Draft version September 17, 2017
Preprint typeset using LaTeX style emulateapj v. 12/16/11

CUBE: A MEMORY-LITE COSMOLOGICAL N-BODY ALGORITHM

Hao-Ran Yu[1,2], Ue-Li Pen[2,3,4,5], Xin Wang[2]
[1]Tsung-Dao Lee Institute, Shanghai Jiao Tong University, Shanghai, 200240, China
[2]Canadian Institute for Theoretical Astrophysics, University of Toronto, Toronto, ON M5H 3H8 Canada
[3]Dunlap Institute for Astronomy and Astrophysics, University of Toronto, Toronto, ON M5S 3H4, Canada
[4]Canadian Institute for Advanced Research, CIFAR Program in Gravitation and Cosmology, Toronto, ON M5G 1Z8, Canada
[5]Perimeter Institute for Theoretical Physics, Waterloo, ON N2L 2Y5, Canada
Draft version September 17, 2017

Abstract

Cosmological large scale structure N-body simulations are computation-light, memory-heavy problems in supercomputing. Traditional N-body simulation algorithms use at least 24-byte memory per particle, of which six 4-byte single precision floating point numbers keep track phase space coordinates of each particle. Here we present the algorithm and accuracy of a new parallel, memory-lite, Particle- Mesh based N-body code, where each particle can occupy as low as 6-byte memory. This is accomplished by storing relative position and relative velocity of each particle, in the format of 1-byte-integer, respect to their averaged value of a mesh-grid. The remaining information is given by complimentary density and velocity fields, which are negligible in memory space, and proper ordering of particles, which gives no extra memory. Our numerical experiments show that this integer based N-body algorithm provides acceptable accuracy compared to traditional algorithm in cosmological simulations. This significant lowering of memory-to-computation ratio breaks the bottleneck of scaling up and speeding up large cosmological N-body simulations on multi-core and heterogenous computing systems.

1. INTRODUCTION

N-body simulation is a powerful tool to solve the highly nonlinear dynamic problems (Hockney & Eastwood 1988). It is widely used in cosmology to model the formation and evolution of the large scale structure (LSS). With the fast development of parallel supercomputers, we are able to simulate a system of more than a trillion (10^{12}) N-body particles. To date the largest N-body simulation in application is the "TianNu" simulation (Yu et al. 2017; Emberson et al. 2017) run on the TianHe-2 supercomputer by cosmological simulation code `CUBEP3M` (Harnois-Déraps et al. 2013). It uses nearly 3×10^{12} particles to simulate the cold dark matter (CDM) and cosmic neutrino evolution through the cosmic age.

The N-body simulations use considerable amount of memory, because the phase space coordinates (x, y, z, v_x, v_y, v_z) of each N-body particle must be stored as, at least, six single precision floating point numbers (24 bytes). Contrarily, their computing workload can be alleviated by many algorithms, like Particle-Mesh [cite] and Tree [cite], scale as $o(N \log N)$ or even $o(N)$. On the other hand, modern supercomputer systems use multi cores, many integrated cores (MIC) and even densely parallelized GPUs, bringing orders of magnitude higher computing power, whereas these architectures usually have limited memory allocation. Thus, the computation-light but memory-heavy applications, compared to matrix multiplication and decomposition calculations, are less suitable for fully usage of the computing power of modern supercomputers. For example, although native and offload modes of `CUBEP3M` are able to run on the Intel Xeon-PHI MIC architectures, with the require ment of enough memory, TianNu simulation were done on TianHe-2 with only its CPUs – 73% of the total memory but only 13% of the total computing power.

We present a new N-body simulation code `CUBE`, using as low as 6 byte per particle (here after we use "bpp" referring to "byte per particle"). We show that it gives accurate results in cosmological LSS simulations. The algorithm is presented in §2, and a comparison between this method and traditional method is shown in §3. Discussions and conclusions are in §4.

2. ALGORITHM

The most memory consuming part of a N-body simulation is usually the phase space coordinates of N-body particles – 24 bpp (4 single precision floating numbers) must be used to store each particles' 3D position and velocity vectors. `CUBEP3M`, an example of a memory-lite parallel N-body code, can use as low as 40 bpp in sacrificing computing speed (Harnois-Déraps et al. 2013). This includes the phase coordinates (24 bpp) for particles in physical domain and buffered region, a linked list (4 bpp), and a global coarse mesh and local fine mesh. 4-byte real numbers are not necessarily adequate in representing the *global* coordinates in simulations. If the box size is many orders of magnitude larger than interactive distance between particles, especially in the field of re-simulation of dense subregions, double precision (8-byte) coordinates are needed to avoid truncation errors. Another solution is to record *relative* coordinates for both position and velocity. `CUBE` replaces the coordinates and linked list 24+4=28 bpp memory usage with an integer based storage, reduces the basic memory usage from 28 bpp down to 6 bpp, described as following subsections.

