1 N-body Methods

We will begin by exploring the gravitational N-body problems and algorithms that have been developed to solve it efficiently.

1.1 Specification of the Problem

Frequently in astrophysics we have systems of “particles” which interact only gravitationally. Examples include dark matter and purely stellar systems (in which each star is a particle). Ignoring relativistic effects (which we can in a wide variety of situations) the evolution of such systems is entirely specified by the Newtonian $1/r^2$ law of gravity, such that the acceleration of particle $i$ is given by:

$$\ddot{x}_i = \sum_{j=1;j\neq i}^{N} -\frac{Gm_j}{|x_{ij}|^2}x_{ij},$$  

(1)

where $x$ is position, $x_{ij} \equiv x_i - x_j$ and a hat indicates a unit vector.

1.1.1 How Big Must $N$ Be?

We typically want $N$, the number of particles, to be as large as possible given available computational resources. When simulating a system of fixed total mass, increasing $N$ will allow each particle to have a smaller mass and, therefore, will allow the simulation to resolve structure on smaller scales (and more closely approach the idealized fluid that we’re attempting to simulate). The number of particles used will also affect how long we can run our simulation for, before the discrete nature of our particle representation becomes a problem. Imagine an isolated, self-gravitating system of particles that is in virial equilibrium (such as a galaxy). In the limit of an infinite number of particles, the potential of this system will be unchanging in time and so each particle will conserve its total energy as it orbits through the system. However, if there are a finite number of particles, the potential will no longer remain constant in time, instead fluctuating as particles move around. This will allow for energy exchange between particles. If our simulation contains fewer particles than the real system (which is almost always will do) this energy exchange is unphysical. In a non-gravitating system this would eventually lead to equipartition and thermal equilibrium. Gravitating
systems, because of their negative specific heat, have no thermal equilibrium and this process will eventually lead to the phenomenon of “core collapse” (observed in globular clusters) in which the core of the system collapses to arbitrarily high densities while a diffuse envelope of material is ejected to large radii.

To estimate how long we can run a simulation before this relaxation process becomes important, we want an order-of-magnitude estimate how long it takes for encounters with other stars to significantly change the energy of a star. Typically we are interested in collisionless systems in which the acceleration of a given particle never receives a dominant contribution from a single other particle, instead being determined by the combined effects of many particles. Consider an encounter between two stars. Assuming the collision is a small perturbation to the motion of the star we can approximate the force experienced by the star as:

$$\dot{v}_\perp = \frac{Gm}{b^2 + x^2} \cos \theta = \frac{Gmb}{(b^2 + x^2)^{3/2}} \approx \frac{Gm}{b^2} \left[ 1 + \left( \frac{vt}{b} \right)^2 \right]^{-3/2}. \quad (2)$$

Integrating over the entire collision gives us the change in velocity of the star:

$$\mathbf{v}_\perp \approx \frac{Gm}{bv} \int_{-\infty}^{\infty} (1 + s^2)^{-3/2} ds = \frac{2Gm}{bv}, \quad (3)$$

which is approximately the force at closest approach times the duration of the encounter, $b/v$. The surface density of stars in a galaxy is of order $N/\pi R^2$, so in crossing the galaxy once, the star experiences

$$\delta n = \frac{N}{\pi R^2} 2\pi b db = \frac{2N}{R^2} b db, \quad (4)$$

encounters with impact parameter between $b$ and $b + db$. The encounters cause randomly oriented changes in velocity, so $\langle \delta \mathbf{v}_\perp \rangle = 0$, but there can be a net change in $v^2_\perp$:

$$\delta v^2_\perp \approx \left( \frac{2Gm}{bv} \right)^2 \frac{2N}{R^2} ddb. \quad (5)$$
Our perturbation approach breaks down if $\delta v_\perp \sim v_\perp$ which occurs is $b \lesssim b_{\text{min}} = G m/v^2$, so integrating over all impact parameters$^1$ from $b_{\text{min}}$ to $R$ (the largest possible impact parameter):

$$
\Delta v_\perp^2 = \int_{b_{\text{min}}}^{R} \delta v_\perp^2 \approx 8 N \left( \frac{G m}{R v^2} \right)^2 \ln \Lambda,
$$

(6)

where $\Lambda = R/b_{\text{min}}$ (“Coulomb logarithm”). The typical speed of a star in a self-gravitating galaxy is

$$
v^2 \approx \frac{G N m}{R}.
$$

(7)

Therefore, we find

$$
\frac{\Delta v_\perp^2}{v^2} = \frac{8 \ln \Lambda}{N},
$$

(8)

and the number of crossings required for order unity change in velocity is:

$$
n_{\text{relax}} = \frac{N}{8 \ln \Lambda}.
$$

(9)

Since $\Lambda = R/b_{\text{min}} \approx R v^2/G m \approx N$ we find $t_{\text{relax}} \approx [0.1 N/\ln N] t_{\text{cross}}$.

Relaxation is important for systems up to globular cluster scales, but is entirely negligible for galaxies.

<table>
<thead>
<tr>
<th>System</th>
<th>$N$</th>
<th>$t_{\text{relax}}/t_{\text{cross}}$</th>
<th>$t_{\text{relax}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Small stellar group</td>
<td>50</td>
<td>1.3</td>
<td></td>
</tr>
<tr>
<td>Globular cluster</td>
<td>$10^5$</td>
<td>870</td>
<td>$10^8$ yr</td>
</tr>
<tr>
<td>Galaxy</td>
<td>$10^{11}$</td>
<td>$4 \times 10^8$</td>
<td>$4 \times 10^7$ Gyr</td>
</tr>
</tbody>
</table>

It’s worth taking a moment to consider what we’re actually doing in an N-body calculation. Specifically, representing a fluid in 6-D phase space by a set of discrete points. We associate a mass with each of those points (usually the same mass for each point) such that we can use the points to compute the density distribution of the fluid and therefore its gravitational potential. Additionally, we move the points in phase space according to that same gravitational potential. As such, there are fundamentally two roles for the points: to represent the mass distribution at any time and to flow as would the actual fluid such that they continue to represent the mass distribution at later times. For most systems of interest (e.g. galaxies, dark matter halos) the number of particles we can use (given current computational techniques and resources) is much less than the number of particles in the real system. For example, a galaxy may contain $10^{11}$ stars, but the best current simulations of galaxies contain only $\sim 10^6$. Similarly, the best current simulations of dark matter halos contain around $10^9$ particles, but the Milky Way’s halo contains maybe $10^{10}$ dark matter particles. Therefore, the relaxation times in simulations will be much shorter than in the real systems (particularly in dense regions) unless we do something to mitigate this problem.

We can break down the N-body problem into three sub-problems:

1. Specifying initial conditions;
2. Computing forces/accelerations on particles;
3. Stepping particles forward in time.

$^1$For a more careful treatment of small $b$ encounters, take a look at the treatment of dynamical friction, §7.1 of Binney & Tremaine.
1.2 Force Calculation

The N-body problem is fundamentally an $O(N^2)$ problem—direct summation over all particles involves a number of calculations that increases as the square of the number of particles. Not surprisingly therefore, force computation is typically the slowest step in any N-body calculation and so numerous techniques have been devised to speed this up. We’ll review these techniques, as well as the direct summation approach (since it’s still useful in some cases).

1.2.1 Direct Summation (Particle-Particle)

The direct summation approach is inherently simple—as we wrote already, given some set of $N$ particles we evaluate the force on each one using

$$\ddot{x}_i = \sum_{j=1,j \neq i}^{N} -\frac{Gm_j}{|\mathbf{x}_{ij}|^2} \mathbf{x}_{ij},$$

You can write yourself a direct summation code in a few lines. If you have a newish desktop computer with a GPU (Graphics Processing Unit) you can write a direct summation code which can handle $\sim 10^4$ particles in a reasonable amount of time. Specialized hardware (GRAvity PipE or GRAPE boards) can handle several million—enough to simulate a globular cluster at one particle per star. The direct summation approach is obsolete for galactic and cosmological simulations.

In cases where we don’t have one particle per star/particle two-body encounters between particles will be much more common in our simulation than in the real system. This can lead to effects (for example, binary formation) which would not occur in the real system on relatively short timescales. To circumvent this problem we recognize that each particle is better thought of as representing an extended distribution of mass. It will therefore not have a point mass potential. Instead, its potential will be “softened”. We might therefore modify the force law to be

$$\ddot{x}_i = \sum_{j=1,j \neq i}^{N} -\frac{Gm_j}{(|\mathbf{x}_{ij}| + \epsilon)^2} \mathbf{x}_{ij},$$

where $\epsilon$ is a length scale (called the softening length) of order the “size” of the particle and which limits the maximum force between two particles. Note that this doesn’t do much to change the relaxation time—the Coulomb logarithm term in our derivation of the relaxation time shows that the relaxation process gains equal contributions from particles in each logarithmic interval of radius, and so softening the force on small scales only slightly reduces the rate of relaxation.

Different functional forms have been used for the softening. For example, a common approach has been to represent particles by so-called Plummer spheres—density distributions of the form $\rho(x) \propto (1 + x^2)^{-5/2}$ (where $x = r/\epsilon$) which have a potential $\Phi(x) \propto 1/\sqrt{\epsilon^2 + x^2}$, such that

$$\ddot{x}_i = \sum_{j=1,j \neq i}^{N} -\frac{Gm_j|x_{ij}|}{(|x_{ij}|^2 + \epsilon^2)^{3/2}} \mathbf{x}_{ij}.$$  

Another common approach is to use a cubic spline density distribution of the form:

$$\rho(x) \propto \begin{cases} 
4 - 6x^2 + 3x^3 & \text{for } x < 1 \\
(2 - x)^3 & \text{for } 1 \leq x < 2 \\
0 & \text{for } x \geq 2.
\end{cases}$$

This form has the advantage that the density distribution is truncated (i.e. goes to zero beyond $x = 2$) and so the force law becomes precisely Newtonian at $2 \geq 2\epsilon$. In general, softening kernels of this type (known as compact kernels) are superior to non-compact kernels (e.g. Plummer).
Choosing a value for the softening length is something of an art form. It shouldn’t be too large as any structure on scales smaller than the softening length will be smoothed away. However, it shouldn’t be too small, or two-body collisional effects will become important again. A good discussion of choosing softening lengths is given by Dehnen [2001], who explores further refinements to this idea, such as compensating the reduced forces on small scales by slightly enhancing forces on larger scales and the possibility of adaptive softening lengths (i.e. making $\epsilon$ a function of, for example, local density).

1.2.2 Particle-Mesh

The fundamental limitation of the direction summation technique is that it is $O(N^2)$. So, let’s explore some ways to reduce the computational load. One of the first ideas was to compute forces by solving Poisson’s equation rather than by direct summation. Suppose we have a density field $\rho(x)$. Poisson’s equation tells us that the gravitational potential is related to this density field by

$$\nabla^2 \Phi(x) = 4\pi G \rho(x). \quad (14)$$

If we can solve this equation, then the acceleration on particle $i$ can be found from the potential using

$$\ddot{x}_i = -\nabla \Phi(x_i). \quad (15)$$

In particular, if we know the density field on a uniform grid then we can use Fourier transforms (in particular, Fast Fourier Transforms) to quickly solve these equations. If we represent the Fourier transform of some quantity $q$ on our grid as

$$q_i = \sum_{k_j} \tilde{q}_j \exp(i k_j \cdot x_i), \quad (16)$$

where the sum is taken over all wavenumbers $k_j$ then from Poisson’s equation we have

$$\sum_{k_j} k_j^2 \tilde{\Phi}_j \exp(i k_j \cdot x_i) = 4\pi G \sum_{k_j} \tilde{\rho}_j \exp(i k_j \cdot x_i), \quad (17)$$

which can only hold in general if

$$-k_j^2 \tilde{\Phi}_j = 4\pi G \tilde{\rho}_j. \quad (18)$$

Thus, to compute the potential, we proceed as follows:

1. Find the density field on a grid;
2. Compute its Fourier transform;
3. Multiply each Fourier component by $-4\pi G/k_j^2$;
4. Take the inverse Fourier transform.

We can actually skip the final step since we’re interested in the force on each particle which is given by

$$\tilde{x}_j = -i k_j \tilde{\Phi}. \quad (19)$$

So, we simply multiply the potential by $-i k_j$, take the inverse Fourier transform and are left with the particle accelerations on a grid. We can interpolate accelerations to the precise location of each particle if necessary.

The density field is, of course, constructed from the particle distribution. There are numerous ways to do this. The most common are nearest grid point (NGP), cloud-in-cell (CIC) and triangular-shaped cloud (TSC) methods which are illustrated in Figure 2.
Particle-mesh algorithms of this sort are $O(N + N_g \log N_g)$ where $N_g$ is the number of grid points and can therefore be substantially faster than direct summation techniques. Their main limitation is that information about the density distribution (and, therefore, the potential and acceleration fields) on scales smaller than the size of a grid cell are lost. Consequently, structures on smaller scales will be lost or fail to form. This loses one of the nice features of the N-body approach, namely that it puts the resolution where it’s needed. More elaborate techniques, using nested grids of different resolution can be used to help mitigate this limitation.

### 1.2.3 Particle-Particle/Particle-Mesh (P3M)

This approach aims to combine the advantages of direct summation (no loss of small scale resolution) and particle-mesh (fast) techniques. Briefly, it computes the forces by dividing contributions from particles nearby and far away. For particles that are “far away” (typically more than about 3 grid cell lengths away) the contribution to the force is computed using the particle-mesh technique. For particles that are closer by a direct summation is used. (Frequently this means doing the full particle-mesh calculation and then, for each nearby particle, subtracting the force it contributed in the particle-mesh approximation before adding its contribution using the direct summation approach). The biggest problem with this approach is that it can easily become dominated by the direct summation (particle-particle) part of the calculation and become about as slow as direct summation. This will happen in any simulation where particles become strongly clustered (and, unfortunately, gravity is attractive...).

### 1.2.4 Tree Algorithms

Tree algorithms take a rather different approach. In a “top down” approach, the entire simulation volume (typically a cubic region, or, in the 2-D example shown in Fig. 3, a square) is placed into the top level cell. The next level in the tree is created by splitting this cell in half in each dimension such that the 2nd level contains 8 cells (in 3-D; a so-called oct-tree) or 4 cells (in 2-D; a quad-tree). Further tree levels are made by repeatedly splitting cells in the next higher level in this way until each cell contains only a single particle. The resulting tree structure is illustrated in Fig. 4. In the original Barnes-Hut [Barnes and Hut, 1986] tree algorithm, the center of mass of the particles in each cell at each level is computed. Then, to compute the force on any given particle, one simply computes the contribution due to the total mass of particles located at the center of mass of a cell. For nearby particles, we will use the finest levels of the tree to accurately determine the forces. For more distant particles we can use a coarser level of the tree and therefore compute the force due to many particles in one go. In this way, the calculation can be made faster than direct summation while retaining good spatial resolution on small scales. The tree approach has an advantage over particle-mesh techniques in that doesn’t waste time on empty regions of the simulation volume (useful if one is simulating the collision of two galaxies for example). However, there is a memory
overhead associated with constructing and storing the tree structure itself.

The original Barnes-Hut tree algorithm computed forces directly by treating particles in each cell as a monopole (i.e., a point mass). More recent algorithms work with the potential instead and account for higher order moments of the particle distribution in each cell. Essentially, the potential due to particles in a cell is described by a multipole expansion—more or less like expanding a function as a Taylor series. The more terms we include, the more accurate the representation (but the slower the calculation), and the functional forms used are easily differentiable making it simple to compute the force from the potential. This method is more accurate than the simple Barnes-Hut method, but is computationally more expensive if higher order multipoles are used.

How do we determine which level of the tree we should use to compute the force on any given particle? First, keep in mind that we're always going to have a trade off between speed and accuracy with the tree algorithm. If we decide to use the tips of each branch (call them “leaves”) then we have precisely one particle per cell and we would have effectively the particle-particle algorithm, which would be very accurate, but very slow (slower than the original particle-particle algorithm because we wasted a whole lot of time building the tree). Alternatively, if we used the base of the tree (call it the “trunk”) then the calculation would be extremely fast, but very inaccurate, as we'd only consider the interaction of each particle with the center of mass of the distribution. Obviously, we want something in between.

The usual approach is to adopt an “opening angle criterion”. Consider the circled particle in Fig. 5. We want to compute the force on it due to all other particles. We can begin at the “trunk” of the tree and ask the following questions: Is the ratio of the size of the tree cell at this level divided by the distance from the particle to the center of mass of the cell less than some angle \( \theta \)? More specifically, we ask if

\[
\frac{d}{r} < \theta
\]  

(20)

where \( d \) is the size of the tree cell and \( r \) is the distance from the particle in question to the center of mass of this cell. We then proceed as follows:

1. If the opening angle condition is satisfied, then compute the force from all particles in this tree cell by treating them as a single particle at the center of mass of the cell.
2. Otherwise, walk along the tree branches to the next most refined level of the tree and recheck the opening angle condition.

In this way, as we get closer to a set of particles we will use more refined branches of the tree to compute the force from them. The parameter \( \theta \) is a parameter that we can tune—making it smaller will make for a more accurate but slower calculation, while making it larger will reduce accuracy and increase speed. Typically, values of \( \theta \lesssim 1 \) are found to work quite well (although this depends on the details of the algorithm, such as whether or not higher order multipoles of the gravitational potential are included or not). A tree algorithm with this type of opening angle criterion is relatively easy to code as a recursive function, which simply walks along the branches of the tree accumulating forces from sets of particles as it goes and truncating the walk along each branch once the opening angle has been specified. From a coding point of view, this makes for a compact and elegant algorithm.

Why this opening angle criterion and not some other criterion? We can think about the physics the underlies it. Consider the mass distribution in a tree cell. We can represent any density distribution as a sum over various multipoles. Therefore, we can write the gravitational potential for this mass distribution as a multipole expansion in spherical coordinates

\[
\Phi(r, \theta, \phi) = \sum_{l=0}^{\infty} \sum_{m=-l}^{l} \left( A_l r^l + B_l r^{l-1} \right) \left[ C_{lm} Y_{lm}^c(\theta, \phi) + S_{lm} Y_{lm}^s(\theta, \phi) \right],
\]  

(21)
Figure 3: Example of the construction of a quad-tree.
where $A_l$, $B_l$, $C_{lm}$ and $S_{lm}$ are coefficients determined by the mass distribution and the $Y_{lm}$’s are the sine and cosine real spherical harmonics. (This is actually a Laplace series, valid outside the mass distribution—it is applicable to any potential which satisfies the Laplace equation $\nabla^2 \Phi = 0$.) Since we expect the gravitational potential to decline with distance from the source then $A_l = 0$ for all $l$. Then we see that the potential due to the $l^{th}$ multipole declines with distance as $r^{-l-1}$ (e.g. for a monopole distribution, $l = 0$, the usual $r^{-1}$ relation is obtained). Therefore, higher order multipole contributions to the potential will be reduced by a factor of approximately $(d/r)^l$ relative to the monopole contribution if $d$ is the characteristic size of the mass distribution. By ensuring that $d/r < \theta$ we guarantee that, for suitably small $\theta$, all terms beyond the monopole will be negligible.

### 1.3 Timestepping Criteria

Having computed the force acting on each of our particles, we need to evolve the particle distribution forward in time. This amounts to solving the following differential equations

\[
\begin{align*}
\dot{x}_i &= v_i \quad \text{(22)} \\
\dot{v}_i &= F_i/m_i \quad \text{(23)}
\end{align*}
\]

where $x_i$ and $v_i$ are the position and velocity of particle $i$, $F_i$ is the force acting on that particle and $m_i$ is its mass. Naively, we could choose some suitably small timestep, $\delta t$, and assume that the force is approximately constant over that period such that

\[
\begin{align*}
x_i &= x_i + v_i \delta t + \frac{1}{2} F_i/m_i \delta t^2 \quad \text{(24)} \\
v_i &= v_i + F_i/m_i \delta t \quad \text{(25)}
\end{align*}
\]

Two problems immediately arise: First, how small should we make $\delta t$ such that our approximation will be valid. Second, since characteristic timescales in gravitating systems scale as $\rho^{-1/2}$, the densest regions of
Figure 5: Opening angle criterion.
our simulation will evolve on much shorter (i.e. orders of magnitude shorter) timescales so they’ll force us to take very short timesteps. The second of these suggests that we perhaps want to have different timesteps for different particles (no problem in principle, but we have to correctly synchronize particles). The first suggests that we figure out some criterion to judge how small the timestep needs to be.

To make having individual timesteps for each the particles computationally manageable it’s often useful to enforce the timesteps to by binary fractions of some overall timestep, $T_0$. That is, each particle will have a timestep $T_0/2^n$ for some integer $n$. Thus, a particle with $n = 1$ will take two steps for every one step taken by a particle with $n = 0$, while a particle with $n = 2$ will take four steps over the same time. The advantage of this approach is that we can easily maintain synchronization between particles (i.e. bring them all to the same time). To ensure sufficiently accurate evolution we then enforce

$$\delta t = \frac{T_0}{2^n} < \eta \frac{1}{\sqrt{G \rho}},$$

(26)

where $\rho$ is an estimate of the locally enclosed density. For a particle orbiting at radius $r$ in a spherical potential for example, $\rho \sim M(r)/r^3$ if $M(r)$ is the mass inside radius $r$. Here, $\eta$ is a numerical parameter that we can adjust to control the size of timesteps. We want timesteps small enough that any results we extract from the calculation are converged (i.e. unaffected by timestep size) but large enough that the calculation completes in a reasonable amount of time.

For specificity, we’ll consider the time integration algorithm used in the original GADGET N-body code [Springel et al., 2001]. There are many other (and better) choices, but most share similar features (see, for example, Springel 2005, Zemp et al. 2007). GADGET-1 updates particle positions and velocities by first predicting the position of a particle at the middle of a timestep $\delta t$:

$$x^{(n+\frac{1}{2})} = x^{(n)} + v^{(n)} \frac{\delta t}{2},$$

(27)

where the superscripts refer to position/velocity at step $n$. This mid-step position is used to find the acceleration of the particle

$$a^{(n+\frac{1}{2})} = -\nabla \Phi |_{x^{(n+\frac{1}{2})}}.$$  

(28)

The particle is then advanced to the next timestep using

$$v^{(n+1)} = v^{(n)} + a^{(n+\frac{1}{2})} \delta t$$

(29)

$$x^{(n+1)} = x^{(n)} + \frac{1}{2} \left[v^{(n)} + v^{(n+1)}\right] \delta t.$$  

(30)

Note that we use the mean velocity through the timestep when updating the position. This is a variant of the so-called “leap-frog” method. Energy (kinetic plus potential) should always be conserved along the particle’s path, but in practice it won’t be precisely since we take a finite timestep. It can be shown that the error in energy for the above time integrator is

$$\Delta E = \frac{1}{4} \frac{\partial^2 \Phi}{\partial x_i \partial x_j} v_i^{(n)} a_j^{(n+\frac{1}{2})} \delta t^3 + \frac{1}{24} \frac{\partial^3 \Phi}{\partial x_i \partial x_j \partial x_k} v_i^{(n)} v_j^{(n)} v_k^{(n)} \delta t^3 + O(\delta t^4).$$  

(31)

That is, this integrator is second order accurate (i.e. the first time in the above is third order in $\delta t$). In principle, the above could be used to choose $\delta t$ to keep the error in energy below some predefined level. In practice this is not very useful as determining the gradients of the potential is expensive and we don’t know for sure that the higher order terms are not important. Instead, GADGET-1 uses a criterion similar to that given above using a local estimate of the density. This local estimate is found by averaging over some number of nearby particles (similar to the approach used for smoothed particle hydrodynamics as we’ll discuss later). This can be problematic in low density regions (where we have to go to large distances to find neighboring particles) and in dense regions resolved by only a small number of particles. For example, consider a cosmological simulation in which a low mass dark matter halo forms from 20 particles. Suppose
we are using the 32 nearest particles to estimate the local density. In this case, the particles in the halo will have their local density estimate contaminated by particles that are not in the halo. As a result, the density estimate will be too low, the resulting timestep too long and so the halo will dissipate because of accumulating errors in energy conservation.

Therefore, GADGET-1 adds a second criterion

\[ \delta t < \eta_a \frac{\sigma}{|a|}, \]  

(32)

where \( \eta_a \) is an adjustable parameter and \( \sigma \) is an estimate of the local velocity dispersion (also found by averaging over nearby particles). These prevents the velocity changing by more than a fixed fraction of the local velocity dispersion in a given timestep. Consider again the case of the low mass dark matter halo described above. If we estimate the local velocity dispersion from the 32 nearest neighbors, then we’ll underestimate it due to the contamination from particles outside the halo. In this case, however, this makes our timestep shorter, and so our integration more accurate. By combining these two criteria it’s possible to maintain good energy conservation through a simulation while minimizing the number of timesteps that have to be taken.

1.4 Initial Conditions

So far we’ve considered how to compute the forces and evolve the particle positions/velocities for some set of particles for which we already know the position and velocity of each particle. But, we have to begin somewhere. Specifically, we need to know the positions and velocities of the particles at some initial time. These are the initial conditions of the calculation. Generally speaking, the goal of creating initial conditions is to set up the particles to represent some physical system while minimizing any numerical effects due to discreteness or assumptions made. We’ll consider two scenarios which are common in the world of N-body simulation: an equilibrium system of dark matter, and the early Universe.

1.4.1 Equilibrium Dark Matter Halo

(Note that this could just as easily by an equilibrium system of stars, i.e. a galaxy.) A not uncommon N-body experiment is to see what happens when two systems, e.g. dark matter halos, merge. To do this, we typically want to create isolated N-body representations of the halos when they are well separated and in internal equilibrium.

For cosmological halos in a cold dark matter universe, the Navarro-Frenk-White profile [Navarro et al., 1997] is a good description of the run of density with radius. This profile has the form

\[ \rho(r) = \frac{\rho_0}{(r/r_s) \left[ 1 + (r/r_s) \right]^2}, \]  

(33)

where \( \rho_0 \) is a normalization and \( r_s \) is a characteristic radius known as the scale radius. We can select values for these constants based on cosmological calculations, but the numerical values don’t matter for our purposes. We want to make an N-body representation of an NFW halo. Let’s assume that the halo is perfectly spherical to keep things simple. Also, we can note that the mass of the NFW profile is logarithmically divergent as we go to larger radii. This is kind of annoying for an N-body simulation because we can’t model an infinite volume. So, let’s introduce a truncation such that

\[ \rho(r) = \frac{\rho_0}{(r/r_s) \left[ 1 + (r/r_s) \right]^2} \times \begin{cases} 
1 & \text{if } r < r_{\text{out}} \\
\exp \left( \frac{-r}{r_{\text{out}}} \right) & \text{if } r > r_{\text{out}}
\end{cases}, \]  

(34)
where \( r_{\text{out}} \) is some outer radius. Like any good computational physicist we’ll be sure to check that our adoption of a truncation radius doesn’t affect our results (for example, by repeating our calculations with a larger value of \( r_{\text{out}} \)).

First, we want to select a position for each particle in our N-body representation. Since the halo is spherical it makes sense to use spherical coordinates. For \( \phi \) things are easy, we just pick a number at random from a distribution that’s uniform between 0 and 2\( \pi \). For \( \theta \) it’s also easy—we know that the surface area on a sphere between \( \theta \) and \( \theta + d\theta \) is proportional to \( \sin \theta d\theta \equiv d\vartheta \). So, if we select a number at random from a uniform distribution between -1 and 1 and call this \( \cos \theta \) then take the inverse cosine to get \( \theta \) we should have the correct distribution.

For the radial coordinate, \( r \), we have to think a little more. If each of our particles has the same mass, then the chance of a particle drawn at random from the halo mass distribution lying between \( r \) and \( r + dr \) should be simply proportional to the fraction of mass in that region. That is

\[
dP(r) = \frac{4\pi r^2 \rho(r) dr}{\int_0^{\infty} 4\pi r^2 \rho(r) dr}.
\]

(35)

if we integrate this, we get the cumulative probability that any randomly selected particle will be found within radius \( r \):

\[
P(r) = \int_0^r \frac{4\pi r'^2 \rho(r') dr'}{\int_0^{\infty} 4\pi r'^2 \rho(r') dr'}.
\]

By definition, \( P(r) \) runs from 0 to 1. Therefore, for each particle we can simply select a random value from a uniform distribution between 0 and 1 and call this \( P(r) \). We then use the above equation to infer the \( r \) that corresponds to this \( P(r) \). In this way we’re guaranteed to build up a halo with the correct density profile. Note that the above integrals are not analytically tractable when the exponential term is included, so they usually have to be solved numerically.

After applying this procedure, we have 3-D positions for all of the particles in the N-body representation of the halo (which we can easily convert to Cartesian coordinates for input into our N-body solver). We also need velocities. For these, we make use of the fact that our halo is assumed to be in equilibrium. In collisionless gravitational dynamics, we know that any equilibrium system must obey Jeans equation:

\[
\frac{d(\rho \sigma_r^2)}{dr} + 2\frac{\beta(r)}{r} \rho(r) \sigma_r^2(r) = -\frac{GM(r)}{r^2} \rho(r),
\]

(37)

where \( \sigma_r(r) \) is the radial velocity dispersion at radius \( r \), \( M(r) \) is the mass enclosed with radius \( r \) and \( \beta(r) \) is the anisotropy parameter defined as

\[
\beta(r) = 1 - \frac{\sigma_\phi^2(r) + \sigma_\theta^2(r)}{2\sigma_r^2(r)},
\]

(38)

with \( \sigma_\phi(r) \) and \( \sigma_\theta(r) \) being the velocity dispersions in the other two directions. Since we know \( \rho(r) \) (and, therefore, \( M(r) \)) we can solve this equation for \( \sigma_r(r) \) if we choose some form for \( \beta(r) \). There are many choices here, but for simplicity let’s assume an isotropic velocity distribution which implies \( \beta(r) = 0 \). The Jeans equation is then easily solved for \( \sigma_r = \sigma_\phi(r) = \sigma_\theta(r) \). If we further assume that the velocity distribution in each coordinate is always a Gaussian then we have a fully specified velocity distribution function at all radii in the halo. Given the radius of a particle, we can then compute the corresponding \( \sigma_r \) and draw a random number from a normal distribution with standard deviation \( \sigma_r \) and repeat for the other two coordinates. This will give us a set of particle velocities consistent with Jeans equation and so consistent with an equilibrium halo.

Before moving on, we should think a little more about the final assumption made above, namely that the velocity distribution is Gaussian. Jeans equation does not tell us that the distribution is Gaussian. It
merely tells us what the dispersion of the velocity distribution should be. We could construct an infinite number of non-Gaussian distributions which all have the same $\sigma$. So, we don’t know for sure that we have the correct velocity distribution. Jeans equation can be extended to higher order moments of the velocity distribution, so we could (in principle) compute all of these higher moments and use them to construct a more accurate velocity distribution. In practice this is sometimes done, but often it is not. The reason is that the Gaussian approximation is often not too bad, and computing higher order moments rapidly becomes numerically challenging. When solving numerical problems we should always keep in mind which of the many approximations we have made is the most severe. For example, is it worth solving the Jeans equation for higher order moments of the velocity distribution when we’ve already approximated the halo as being spherically symmetric? Real halos are not spherically symmetric, so maybe this assumption is the biggest factor limiting the accuracy of our results. Maybe the fact that we’re modelling purely dark matter with no gas component is a bigger limitation... There’s no point wasting time doing one part of the calculation to arbitrarily high precision if other aspects are solved in a much cruder way.

1.4.2 Cosmological Initial Conditions

For cosmological calculations we’re interested in setting up initial conditions that represent the distribution of matter in the early stages of the Universe. At early times, the Universe is almost perfectly uniform with just small perturbations in the density as a function of position (these are the source of the $10^{-5}$ level ripples seen in the cosmic microwave background). In our standard cosmological model there are two features that make the situation simpler still. First, if inflation is correct then the density perturbations should be Gaussian—that is the phases of individual Fourier modes of the density field are uncorrelated and so the density field is completely described (statistically) by a power spectrum (which measures the mean amplitude of Fourier modes of a given wavelength). Second, in cold dark matter models the particles begin with zero random velocities—i.e. there is no velocity dispersion at a given point and so velocity is an entirely deterministic function of position, controlled entirely by the density field. A further useful feature is that, because the initial density perturbations are small they can be treated with a linear perturbation theory analysis. Providing we begin our simulation at an early enough time in the universe we can therefore use a linear analysis to set our initial conditions. We won’t discuss the details of cosmological perturbation theory in this class (take Ay 127 for that). A good discussion of the details (including the unpleasant relativistic aspects that we’re going to completely ignore!) can be found in Ma and Bertschinger [1995], while a code for generating such initial conditions is given by Bertschinger.

Briefly, the standard method to construct initial conditions for such scenarios is as follows:

1. Create a set of “pre-initial conditions” which is a random, but homogeneous set of particle positions in a periodic cube. We can construct this by simply selecting particle positions $(x, y, z)$ at random within a cubic region.

2. Compute the power spectrum, $P(k)$, that gives the mean amplitude of each Fourier mode of the desired density field. For a Gaussian random field the distribution of amplitude, $\delta$, for a given mode can be shown to be a Rayleigh distribution

$$P(\delta)d\delta = \frac{\delta}{\sigma^2} \exp\left(-\frac{\delta^2}{2\sigma^2}\right) d\delta,$$

(39)

Frequently, this is not actually what is done, because it introduces fluctuations in the density field due to Poisson statistics, i.e. fluctuation in particle number in any region. If we began evolving this “homogeneous” distribution regions that randomly happen to be denser would quickly collapse under their own gravity. An alternative is to use a distribution in which the force on each particle is zero. For example, a regular lattice satisfies this condition (although it’s an unstable equilibrium). Another common approach is to use “glass” initial conditions which can be made by starting with a Poisson-random distribution, then evolving it forward in time using a repulsive gravitational force. The particles will then all try to move apart until they reach positions of zero net force.
where \( \sigma^2 = VP(k)/2 \) and \( V \) is the volume of the simulation cube\(^3\).

3. Take the Fourier transform to convert the \( \hat{\delta} \) Fourier components into an actual density field. We then want to move the particles to recreate this density field in our N-body representation. For this we can use the Zel’dovich approximation which relates the Lagrangian position (position in the initial conditions) of a particle to its Eulerian position. We won’t discuss the details, but note that this is a linear perturbation theory solution to the cosmological equations governing the growth of density perturbations. The Zel’dovich approximation states that \( x = q + \Psi(q) \) where \( q \) is the initial (Lagrangian) position of a particle and \( x \) the Eulerian position. We can compute the gravitational potential \( \Psi(q) \) from our density field and so can find \( x \). We can also take the derivative of this equation to get the velocity \( \dot{x} \).

There are problems with this method that make constructing accurate cosmological methods very difficult. For example, what about the contribution from modes with wavelengths longer than the size of our simulation box? If we compute the density field on a grid, then we will only get Fourier modes whose wavelength is an integer number of grid cell lengths—do this missing modes matter? Fortunately, these issues have been considered extensively (see, for example, Sirko 2005) and accurate initial condition generating codes exist.

1.5 Parallelization

If you can do something with one computer, you can do it bigger and better with many computers. At least, you can in principle…

N-body codes are computationally demanding and so it’s no surprise that a lot of work has been put into designing them to run efficiently on parallel computers. The idea is to divide the calculations that must be performed between a large number of processors and have the each solve part of the problem. To do this well there are a few considerations:

- How do we best divide the work between the processors? This may be limited by the amount of memory available to each processor (which will limit, for example, how many particles can be stored on each processor). Most importantly though, we want each processor to have about the same amount of work to do—if we have a situation where one processor is still crunching through its tasks while all others have already finished then we’re not using the processors most efficiently. This is referred to as load balancing.

- How can we minimize the amount of communication between processors? Processors will need to communicate their results to each other (since, for example, particles on one processor will need to know the forces they feel from particles on some other processor). Communication is a relatively slow task so we want to minimize it as much as possible.

Dubinski [1996] gives a description of an approach to parallelizing a tree code. We’ll follow his discussion. The basic approach is one of domain decomposition in which the simulation volume is broken up into sub-volumes and each sub-volume is assigned to a separate processor. The task is to find a sufficiently optimal domain decomposition to meet the criteria described above.

Let us assign to each particle a cost, \( C_i \), which is a measure of the computational work required for this particle. In practice, this could be the number of gravitational force evaluations to evolve it over some

\(^3\)Note that since the density field must be real, the Fourier components must satisfy the Hermitian constraints: \( \hat{\delta}(k) = \hat{\delta}^*(-k) \).
fixed time period for example. The cost is something we can compute relatively easily for each particle as the simulation proceeds. At the start of the calculation we don’t know $C_i$, so we’ll set $C_i = 1$ for all particles initially. We want to choose domains such that $\sum C_i$ (where the sum is taken over all particles in the domain) is about the same for all domains. To do this, we can use a technique to the tree construction algorithm we’ve used before. Suppose we have $2^n$ processors, with $n$ some integer. We begin by splitting the simulation volume in two, with the division made such that $\sum C_i$ is the same (or almost the same) on each side of the division. We then proceed to split each of these two domains into two sub-domains again balancing $\sum C_i$ in each subdomain. Repeating this process $n$ times we get $2^n$ domains, corresponding to rectangular regions of the simulation cube, each with the same total computational cost.

As the simulation proceeds, the cost for each particle will change. For example, if a particle moves into a denser region it will require more computations to evolve its position and so its cost will increase. Therefore, our initially load balanced domain decomposition will not remain load balanced. Therefore, after each timestep (or, perhaps, once the load balancing becomes sufficiently bad) we can attempt to repartition into new domains to re-balance the workload. This means moving some particles from one processor to another, and therefore requires communication. We want to minimize this communication by moving as few particles as possible. Deciding which particles to move and when depends on the details of the simulation (e.g. how clustered the particles become), the algorithm used (which affects the distribution of costs) and the hardware used (which determines the relative costs of communication vs. work load imbalance). So, this usually involves some tuning of parameters to find the optimal way in which to move particles from one processor to another.

An additional problem arises with the need to communicate gravitational forces between domains. Once we’ve performed the domain decomposition it’s easy for each processor to build an oct-tree for its local set of particles. We could then have each processor share its local oct-tree with every other processor. Then, each processor would have the full tree and so could proceed to compute accelerations and update positions for all of its particles. In practice this is a bad idea for two reasons. First, the memory requirements for the full tree may be more than a single processor can handle. Second, this involves a lot of communication which slows down the calculation. However, we can make use of the opening angle criterion to limit the communication and memory requirements. Consider two widely spaced domains, which we’ll label 1 and 2. Since all of the particles in domain 1 are far from all of the particles in domain 2 we’ll only need to use the coarsest levels of the domain 2 tree to evaluate forces for particles in domain 1 (and vice versa). Therefore, we can apply the opening angle criterion to domain 2 in a conservative way to find the finest level of the domain 2 tree that will ever be needed by any particle in domain 1. We then communicate only those levels of domain 2 to the processor working on domain 1, and none of the finer levels of the tree. This allows the essential tree for domain 1 to be built—it contains just enough information to compute the forces on domain 1 particles at the required level of accuracy. This minimizes the communication and memory requirements.

Modern N-body codes use more complicated algorithms, but the basic ideas are the same.

2 Smoothed Particle Hydrodynamics

So far, I’ve only discussed gravitational forces in the N-body technique. This is OK if we’re dealing with dark matter and/or stars since these both act as collisionless particles which experience only gravitational forces. But, in general, we want to include gas as well, and so we need to include hydrodynamic forces and, perhaps, things like radiative energy losses etc. You’ve already discussed hydrodynamical forces and their numerical solution using grid-based methods (i.e. Eulerian methods). We can certainly use those in conjunction with an N-body representation of dark matter for example. But, since we’re dealing with particles anyway, it’s worth considering if we can use a particle-based approach for hydrodynamical forces also. One advantage of this might be that a particle based approach naturally concentrates particles in dense regions, so we
automatically get better resolution in dense regions of our simulation.

The first thing to note in considering this is that hydrodynamical forces involve things like density and pressure gradients. These are quantities which fundamentally can’t be specified for a single particle. Instead, they depend on the relation between a particle and other nearby particles. For example, the density in a region clearly depends on how many particles are present in that region. So, we’re going to need to consider collections of particles to estimate these quantities. The Smooth Particle Hydrodynamics (SPH) technique does this by smoothing over the local particle distribution in a specific way that we’ll examine next. We’ll follow the notation of Springel [2010] who gives a more detailed review of the SPH technique.

2.1 Smoothing Kernels

We begin by realizing that, for any field \( F(\mathbf{r}) \) (which could be density, pressure etc.), we can write a smoothed version of this field as a convolution

\[
F_s(\mathbf{r}) = \int F(\mathbf{r}) W(\mathbf{r} - \mathbf{r}', h) d\mathbf{r}',
\]

where \( W(\mathbf{r} - \mathbf{r}', h) \) is a smoothing kernel and \( h \) is the characteristic length scale of the kernel. In the limit \( h \to 0 \) the kernel should approach a Dirac delta function in which case \( F_s(\mathbf{r}) = F(\mathbf{r}) \). To be useful for our purposes we’ll soon see that we want a kernel that is symmetric (i.e. no preferred direction) and is differentialable at least twice (so we can take derivatives of the smoothed field). In the early days of SPH people used Gaussian kernels, but most modern codes use a kernel with “finite support” (i.e. they’re zero beyond some finite radius), typically a cubic spline. Writing \( W(\mathbf{r}, h) = w(|\mathbf{r}|/2h) \) (so that our kernel is automatically symmetric and scales linearly with \( h \)) we can write

\[
w(q) = \frac{8}{\pi} \begin{cases} 
1 - 6q^2 + 6q^3, & 0 \leq q \leq \frac{1}{2} \\
2(1 - q)^3, & \frac{1}{2} < q \leq 1, \\
0, & q > 1.
\end{cases}
\]

This kernel (which belongs to a broader class of similar kernels) goes to zero beyond \( r = 2h \).

In the case of a particle distribution our field \( F(\mathbf{r}) \) is actually a collection of delta functions. If each particle has a mass \( m_i \) and a density \( \rho_i \) (we’ll worry about how we get this later) then they are associated with a volume \( \Delta \mathbf{r}_i \sim m_i/\rho_i \). In this case, we can replace the convolution integral with a sum:

\[
F_s(\mathbf{r}) \approx \sum_j \frac{m_j}{\rho_j} F_j W(\mathbf{r} - \mathbf{r}_j, h).
\]

This is then a smooth, differentialable representation of the field \( F \) described by our particles. We should be clear that this is an approximation, and so has some error compared to the true field (i.e. that we would obtain with infinite particles). If the particles are uniformly spaced by a distance \( d \) in 1-D, it can be shown that the interpolation is 2nd order accurate if \( h = d \). For real distributions in 3-D it’s not so simple to prove this, but it seems reasonable that we’d need \( h \geq d \) which means \((4\pi/3)2^3 \approx 33\) neighbor particles should be within the non-zero part of the smoothing kernel.

So what about the density? We’ve already stated that we don’t know the density for an individual particle, yet it appears in the above sum. Fortunately, we can use the SPH kernel to evaluate the density. Let \( F_i = \rho_i \), then the smoothed density field becomes

\[
\rho_s(\mathbf{r}) \approx \sum_j m_j W(\mathbf{r} - \mathbf{r}_j, h),
\]

which we can estimate knowing only the masses of the particles. The density for each particle is then found from the smoothed density field at the position of the particle.
We are free to choose whatever \( h \) we want and, in fact, to make it a function of position. This is advantageous because we can make \( h \) small in regions of high density and thereby avoid smoothing away too much information in these regions. There are essentially two choices for computing a suitably position-dependent \( h \). In the “scatter” approach we use a kernel \( W[r - r_j, h(r)] \) (i.e. \( h \) depends on the position \( r \)) while in the “gather” approach we use \( W[r - r_j, h(r_j)] \) (i.e. \( h \) is determined at the position of the particle for which we’re computing a smoothed field). The gather approach has the advantage that density of particle \( i \) can be determined using only \( h_i \), the smoothing length for that same particle\(^4\). Using the gather method we have

\[
\rho_i \approx \sum_j m_j W(r - r_j, h_i),
\]

where we’ve dropped the subscript “s”. From this, it becomes clear why kernels with compact support are useful—once we know \( h_i \) we need only find those particles within a distance \( 2h_i \) of particle \( i \) in order to find its density. This makes the calculation \( O(N_{\text{ngb}}N) \) where \( N \) is total particle number and \( N_{\text{ngb}} \) is the number of neighbors within \( 2h_i \) (which will be about the same for all particles since we choose \( h_i \) based on the distance to these nearest neighbors). The \( N_{\text{ngb}} \) parameter is a key input to SPH simulations and it should always be checked to be sufficiently large to give converged answers, as ultimately the accuracy of SPH will be limited by the accuracy of the smoothing/interpolation that the kernel approach implies.

Note that we can apply this approach to any field. For example, the velocity field is

\[
v_i = \sum_j \frac{m_j}{\rho_j} v_j W(r_i - r_j, h),
\]

which, since this is now a smooth and continuous field, we can take the derivative of to get the local velocity divergence

\[
(\nabla \cdot v)_i = \sum_j \frac{m_j}{\rho_j} v_j \cdot \nabla_i W(r_i - r_j, h).
\]

Alternatively, we can make use of the identity \( \rho \nabla \cdot v = \nabla (\rho v) - v \cdot \nabla \rho \) and estimating the smoothed versions of the two fields on the right separately. This gives

\[
(\nabla \cdot v)_i = \frac{1}{\rho_i} \sum_j m_j (v_j - v_i) v_j \cdot \nabla_i W(r_i - r_j, h),
\]

which turns out to be more accurate in practice and has the useful feature that it goes precisely to zero when all velocities are equal.

### 2.1.1 Variational Derivation

It’s possible to derive the SPH equations from a Lagrangian. Why would we do this? It guarantees the various conservation laws hold (since they’re explicitly encoded in the Lagrangian) and also ensures that the phase space structure imposed by Hamiltonian dynamics is retained. Eckart [1960] showed that the Euler equations for inviscid ideal gas flow follow from the Lagrangian

\[
L = \int \rho \left( \frac{v^2}{2} - u \right) dV.
\]

This idea was first fully exploited by Springel and Hernquist [2002] who discretized this Lagrangian in terms of the SPH particles, yielding

\[
L_{\text{SPH}} = \sum_i \left( \frac{1}{2} m_i v_i^2 - m_i u_i \right),
\]

\(\text{Technically if we integrate over the smoothed density field when using the gather method we don’t get the correct total mass, so this method does not correctly conserve mass. This doesn’t matter though since we explicitly conserve mass because mass is tied to particles which are conserved by construction.}\)
where \( u_i \) is the thermal energy per unit mass of the particle. For a gas with adiabatic index \( \gamma \) we can write

\[
P_i = A_i \rho_i^\gamma = (\gamma - 1) \rho_i u_i,
\]

where we’ll refer to \( A_i \) as the “entropy”. More specifically, \( A_i \) is uniquely determined by the specific entropy of the particle and so is conserved in adiabatic flow. So, assuming \( A_i \) to be constant we can write

\[
u_i = A_i \rho_i^{\gamma - 1}.
\]

We then use the standard variational approach to get our equations of motion:

\[
\frac{d}{dt} \frac{\partial L}{\partial \dot{r}_j} - \frac{\partial L}{\partial r_j} = 0.
\]

Some care is needed because the smoothing scale, \( h \), varies from particle to particle (and so we can represent it too as a smooth field using our standard kernel). But, the bottom line is

\[
\frac{dv_i}{dt} = - \sum_{j=1}^{N} m_j \left[ f_i \frac{P_i}{\rho_i^2} \nabla_i W_{ij}(h_i) + f_j \frac{P_j}{\rho_j^2} \nabla_i W_{ij}(h_j) \right],
\]

where

\[
f_i = \left[ 1 + \frac{h_i}{3 \rho_i \partial \rho_i} \right]^{-1},
\]

and \( W_{ij}(h) \equiv W(|r_i - r_j|, h) \). What’s nice about this is that we’ve converted the set of partial differential equations that described inviscid flow of an ideal gas into a much simpler set of ordinary differential equations. Mass and energy conservation are handled automatically (mass because we have a fixed number of particles and energy because our flow is adiabatic so the energy is determined by the (constant) entropy).

It’s also possible to derive relativistic implementations of SPH using this variational principle approach.

### 2.2 Other Issues

#### 2.2.1 Artificial Viscosity

In all of the above we’ve explicitly assumed that the specific entropy of a particle is conserved, i.e. that the flow is adiabatic. However, even if starting from smooth initial conditions the Euler equations can lead to shocks and contact discontinuities at which the differential form of the Euler equations breaks down and their integral form (conservation laws) must be applied instead. Analysis shows that in shocks the entropy is always increased. How can we introduce this behavior into our SPH calculation? The usual approach is to introduce an artificial viscosity term. The role of this artificial viscosity is to broaden the shocks into a resolvable layer and dissipate kinetic energy into heat, thereby increasing the entropy. If introduced as a conservative force then the conservation laws implicit in the Euler equations guarantee that the correct amount of entropy is generated, independent of the details of the viscosity used. Of course, the problem with this is that shocks are now not perfect thin—instead they will be resolved by a few smoothing lengths.

The viscous forces is usually added in the form

\[
\frac{dv_i}{dt} \bigg|_{\text{visc}} = - \sum_{j=1}^{N} m_j \prod_{ij} \nabla_i W_{ij},
\]
where
\[
\bar{W}_{ij} = \frac{1}{2}[W_{ij}(h_i) + W_{ij}(h_j)]
\] (56)
is a symmetrized kernel. Provided that \(\prod_{ij}\) is symmetric in \(i\) and \(j\) this viscous force is anti-symmetric between pairs of particles so always conserves linear and angular momentum. It does not conserve energy (that’s why we’re including it) so a compensating change in the internal energy (or, equivalently, entropy) is introduced to balance this. We then have the desired conversion of kinetic to thermal energy and a corresponding entropy production.

A common form used for the viscosity factor is
\[
\prod_{ij} = \begin{cases} 
-\alpha c_{ij} \mu_{ij} + \beta \mu_{ij}^2 / \rho_{ij} & \text{if } v_{ij} \cdot r_{ij} < 0 \\
0 & \text{otherwise,}
\end{cases}
\] (57)
with
\[
\mu_{ij} = h_{ij} v_{ij} \cdot r_{ij} / |r_{ij}|^2 + \epsilon h_{ij}^2.
\] (58)
Here, \(h_{ij}\) and \(\rho_{ij}\) are arithmetic means of the corresponding quantities for particles \(i\) and \(j\), and \(c_{ij}\) being the mean sound speed. The strength of the viscosity is controlled by parameters \(\alpha\) and \(\beta\). The goal is to make this viscosity significant in shocks but weak elsewhere (so that we don’t dissipate kinetic energy in regimes that are evolving adiabatically). Typical values of \(\alpha \approx 0.5-1.0\) and \(\beta = 2\alpha\) are used. The parameter \(\epsilon \approx 0.01\) is introduced to avoid a singularity if two particles get very close.

### 2.2.2 Self-Gravity

Our SPH gas particles feel the force of gravity and, in particular, there is a self-gravity between the SPH particles. We can make use of standard N-body techniques to compute the gravitational forces, but there’s also a neat way to do this using the variational approach discussed above. We simply add the gravitational self-energy of the SPH particles:
\[
E_{\text{pot}} = \frac{1}{2} \sum_i m_i \Phi(r_i) = \frac{G}{2} \sum_{ij} m_i m_j \phi(r_{ij}, \epsilon_j)
\] (59)
to the SPH Lagrangian giving
\[
L_{\text{SPH}} = \sum_i \left( \frac{1}{2} m_i v_i^2 - m_i u_i \right) - \frac{G}{2} \sum_{ij} m_i m_j \phi(r_{ij}, \epsilon_j).
\] (60)
When we apply the variational principle the equations of motion pick up extra terms due to gravity which look like:
\[
m_i a_i^{\text{grav}} = -\frac{\partial E_{\text{pot}}}{\partial r_i} = -\sum_j G m_i m_j \frac{r_{ij}}{r_{ij}} \left[ \phi'(r_{ij}, \epsilon_i) + \phi'(r_{ij}, \epsilon_j) \right] / 2
\]
\[-\frac{1}{2} \sum_{jk} G m_j m_k \frac{\partial \phi(r_{jk}, \epsilon_j)}{\partial \epsilon_j} \frac{\partial \epsilon_j}{\partial r_i}
\] (61)
The first time here is what you’d guess—it’s a symmetrized gravitational interaction. The second term shows up if the softening length, \(\epsilon\), is allowed to be different for each particle (and so varies with position), and is required to make the interaction conservative in such cases.
2.2.3 Time Steps

One final issue: What time steps should we use when evolving the SPH particles? The usual approach is to impose a so called Courant time step criterion which requires:

$$\Delta t_i = C_{CFL} \frac{h_i}{c_i}$$

(62)

where $c_i$ is the sound speed and $C_{CFL} \sim 0.1$–0.3. This limits each particle to moving significantly less than one smoothing length per timestep providing the particle is moving subsonically. This does not work so well in cases such as a blast wave propagating into cold gas (for which $c_i$ will be small, allowing large timesteps which can result in cold gas particles moving through the blast wave). Improved algorithms attempt to catch such cases. This timestep is usually augmented by checking the gravitational timestep (and taking the minimum of the two) using the usual N-body techniques. In fact, in an N-body tree code the tree structure turns out to be extremely useful for one of the key SPH operations, namely finding the nearest neighbors. The cost of the operation can be reduced from $O(N^2)$ for a simple search of all particles to $O(N_{ngb}N \ln N)$ when using the tree.

References


