Lombardi et al. (1997), we determine that for a triaxial ellipsoidal star whose center of mass is at some distance x_0 from the origin, traveling initially in the y-direction with orbital velocity Ωx_0, and with semi-major axes a_1 and a_2 in the x- and y-directions, a particle located at coordinates (x, y) with respect to the center of mass of the star has a velocity in the inertial frame given by

$$\begin{aligned} v_x &= \frac{2a_1^2}{a_1^2+a_2^2}\Omega y - \Omega y = \frac{a_1^2-a_2^2}{a_1^2+a_2^2}\Omega y, \\ v_y &= -\frac{2a_2^2}{a_1^2+a_2^2}\Omega x + \Omega(x_0+x) = \frac{a_2^2-a_1^2}{a_1^2+a_2^2}\Omega x + \Omega x_0. \end{aligned} \tag{4-2}$$

We note that for circular stars, this reduces to simple translationary motion. For elliptical stars, the surface shape rotates with the orbital velocity in such a way that the surface profile is invariant in the corotating frame. Since the code cannot perform a relaxation loop to solve for Ω, we estimate the proper binary angular velocity by calculating the net inward force on both components of the binary, analogously to the corotating case.

As in the case of corotating systems, the initial binary separation is set with the parameter `SEP0`, and the mass ratio is given by `QDAR`.

4.4.1. Equal-mass Binaries

To construct a $q = 1.0$ irrotational binary configuration, you can use the "`setup2i1`" subroutine, which is called by setting the three letter code in `sph.init` to "`2i1`".

The particle positions and other important parameters are read in from a file called "`pos.sph`", which you will need to create. We have supplied the code to make an executable called "`b2pos`", which will write this file for you. Based on responses to two questions, it can convert binary output files, in the form "`out012.sph`" (or any other 3-digit number) or "`restart.sph`" into the ASCII `pos.sph` file. For more on file I/O, please see Sec. 5.

Note that the parameter "`N`" used to initialize the run is essentially ignored, since the number of particles is set by the configuration read in from the files.

Additionally, the run will need you to create an input file called "`axes1.sph`". This file, in ASCII format, consists of three numbers representing the amount by which the spherical model is deformed in each of the three principal axes (x: the center of mass separation vector, y: the orbital velocity, z: the angular momentum axis). Typically, all three values should be ~ 1, and more often than not, $a_1 > a_2 > a_3$. This file will be created for you by the `testinput` executable discussed above.

4.4.2. Unequal-mass Binaries

To construct a irrotational binary configuration with unequal-mass stars, you can use the "`setup2iq`" subroutine, which is called by setting the three letter code in `sph.init` to "`2iq`".

The particle positions and other important parameters are read in from files called "`pos.sph`" and "`pos2.sph`", which you will need to create. We have supplied the code to make executables called "`b2pos`" and "b2pos2", which will write these files for you. Based on responses to two questions, it can convert binary output files, in the form "`out012.sph`" (or any other 3-digit number) or "`restart.sph`" into the ASCII `pos.sph` and `pos2.sph` files. For more on file I/O, please see Sec. 5.

Note that the parameter "`N`" used to initialize the run is essentially ignored, since the number of particles is set by the configuration read in from the files.

Additionally, the run will need you to create input files called "`axes1.sph`" and "`axes2.sph`". These files, in ASCII format, each consist of three numbers representing the amount by which the spherical model is deformed in each of the three principal axes (x: the center of mass separation vector, y: the orbital velocity, z: the angular momentum axis) for the first and second star. Typically, all three values should be ~ 1, and more often than not, $a_1 > a_2 > a_3$. The axis ratios should differ between the two stars, especially when the mass ratio diverges from unity. These files will be created for you by the `testinput` executable discussed above.

4.5. Hyperbolic Collisions

In addition to circular binaries, we have written routines which set up two stars on a hyperbolic trajectory. As was the case with irrotational binaries, there is no way to perform relaxation, so you will need to create relaxed single-star models first. If the binaries is started from a sufficiently large separation, tidal deformations are expected to be small, and spherical initial conditions are perfectly acceptable.

For stars with mass M_1 and M_2, we define the total and reduced masses by $M_T \equiv M_1+M_2$ and $\mu = M_1 M_2/M_T$. For a system with fixed masses, there are two free parameters which describe the initial orbit. We choose to define out initial conditions via the periastron distance r_p and relative velocity at infinity $v_\infty \equiv |\vec{v}_1 - \vec{v}_2|$, set by the parameters “`RP`” and “`VPEAK`”, respectively. The initial binary separation r_0 is given as usual by `SEP0`, and determines where on the orbital path we start the calculation. For such a system, we know the conserved total energy must be

$$E_T = \frac{1}{2}\mu v_\infty^2, \tag{4-3}$$

and by energy conservation the initial relative velocity is

$$v_0 = \sqrt{v_\infty^2 + \frac{2GM_T}{r_0}}, \tag{4-4}$$

and the velocity at periastron is

$$v_p = \sqrt{v_\infty^2 + \frac{2GM_T}{r_p}}. \tag{4-5}$$

From the latter expression, we find that the conserved system angular momentum is given by

$$J_T = \mu v_p r_p = \mu\sqrt{r_p^2 v_\infty^2 + 2GM_T r_p}, \tag{4-6}$$

which yields immediately the impact parameter b

$$b = \frac{J_T}{\mu v_\infty} = \sqrt{r_p^2 + \frac{2GM_T r_p}{v_\infty^2}}. \tag{4-7}$$

At our initial separation r_0, we know that the transverse velocity is given by

$$v_t = \frac{J_T}{\mu r_0} = \frac{\sqrt{r_p^2 v_\infty^2 + 2GM_T r_p}}{r_0} = \frac{v_\infty b}{r_0}, \tag{4-8}$$

and the radial velocity by $v_r = \sqrt{v_0^2 - v_t^2}$.

In order to use the most precise gravity grid possible, i.e., one that covers the smallest physical volume possible, we start both stars in opposite corners of the grid, rather than separated along the x-axis. The transverse velocity is projected perpendicular to the separation vector in such a way that $v_x = v_y$. Technically, it is possible to have an initial velocity which could take the stars off the gravity cubic gravity grid, but only if $v_t/v_0 > \sqrt{1/3}$. If you find this is the case, you may want to start the stars further apart!

Please note that the gravitational force calculated for particles can have a weak dependence on the physical size represented by a grid cell. Since the grid will need to be large in these calculations, you may want to make it larger than normal when constructing single star models.

4.5.1. Equal-mass Hyperbolic Binaries

To construct a $q = 1.0$ hyperbolic binary configuration, you can use the "`setuphyp1`" subroutine, which is called by setting the three letter code in `sph.init` to "`hy1`".

The particle positions and other important parameters are read in from a file called "`pos.sph`", which you will need to create. We have supplied the code to make an executable called "`b2pos`", which will write this file for you. Based on responses to two questions, it can convert binary output files, in the form "`out012.sph`" (or any other 3-digit number) or "`restart.sph`" into the ASCII `pos.sph` file. For more on file I/O, please see Sec. 5.

Given the possible complexities involved in setting the initial parameters, we strongly recommend using `testinput` to help set the gravity grid limits.

4.5.2. Unequal-mass Hyperbolic Binaries

To construct a hyperbolic binary configuration with unequal-mass stars, you can use the "`setuphypq`" subroutine, which is called by setting the three letter code in `sph.init` to "`hyq`".