2.1. *Particle position storage*

haoran@cita.utoronto.ca

We construct a uniform mesh throughout the space and each particle belongs to its parent cell of the mesh. Instead of storing global coordinates of each particle, we store its offset relative to its parent cell which contains the particle. We divide the cell, in each dimension d, evenly into $2^8 = 256$ bins, and use a 1-byte (8 bits) integer $\chi_d \in \{-128, -127, ..., 127\}$ to indicate which bin it locates in this dimension. The global locations of particles are given by cell-ordered format in memory space, and a complimentary number count of particle number in this mesh (density field) will give complete information of particle distribution in the mesh. Then the global coordinate in dth dimension x_d is given by $x_d = (n_c - 1) + (\chi_d + 128 + 1/2)/256$, where $n_c = 1, 2, ..., N_c$ is the index of the coarse grid. The mesh is chosen to be coarse enough such that the density field takes negligible memory. This coarse density field can be further compressed into 1-byte integer format, such that a 1-byte integer show the particle number in this coarse cell in range 0 to 255. In the densest cells (rarely happened) where there are ≥ 255 particles, we can just write 255, and write the actual number as a 4-byte integer in another file.

In a simulation with volume L^3 and N_c^3 coarse cells, particle positions are stored with a precision of $L/(256N_c)$. The force calculation (e.g. softening length) should be configured much finer than this resolution, discussed in later sections. On the other hand, particle position can also be stored as 2-byte (16 bits) integers to increase the resolution. In this case, each coarse cell is divided into $2^{16} = 65536$ bins and the position precision is $L/(65536N_c)$, precise enough compared to using 4-byte global coordinates, see later results. We denote this case "x2" and denote using 1-byte integers for positions "x1".

We collectively write the general position conversion formulae

$$\chi_d = \left[2^{8n_\chi}(x_d - [x_d])\right] - 2^{8n_\chi - 1}, \quad (1)$$

$$x_d = (n_c - 1) + 2^{-8n_\chi}\left(\chi_d + 2^{8n_\chi - 1} + 1/2\right), \quad (2)$$

where [ ] is the operator to take the integer part. $n_\chi \in \{1, 2\}$ is the number of bytes used for each integer, x_d and χ_d are floating and integer version of the coordinate. Similarly, in section 2.2, the velocity counterpart of them are $n_\nu = 1, 2$, v_d and ν_d. Position resolution for n_χ-byte integer, "xn_χ", is $2^{-8n_\chi}L/N_c$.

2.2. *Particle velocity storage*

Similarly, actual velocity in dth dimension v_d is decomposed into an averaged velocity field on the same coarse grid v_c and a residual Δv relative to this field: $v_d = v_c + \Delta v$. v_c is always recorded and kept updated, and occupies negligible memory. We then divide velocity space Δv into uneven bins, and use a n_ν-byte integer to indicate which Δv bin the particle is located.

The reason why we use uneven bins is that, slower particles are more abundant compared to faster ones, and one should better resolve slower particles tracing at least linear evolution. On the other hand, there could be extreme scattering particles (in case of particle-particle force), and we can safely ignore or less resolve those non-physical particles. One of the solution is that, if we know the probability distribution function (PDF) $f(\Delta v)$ we divide its cumulative distribution function (CDF) $F(\Delta v) \in (0, 1)$ into 2^{8n_ν} bins to determine the boundary of Δv bins, and particles should evenly distribute in the corresponding uneven Δv bins. Practically we find that either $f(v_d)$ or $f(\Delta v)$ is close to Gaussian, so we can use Gaussian CDF, or any convenient analytic functions which are close to Gaussian, to convert velocity between real numbers and integers.

The essential parameter of the velocity distribution is its variance. On non-linear scale, the velocity distribution function is non-Gaussian. However, to the first order approximation, we simply assume it as Gaussian and characterized by the variance

$$\sigma_v(a, k) = \frac{1}{3}(aHfD)^2 \int_0^k d^3q \frac{P_L(q)}{q^2}, \quad (3)$$

where $a(z)$ is the scale factor, $H(z)$ the Hubble parameter, D is the linear growth factor, $f = d\ln D/d\ln a$, and $P_L(k)$ is the linear power spectrum of density contrast at redshift zero. $\sigma_v(a, k)$ is a function of cosmic evolution a and a smoothing scale k, or r. Δv is the velocity dispersion relative to the coarse grid, so we approximate its variance as

$$\sigma_\Delta^2(a) = \sigma_v^2(a, r_c) - \sigma_v^2(a, r_p), \quad (4)$$

where r_c is the scale of coarse grid, and r_p is the scale of average particle separation. In each dimension of 3D velocity field, we use $\sigma_\Delta^2(a)/3$ according to the equipartition therorem. On different scales, we measure the statistics of v_d, v_c and Δv and find good agreement with the above model.

The simulation results are very insensitive if we manually tune the variance of the model σ_Δ within an order of magnitude. However, the method of using uneven bins gets much better results than simply using equal bins between minimum and maximum values $[\min(\Delta v), \max(\Delta v)]$. So, one can safely use a standard ΛCDM (Cold Dark Matter with a cosmological constant) for slightly different cosmological models, in equation (3). In `CUBE`, the velocity conversion takes the formula

$$\nu_d = \left\lfloor (2^{8n_\nu} - 1)\pi^{-1}\tan^{-1}\left(v_d\sqrt{\pi/2\sigma_\Delta^2}\right)\right\rfloor, \quad (5)$$

$$v_d = \tan\left(\frac{\pi\nu_d}{2^{8n_\nu} - 1}\right)\sqrt{2\sigma_\Delta^2/\pi}, \quad (6)$$

where $\lfloor\ \rfloor$ is the operator to take the nearest integer. Tangent functions are convenient and compute very fast. Compared to error functions used in Gaussian case, they take the same variance at $v_d = 0$ but resolve high velocities relatively better. Note again that proper choice of conversion formulae and σ_Δ only optimizes the velocity space sampling, but does not affect the physics.

Initially, particles are generated by initial condition generator, at a higher redshift. The coarse grid density field v_c is also generated at this step by averaging all particles in the coarse cell. A global σ_Δ is calculated by equation (4), where linear approximation is hold. Then velocities are stored by equation (5). During the simulation, v_c is updated every time step, and a nonlinear σ_Δ is measured directly from the simulation, and can be simply used in the next time step, after scaled by the

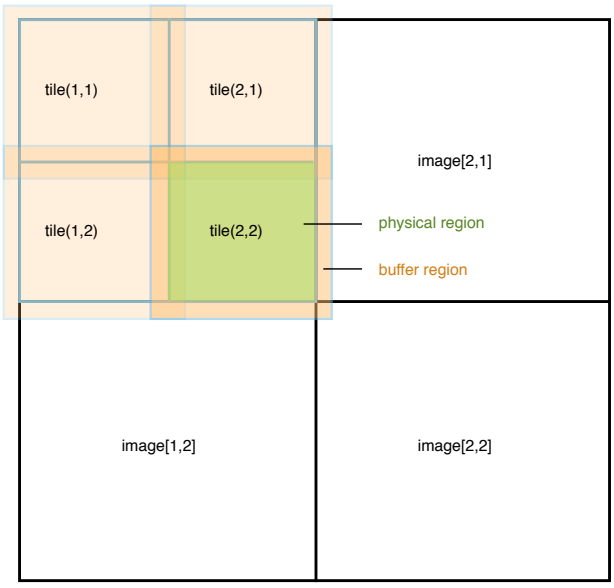

FIG. 1.— Spacial decomposition.

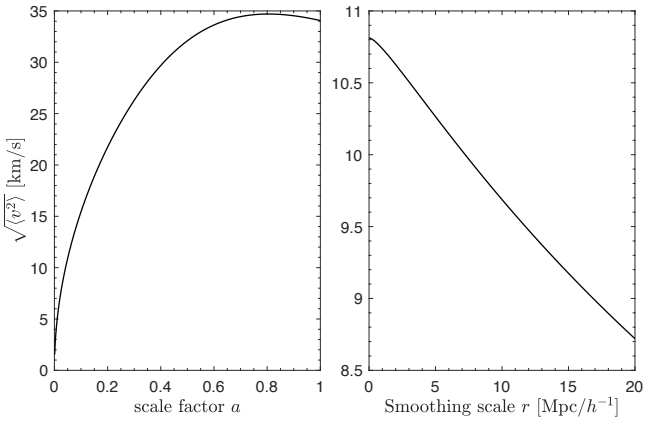

FIG. 2.— Velocity dispersion.

ratio of growth factors between two adjacent time steps. More details see section 2.4.

2.3. *Spacial decomposition*

`CUBE` uses cubic decomposition structures. The global simulation volume is decomposed into $\texttt{nn}^3$ cubic sub-volumes, and each of these are assigned to a coarray *image*[1]. Inside of an image, the sub-volume is further decomposed into $\texttt{nnt}^3$ cubic *tiles*. Each tile is surrounded by a *buffer* region which is usually `ncb` coarse cells thick. The buffer is designed for two reasons: (1) computing the fine mesh force, whose cut-off $\texttt{nforce\_cutoff} \leq \texttt{ncb}$, and (2) collecting all possible particles travelling from a tiles buffer region to its center, *physical* region. Thus, a integer-based coarse mesh density is defined as

$$\texttt{integer(1) rho\_c(nex, nex, nex, nnt, nnt, nnt)[nn, nn, *]}$$

where $\texttt{nex} = \texttt{nt} + 2\texttt{ncb}$ covers the buffer region on both sides, `nnt` is the tile dimemsions, and `nn` is the image *codimensions*[2]. Particles' phase coordinates `xp` and `vp` are copied between tiles and images, when necessary. