Creating Lightcones in HACC

Joe Hollowed, Cosmological Physics and Advanced Computing Group

Last edited March 8, 2018

This document outlines our current approach for generating lightcones from HACC simulation products. In § *1*, we derive the condition which defines all spacetime events that can be seen by an observer at any given time (we parameterize the surface of the observer's past lightcone). In § *2*, we describe how we apply this condition to find lightcone-crossing times of objects whose positions and velocities are known only at discrete time snapshots, and nonlinear effects and corrections are discussed in § *3*. § *4* describes the details of our implementation of this problem in HACC. Finally, a summary is given in § *5*. The code which implements the methods described here can be found in the HACC repository at
`https://svn.alcf.anl.gov/repos/DarkUniverse/hacc/` (see § *4* for details). § *1* - § *2.1* of this document's content is based on in-line documentation in the above mentioned code, written by Hal Finkel.

1 Parameterizing the Lightcone Surface

At any given time t, an observer will see only what is on the surface of her past-lightcone. Any event located inside the lightcone surface has been seen in the past, and any event outside will be seen in the future, in general. Hence, our goal is to parameterize this surface in terms of quantities that are available to us as simulation outputs. Applying this parameterization to simulation datasets will allow us to locate all particles on an observer's past-lightcone at any given time (yielding the observable universe).

Let us first consider the problem in a static Euclidean space. Recall that the spacetime separation between two events occurring at (t, r, θ, φ) and $(t+dt, r+dr, \theta+d\theta, \varphi+d\varphi)$, according to the laws of special relativity, is given by the Minkowski metric:

$$ds^2 = -c^2dt^2 + dr^2 + r^2d\Omega^2, \tag{1}$$

where $d\Omega^2$ includes the change of variables from Cartesian to spherical coordinates as

$$d\Omega^2 \equiv d\theta^2 + \sin^2\theta d\varphi^2. \tag{2}$$

An event being located on the lightcone surface is defined by the fact that the line joining it to the observer is a null geodesic, where $ds^2 = 0$. If we impose the constraint that the observer is located at the spatial origin (which is always done in practice), then this null geodesic is purely radial, and thus $d\Omega^2 = 0$. Objects located on the lightcone therefore satisfy the condition

$$0 = -c^2dt^2 + dr^2 \tag{3}$$

which can be rearranged to express the distance to the event in terms of the light travel time as

$$c^2dt^2 = dr^2 \quad \rightarrow \quad \int_{t_{\rm event}}^{t_{\rm obs}} c\,dt = c(t_{\rm obs} - t_{\rm event}) = ||\mathbf{r}||\,, \tag{4}$$

where $t_{\rm obs}$ and $t_{\rm event}$ are the time coordinates of the observer and any observed event, respectively. $||\mathbf{r}||$ represents the magnitude of the spatial distance to the observed event.

So, we have parameterized the lightcone surface by the rightmost equality in Eq.(*4*) for a static universe. However, the universe *is not* static, and so light travel time between two events is *not* $||\mathbf{r}||/c$, but should effectively be increased by the scale factor $a(t)$. Obtaining the correct condition for finding an event on the lightcone, then, must require repeating the simple procedure in Eq.(*1-4*) using the more general Robertson-Walker metric:

$$ds^2 = -c^2dt^2 + a(t)^2\left[dr^2 + S_\varkappa(r)^2 d\Omega^2\right]. \tag{5}$$

Here, $S_\varkappa(r)$ defines the curvature of space, which can be ignored since $d\Omega^2 = 0$, as per our assumptions stated above. In that case, a null geodesic spacetime separation under this metric (again set $ds^2 = 0$ and solve) gives

$$\int_{t_{\rm event}}^{t_{\rm obs}} c\frac{dt}{a(t)} = ||\mathbf{r}||\,. \tag{6}$$

We can make this condition a bit cleaner by casting the variable of integration to the scale factor, as a proxy for time, if we note the following substitutions:

$$dt = \frac{dt}{da}da = \frac{da}{\dot{a}} \quad \text{and} \quad H = \frac{\dot{a}}{a}\,. \tag{7}$$

Combining Eq.(*6-7*) results in

$$\int_{a_{\rm event}}^{1} c\frac{da}{a^2 H} = ||\mathbf{r}||\,. \tag{8}$$

Eq.(*8*) is our sought after parameterization[a]; all events whose spacetime coordinates satisfy this condition are found on the lightcone surface and are visible to the observer, at the spatial origin, at time $t_{\rm obs}$.

2 Crossing the Lightcone

Given pristine knowledge of particle positions resultant from cosmological simulations, Eq.(*8*) appears straightforward to apply: for any time t, find the associated scale factor $a(t)$, solve the integral on the left hand side of Eq.(*8*) to obtain some value R. Then, check all particles in the simulation for the condition that $||\mathbf{r}_n|| = R$, where the subscript n denotes the nth particle. If the condition is satisfied for any particle n, then that particle crossed the lightcone and is seen by the observer at time t. Repeat this for all times t to fill the lightcone.

Unfortunately, simulation outputs do not include "all times t". Since we only have of order *100* time snapshot outputs from $a \approx 0$ to $a = 1$, the naive approach described

[a] Eq.(*8*) is actually the parameterization of a sphere in three-dimensional space, whose radius is dependent on the parameter a (A more readable form of a general expanding/contracting sphere is $||\mathbf{r}|| = \sqrt{x(t)^2 + y(t)^2 + z(t)^2} = f(t)$). Actually visualizing such a sphere requires projecting the object into two spatial dimensions and allowing time to occupy the third axis— this results visually in a cone, hence the name "lightcone". Indeed the lightcone surface is truly a spherical object.

in the previous paragraph is not applicable. Consider a particle n that crosses the lightcone surface (is seen) at time $t_{\rm cross}$ (ideally true of all particles)[b]. This event is only captured in the simulation outputs by the fact that particle n is inside the observer's past lightcone at some snapshot time $t_1 < t_{\rm cross}$, and outside of the lightcone at the next snapshot time $t_2 > t_{\rm cross}$. Ultimately, the problem that we need to solve in order to construct the lightcone is finding $t_{\rm cross}$ for all particles. This has traditionally been done in the HACC codebase using particle extrapolation from time t_1, which is discussed in § *2.1*. More recently implemented is an improved approach which rather interpolates between times t_1 and t_2, described in § *2.3*.

2.1 Particle Extrapolation

Since we would ultimately like to solve Eq.(*8*) for $a_{\rm event}$ (equivalently $t_{\rm cross}$), we must first write the relevant quantities in terms of simulation data products. Let us consider a single simulation timestep for the remainder of this section, which begins at snapshot i and ends at snapshot f. The time it spans is $t_f - t_i$, and the scale factor evolution is $a_f - a_i$.

Each particle in snapshot i has six quantities that are relevant to our purposes; three comoving Cartesian coordinates and corresponding velocities

$$\mathbf{r}_i = \begin{bmatrix} x_i \\ y_i \\ z_i \end{bmatrix} \quad , \qquad \mathbf{v}_i = \dot{\mathbf{r}}_i \ . \tag{9}$$

Note that $||\mathbf{r}_i|| \neq ||\mathbf{r}||$. In Eq.(*8*) $||\mathbf{r}||$ is the distance to the particle at the time it *crosses* the lightcone surface (unknown), somewhere between t_i and t_f, whereas $||\mathbf{r}_i||$ is the distance to the particle at time t_i (known). Likewise, $a_i \neq a$ as found in Eq.(*8*). We now define $||\mathbf{r}||$ by linear extrapolation from the particle's position and velocity at t_i:

$$\begin{aligned} ||\mathbf{r}|| &= ||\mathbf{r}_i + \mathbf{v}_i dt|| \\ &= \left[(x_i + \dot{x}_i dt)^2 + (y_i + \dot{y}_i dt)^2 + (z_i + \dot{z}_i dt)^2\right]^{1/2} \end{aligned} \tag{10}$$

where we have introduced the quantity dt as $dt = t_{\rm cross} - t_i$. Let us also obtain an expression for a in Eq.(*8*), the scale factor at the time $t_i + dt$:

$$a = a(t_i + dt) = a_i + \dot{a} dt \ . \tag{11}$$

Finally, let's describe a_f in terms of a_i and the timestep width $\tau = t_f - t_i$, as

$$a_f = a_i + \dot{a}\tau \ . \tag{12}$$

Refer to Figure *1* for a visual summary of the important quantities discussed above. Again, our ultimate goal is to find $t_{\rm cross}$ for all particles. The remainder of this section will do this by solving for the quantity dt.

[b] To say that the particle "is seen" at time $t_{\rm cross}$ is shorthand for the more accurate statement: "photons emitted from the particle's position at time $t_{\rm cross}$ are seen by the observer at time $t_{\rm obs}$". The observer of course observes the entire sky at a unique time $t_{\rm obs}$.

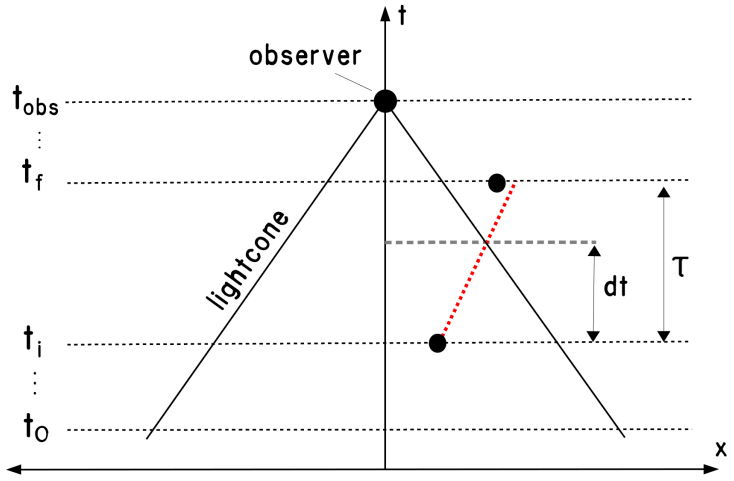

Figure *1*: A schematic of linear particle position extrapolation. This figure focuses on an arbitrary timestep bounded by snapshots i and f, where the vertical axis is time and the horizontal axis is space. Time increases upward. The particle (blue disc) is found inside the lightcone at time t_i, and outside the lightcone at the time t_f of the following snapshot. It is estimated to cross the lightcone, by extrapolation (red dashed line), at time $t_i + dt$. Note that the extrapolation does not in general estimate the correct position of the particle at time t_f.

At this point, we will take the assumption that $0 \leq dt \leq \tau \ll 1$, and therefore drop any nonlinear terms in dt[c]. This allows us to approximate $||\mathbf{r}||$ as

$$\begin{aligned} ||\mathbf{r}|| &= ||\mathbf{r}_i + \mathbf{v}_i dt|| \\ &\approx ||\mathbf{r}_i|| + \frac{dt(x_i\dot{x}_i + y_i\dot{y}_i + z_i\dot{z}_i)}{||\mathbf{r}_i||} \end{aligned} \tag{13}$$

in order to isolate dt[e]. Next, we can break the left-hand-side of Eq.(*8*) into two integrals; one that spans the time from the particle's lightcone intersection (at a) to the end of the snapshot t_f (at a_f)), and one that spans the remaining history of the universe:

$$\int_a^1 c\frac{da}{a^2H} = \int_a^{a_f} c\frac{da}{a^2H} + \int_{a_f}^1 c\frac{da}{a^2H}\,. \tag{14}$$

The first thing to note, here, is that the the bounds and integrand of the latter piece over $[a_f, 1]$ are entirely known in the simulation parameters and snapshot information, when we express H as

$$H = \frac{H_0}{a^{3/2}}\sqrt{\Omega_m + \Omega_\Lambda a^3}\,. \tag{15}$$

So, we can calculate this integral numerically via Simpson's rule quadrature, and will henceforth refer to that result as

$$\Theta \equiv \text{Simpson}\left(\int_{a_f}^1 c\frac{da}{a^2H}\right)\,. \tag{16}$$

[c] With certain assumptions made about our time units, the assertion that $\tau \ll 1$ can seem absurd; know that our temporal units will ultimately be *program time*, which is essentially the scale factor. See § *4.4* for details.

[d] In the case that the lightcone is being constructed in-situ, this assumption is likely appropriate. Constructing the lightcone as a post-processing task on the output of the simulation, however, operates on much more coarsely resolved time steps (the output snapshots are only a downsampling of the total number of time steps taken in the simulation), and this approximation may be bad. For simplicity, we will retain it throughout this section, but see § *3* for a detailed discussion of nonlinear effects.

[e] A subtle approximation made here; write it as $||\mathbf{r}||^2 = (\mathbf{r}_i + \mathbf{v}_i dt) \cdot (\mathbf{r}_i + \mathbf{v}_i dt)$, expand, drop dt^2 terms, and apply the binomial series to first order.

Turning our attention to the $[a, a_f]$ piece of Eq.(*14*), we see that we certainly cannot numerically evaluate this integral, since we do not know a. In our interest in finding dt, let's instead solve the integral explicitly, and then approximate the result by dropping dt^2 terms. We do this by replacing the limits of integration with Eq.(*11-12*), and again note that $H = \dot{a}/a$;

$$\begin{aligned}\int_a^{a_f} c\frac{da}{a^2 H} &= \int_{a_i+\dot{a}dt}^{a_i+\dot{a}\tau} c\frac{da}{(a_i+\dot{a}dt)\dot{a}} \\ &= \frac{c(\tau - dt)}{a_i + \dot{a}dt} \\ &\approx \frac{c\left[a_i(\tau - dt) - \dot{a}dt\tau\right]}{a_i^2}\end{aligned} \tag{17}$$

With that, we have all of the pieces required to cast Eq.(*8*) in terms of quantities available in our simulation output. The left hand side of Eq.(*8*) is replaced with Eq.(*16-17*), while the right hand side is replaced with Eq.(*13*):

$$\int_{a_{\text{event}}}^{1} c\frac{da}{a^2 H} = ||\mathbf{r}|| \tag{8}$$

$$\downarrow$$

$$\Theta + \frac{c\left[a_i(\tau - dt) - \dot{a}dt\tau\right]}{a_i^2} = ||\mathbf{r}_i|| + \frac{dt(x_i\dot{x}_i + y_i\dot{y}_i + z_i\dot{z}_i)}{||\mathbf{r}_i||} \tag{18}$$

which we can then finally solve for dt:

$$\begin{aligned}\Theta + \frac{c(a_i\tau)}{a_i^2} - ||\mathbf{r}_i|| &= dt\left[\frac{(x_i\dot{x}_i + y_i\dot{y}_i + z_i\dot{z}_i)}{||\mathbf{r}_i||} + \frac{c(a_i + \dot{a}\tau)}{a_i^2}\right] \\ &\downarrow \\ dt &= \left[\Theta + \frac{c(a_i\tau)}{a_i^2} - ||\mathbf{r}_i||\right] \Bigg/ \left[\frac{(x_i\dot{x}_i + y_i\dot{y}_i + z_i\dot{z}_i)}{||\mathbf{r}_i||} + \frac{c(a_i + \dot{a}\tau)}{a_i^2}\right] .\end{aligned} \tag{19}$$

We can now solve for $t_{\text{cross}} = t_i + dt$ for all particles in any snapshot of the simulation using extrapolation with first order approximations.

If we compute dt in this way for a given particle at snapshot i, and the result is that $t_{\text{cross}} > t_f$ (or equivalently $dt > \tau$), then we discard the recovered dt, since we will almost certainly obtain a better answer if we extrapolate from the next snapshot f instead. If $t_i < t_{\text{cross}} \leq t_f$, then we save it (an entry for this particle is created in the lightcone output). After doing this for every particle in snapshot i, we then advance to snapshot f and repeat the process again. After advancing through all snapshots, the lightcone will be filled.

2.2 Particle Duplication

Extrapolating particle positions from their velocities at snapshot i to find their lightcone-crossing time will work in general, albeit potentially inaccurate because of our approximations. Upon more careful consideration, however, particle extrapolation suffers from the particularly serious ailment of outputting the same particle at two or more times in the final lightcone.

To understand why this "particle duplication" happens, consider the diagram shown in Figure *2*. Here, we have a particle whose velocity $\mathbf{v}_i$ at snapshot i suggests it to be moving in the $+x$ direction (in the diagram, x is representing all of space rather than a Cartesian component). Extrapolating it's position places it outside of the lightcone at time t_f of the next snapshot, and estimates it to have crossed the lightcone at time $t_i + dt$. This result is saved and seen in the lightcone output. What *actually* occurred is that the particle underwent some kind of interaction, which is temporally unresolved in the downsampled snapshot outputs, that altered it's trajectory. Therefore, it's real path diverges significantly from it's extrapolated estimate. In the extreme case as shown in Figure *2*, the particle won't actually cross the lightcone at all until some later time $t_{\text{cross}} > t_f$, which will also be saved in the output of some later snapshot. The net effect is that the extrapolation method described in § *2.1* places this particle on the lightcone *twice*[f], meaning the observer sees the same object at two different redshifts. This of course should be impossible without the object exceeding the speed of light. The converse situation is also true, and equally as serious, in which certain particles *never* appear in the lightcone (the true path crosses the lightcone, while the extrapolated estimate does not).

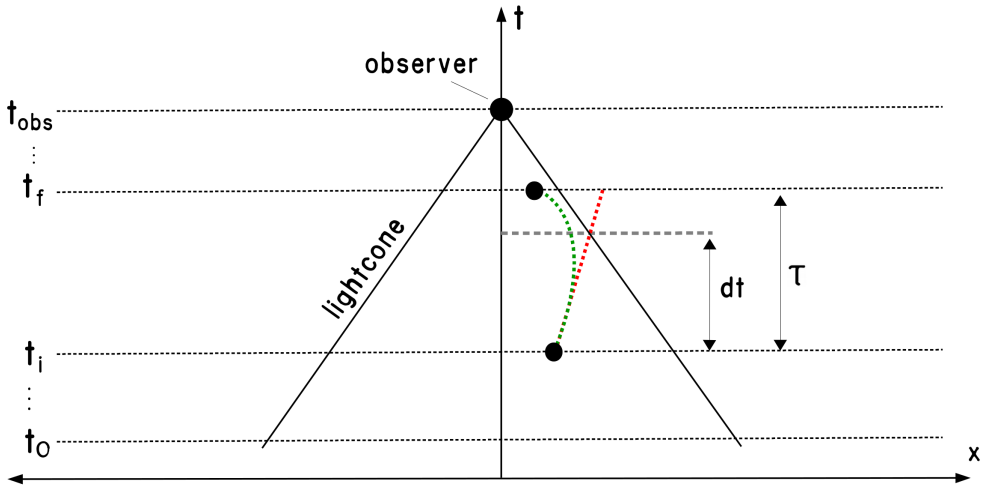

Figure *2*: A schematic showing the origin of particle duplication issues with the extrapolation approach. The vertical axis is time, and the horizontal is space. Time increases upward. The particle is found at snapshot i with an initial velocity pointing in the $+x$ direction. Extrapolating the position based on this velocity yields the estimated trajectory in red, which crosses the lightcone at time $t_i + dt$. The particle's true path is shown in green, which experienced some kind of interaction shortly after the time t_i. Since the particle is actually found *inside* the lightcone at time t_f, it must end up crossing the lightcone again[g], thus being seen by the observer at two different times. The path shapes are greatly exaggerated— no simulation particles reach such relativistic speeds.

[f] Or even more than twice, if the duplication artifact appears more than once for a particular particle.

[g] Unless the particle then also experiences the opposite extrapolation artifact in which the lightcone crossing is lost altogether.

This duplication error is significant, and effects a few percent of particles, with an even worse effect on halo lightcone catalogs near *5-10*%. The obvious fix for this is to calculate lightcone-crossing times by particle interpolation rather than extrapolation (in Figure *2*, a new estimated path connecting the two particle positions would be a much better representation of the true trajectory in green). This approach was not immediately implemented due to the simplicity of the extrapolation method, and lack of foresight into the presently discussed bugs of which we are now well acquainted. Building the lightcone in-situ makes the extrapolation method especially appealing since no knowledge of the upcoming timestep is needed. In any case, particle position interpolation is a necessary requirement for clean and reliable lightcone catalogs, which is described next in § *2.3*.

2.3 Particle Interpolation

We have taken a very simple approach to the position-interpolation improvement, so that we would avoid having to perform significant code refactoring (with respect to the original implementation described in § *2.1*). Consider again a particle at position $\mathbf{r}_i$ (velocity $\mathbf{v}_i$) at snapshot i, and position $\mathbf{r}_f$ (velocity $\mathbf{v}_f$) at snapshot f. We first find an "equivalent linear velocity"– a constant velocity such that would move the particle from $\mathbf{r}_i$ to $\mathbf{r}_f$ in time $\tau = t_f - t_i$:

$$\mathbf{v}_{\text{linear}} = \frac{\mathbf{r}_f - \mathbf{r}_i}{\tau} \,. \tag{20}$$

This of course requires that, when working on a particular snapshot i in the course of the lightcone construction, we must also load in snapshot f (not necessary in the original implementation). Once we have done this, the result of Eq.(*20*) can simple be fed into the framework of § *2.1*, where v_{linear} would replace any occurrence of v_i. In this case, Eq.(*10*), which was

$$\begin{aligned} ||\mathbf{r}|| &= ||\mathbf{r}_i + \mathbf{v}_i dt|| \\ &= \left[(x_i + \dot{x}_i dt)^2 + (y_i + \dot{y}_i dt)^2 + (z_i + \dot{z}_i dt)^2\right]^{1/2} \,, \end{aligned} \tag{10}$$

is exactly a linear interpolation. Eq.(*13*) is then the approximated interpolated position which allows us to solve for dt, and the rest of the procedure is unchanged.

3 Nonlinear Effects

Recall the assumption made in § *2.1* that $0 \leq dt \leq \tau \ll 1$. This assumption allowed us to drop any higher order dt terms, and enabled the derivation of an explicit expression for the lightcone crossing time dt as given in Eq.(*19*). If dt^2 and higher terms are retained (approximate equalities in Eq.(*13*) and Eq.(*17*) will not hold), then the problem turns out to have no analytical solution. If we wish to include such nonlinearities in our answer, then we must implement an iterative solver for the crossing time.

Eq.(*8*), which was the condition for finding an object on the lightcone given its space-time coordinates, was rewritten using quantities specifically exposed by our simulations in Eq.(*18*), and included the $dt \ll 1$ approximation. Rewriting this without

imposing the approximation gives

$$\int_{a_{\text{event}}}^{1} c\frac{da}{a^2 H} = ||\mathbf{r}|| \tag{8}$$

$$\downarrow$$

$$\Theta + \frac{c(\tau - dt)}{a_i + \dot{a}dt} = \left[(x_i + \dot{x}_i dt)^2 + (y_i + \dot{y}_i dt)^2 + (z_i + \dot{z}_i dt)^2\right]^{1/2} , \tag{21}$$

where Θ is defined in Eq.(*16*). As stated above, this form has no solution for dt which does not require iterative methods to evaluate[h]. Thus, we employ the Newton-Raphson root-finding routine on the function

$$f(dt) = ||\mathbf{r}_i + \mathbf{v}_i dt|| - \left(\Theta + \frac{c(\tau - dt)}{a_i + \dot{a}dt}\right) . \tag{22}$$

Note that we've moved back to vector notation for brevity. Newton's method converges (to desired tolerance) on the root of a general function $g(x)$, after being given an initial guess near the root, by the iterative perscription

$$x_{n+1} = x_n - \frac{g(x_n)}{g'(x_n)} , \tag{23}$$

with the subscript n denoting the iteration. Often, the drawback in using this method is the requirement of having the derivative $g'(x)$. If the function $g(x)$ is not differentiable, further complexity can be added by approximating $g'(x)$, reducing the convergence rate to something akin to the secant method. Fortunately, Eq.(*22*) is analytically differentiable as

$$\frac{\partial}{\partial dt} f(dt) = \frac{\left[dt^2(\mathbf{r}_i^2 \cdot \mathbf{v}_i^2)\right]^{1/2}}{dt} + \frac{c(a_i + \dot{a}\tau)}{a + \dot{a}_i dt} . \tag{24}$$

A few words on convergence and stability; there are several situations in which the Newton-Raphson method will fail to converge, or even diverge. These issues are mitigated to some degree by the fact that our root must lie in the specific range $(0, \tau]$ of the timestep under consideration, and even more so by the fact that $f(dt)$ should always be a very smooth function of multiplicity *1*. This claim is supported by the form of Eq.(*22*); the first term is an expression of the motion of a simulation particle/object, which is negligible when compared to the speed of light (the "motion" of the lightcone-surface). The factor involving c in the second term, which is first order in dt, then, dominates the behavior of $f(dt)$. Under these conditions, our root finding method should always be sufficiently stable, and offers quadratic convergence.

After applying Eq.(*22*) and (*24*) to the Newton-Raphson method of root finding, we will obtain an evaluation of dt that is free from any errors that may have been introduced by the linear approximation $dt \ll 1$ (to order of the root finding accuracy).

[h] That is to say, no analytic solution is known to be possible, and standard CAS packages are unable to produce a result which does not contain non-elementary functions requiring numerical evaluations.

4 Implementation

This section will describe where and how, in HACC, each of the components of the lightcone construction (§ *1* - § *3*) are performed. What follows is written with frequent reference to the structure of the HACC framework and its important classes, though familiarity is not necessarily assumed, on the part of the reader. Special attention will be given to discussing the units of various quantities involved, which proves to be a nuanced and sub-optimally documented ingredient of the codebase. If one finds difficulty in understanding, or developing, any lightcone-relevant codes such as those described here, time should be afforded for careful dimensional analysis.

§ *4.1* introduces the lightcone driver and gives an overview of the input parameters and options. § *4.2* gives a detailed look at the problem geometrically, explaining the details of simulation box duplication, and § *4.3* describes the location of all components of the solver. § *4.4* makes sense of the units of the important quantities in the problem, and § *4.5* finally describes all output products.

4.1 Parameters & Options

The source files for this project are located in `/hacc/nbody`. The driver for the interpolation scheme (§ *2.3*) is at `/nbody/simulation/driver_lc_interpolation.cxx`, and its `main()` function takes six arguments:

1.	`<paramName>`	path to a HACC `param` file
2.	`<outName>`	output file path
3.	`<inName>`	input file path for snapshot i particle data
4.	`<stepNumber>`	step number i
5.	`<nextInName>`	input file path for snapshot f particle data
6.	`<nextStepNumber>`	step number f

Table *1*: Input parameters for the lightcone driver's `main()` function

Likewise, the driver for the original extrapolation code (§ *2.1*) is found at `/nbody/simulation/driver_lc.cxx`, and takes the same arguments as listed above, excluding `<nextInName>` (finding lightcone crossing times using extrapolation does not require knowledge of a particle's position at step f). The remainder of this section will apply equally to either the approaches of § *2.1* or § *2.3* unless otherwise noted.

Running the driver code begins by initializing MPI, and then creating an instance of the `MC3Options` class. This object is known to the driver as `options`, and is the interface through which run parameters are stored, set, and retrieved. The state of the `options` object is modified, with respect to the default constructor, by the content of a `Basedata` object in charge of reading the parameter file (argument *1* in Table *1*), which is expected to have been passed by the caller of the lightcone driver. Valid parameter file options most relevant to the building of lightcones using this code are given in Table *2* below.

At this point, any reader that finds the content of Table *2* to be clear may decide to skip to § *4.3*. Otherwise, § *4.2* gives a very detailed description of the geometry of the problem we are trying to solve, which many of the options in Table *2* concern themselves with.

Table *2*: Parameter file options most relevant to lightcone construction

	Option	Type	Description
1.	`LCX`, `LCY`, `LCZ`	`float`	The cartesian `x`, `y`, or `z` position, respectively, of the lightcone origin (observer) in physical Mpc/h
2.	`LCAllOctantsX`, `LCAllOctantsY`, `LCAllOctantsZ`	`int` [i]	Whether or not to generate lightcone output in both sky octants along the cartesian `x`, `y`, or `z` directions, respectively [j]
3.	`LCReplicants`	`int`	The number of simulation boxes to replicate along one axis, in the positive direction, not including the initial volume [k]
4.	`LCRedshift`	`float`	The extent of the lightcone specified as a redshift. This value is used to determine `LCReplicants` automatically if `LCUseRedshift` is `TRUE`
5.	`LCUseRedshift`	`bool`	Whether or not to allow the driver to calculate `LCReplicants` itself, given the value of `LCRedshift` [l]
6.	`LCAutoRedshift`	`bool`	Whether or not to allow the driver to automatically set `LCRedshift` to be the redshift of the input step, with a *10*% padding, given that `LCUseRedshift` is also `TRUE` [m]
7.	`LCRotate`	`int`	Whether or not to apply random rotations to simulation box replications
8.	`LCRotateSeed`	`int`	Seed for random box rotations
9.	`LCFull`	`int`	Whether or not to use the full input dataset (as given by parameter *3.* in Table *1*). If `FALSE`, masking is applied to remove undesired particles/objects by some criteria.
10.	`LCSolverNonlin`	`bool`	Whether or not to use the nonlinear lightcone crossing-time solver (as described in § *3*)
11.	`LCMinMass`	`float`	The minimum mass that a simulation object must be to be included in the lightcone output [n]
12.	`LCWriteOrigPos`	`int`	Whether or not to write out the original object positions (from the base snapshot), rather than only the extrapolated or interpolated positions found at the lightcone crossing time.

[i] Several of these options are present in HACC as `int` types, though their descriptions make it clear that they are meant to convey boolean quantities. In this case, the possible values of the option are `[0,1]`, and this should be clear by context.

[j] If the `x`, `y`, and `z` versions of this option are all set to `FALSE`, then the lightcone generated is one octant of the sky. Having only one of these options set to `TRUE` will produce a quarter sky, etc., and all of them being set to `TRUE` will output a full-sky lightcone.

[k] Though this option specifies the number of replications along one axis, the box replication actually occurs along all three cartesian axes. It also occurs in both the negative *and* positive direction for each axis *k* where `LCAllOctantsK = TRUE`. Thus, finding the total number of replications requires a calculation, which is described in § *4.2*.

[l] If this option is `TRUE`, then the param file entry for `LCReplicants` is ignored. Otherwise, `LCRedshift` is ignored

[m] Therefore, if both this option and `LCUseRedshift` are set to `TRUE`, the decision of the number of simulation box duplicates is fully automated, and the param file entries for both `LCReplicants` and `LCRedshift` are ignored

[n] This option is relevant in the case that the lightcone is being built from halo datasets, or some other object that varies in mass, rather than gravity-only mass particles.

4.2 Geometrical Considerations

Before touring the rest of the code, we should discuss how best to visualize the geometry of the problem in the context of cubic cosmological simulation volumes. This discussion will highlight the purpose and usage of most of the options in Table *2* that may appear somewhat abstruse.

An important characteristic of numerical simulations that have guided our problem-solving thus far is that information is only known at discrete time snapshots. Our approach, conclusions (most prominently Eq.(*19*)), and notation all acknowledge this limitation. However, we have, thus far, implicitly assumed that we have infinite spatial information, or at least that we have spatial information out to the furthest possible extent of our lightcone. Of course, this is not true; the largest simulations volumes that have been run are expanding cubes whose comoving side-lengths are in the neighborhood of *4*Gpc/h. If we place an observer at the corner of one such simulation volume, to what extent can our lightcone reach? The lightcone surface is a spherical object centered on the observer, and it's radius could not exceed, in this case, a comoving distance *4*Gpc/h— a distance which light will cross in a reasonable amount of time, and, assuming a WMAP*7* cosmology, allows our observer to see to a maximum redshift of $\sim$*2.3*.

On top of this distance limitation, we have also introduced a direction limitation; situating our observer in the corner of the box maximizes the radius of the lightcone, but only allows one octant of the sky to be seen. Placing an observer, instead, in the center of the box allows for a full-sky lightcone to be generated, but in that case the lightcone extent would be reduced to redshift $\sim$*0.8*. These issues are of course only more constraining with smaller simulation volumes.

Surely we would like to fill lightcones with simulation particles/objects to any arbitrary redshift we choose, and surely we would also like the option to do so over the entire angular domain of any observer. Our solution is to grow the spatial extent of our simulation output, effectively, by replicating the box many times and "tiling" these replications in space. This idea is shown visually in Figure *3*. For example, if we'd like to fill an all-sky lightcone out to a redshift of *2* (a comoving distance of $\sim$*3.7*Mpc/h given WMAP*7*) with particle output from a *2*Gpc/h simulation, we would replicate the box once in each cartesian direction for a limiting lightcone radius of *4*Gpc/h.

While implementing this replication technique does solve the issues described above, and allows us to generate a lightcone to any arbitrary redshift, it introduces some further complications of its own. If we carelessly replicate the simulation volume many times, the presence of repeating large-scale structure will start to become visually obvious, and inject artifacts into various statistics, e.g. power spectra, redshift distributions, etc. For this reason, the code has been given the capability to randomly rotate each replication by swapping the x, y, and/or z positions and velocities of its particles. This solution, in turn, has yet another side-effect: before rotating the replicated boxes, they are all seamlessly joined to each other due to the periodic boundary condition of each box edge. This property will be lost after randomly-rotating each volume, and discontinuities can be created at box edges (filaments and halos may be clipped).

In short, we are forced to decide between introducing either repeating large-scale structure, or large-scale edge effects, into the lightcone output. The latter has tended to be the less significant problem. These issues are further detailed in Figure *4*, where

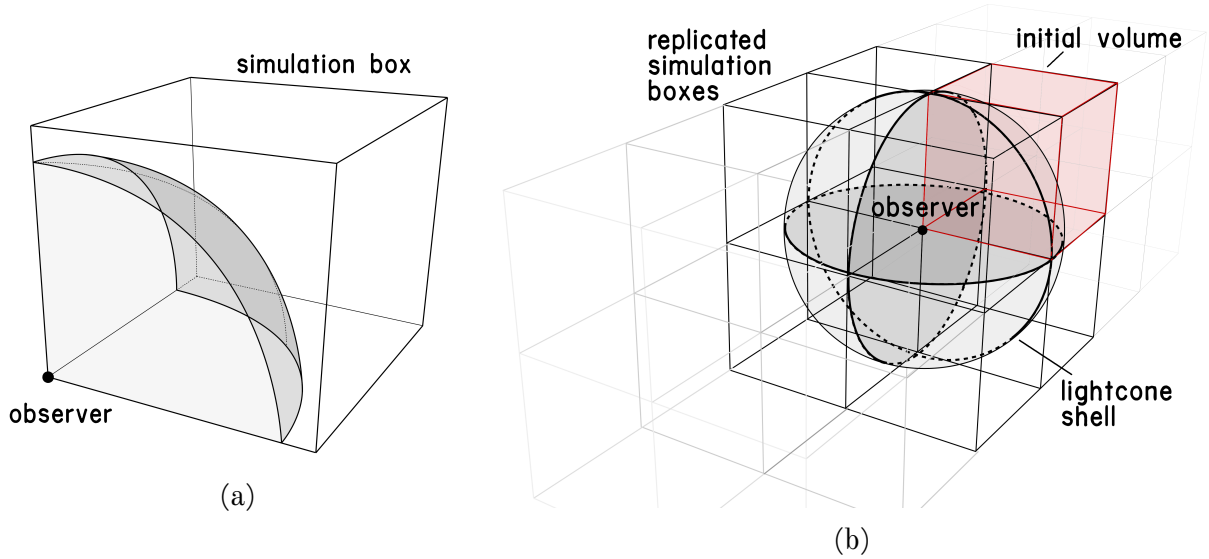

Figure *3*: A comparison of the problem geometry in the case that one simulation volume is used, with the case that the volume is replicated and tiled. **Fig(*3*a)**: A view of a simulation volume containing a lightcone, filling an octant of the sky, being generated with respect to an observer placed at the origin (bottom left corner). The spherical shell is the extent of the lightcone (a surface upon which lie the furthest, and youngest, objects that the observer can see) at some time t. Since the extent of the lightcone has not reached the other side of the box yet (light from the other side of the box has yet to reach the observer), it is true that $t < L/c$, where L is the box length. Notably, we *cannot* generate a lightcone, using this simulation volume, to allow the observer to see events occurring beyond a time $t_{\rm max} > L/c$ into the past[o]. **Fig(*3*b)**: A view of a full-sky lightcone now being generated across a tiling of duplicated simulation boxes. The red volume indicates the initial volume shown in Fig(*3*a). Notice that now we have eight times the sky-coverage, without having had to reduce the radius of the lightcone. We can replicate the simulation boxes as many times as we'd like to produce an arbitrarily large lightcone.

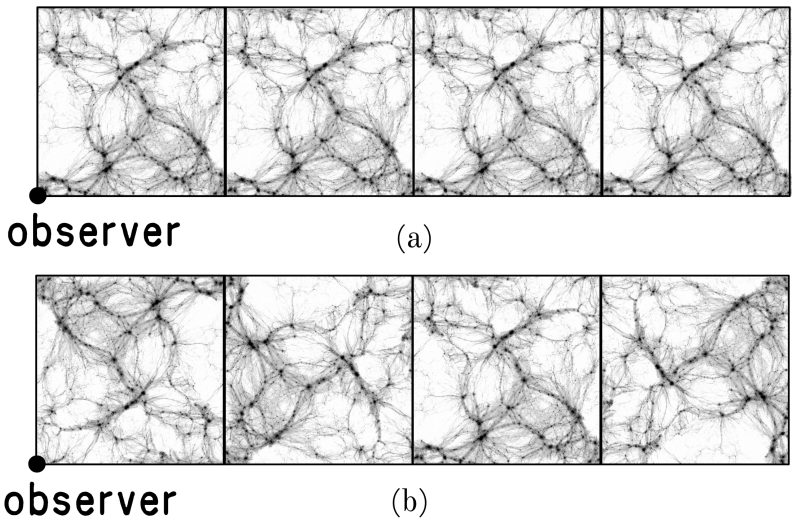

Figure *4*:[p] A comparison of the consequences of simulation box replications, with (Fig(*4*b)) and without (Fig(*4*a)) enabling random rotations. The problem of repeating LSS is clear in Fig(*4*a), as the density along the lower box edge is noticeably higher than the top edge. Repeating structures along this lower edge have very little angular separation in the observer's line of sight, making the effect particularly strong. These issues are cured in Fig(*4*b), though some density discontinuities are obvious (notedly along the vertical edge separating the initial and first replicated volumes).

[o] More accurately, if we consider cosmological dynamics as discussed in § *1*, it is true that setting $t_{\rm obs} = t$ in Eq.(*6*) will yield a distance of $||\mathbf{r}|| < L$, and that we cannot enable the observer, using this simulation volume, to see events occurring more than a time $t_{\rm max}$ into the past, such that setting $t_{\rm obs} = t_{\rm max}$ in Eq.(*6*) gives $||\mathbf{r}|| = L$.

[p] Credit for the cosmic web image used in the creation of this figure to Benedikt Diemer: http://www.benediktdiemer.com/visualization/images/

the box replications along a single axis are visualized.

With all of this in place, we are equip to parse the meaning and usage of the options listed in Table *2*. Option *7* in Table *2*, `LCRotate`, is simply a boolean value which turns the box replication rotation, as described above, on or off. Option *8*, `LCRotateSeed` is self-explanatory, though it is worth taking a moment to note that the random number generator at work here, given a constant seed, gives reproducible results in general when running in parallel; this means that the MPI topology is free to change without effecting the spatial configuration of the output (for instance, one may desire to run a full particle lightcone to obtain shear maps, and also a corresponding galaxy lightcone, where the galaxy lightcone requires many fewer nodes to compute in a reasonable amount of time. The rotations of each simulation box replication would be preserved between these two runs).

Options *2-6* in Table *2* ought to be treated more carefully. Option *3* gives the number of box replications desired, *along one axis, in the positive direction*. Options *2* deliver three boolean values to the code which control the sky-coverage of the lightcone (see footnote j). How these options are actually operating on the spatial configuration of the problem are as follows: setting `LCAllOctantsX` to `TRUE` will replicate boxes in *both* the positive and negative x directions, and likewise for the `Y` and `Z` variants of this option. Therefore, in order to actually calculate the number of replications, something like following operation is performed:

```
int NumRepsX = LCReplicants+1 + (LCAllOctantsX ? 1 : 0)*(LCReplicants+1);
int NumRepsY = LCReplicants+1 + (LCAllOctantsY ? 1 : 0)*(LCReplicants+1);
int NumRepsZ = LCReplicants+1 + (LCAllOctantsZ ? 1 : 0)*(LCReplicants+1);
int numBoxesTotal = NumRepsX*NumRepsY*NumRepsZ;
```

Let's dissect the declaration of `NumRepsX`; why is it that each time `LCReplicants` is used (the value of Option *3* defined by the user), it is incremented by one? The leftmost `+1` is adding the initial simulation volume to the total box count, and the rightmost `+1` is to keep the total replicated volume symmetric across the y-z plane. This means that if `LCAllOctantsX` is true, then the number of replications in the x direction is doubled, which makes sense. This all of course holds for the `Y` and `Z` variants of the quantity as well.

The potential clumsiness of the last few paragraphs suggest that a figure would be most helpful at this point, so let's return to our example as stated on page *11*. There, we imagined constructing a full-sky lightcone out to a redshift of *2* (comoving distance of $\sim$*3.7*Mpc/h) using a *2*Gpc/h simulation. We said that this would require one box replication, in each cartesian direction, yielding a volume capable of building, at most, a lightcone with an extent of *4*Gpc/h. Figure *5* shows the x-y plane of the presently discussed situation. Does this plan make sense, given the behavior of Options *2* and *3* described above? It does, if we define these options properly. The problem outline above suggests that we use the option values

```
LCReplicants = 1
LCAllOctantsX = LCAllOctantsY = LCAllOctantsZ = TRUE .
```

Were we only looking to generate an octant of the sky, we would change the latter

three of these options to

```
LCAllOctantsX = LCAllOctantsY = LCAllOctantsZ = FALSE .
```

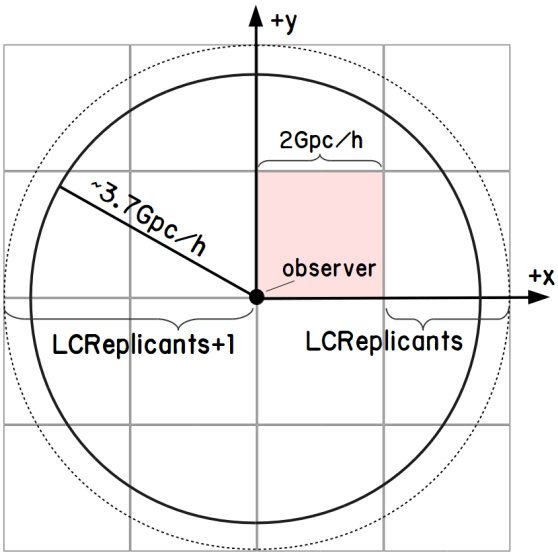

Figure *5*: A visualization of the spatial configuration of the simulation boxes being duplicated in the example problem discussed on page *13*. The initial *2*Gpc/h volume is tinted red. Recall that `LCReplicants` was set to `1` in this example, meaning that one box will be replicated in the *positive* direction of some single cartesian axis. With the added consideration of the initial volume, that makes two box lengths of data from the observer at the origin to the lightcone extent. Since we have specified that we would like to generate a full sky, we replicate two more boxes in the *negative* directions. The extent of the lightcone that we desire to build is shown by the dark circle, at ~*3.7*Mpc/h, and the largest possible lightcone that this box configuration could accommodate is shown as the dashed circle, at ~*4*Mpc/h. In this example, we see *16* total simulation volumes, or *64* total volumes if we also consider those replicated in the *z* direction (omitted from this figure). Solving the expression for `numBoxesTotal` on pg. *13*, given our parameter choices, agrees with this figure.

With all of this said, it is clear that the values of Option *1* in Table *2* (the position of the observer) should really always be that of the origin (`LCX`=`LCY`=`LCZ`=*0*), even though the user *is* allowed to specify otherwise. Perhaps this unattractive feature will be addressed in the future. Finally, as a last consistency check, it should now be easy to understand that the options necessary to generate the configuration shown by the bold-faced boxes in Figure *3*b would be

```
LCReplicants = 0
LCAllOctantsX = LCAllOctantsY = LCAllOctantsZ = TRUE .
```

In summary, the behavior of `LCReplicants` isn't necessarily intuitive— it acts this way so that one need only find the number of box-lengths required to reach the desired lightcone radius, in one dimension, to specify the number of replications (remembering that the initial volume does not count as a "replication").

Finally, we can briefly describe Options *4-6*. If Option *5*, `LCUseRedshift`, is set to `TRUE`, then the code will look toward Option *4*, `LCRedshift` to calculate `LCReplicants` internally. This was implemented as a convenience to the user— since lightcones, as a part of science projects, are usually planned to extend to a particular redshift, rather than a distance, the user would have to manually convert this redshift to a distance under the cosmology of the simulation run, and then find the necessary value of `LCReplicants`. Specifying the redshift of the lightcone instead, via Options *4*&*5*, prompts the code to use Eq.(*16*) to compute the necessary `LCReplicants`, by set-

ting the lower integration bound a_f to `1/(1+LCRedshift)` (recall the discussion about Eq.(*8*) to see why this works).

This *still* leaves work for the user, however, since the lightcone code is run on a snapshot-by-snapshot basis. If a user is running the snapshot output of step *487* ($z \approx 0.02$) through the lightcone solver, as part of an effort to build a lightcone to $z = 3$, then they will cerainly be doing far more calculations than is necessary if `LCRedshift=3` (which will replicate the volume many times). To combat this last complication, Option *6* is offered. If both Option *5* and Option *6*, `LCAutoRedshift`, are set to `TRUE`, then the `LCRedshift`, and hence `LCReplicants`, will be computed internally. This works by simply calculating the redshift of the input snapshot number, plus a *10*% redshift padding, to set the value of `LCRedshift`.

In summary, generating a lightcone using this code offers three distinct parameter usages, as they relate to box replication:

1. `LCUseRedshift`=`LCAutoRedshift`=`FALSE`
   `LCReplicants` = `some integer`
2. `LCAutoRedshift` = `FALSE`
   `LCUseRedshift` = `TRUE`
   `LCRedshift` = `some value`
3. `LCReplicantsLCUseRedshift` = `LCAutoRedshift` = `TRUE`

Use case *1* in the list above will offer full control to the user in specifying the details of the problem configuration. Use case *2* is slightly more convenient, while use case *3* will minimize the calculations demanded from the user, lowering the potential for error, and possibly allow the user to forget most of the present section of this document.

4.3 Code Layout

Let us recall the discussion as left at the end of § *4.1*. After the first few tasks performed by the lightcone driver, we now have MPI initialized, with `MC3Options` and `Basedata` objects storing our run parameters (including those so explicitly treated in § *4.2*). In a moment, two more important objects will be introduced, though I will not subsequently continue to detail every operation and all HACC objects created by the driver, as that is a job better left to the in-line documentation, of which there is plenty. This section will, instead, offer a high level road-map of where the various steps of the problem solution are carried out.

To borrow the notation introduced in § *2.1*, consider running the driver on a timestep bounded by snapshots i and f. The original extrapolation version of the driver (`nbody/simulation/driver_lc.cxx`) continues by reading the snapshot i particle data from a specified `gio` file (parameter *3* in Table *1*). Now, for the two important objects mentioned above; a `ParticleActions` object (henceforth `pa`) is then created, as well as two `TimeStepper` instances for keeping track of z-dependent cosmological parameters at times t_i and t_f. The member functions of `pa` are what perform most of the calculations that we care about in solving for lightcone crossing times, and they are found at `nbody/cpu/ParticleActions.cxx`.

Moving on; if `LCUseRedshift` and/or `LCAutoRedshift` (see Table *2*) are given as `TRUE` by the user, then the function `pa.calcLCReplicants()` is called, which performs

some of the duties laid out in § *4.2*. Ultimately, the particle data is passed to `pa.map1()` , which in turn calls the solver code at `pa.update_lc()`.

`pa.update_lc()` is where most of the magic happens. The box replications are assigned random rotations, positions and velocities are scaled to the proper units (see § *4.4*), and some prerequisite computations are done to prepare for running the solver code. Eq.(*15*), the time-dependent Hubble parameter, is calculated in `pa.Hat()`. Eq.(*16*), a Simpson's rule numerical integration giving the distance from the observer to the lightcone extent at t_f (which the code denotes as A rather than Θ), is found in `pa.calcA()`.

With all of this in place, the particle data can be passed to the solver function, either `pa.calc_dt()` or `pa.calc_dt_nonlinear()`, depending on the value of the option `LCSolverNonlin`. The former of these solver functions implements Eq.(*19*) in § *2.1*, and the latter invokes the Newton-Raphson root finding methon on Eq.(*22*) as explained in § *3*. After this routine has been visited by every particle, in every box replication, the results are written out in `gio` format via `pa.writeLC()`. The specific form and content of this output is described in § *4.5*.

The particle interpolation version of the driver (`driver_lc_interpolation.cxx`) does all of the things discussed thus far, with some additions. In that case, we need particle position information at *both* snapshots i and f, so two `gio` files are now read in (the second given by parameter *5* in Table *1*). For each particle belonging to snapshot i, we use a binary search algorithm to find its location also in snapshot f. With this mapping completed, we may interpolate the particle's position, which is done exactly as described in § *2.3*. If any particle from snapshot i is found to have no match in f, which can happen, then we default the calculation of dt for that particle to the extrapolation method (we do not update its velocity).

4.4 Units

This section, as promised, gives a careful look at the units of the important quantities in this problem, and their transformations. Particle coordinate values are converted between different systems in HACC via the `ParticleCoords` class, whose constructor is called as

```
ParticleCoords(units, scope, periodic, symplectic)
```

where the latter two arguments are expected to be boolean values. The particle data is read in from the input snapshot files by the driver in the same form that they are written out by the simulation, where `units=PHYSICAL, scope=GLOBAL, periodic=true, symplectic=false`. `PHYSICAL` specifies that particle coordinates are given in physical units (Mpc/h for positions, and km/s for velocities) and `GLOBAL` says that those coordinates are given with respect to the global simulation origin (as opposed to the local origin of the MPI rank to which the particle belongs). `periodic` refers to the boundary conditions of the coordinates, while `symplectic` specifies whether or not velocities are multiplied by a factor of a^2.

Now, are these the proper units for our quantities to be in with respect to our solution strategy derived in § *1*-§ *2.3*? The answer is no, and some transformations are going to be required. It should at least be immediately clear that the unit mismatch between

our particle positions and velocities is of no help, since we seek to use those velocities to approximate new positions (either by extrapolation or interpolation). To understand what is about to happen, let's again print the last line of the linear solution, Eq.(*19*):

$$dt = \left[\Theta + \frac{c(a_i\tau)}{a_i^2} - ||\mathbf{r}_i||\right] \Big/ \left[\frac{(x_i\dot{x}_i + y_i\dot{y}_i + z_i\dot{z}_i)}{||\mathbf{r}_i||} + \frac{c(a_i + \dot{a}\tau)}{a_i^2}\right] \ . \tag{19}$$

All of the summands in the numerator on the right hand side of Eq.(*19*) clearly need to have dimensions of length, which is asserted by the $||\mathbf{r}_i||$. The quantity on the left hand side is a time, implying that the units of the denominator on the RHS must have dimensions of length/time. So, we can fix our velocity units to be km/s, and apply conversions to our times and lengths, or vice-verse. The latter is the obvious choice, since we should certainly like to keep our lengths as a comoving unit, and our time units defined with respect to the scale factor a.

τ is the temporal width of the timestep bounded for snapshots i and f (continuing to follow the notation as in § *4.3*), which is supplied in HACC via the `TimeStepper` class as a dimensionless *program time* (henceforth pt), which is some power of the scale factor; pt$=a^\alpha$, where α is typically chosen to be *1*[q]. Given that our distances are in units of Mpc/h, our velocities must be scaled to units of (Mpc/h)/pt, including that of c. Note that occurrences of H and $\dot{a}$ must also undergo this conversion, since their unit definitions contain km/s, as given by the `TimeStepper` and Eq.(*15*).

Carrying out the dimensional analysis of Eq.(*19*) with these changes does indeed give the units of dt as dimensionless program time, which is desired. For further convincing, unit analyses can be performed on Eq.(*20*) and Eq.(*21*) to prove that these dimension choices are also correct for the interpolation routine, and the nonlinear solver.

Now, the conversion described above turns out to be slightly involved. Assuming that we have velocities in km/s, then we need a conversion factor with units of s/(h·pt). Multiplying our velocities by this value should produce the dimensions we are after. We can start by trying to obtain an expression relating physical time to program time; we know that

$$H = \frac{\dot{a}}{a} = \frac{da}{dt}a^{-1} \ , \tag{25}$$

and we can use this to find the operator d/dt:

$$\begin{aligned} aH &= \frac{da}{dt} \\ dt(aH) &= da \\ \frac{d}{dt} &= aH\frac{d}{da} \end{aligned} \tag{26}$$

which is what we stated in Eq.(*7*). We can use the presence of the scale factor a, here, to involve the program time pt$=a^\alpha$ via the chain rule:

$$\frac{d}{da} = \frac{d}{d\mathrm{pt}}\alpha a^{\alpha-1} \ . \tag{27}$$

[q] α, here, is an input parameter given by the user running the simulation. To understand the choice to use a power of the scale factor as the program time variable, see the RRU Simulation and Code Overview.

Combining Eq.(*26*) and Eq.(*27*) leaves us with

$$\frac{d}{dt} = aH\alpha a^{\alpha-1}\frac{d}{d\mathrm{pt}} \,. \tag{28}$$

Eq.(*28*) has a time differential operator on each side, and acting those operators on a generic position r gives us

$$\frac{dr}{dt} = v_{\mathrm{phys}} = aH\alpha a^{\alpha-1}\frac{dr}{d\mathrm{pt}} = aH\alpha a^{\alpha-1}v_{\mathrm{pt}} \tag{29}$$

which makes it clear that our conversion factor must be

$$\frac{1}{aH\alpha a^{\alpha-1}} \tag{30}$$

and must have units of s/pt (since the units of v_{phys} and v_{pt} are km/s and km/pt, respectively). We must not forget that we'd also like our distances in comoving coordinates, so finally we multiply the conversion factor Eq.(*30*) by h/h, where h in the numerator becomes part of the value, and the h in the denominator becomes part of the units (reminded by the italics, or lack thereof). In summary, we learn that we can convert our physical velocities (including c) to dimensions of [comoving distance]/[program time] by multiplying them by

$$\frac{h}{aH\alpha a^{\alpha-1}} \quad \frac{\mathrm{s}}{\mathrm{h}\cdot\mathrm{pt}} \tag{31}$$

and also that, we can't forget, we can also convert our values of H and $\dot{a}$ to the proper units by multiplying by Eq.(*30*). This is precisely what the lightcone solver does; in `pa.update_lc()`, the variable `s_pt` holds the conversion factor obtained from evaluating Eq.(*30*), and likewise the variable `scale_vel` for Eq.(*31*). Any outstanding opaqueness should be able to be cleared via a careful reading of the in-line documentation, and a sufficient understanding of HACC's coordinate conversion utilities (namely, the member functions of `ParticleCoords`, which make use of the various `Domain::grid2phys_()` offerings).

4.5 Output

The output of the lightcone code looks very much like simulation snapshot output; GIO files are written out for each snapshot that the code is run on, with a data row per particle/object. Rows have at most *17* columns, each corresponding to some property of the object, which are enumerated in Table *4*. The structure of the output directory is fully dependent on the nature of the job submission script written by the user (where parameter *2* of Table *1* is defined), though the convention is to end up with something that looks like Figure *6*.

Most of the output quantities described in Table *4* are sufficiently clear, though two of them, `rotation` and `replication` require further explanation. The former gives

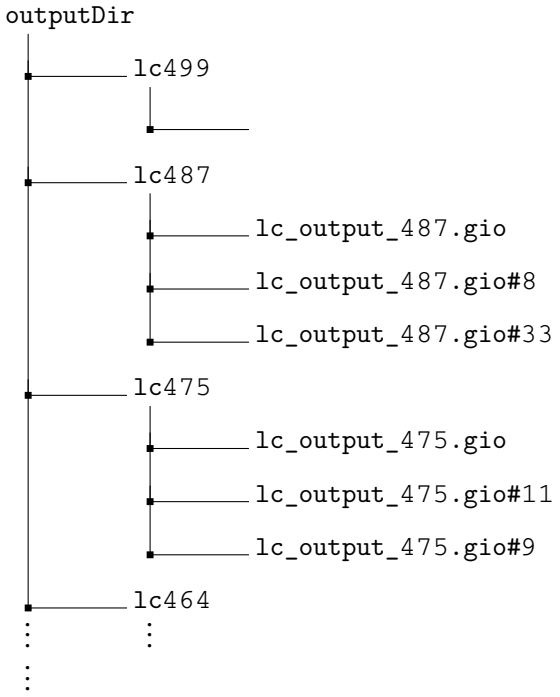

Figure *6*: The conventional output directory structure resultant from running the lightcone solver on at least the first four snapshots of some simulation which ran through a total of *500* full timesteps. The hashed (#) filenames indicate portions of the output that were written by different MPI ranks; only the unhashed filename (the "header") need be passed to GIO in order to read the output. Notice that step *499* has no lightcone output—this makes sense; step *499* corresponds to $z = 0$, which is the time of observation. The lightcone volume, then, at step *499* is of course zero (setting $a_{\text{event}} = 1$ in Eq.(*8*) gives $||\mathbf{r}|| = 0$, telling us that the only events that the observer can see at $z = 0$ are those that have zero spatial separation from them self).

an identifier, for each object, denoting the coordinate transformation that was applied to the simulation replication which hosts the object. This identifier is an integer i where $i \in [0, 5]$. The meaning of those values are as follows:

Table *3*

value of `rotations`	0	1	2	3	4	5
transformation applied	none	$x \leftrightarrow y$	$y \leftrightarrow z$	$x \leftrightarrow y$, $y \leftrightarrow z$	$z \leftrightarrow x$	$x \leftrightarrow y$, $z \leftrightarrow x$

where the "swap operator" $\leftrightarrow$ acting on two dimensions n and m is $n \leftrightarrow m = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} n \\ m \end{pmatrix}$, and a comma between two of these transformations means "followed by".

The latter output product, `replication`, is again some integer identifier for each object, this time specifying the box replication to which the object belongs. This is useful information to have because, for example, one may find multiple particles , let's call them a and b, with the same `id` in some lightcone output. If a resides in the initial simulation volume, while b resides in some distant replication of the volume, then all is well (a and b are effectively different objects)[r]. If a and b reside in the *same* volume, however, then that would be indicative of a bug of the § *2.2* variety. In general, someone doing analysis with lightcone catalogs may need this information.

So, we give each replication a unique *32*-bit integer identifier, within which is packed it's spatial coordinates in units of box-lengths. Each coordinate (x, y, z) is allotted ten of these bits (*2* bits leftover), with the specific anatomy of the value shown in

[r] in fact, one should expect exactly as many duplications of particle a as there are simulation boxes, that is, `numBoxesTotal` as computed on pg. *13*. The values of `replication` for each of these particles should be unique.

Figure *7*. Note that, under this scheme, the initial volume containing the observer is *not* at (*0*,*0*,*0*). Rather, the origin of this setup is at the positive corner replication (for example, the top-right box in Figure *5*), which puts the initial volume at $x = y = z =$ `LCReplicants`. This is done to avoid having to bit-shift negative signed integers.

Figure *7***:** The anatomy of the *32* bit integer, `replication`, as found in the lightcone output. The first three sets of ten bits, starting from the left, are the x, y, and z positions of the replication, respectively, in units of box lengths. In this example, the considered replication is found at (*2*, *1*, *1*). There are two unused trailing bits.

`replication =` $\underbrace{\texttt{0000000010}}_{x}$ $\underbrace{\texttt{0000000001}}_{y}$ $\underbrace{\texttt{0000000001}}_{z}$ `00`

One can therefore easily extract the coordinates of each box replication by bit shifting and masking:

$x =$ `replication >> 20`
$y =$ `(replication >> 10) & 0x3ff`
$z =$ `replication & 0x3ff`

It is perhaps worth noting that, since each box coordinate is allowed a maximum of ten bits, we must have `LCReplicants` $\leq$ *1024* for an octant sky, and `LCReplicants` $<$ *512* for a full-sky lightcone. This is no problem, as our number of replications will never be nearly this large; even with a small simulation volume of *256*Mpc/h, a lightcone built to utilize all *512* box replications along it's radial axis would reach far beyond the redshift limits of typical simulation output (where $z \leq 200$).

Table *4*: The output properties for each particle/object that passed through the lightcone solver

	Output Column	Type[s]	Description
1.	`x`, `y`, `z`	`float`	The cartesian `x`, `y`, or `z` position, respectively, of the object at the time $t_i + dt$ (given by either interpolation, § *2.3*, or extrapolation, § *2.1*) in comoving Mpc/h.
2.	`vx`, `vy`, `vz`	`float`	The `x`, `y`, or `z` velocity components, respectively, of the object at the time $t_i + dt$ (given as v_{linear} if interpolation was used (§ *2.3*) or as v_i if extrapolation was used (§ *2.1*)) in physical km/s.
3.	`id`	`int`	The id of the object as found in the input object catalog (perhaps particle ids coming from snapshot files, or halo tags coming from halo catalogs, etc.).
4.	`a`	`float`	the scale factor $a(t)$ at the time the object crossed the lightcone, $t = t_{\text{cross}} = t_i + dt$.
5.	`step`	`int`	The simulation snapshot from which this object originated (where it's position at time t_{cross} was approximated from).
6.	`rotation`	`int`	An identifier giving the coordinate rotation that was applied to the simulation box replication in which this particle resides. Possible values are *0-5*, which are discussed in the text.

Continued on next page

Table *4* – *Continued from previous page*

	Output Column	Type[s]	Description
7.	`replication`	`int`	A unique identifier specifying which simulation box replication this object resides in. A method for extracting information about the spatial position of this box replication identifier is given in the text.
8.	`mask`	`int`	A *16* bit mask, where each bit represents the state of some flag relevant to the simulation run. In the lightcone output, this value is simply set to *1* for all particles, though it could conceivably be used to specify flags that may be useful.
9.	`phi`	`float`	The scalar potential used by the gravity solver during the simulation run. This value is not useful in lightcone outputs, and is set to *0* for all entries. The column is retained to maintain consistency with the form of particle snapshot outputs.
10.	`mass`	`float`	The mass of the object, in the case that the code was run on a halo catalog or merger tree.
11.	`origX`, `origY`, `origZ`	`float`	The cartesian `x`, `y`, or `z` position, respectively, of the object at the time t_i (the objects original position in the input catalog snapshot before being permuted by the lightcone solver) in comoving Mpc/h.

[s]The types listed here are also only conventional, and can be changed by the user. Positions and velocities, for example, can alternatively be given as `doubles`. Look toward `common/HaccTypes.h` for some relevant type declarations.

5 Too Long; Didn't Read

Assume an observer located at the spatial origin of a comoving coordinate system. Any event located on this observer's past-lightcone will satisfy the condition

$$\int_{a_{\text{event}}}^{1} c\frac{da}{a^2 H} = ||\mathbf{r}|| \,. \tag{8}$$

where a is the scale factor, $H = \dot{a}/a$ (the Hubble parameter), and $\mathbf{r}$ is the displacement vector from the observer to the event. Refer to § *1* for a full derivation. Now, consider such an event in the context of cosmological simulation output; a particle crosses the lightcone somewhere between snapshot i and snapshot f (constituting a timestep of width $\tau = t_f - t_i$) at an unknown time t_{cross}. We define a quantity dt such that $dt = t_{\text{cross}} - t_i$. If we assume that $dt \ll 1$[t], then we can obtain the solution

$$dt = \left[\Theta + \frac{c(a_i\tau)}{a_i^2} - ||\mathbf{r}_i||\right] \Bigg/ \left[\frac{(\mathbf{r}_i \cdot \dot{\mathbf{r}}_i)}{||\mathbf{r}_i||} + \frac{c(a_i + \dot{a}\tau)}{a_i^2}\right] \,. \tag{19}$$

where Θ is the Simpson's-rule numerical evaluation of a definite integral similar to that given above in Eq.(*8*):

$$\Theta \equiv \text{Simpson}\left(\int_{a_f}^{1} c\frac{da}{a^2 H}\right) \,. \tag{16}$$

The ultimate utility of the lightcone code is to compute dt for every particle at every step of the simulation (find each particle's lightcone-crossing time). The solution presented in Eq.(*19*) is derived by extrapolating the position of a given particle, using it's velocity, from time t_i (known) to time $t_i + dt$ (unknown). If it is found, after solving for dt, that $0 < dt \leq \tau$, then the result is saved. If rather $dt > \tau$ or $dt < 0$, the result is ignored (the first case means we will almost certainly obtain a better estimate of dt if we instead extrapolate from time t_f during the next step of the lightcone construction, and the second, that the particle should have already crossed the lightcone at some earlier time). This procedure is discussed at length in § *2.1*.

For a cleaner result, we can interpolate the position of the particle between snapshot i (t_i) and snapshot f (t_f), rather than extrapolating. This simply requires that we replace the velocity $\dot{\mathbf{r}}_i$ in Eq.(*19*) with the equivalent linear velocity

$$\mathbf{v}_{\text{linear}} = \frac{\mathbf{r}_f - \mathbf{r}_i}{\tau} \,, \tag{20}$$

as more thoroughly described in § *2.3*. Finally, we can obtain a higher order solution by relaxing the assumption that $dt \ll 1$, though this prohibits an analytical solution for dt as given in Eq.(*19*). In this case, we preform a Newton-Raphson root-finding method on the function

$$f(dt) = ||\mathbf{r}_i + \mathbf{v}_i dt|| - \left(\Theta + \frac{c(\tau - dt)}{a_i + \dot{a}dt}\right) \,. \tag{32}$$

[t] This approximation is allowed in the case that we use a normalized time unit, such as the scale factor $a(t)$, or more specifically a power of a called the program time; a^α

to find *dt*. Refer to § *3* for further explanation.

This problem is implemented in HACC in `nbody/cpu/ParticleActions.cxx`, and driven by `nbody/simulation/driver_lc.cxx`, or `driver_lc_interpolation.cxx`, for the extrapolation (§ *2.1*) and interpolation (§ *2.3*) approaches, respectively. The driver's `main()` function expects at most six arguments, as given below in § *4.1*, Table *1*, which is shown below (where only the interpolation driver requires argument *5*). There are also many other run parameters that can be set by the user in a `.param` file, several of which are ultimately crucial to the structure of the lightcone output. Not all of these parameters will be visited in this summary section, but are given in Table *2*, and discussed in high detail in § *4.1*. Briefly, those parameters which are the most influential to the spatial configuration of the problem are `LCAllOctants`, `LCReplicants`, and `LCRotate`. The purpose of these three options can be understood under the follo-

1.	`<paramName>`	path to a HACC `param` file
2.	`<outName>`	output file path
3.	`<inName>`	input file path for snapshot i particle data
4.	`<stepNumber>`	step number i
5.	`<nextInName>`	input file path for snapshot f particle data
6.	`<nextStepNumber>`	step number f

Table *1*: Input parameters for the lightcone driver's `main()` function

wing considerations: we are limited in the extent to which any observer's lightcone can reach (how far back in time they can see) by the dimensions of our simulations volume. For example, if we are interested generating the observable universe to a redshift of $z = 3$, then our observer's past-lightcone will be a spherical object whose radius is ~*4.6*Mpc/h, assuming a WMAP*7* cosmology. To get around the fact that we almost never deal with simulations larger than *4*Mpc/h (and often much smaller) we decide to replicate and tile the box in space to cover the volume that we require (as thoroughly detailed in § *4.2*, and seen in Figure *3*).

The option `LCReplicants` (Table *2*, option *3*), then, sets the number of simulation boxes required to reach the extent of the lightcone in one-dimension, not including the initial volume. `LCRotate` (Table *2*, option *7*) is a boolean value turning on or off the functionality of applying random rotations to each of these replications, which intends to avoid regular repetition of large-scale structures. Finally, `LCAllOctants` (Table *2*, options *2*) allows the user to specify the angular domain of the observer, from octant to full-sky. Again, more details are offered in § *4.1*-§ *4.3*. Those seeking to run the code and build lightcones should at least read through Table *2*.

The output of the presently discussed code is given in Table *4*, immediately preceding this summary section, with finer details being discussed in the text of § *4.5*. In summary, the output includes the following: the scale factor corresponding to the calculated lightcone crossing time, $t_{\text{cross}} = t_i + dt$; the approximated positions of each object at the time t_{cross}; velocities (also approximated in the case that interpolation was used); and ids. Also written out is information on the location and rotation of the simulation box replication in which each object is found, and other optional and/or incidental data products.