The particle positions and other important parameters are read in from files called "`pos.sph`" and "`pos2.sph`", which you will need to create. We have supplied the code to make executables called "`b2pos`" and "b2pos2", which will write these files for you. Based on responses to two questions, it can convert binary output files, in the form "`out012.sph`" (or any other 3-digit number) or "`restart.sph`" into the ASCII `pos.sph` and `pos2.sph` files. For more on file I/O, please see Sec. 5.

Given the possible complexities involved in setting the initial parameters, we strongly recommend using `testinput` to help set the gravity grid limits.

4.6. Restarting From a Previous Run

If you are restarting directly from where a previous run stopped, the method is slightly different. Note that these instructions do not apply for taking a single-star configuration and using it as the basis for the components of a binary, see the sections above for more on this. To restart a run, take the required output file, in the form "`restart.sph`" or "`outxxx.sph`', where "xxx" is a 3-digit integer, as described in Sec. 5.2.1, and rename the file "`restart.sph`". You will not need a file named "`sph.init`", as it will be ignored by the program if `restart.sph` is found. You will need a new version of `sph.input` as well. Most likely, you will want to adjust the parameter `TF`, the time at which the run ends, to increase the value. Other parameters can be changed, but be careful. We recommend you use the `testinput` routine the first time you try this to see how it works.

5. File Input/Output

As could be expected for a code of some complexity, there are several files required to start a run, and several files written during the run. We summarize here the purpose and content of each.

5.1. Input Files

Some input files are required for the code to work properly, some are optional. Please check carefully to make sure you have all the files you need.

5.1.1. Key System Parameters: sph.input

The file `sph.input` is used to enter in most of the key run parameters into the code. It is read in during all of the initialization subroutines, as well as when a run is being restarted. For your convenience, the `testinput` utility can create this file for you, based on your responses to its questions.

The `sph.input` file is stored in ASCII format, such that it can be read into the program in the form of the "`INPUT`" namelist, defined at the end `spha.h`. It includes the following variables:

TF: The time at which the run will end, in code units. New runs start from $t = 0$, restarted runs start from the time reached when the `restart.sph` file was written in the previous run.

DTOUT: The time interval until a new permanent checkpointing file named `outxxx.sph` is written (see Sec. 5.2.1), where "xxx" is a 3-digit integer set by the code parameter `NOUT`.

GAM: The adiabatic index Γ, used in the equation of state $P_i = A_i \rho_i^\Gamma$.

N: Typically the number of particles to be used per star during the run, but the usage varies depending on which subroutine is called. This parameter is basically ignored if you are restarting a run. See the initialization subroutines for more on the meaning of this parameter for a given run.

NNOPT: The optimal number of neighbors for each particle, which should generally be a factor of two less than the maximum allowed number, set by the parameter `NNMAX` in `spha.h`. We have found good results for `NNOPT`$= 100$ for $N = 10^5$ particles, letting the optimal number of neighbors increase or decrease by a factor of two for each order of magnitude change in the number of particles used. See Sec. 3.2 for more info.

NAV: Integer parameter used to set the choice of AV routine, or lack thereof. See Sec. 3.5 for more info.

ALPHA,BETA,ETA2: Parameters used by the AV schemes. Proper choices for each are discussed in Sec. 3.5. These can all be set to zero if AV is off.

NGR: Sets the gravity grid boundaries to be used by the run, see Sec. 3.3.2 for the complete list.

XGRMIN,XGRMAX,YGRMIN,YGRMAX,ZGRMIN,ZGRMAX: Usually, these are the minimum and maximum x, y, and z-values used for the gravity grid, but usage varies depending on the choice of `NGR`.

XGRLIM,YGRLIM,ZGRLIM: For some choices of `NGR`, the extent to which the gravity grid is allowed to expand, but see Sec. 3.3.2 for the exact usage.

HMIN,HMAX: The minimum and maximum allowed smoothing lengths for any particle. If `HMIN`< 0, there is no minimum. This is actually the implementation we usually prefer.

NRELAX: Sets the choice of the relaxation scheme being used. `NRELAX`$= 0$ means no relaxation, 1 means single-star relaxation, and 2 is used for corotating binaries. See Sec. 3.4 for more info.

TRELAX: The relaxation timescale used in Eqs. 3-12 and 3-13.

SEP0: The initial binary separation for corotating and irrotational binaries, as well as for binaries on hyperbolic orbits.

QDAR: The binary mass ratio, $q = M_2/M_1$, typically defined such that $q < 1.0$.

AMNS,RNS: Generally, the mass and radius of a single star or the primary in the binary, in code units.

RP,VPEAK: The perihelion separation and relative velocity at infinite separation for binaries on hyperbolic orbits. See Sec. 4.5 for more info.

TRELOFF: The time at which relaxation turns off, for runs which go into a dynamical phase after a relaxation phase. For runs where relaxation should last for the entire run, set `TRELOFF`>`TF`.

NGRAVRAD: Integer parameter, used to determine if radiation reaction, as defined by the BDS formalism, will be included (`NGRAVRAD`=1) or not. See Sec. 3.6 for more info.

SOL: For runs with radiation reaction in use, the speed of light in code units, not necessarily equal to unity.

5.1.2. Starting New Runs: sph.init

The file `sph.init` is used to choose which initialization subroutine will be used for new runs. It is necessary unless a restart file named “`restart.sph`” is present, in which case it will be ignored.

For your convenience, the `testinput` utility can create this file for you, based on your responses to its questions. The file format is defined by the `INITT` namelist in `init.f`, and the file has the form

```
&INITT
INAME='2qs'
&END
```

where the complete list of options works as follows:

- `1em`: Single Star, equal-mass particles- subroutine `SETUP1EM`
- `1es`: Single Star, equal-mass particles- subroutine `SETUP1ES`
- `2cm`: Corotating binary, stars of equal mass, equal-mass particles- subroutine `SETUP2CM`
- `2cr`: Corotating binary, stars of equal mass, previously calculated stellar model- subroutine `SETUP2CR`
- `2cs`: Corotating binary, stars of equal mass, equally-spaced particles- subroutine `SETUP2CS`
- `2i1`: Irrotational binary, stars of equal mass- subroutine `SETUP2I1`
- `2iq`: Irrotational binary, stars of unequal mass- subroutine `SETUP2IQ`
- `2qm`: Corotating binary, stars of unequal mass, equal-mass particles- subroutine `SETUP2QM`
- `2qr`: Corotating binary, stars of unequal mass, previously calculated stellar models- subroutine `SETUP2QR`
- `2qs`: Corotating binary, stars of unequal mass, equally-spaced particles- subroutine `SETUP2QS`
- `hy1`: Binary on hyperbolic orbit, stars of equal mass- subroutine `SETUPHYP1`
- `hyq`: Binary on hyperbolic orbit, stars of unequal mass- subroutine `SETUPHYPQ`

5.1.3. Restarting From a Previous Run: restart.sph

A binary file named `restart.sph` is used to restart a calculation by picking up where a previous run had left off. Please note that using a previously calculated single star model as an initial model for a component of a binary DOES NOT count for this purpose.

There are two kinds of files which will work as a restart file. First are the `restart.sph` files created by the previous run. These files, produced by the `CHKPT` subroutine every `NITCH` iterations, are constantly rewritten to save memory while remaining up to date. More practical however, are the "`outxxx.sph`" files, where "xxx" is a 3-digit integer, produced by the `DUOUT` subroutine at code time intervals of `DTOUT`, and when a run terminates. To use these files, in the same exact format as the restart file above, simply rename the file "`restart.sph`".

5.1.4. Using a Single-star Configuration in a Binary: pos.sph

Since relaxation is not an option for irrotational or hyperbolic binaries, we require the use of a relaxed single-star configuration. These are optional as inputs for corotating binaries as well. These configurations are passed as ASCII files to the input routines from a file named `pos.sph`, and from a file named `pos2.sph` if the components of the binary are different. To create these files, take a restart file of either form mentioned in the previous section, and run the utility "b2pos" on it, producing a file named "`pos.sph`" (and "`b2pos2`" if necessary, it is the same executable, but produces a file named "`pos2.sph`"). The first line of the file contains the parameters `N, NNOPT, HMIN, HMAX, GAM`, and `TRELAX` from the previous run. Following that is a list, one line per particle, containing the position, mass, smoothing length, density, and entropic variable of each particle. The file closes with the number of particles repeated, to be used as a safety check.

5.1.5. Defining Triaxial Ellipsoids for Irrotational Binaries: axes1.sph and axes2.sph

When constructing an irrotational binary, the code takes a spherical single-star model and deforms it into a triaxial ellipsoid. To do so, the code reads from a file named `axes1.sph` an ASCII list of three integers, representing the deformation in the three principal directions (it reads from `axes2.sph` as well if you declare the binary contains stars of unequal-mass). This file will also be created by `testinput` for your convenience. See Sec. 4.4 for more info.

5.2. Output Files

Our code pipes more than a bit of information to stdout. It also can write several other files containing useful quantities. To avoid trouble with MPI, only the lowest rank process opens and writes to these files.

5.2.1. restart.sph and outxxx.sph

We have two different routines which write a binary file containing all the information necessary to restart the run at a later time. Subroutine `CHKPT` writes such a file every `NITCH` iterations, where the parameter is set in the `CHKPT` routine. This file, whose format is given in the `DUMP` subroutine, stores all the important scalar parameters for the run, and then stores all the particle-based quantities, except for the neighbor list information, which can be reconstructed from scratch during a new calculation. The `restart.sph` file is overwritten each time, to save on memory.

The code is also set to checkpoint every time `DTOUT` time units pass, writing a file named `outxxx.sph`, where "xxx" is set as a 3-digit integer equal to `NOUT`, which is iterated every time the file is written. These files are useful for recording and reconstructing the progress of a run later.

5.2.2. *energy.sph, biout.sph, gwdata.sph, com.sph, and spin.sph*

The code also writes to a number of ASCII output files every timestep.

The file `energy.sph` lists the current time, and then the total system potential energy, kinetic energy, internal energy, total energy, total entropy, total angular momentum, and the maximum SPH density for any particle. These quantities are defined by the sums:

$$P.E. = 0.5\sum_i m_i\Phi_i, \tag{5-1}$$

$$K.E. = 0.5\sum_i m_i v_i^2, \tag{5-2}$$

$$I.E. = \frac{1}{\Gamma-1}\sum_i A_i\rho_i^{\Gamma-1}, \tag{5-3}$$

$$E_T = P.E. + K.E. + I.E., \tag{5-4}$$

$$S_T = \frac{1}{\Gamma-1}\sum_i m_i \ln\left(\frac{A_i}{\Gamma-1}\right). \tag{5-5}$$

The file `biout.sph` lists the current time, the binary separation, the angle between the x-axis and the line connecting the stars' centers-of-mass, the distances for each star between the star's center-of-mass and the furthest particle from it, and the angles for each star between the center-of-mass and the furthest particle.

The file `gwdata.sph` contains the current time, and then the second time derivatives of the quadrupole moments. Note that we do not subtract out the trace of the quadrupole tensor beforehand. To calculate the gravitational waveform, take the appropriate linear combination of these terms. See Faber & Rasio (2000) for the way we have done this. We have generally found the first and last value for the quadrupole moments to be a bit inaccurate, especially if you wish to finite difference to find the gravitational wave luminosity, Feel free to discard the first and last lines of each file.

The file `com.sph` stores the current time, and then the position and velocity of the centers-of-mass of each star.

The file `spin.sph` stores the current time, and then the angular momentum of each star around its own center of mass. Note that we give the angular momentum in 5 directions, x, y, and z, as well as two directions rotated 45 degrees from the x and y directions in the x-y plane. These are appropriate to the case of hyperbolic binaries, for which the self-spin vectors lie primarily in the "plus" direction due to the way we construct the initial orbit. See Sec. 4.5 for more info about this.

All of these files can be safely commented out or edited to meet your needs. Note that they are opened upon the start of the code, and not closed until it terminates. Thus, data is only written to them every time the buffer size exceeds a system-dependent value, and a run which is stopped midway through will not contain all of the calculated values.

5.2.3. Subroutine densplot

As an example of what can be done to output interesting information from the code, we include a routine named `DENSPLOT` in `densplot.f`. It calculates the SPH density for a 3-d grid, whose size is set by parameters within the file. The results are saved by converting the data to an integer between 0 and 255, and converting this to a character, which is stored in binary format, taking up one byte per element. Results can be scaled against either the maximum density present on the grid, or a user-defined value. Output files are saved in the format "`densxxxx.dat`", where "xxxx" is a 4-digit integer equal to the parameter `NIT`, the iteration count.

6. Common Block Variables and User-defined Parameters

For your convenience, we list the variables contained within the common blocks, as well as the parameters set prior to runtime.

6.1. Common Block Parameters in spha.h

In the file `spha.h`, you must define the following parameters prior to compiling the code:

NPROCS: Set this equal to the number of processors on which you will tell the code to run. It is often best of this number as a power of two, since it affects the FFT.

NMAX: Set this number equal to or larger than the number of particles you will use. It is used to allocate the sizes of the vectors defined for particle quantities.

NMAXP: This is determined from the code by the equation $\texttt{NMAXP}=(\texttt{NMAX}-1)/\texttt{NPROCS}+1$. It represents the number of particles stored on each processor, and is used to set the size of the arrays containing the neighbor lists.

NNMAX: This is the maximum allowed number of neighbors. It is used to allocate the size of the neighbor arrays. Make sure t is $\sim 2\times$ greater than the optimal number of neighbors in `sph.input`.

NTAB: This is the size of the vector which stores the tabulated values for the smoothing kernel function.

CN: We multiply the minimum timestep by this factor, to ensure the stability of the evolution. We have found $\texttt{CN}=0.6$ to be appropriate.

PI: This one is obvious!

NGRAV: The size of the data portion of the gravity grid, in each dimension.

NNGRAV: Defined as $2\times\texttt{NGRAV}$, it is the actual size in each dimension of the gravity grid.

N1GRAV, NNP: Defined as $\texttt{NGRAV}+1$ and `NNGRAV/NPROCS`, respectively, these are needed as array sizes in the gravity routines.

6.2. Other Code Parameters

Other parameters are scattered around the code. They are:

TINY, UTINY, RHOTINY: These are used in subroutines `TSTEP`, `ADVANCE`, and `RHOS`, respectively, all of which are in `advance.f` to avoid dividing by zero.

NGRIDX, XMIN, XMAX, NGRIDZ, ZMIN, ZMAX: These are used in `densplot.f` to define the memory size and physical size of the grid used for writing the density values.

MAXRHO: This is the maximum density used to scale the density values in `densplot.f`.

NAMX: This is used by subroutines `QGRAV`, `QGRAV2`, and `QGRAVRAD` to estimate the amount of mass contained within a given range of values for grid settings which only include a certain percentage of the mass.

TGR: This is used by subroutines `QGRAV`, `QGRAV2`, and `QGRAVRAD` when you choose `NGR`=21. After time `TGR` has passed, the run will switch from `NGR`=2 to `NGR`=1. See Sec. 3.3.2 for more info.

NCMX, NCMY, NCMZ: These define the memory size of the grid used to calculate the neighbor lists in subroutine `NENE` in `nelist.f`.

NITCH: This defines the number or iterations between writing data to the file `restart.sph` in subroutine `CHKPT` in `output.f`.

TINY: In subroutine `ENOUT` in `output.f`, this is used to avoid dividing by zero.

XMAX, MAXSTP, PI, NRM, NMAX: These are used in `poly.f` to define parameters for the Runge-Kutta routine which solves for the polytropic density profile. Edit them at your own risk!

IA, IM, AM, IQ, IR, NTAB, NDIV, EPS, RNMX: These are used in `ran1.f` the random number generator. Leave them alone.

MAXTRY, NRGRID: These are used in the initialization routines to avoid infinite loops. They should never affect the code at all. If they come into play, there are bigger problems afoot.

6.3. Common Block Variables

The remaining common block variables in `spha.h`, to which almost all subroutines have access, are listed here:

NAV, ALPHA, BETA, ETA2: These are all described in Sec. 5.1.1, and set values for the AV routines.

NGR, XGRMIN, XGRMAX, YGRMIN, YGRMAX, ZGRMIN, ZGRMAX, XGRLIM, YGRLIM, ZGRLIM: These are all described in Sec. 5.1.1, and set values for the gravity routines.

NRELAX, TRELAX, TRELOFF: These are all described in Sec. 5.1.1, and set values for the relaxation routines.

OMEGA2: This is Ω^2, the square of the orbital velocity.

NLEFT: This is the number of particles in the primary. Loops which distinguish between the stars in a binary loop from $1 \leq i \leq$ `NLEFT`, and `NLEFT` $+ 1 \leq i \leq$ `N`.

SEP0, QDAR, AMNS, RNS, RP, VPEAK: These are all described in Sec. 5.1.1, and set parameters for the two stars.

N, NNOPT, GAM, HMIN, HMAX: These are all described in Sec. 5.1.1, and set some global parameters.

NOUT, NIT: These are respectively, the current count of output files written by subroutine `DUOUT`, and the number of iterations completed.

DT, T: These are the current timestep and the current time in code units.

VXS, VYS, VZS, XM2, YM2, ZM2: These are the velocities and positions of each particle, corrected to subtract off shifts in the system center-of-mass and the leapfrog timestep, used in the output routines.

TF, DTOUT: These are all described in Sec. 5.1.1, representing the time at which the run ends, and the time interval between writing output files.

WTAB, DWTAB, CTAB: The tabulated kernel function and its derivative, and a scalar representing the stepsize within the tabulated vector.

X, Y, Z, VX, VY, VZ, AM, HP, A, RHO, POR2, GRPOT: Vectors containing, for each particle, the position, velocity, mass, smoothing length, entropic variable, density, the quantity P/ρ^2, and the gravitational potential.

VXDOT, VYDOT, VZDOT, ADOT: The acceleration and rate of change of the entropic variable, for every particle.

GX, GY, GZ: The gravitational force for every particle.

UX, UY, UZ: Alternate velocity used when radiation reaction is included, see Sec. 3.6 and Eq. 3-31 for more info.

XIJ, YIJ, ZIJ: The vector difference in position between a particle and its neighbors, calculated once to save time, at the cost of memory usage.

NNI: An array of each particle's neighbors.

NN: A vector containing the number of neighbors each particle has.

MYRANK: The rank of each processor, returned by MPI_COMM_RANK.

N_LOWER, N_UPPER: The range of particles covered by each processor for parallel distributed computational loops.

KSTART, KOFFSET: Parameters used to section the 3-d gravity grids for parallel FFTs.

Q3XX, Q3XY, Q3XZ, Q3YY, Q3YZ, Q3ZZ: Third time-derivative of the quadrupole tensor, used for calculating radiation reaction.

Q2XX, Q2XY, Q2XZ, Q2YY, Q2YZ, Q2ZZ: Second time-derivative of the quadrupole tensor, used for calculating gravitational waveforms.

NGRAVRAD, SOL: Parameter used to determine whether radiation reaction will be used, and the speed of light in code units, see Sec. 5.1.1.

INITGR: Integer used to delay turning on radiation reaction for two iterations when dynamical run starts, since initial values are often inaccurate.

FREACX, FREACY, FREACZ: Radiation reaction force on each particle.

GXX, GXY, GXZ, GYY, GYZ, GZZ: Second derivatives of the gravitational potential, used in calculating the radiation reaction force.

DXPNR, DYPNR, DZPNR: Derivatives of one of the radiation reaction potentials.

7. Subroutines

For your convenience, we list the subroutines which comprise our code, with a brief description of each one.

Subroutine ADJUST: Changes the smoothing length `HP` of each particle to try to force each particle to maintain `NNOPT` neighbors, subject the rule `HPMIN` $\leq$ `HP` $\leq$ `HPMAX`. In practice, we change the smoothing length of each particle to the arithmetic average of the old value, and the power-law guess at the new value, given by Eq. 3-9 as:

$$h_{new} = \frac{1}{2} h_{old} \times \left[1 + \left(\frac{N_{opt}}{N_N} \right)^{\frac{1}{3}} \right]. \tag{7-1}$$

- See Sec. 3.2 for more details.
- In file `nelist.f`
- Calls no subroutines
- Called by subroutine `MAINIT`

Subroutine ADOTS: Selects which AV routine to use to calculate the entropy evolution equation, Eq. 3-16

$$\frac{dA_i}{dt} = \frac{\Gamma - 1}{2\rho_i^{\Gamma-1}} \sum_j m_j \Pi_{ij} (\vec{v}_i - \vec{v}_j) \cdot \nabla W_{ij}. \tag{7-2}$$

The choice of AV subroutine is set by the parameter `NAV`. Valid choices are:

- `NAV`=0: No AV is used.
- `NAV`=1: `BALADOTS`: Balsara's form, see Sec. 3.5.3
- `NAV`=2: `CLAADOTS`: Monaghan's form, see Sec. 3.5.1
- `NAV`=3: `NEWADOTS`: Hernquist and Katz's form, see Sec. 3.5.2

Other choices of `NAV` will produce an error.

- See Sec. 3.5 for more details.
- In file `dots.f`
- Calls subroutines `BALADOTS, CLAADOTS, NEWADOTS`
- Called by subroutine `ADVANCE`

Subroutine ADVANCE: Evolves all the hydro quantities each timestep. First, the positions are advanced half a timestep, and used to calculate the rate of change of the entropic variable for each particle (`ADOTS`), before being advanced to the end of the full timestep. Next, velocities are advanced half a timestep, accelerations are calculated (`VDOTS`), and then the velocities are advanced using the new acceleration a full timestep from their original values.

- See Sec. 3.1.5 for more details.
- In file `advance.f`
- Calls subroutines `NENE, RHOS, ADOTS, CMADJ, VDOTS`
- Called by subroutine `MAINIT`

Subroutine BALADOTS: Calculates the change in the entropic variable A_i when the Balsara AV prescription is used (Balsara 1995). See Eqs. 3-16, 3-23, 3-24, and 3-25. See Sec. 3.5.4 for details about parallelization.

- See Sec. 3.5.3 for more details.
- In file `balAV.f`
- Calls subroutines `CALCDIVV, GETCURLV`
- Called by subroutine `ADOTS`

Subroutine BALVDOTS: Calculates the AV acceleration term when the Balsara AV prescription is used (Balsara 1995). See Eqs. 3-15, 3-23, 3-24, and 3-25. See Sec. 3.5.4 for details about parallelization.

- See Sec. 3.5.3 for more details.
- In file `balAV.f`
- Calls subroutines `CALCDIVV, GETCURLV, DOQGRAV, QGRAV`
- Called by subroutine `VDOTS`

Subroutine BIOUT: Writes file "`biout.sph`" with binary-related quantities

- See Sec. 5.2.2 for more details.
- In file `output.f`
- Calls no subroutines
- Called by subroutine `OUTPUT`

Subroutine CALCDIVV: Calculates the SPH form of the divergence of the velocity at each particle position. This is given by Eq. 3-22 as

$$(\vec{\nabla} \cdot \vec{v})_i = \frac{1}{\rho_i} \sum_j m_j (\vec{v}_j - \vec{v}_i) \cdot \vec{\nabla} W_{ij}. \tag{7-3}$$

This routine is parallelized like other hydro loops, see Sec. 3.1.3.

- See Sec. 3.5.2 for more details.
- In file `calcdivv.f`
- Calls no subroutines

- Called by subroutines `BALADOTS, BALVDOTS, NEWADOTS, NEWVDOTS`

Subroutine CHECKPT: Writes a binary file called "`restart.sph`" every `NITCH` iterations.

- See Sec. 5.2.1 for more details.
- In file `output.f`
- Calls subroutine `DUMP`
- Called by subroutine `MAINIT`

Subroutine CLAADOTS: Calculates the change in the entropic variable A_i when the Monaghan AV prescription is used (Monaghan 1989). See Eqs. 3-16, 3-17, 3-18, and 3-19. See Sec. 3.5.4 for details about parallelization.

- See Sec. 3.5.1 for more details.
- In file `claAV.f`
- Calls no subroutines
- Called by subroutine `ADOTS`

Subroutine CLAVDOTS: Calculates the AV acceleration term when the Monaghan AV prescription is used (Monaghan 1989). See Eqs. 3-15, 3-17, 3-18, and 3-19. See Sec. 3.5.4 for details about parallelization.

- See Sec. 3.5.1 for more details.
- In file `claAV.f`
- Calls no subroutines
- Called by subroutine `VDOTS`

Subroutine CMADJ: Adjusts the system center of mass back to the origin when relaxation is on.

- See Sec. 3.4 for more details.
- In file `advance.f`
- Calls no subroutines
- Called by subroutine `ADVANCE`

Subroutine crfft: This C-language routine performs a 3-d Real $\rightarrow$ Complex inverse transform using the FFTW libraries.

- See Sec. 3.3.4 for more details.
- In file `fftw_f77.c`
- Calls the FFTW routines

- Called by subroutines `QGRAV, QGRAV2, QGRAVRAD`

Subroutine DENSPLOT: Generate a compact data file at every iteration for use in animations and other displays. The SPH density is calculated on a grid, and stored in binary form as an integer in the range 0-255, converted to a character.

- See Sec. 5.2.3 for more details.
- In file `densplot.f`
- Calls no subroutines
- Called by subroutine `OUTPUT`

Subroutine DOQGRAV: Calculates the gravitational force as well as the radiation reaction force on each particle, when radiation reaction is turned on by setting `NGRAVRAD`=1. Our scheme for radiation reaction is taken from Blanchet et al. (1990). See Eqs. 3-27, 3-28, 3-29, and 3-30. Note that the speed of light `SOL`, in code units, must be set for this routine to work.

- See Sec. 3.6 for more details.
- In file `doqgrav.f`
- Calls subroutines `QGRAV2, QGRAVRAD`
- Called by subroutines `BALVDOTS, CLAVDOTS, NEWVDOTS`

Subroutine DUMP: Writes to the logical unit passed an integer input variable (i.e., a previously opened file) the unformatted data which can be used to restart the run. MAKE SURE THE FORMAT AGREES WITH SUBROUTINE INIT, OR YOU WILL NOT BE ABLE TO RESTART RUNS!

- See Sec. 5.2.1 for more details.
- In file `output.f`
- Calls no subroutines
- Called by subroutines `CHECKPT, DUOUT`

Subroutine DUOUT: Writes a binary file of the form "`outxxx.sph`", where "xxx" represents a three-digit integer, every time an additional time interval of "`DTOUT`" passes.

- See Sec. 5.2.1 for more details.
- In file `output.f`
- Calls subroutine `DUMP`
- Called by subroutine `OUTPUT`

Fuction DW: Calculates the derivative of the SPH smoothing kernel function W, which itself is given by Eq. 3-2.

- See Sec. 3.1.2 for more details.
- In file `kernels.f`
- Calls no subroutines
- Called by subroutine `TABULINIT`

Subroutine ENOUT: Writes file "`energy.sph`" with various energies and other thermodynamic global quantities.

- See Sec. 5.2.2 for more details.
- In file `output.f`
- Calls no subroutines
- Called by subroutine `OUTPUT`

Subroutine GETCURLV: Calculates the SPH form of the curl of the velocity at each particle position. This is given by Eq. 3-26 as

$$(\vec{\nabla} \times \vec{v})_i = \frac{1}{\rho_i} \sum_j m_j (\vec{v}_i - \vec{v}_j) \times \vec{\nabla} W_{ij}. \tag{7-4}$$

This routine is parallelized like other hydro loops, see Sec. 3.1.3.

- See Sec. 3.5.3 for more details.
- In file `getcurlv.f`
- Calls no subroutines
- Called by subroutines `BALADOTS, BALVDOTS`

Subroutine GRAVQUANT: Calculates `n_lower` and `n_upper`, which determine the range of particles whose neighbor lists are stored on a specific processor, since `N`, the number of particles, is often set at runtime, and can differ from the user-specified value found in `sph.input`. It also calculates the grid ranges used in the gravity routines, and runs consistency checks on several other numbers, to ensure the runs will not crash. GRAVQUANT MUST BE CALLED AFTER N IS SET, AND BEFORE NENE IS RUN, OR THE CODE WILL CRASH AND DIE IN A HORRIFIC FASHION!

- See Sec. 3.1.3 for more details.
- In file `main.f`
- Calls no subroutines
- Called by subroutines `INIT, SETUP1EM, SETUP1ES, SETUP2CM, SETUP2CS, SETUP2CR, SETUP2I1, SETUP2IQ, SETUP2QM, SETUP2QS, SETUP2QR, SETUPHYP1, SETUPHYPQ`

Subroutine GWOUT: Writes file "`gwdata.sph`" with gravity wave-related quantities

- See Sec. 5.2.2 for more details.
- In file `output.f`
- Calls no subroutines
- Called by subroutine `OUTPUT`

Subroutine INIT: Initializes the run. If a file named `restart.sph` exists, it is used to continue a run which had been started previously. If not, it reads in a 3-character code from `sph.init` and launches the proper subroutine. See Sec. 5.1.2 for the complete list.

After the chosen subroutine finishes, it runs the neighbor finding routine and writes out the parameters for the run.

- See Sec. 4 for more details.
- In file `main.f`
- Calls subroutines `TABULINIT, GRAVQUANT, NENE, SETUP1EM, SETUP1ES, SETUP2CM, SETUP2CS, SETUP2CR, SETUP2I1, SETUP2IQ, SETUP2QM, SETUP2QS, SETUP2QR, SETUPHYP1, SETUPHYPQ`
- Called by program `MAIN`

Subroutine LFSTART: initializes the leapfrog timing algorithm.

- See Sec. 3.1.5 for more details.
- In file `init.f`
- Calls subroutines `GRAVQUANT, NENE, RHOS, VDOTS, TSTEP`
- Called by subroutines `SETUP1EM, SETUP1ES, SETUP2CM, SETUP2CS, SETUP2CR, SETUP2I1, SETUP2IQ, SETUP2QM, SETUP2QS, SETUP2QR, SETUPHYP1, SETUPHYPQ`

Program MAIN: Initializes MPI, initializes the hydro routines, opens up the output files, launches the iteration loop, and at the end, closes all files and shuts down MPI.

- In file `main.f`
- Calls subroutines `INIT, MAINIT`

Subroutine MAINIT: Runs a single iteration of the SPH code.

- In file `main.f`
- Calls subroutines `TSTEP, ADVANCE, ADJUST, CHECKPT, OUTPUT`
- Called by program `MAIN`

Subroutine makeplans: This C-language subroutine initializes the FFTW library FFT routines, and returns the memory addresses of structures containing plans for calculating the actual Fourier Transforms.

- See Sec. 3.3.4 for more details.

- In file `fftw_f77.c`
- Calls the FFTW routines
- Called by subroutines `QGRAV, QGRAV2, QGRAVRAD`

Subroutine NENE: Computes the nearest-neighbor list for each particle, and saves the information in an array which is distributed over the processors.

We compute the neighbor list using a head-of-cell/linked list method, which assigns all particles to a 3-d array, then loops over the neighboring cell to each particle to construct the neighbor list. All neighbors must lie within two smoothing lengths of the particle.

- See Sec. 3.2 for more details.
- In file `nelist.f`
- Calls no subroutines
- Called by subroutines `ADVANCE, INIT, LFSTART, OPTHP2, OPTHP3, SETUP2I1, SETUP2IQ`

Subroutine NEWADOTS: Calculates the change in the entropic variable A_i when the Hernquist and Katz AV prescription is used (Hernquist & Katz 1989). See Eqs. 3-16, 3-20, and 3-21. See Sec. 3.5.4 for details about parallelization.

- See Sec. 3.5.2 for more details.
- In file `newAV.f`
- Calls subroutine `CALCDIVV`
- Called by subroutine `ADOTS`

Subroutine NEWVDOTS: Calculates the AV acceleration term when the Hernquist and Katz AV prescription is used (Hernquist & Katz 1989). See Eqs. 3-15, 3-20, and 3-21. See Sec. 3.5.4 for details about parallelization.

- See Sec. 3.5.2 for more details.
- In file `newAV.f`
- Calls subroutines `CALCDIVV, DOQGRAV, QGRAV`
- Called by subroutine `VDOTS`

Subroutine OPTHP2: Calculates the approximately optimal particle smoothing length for runs in which equal-mass particles are laid down Monte Carlo style. The equation is given by

$$h_i = 0.9R\left(\frac{3}{32\pi}\frac{N_{opt}M}{N\rho_i}\right)^{\frac{1}{2}}. \tag{7-5}$$

where M and R are the star's mass and radius, ρ_i is the ideal density at that particle's position, and N and N_{opt} are the total number of particles in the star and the optimal number of neighbors, respectively.

- See Sec. 3.2 for more details.
- In file `setup1ns.f`
- Calls subroutines `POLY, NENE, ADJUST`
- Called by subroutines `SETUP1EM, SETUP2EM, SETUP2QM`

Subroutine OPTHP3: Calculates the approximately optimal particle smoothing length for the secondary star for unequal-mass stars with equal-mass particles (subroutine `SETUP2QM`), using Eq. 7-5, with R and M appropriate for the secondary.

- See Sec. 3.2 for more details.
- In file `setup2q.f`
- Calls subroutines `POLY, NENE, ADJUST`
- Called by subroutine `SETUP2QM`

Subroutine OUTPUT: Calls out all the subroutines which write data to files, having first synchronized the velocities to the same time values as the positions according to the leapfrog algorithm (see Sec. 3.1.5).

- See Sec. 5.2.2 for more details.
- In file `output.f`
- Calls subroutines `ENOUT, BIOUT, GWOUT, DENSPLOT, DUOUT`
- Called by subroutine `MAINIT`

Subroutine POLY: Calculates the value of the entropic variable A_i and the density profile for a spherical polytrope with specified index, mass and radius.

This is one of the very few subroutines which has values passed to it during the subroutine call. They are, in order, the adiabatic index $n \equiv 1/(\Gamma - 1)$, the stellar mass, the stellar radius, and the size of the vector containing the density profile. It returns the density profile as a vector, and the vale of the entropic variable for the star. Note that this routine does NOT include the `spha.h` header file, and thus has no access to common block variables. Note also that it calls subroutines `DERIVS, RKTAB, RK4`, which are the standard Numerical Recipes routine for doing a fourth order Runge-Kutta integration (Press et al. 1992).

- In file `poly.f`
- Calls Numerical Recipes subroutines
- Called by subroutines `SETUP1EM, SETUP1ES, SETUP2CM, SETUP2CS, SETUP2QM, SETUP2QS`

Subroutine QGRAV: Calculates the gravitational force on each particle by means of an FFT convolution algorithm. We use the MPI-parallelized routines of the FFTW software package, available free at `http://www.fftw.org`.

The Green's function for the transform requires a grid twice as large as the grid which holds the data, a method known as zero-padding the data. Essentially, we need to handle independently the effect of particles located at both higher and lower x, y, and z, and one can check that the range of possible values of $\vec{r_i} - \vec{r_j}$ is twice as large in every direction as the data itself.

Densities are uploaded to the grid by a cloud-in-cell algorithm, i.e. densities are assigned with weights to the 8 grid points around any particle. We then calculate the FFT of the data, convolve with the $1/r$ Green's function, and inverse FFT. As this is a parallel FFT, with 3-d arrays distributed along the z-direction, we need to take additional steps to finite difference in the z-direction when we are near the edge of each processor's data segment.

- See Sec. 3.3 for more details.
- In file `grav.f`
- Calls subroutines `makeplans, rcfft, crfft`
- Called by subroutines `BALVDOTS, CLAVDOTS, NEWVDOTS`

Subroutine QGRAV2: Calculates the gravitational force on each particle by means of an FFT convolution algorithm, as well as all second derivatives of the gravitational potential, which are needed for calculating the radiation reaction force. Other than that, this routine is an exact replica of subroutine `QGRAV`

- See Secs. 3.3 and 3.6 for more details.
- In file `grav2.f`
- Calls subroutines `makeplans, rcfft, crfft`
- Called by subroutine `DOQGRAV`

Subroutine QGRAVRAD: Calculates the radiation reaction potential given by Eq. 3-30, in the notation of the BDS radiation reaction formalism (Blanchet et al. 1990).

- See Sec. 3.6 for more details.
- In file `gravrad.f`
- Calls subroutines `makeplans, rcfft, crfft`
- Called by subroutine `DOQGRAV`

Function RAN1: Random number generator, returning a value $0 <$ `RAN1` < 1. Use IDUM=-2391 as a seed. Note that the generator is fully deterministic, and thus your runs are repeatable.

- In file `ran1.f`
- Called by subroutines `SETUP1EM, SETUP1ES, SETUP2CM, SETUP2CS, SETUP2QM, SETUP2QS`

Subroutine rcfft: This C-language routine performs a 3-d Complex → Real forward transform using the FFTW libraries.

- See Sec. 3.3.4 for more details.
- In file `fftw_f77.c`
- Calls the FFTW routines
- Called by subroutines `QGRAV, QGRAV2, QGRAVRAD`

Subroutine RELAX: Adds the relaxation drag terms to the equations of motion. For single stars, the drag force is given by

$$\frac{d\vec{v}}{dt} = \vec{a} - \frac{\vec{v}}{t_{relax}}, \tag{7-6}$$

where the user sets t_{relax}. For binary stars, the calculation is performed in the corotating frame, and the relaxation force takes the form

$$\frac{d\vec{v}}{dt} = \vec{a} + \Omega^2\vec{r} - \frac{\vec{v}}{t_{relax}}, \tag{7-7}$$

where Ω is calculated from the inward force on each star.

- See Sec. 3.4 for more details.
- In file `advance.f`
- Calls no subroutines
- Called by subroutine `ADVANCE`

Subroutine RHOS: Calculates the SPH expression for the particle density, given by

$$\rho_i = \sum_j m_j W_{ij}, \tag{7-8}$$

where the sum is taken over all the neighbors of a particle. We use a "gather-scatter" algorithm, i.e., for each particle-neighbor pair, we count half of the density contribution to the particle, and half to the neighbor. This routine uses MPI to distribute the computation.

- See Sec. 3.1.2 for more details.
- In file `advance.f`
- Calls no subroutines
- Called by subroutines `ADVANCE, LFSTART, SETUP2I1, SETUP2IQ`

Subroutine SETUP1EM: Constructs a single star with equal-mass particles, laid down Monte-Carlo style weighted by the density at a given radius according to a polytropic fit. Called by using the 3-letter code "`1em`" in `sph.init`.

- See Sec. 4.2.1 for more details.
- In file `setup1ns.f`
- Calls subroutines `POLY, GRAVQUANT, LFSTART, OPTHP2`

- Called by subroutine `INIT`

Subroutine SETUP1ES: Constructs a single star with equally-spaced particles of varying mass, which yield a density profile very near to a polytropic fit. Called by using the 3-letter code "`1es`" in `sph.init`.

- See Sec. 4.2.2 for more details.
- In file `setup1ns.f`
- Calls subroutines `POLY, GRAVQUANT, LFSTART`
- Called by subroutine `INIT`

Subroutine SETUP2CM: Constructs a corotating binary containing equal-mass stars, using equal-mass particles, laid down Monte-Carlo style weighted by the density at a given radius according to a polytropic fit. Called by using the 3-letter code "`2cm`" in `sph.init`.

- See Sec. 4.3.1 for more details.
- In file `setup2ns.f`
- Calls subroutines `POLY, GRAVQUANT, LFSTART, OPTHP2`
- Called by subroutine `INIT`

Subroutine SETUP2CR: Constructs a corotating binary containing equal-mass stars, using the output file from a single star run named "`pos.sph`". Called by using the 3-letter code "`2cr`" in `sph.init`.

- See Sec. 4.3.3 for more details.
- In file `setup2ns.f`
- Calls subroutines `GRAVQUANT, LFSTART`
- Called by subroutine `INIT`

Subroutine SETUP2CS: Constructs a corotating binary containing equal-mass stars, using equally-spaced particles of varying mass, which yield a density profile very near to a polytropic fit. Called by using the 3-letter code "`2cs`" in `sph.init`.

- See Sec. 4.3.2 for more details.
- In file `setup2ns.f`
- Calls subroutines `POLY, GRAVQUANT, LFSTART`
- Called by subroutine `INIT`

Subroutine SETUP2I1: Constructs an irrotational binary containing equal-mass stars, using the output file from a single star run named "`pos.sph`", and axis ratios specified by a field called "`axes1.sph`". Called by using the 3-letter code "`2i1`" in `sph.init`.

- See Sec. 4.4.1 for more details.
- In file `setupirr.f`
- Calls subroutines `NENE, ADJUST, RHOS, VDOTS, GRAVQUANT, LFSTART`
- Called by subroutine `INIT`

Subroutine SETUP2IQ: Constructs an irrotational binary containing unequal-mass stars, using the output filed from single star run named "`pos.sph`" and "`pos2.sph`", and axis ratios specified by a field called "`axes1.sph`" and "`axes2.sph`". Called by using the 3-letter code "`2iq`" in `sph.init`.

- See Sec. 4.4.2 for more details.
- In file `setupirr.f`
- Calls subroutines `NENE, ADJUST, RHOS, VDOTS, GRAVQUANT, LFSTART`
- Called by subroutine `INIT`

Subroutine SETUP2QM: Constructs a corotating binary containing unequal-mass stars, using equal-mass particles, laid down Monte-Carlo style weighted by the density at a given radius according to a polytropic fit. Called by using the 3-letter code "`2qm`" in `sph.init`.

- See Sec. 4.3.4 for more details.
- In file `setup2q.f`
- Calls subroutines `POLY, GRAVQUANT, LFSTART, OPTHP2, OPTHP3`
- Called by subroutine `INIT`

Subroutine SETUP2QR: Constructs a corotating binary containing unequal-mass stars, using the output files from single star runs named "`pos.sph`" and "`pos2.sph`". Called by using the 3-letter code "`2qr`" in `sph.init`.

- See Sec. 4.3.6 for more details.
- In file `setup2q.f`
- Calls subroutines `GRAVQUANT, LFSTART`
- Called by subroutine `INIT`

Subroutine SETUP2QS: Constructs a corotating binary containing unequal-mass stars, using equally-spaced particles of varying mass, which yield a density profile very near to a polytropic fit. Called by using the 3-letter code "`2qs`" in `sph.init`.

- See Sec. 4.3.5 for more details.
- In file `setup2q.f`
- Calls subroutines `POLY, GRAVQUANT, LFSTART`

- Called by subroutine `INIT`

Subroutine SETUPHYP1: Constructs a binary on a hyperbolic orbit containing equal-mass stars, using the output file from a single star run named "`pos.sph`". Called by using the 3-letter code "`hy1`" in `sph.init`.

- See Sec. 4.5.1 for more details.
- In file `setuphyper.f`
- Calls subroutine `LFSTART`
- Called by subroutine `INIT`

Subroutine SETUPHYPQ: Constructs a binary on a hyperbolic orbit containing unequal-mass stars, using the output files from single star runs named "`pos.sph`" and "`pos2.sph`". Called by using the 3-letter code "`hyq`" in `sph.init`.

- See Sec. 4.5.2 for more details.
- In file `setuphyper.f`
- Calls subroutine `LFSTART`
- Called by subroutine `INIT`

Subroutine TABULINIT: calculates the SPH smoothing kernel W, used to calculate all hydrodynamic quantities, and its derivative. The kernel function we use is given by

$$W(r,h) = \frac{1}{\pi h^3}\begin{cases} 1 - \frac{3}{2}\left(\frac{r}{h}\right)^2 + \frac{3}{4}\left(\frac{r}{h}\right)^3, & 0 \le \frac{r}{h} < 1, \\ \frac{1}{4}\left[2 - \left(\frac{r}{h}\right)\right]^3, & 1 \le \frac{r}{h} < 2, \\ 0, & \frac{r}{h} \ge 2. \end{cases} \tag{7-9}$$

- See Sec. 3.1.2 for more details.
- In file `kernels.f`
- Calls functions `W, DW`
- Called by subroutine `INIT`

Subroutine TSTEP: Calculates the time step used in the code.

- See Sec. 3.1.5 for more details.
- In file `advance.f`
- Calls no subroutines
- Called by subroutines `MAINIT, LFSTART`

Subroutine VDOTS: Selects which AV routine to use to calculate the all forces on the particle, including the AV acceleration, Eq. 3-15. The choice of AV subroutine is set by the parameter `NAV`. Valid choices are:

- `NAV`=0: `BALVDOTS`, but no AV acceleration term is used.
- `NAV`=1: `BALVDOTS`: Balsara's form, see Sec. 3.5.3
- `NAV`=2: `CLAVDOTS`: Monaghan's form, see Sec. 3.5.1
- `NAV`=3: `NEWVDOTS`: Hernquist and Katz's form, see Sec. 3.5.2

Other choices of `NAV` will produce an error.

- See Sec. 3.5 for more details.
- In file `dots.f`
- Calls subroutines `BALVDOTS, CLAVDOTS, NEWVDOTS`
- Called by subroutines `ADVANCE,LFSTART, SETUP2I1, SETUP2IQ`

Function W: Calculates the SPH smoothing kernel function W, which is given by Eq. 3-2.

- See Sec. 3.1.2 for more details.
- In file `kernels.f`
- Calls no subroutines
- Called by subroutine `TABULINIT`

8. References to Cite and Legal Information

Anyone using this code should say something to the effect: "This code was developed originally by Rasio and Shapiro to study merging binaries (Rasio & Shapiro 1992, 1994, 1995), and parallelized by Faber and Rasio for use in post-Newtonian calculations of coalescing neutron stars (Faber & Rasio 2000; Faber et al. 2001; Faber & Rasio 2002)." All six references should be considered mandatory.

Mention should always be made of the FFTW, please see their webpage for proper citation forms, at `http://www.fftw.org`. A short list of references should include the following: Frigo & Johnson (1997, 1998).

Those using our radiation reaction terms need to reference Blanchet et al. (1990). Those using artificial viscosity need to reference Lombardi et al. (1999), as well as whichever of Monaghan (1989); Hernquist & Katz (1989); Balsara (1995) is appropriate.

Also, please mention the name of the website from which you got the code, so that others may find it more easily. Thanks again.

8.1. Gnu General Public License

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it,

Terms and Conditions For Copying, Distribution and Modification

an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

    (a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

    (b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

    (c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

   These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

   Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

   In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

9. References

REFERENCES

Ayal, S., Piran, T., Oechslin, R., Davies, M. B., & Rosswog, S. 2001, Astrophys. J., 550, 846

Balsara, D. S. 1995, Journal of Computational Physics, 121, 357

Blanchet, L., Damour, T., & Schaefer, G. 1990, Mon. Not. R. Astron. Soc., 242, 289

Faber, J. A., & Rasio, F. A. 2000, Phys. Rev. D, 62, 064012

—. 2002, Phys. Rev. D, 65, 084042

Faber, J. A., Rasio, F. A., & Manor, J. B. 2001, Phys. Rev. D, 63, 044012

Frigo, M., & Johnson, S. G. 1997, The Fastest Fourier Transform in the West, Tech. Rep. MIT-LCS-TR-728, Massachusetts Institute of Technology

Frigo, M., & Johnson, S. G. 1998, in Proc. 1998 IEEE Intl. Conf. Acoustics Speech and Signal Processing, Vol. 3 (IEEE), 1381–1384

Gingold, R. A., & Monaghan, J. J. 1977, Mon. Not. R. Astron. Soc., 181, 375

Gourgoulhon, E., Grandclément, P., Taniguchi, K., Marck, J., & Bonazzola, S. 2001, Phys. Rev. D, 63, 064029

Hernquist, L., & Katz, N. 1989, Astrophys. J. Suppl., 70, 419

Lai, D., Rasio, F. A., & Shapiro, S. L. 1993a, Astrophys. J., 412, 593

—. 1993b, Astrophys. J. Suppl., 88, 205

—. 1993c, Astrophys. J. Lett., 406, L63

—. 1994a, Astrophys. J., 423, 344

—. 1994b, Astrophys. J., 420, 811

—. 1994c, Astrophys. J., 437, 742

Lombardi, J. C., Rasio, F. A., & Shapiro, S. L. 1995, Astrophys. J. Lett., 445, L117

—. 1996, Astrophys. J., 468, 797

—. 1997, Phys. Rev. D, 56, 3416

Lombardi, J. C., Sills, A., Rasio, F. A., & Shapiro, S. L. 1999, Journal of Computational Physics, 152, 687

Lombardi, J. C., Warren, J. S., Rasio, F. A., Sills, A., & Warren, A. R. 2002, Astrophys. J., 568, 939

Lucy, L. B. 1977, Astron. J., 82, 1013

Monaghan, J. J. 1989, Journal of Computational Physics, 82, 1

—. 1992, Ann. Rev. Astron. Astrophys., 30, 543

Press, W. H., Teukolsky, S. A., Vetterling, W. T., & Flannery, B. P. 1992, Numerical recipes in FORTRAN. The art of scientific computing (Cambridge: University Press, c. 1992, 2nd ed.)

Rasio, F. A., & Lombardi, J. C. 1999, J. of Comp. and Applied Math., 109, 213

Rasio, F. A., & Shapiro, S. L. 1991, Astrophys. J., 377, 559

—. 1992, Astrophys. J., 401, 226

—. 1994, Astrophys. J., 432, 242

—. 1995, Astrophys. J., 438, 887

Ruffert, M., Janka, H.-T., & Schaefer, G. 1996, Astron. & Astrophys., 311, 532

Ruffert, M., Janka, H.-T., Takahashi, K., & Schaefer, G. 1997a, Astron. & Astrophys., 319, 122

Ruffert, M., Rampp, M., & Janka, H.-T. 1997b, Astron. & Astrophys., 321, 991

Sills, A., Faber, J. A., Lombardi, J. C., Rasio, F. A., & Warren, A. R. 2001, Astrophys. J., 548, 323

Sills, A., Lombardi, J. C., Bailyn, C. D., Demarque, P., Rasio, F. A., & Shapiro, S. L. 1997, Astrophys. J., 487, 290

Taniguchi, K., & Gourgoulhon, E. 2002, Phys. Rev. D, 65, 044027

Taniguchi, K., Gourgoulhon, E., & Bonazzola, S. 2001, Phys. Rev. D, 64, 064012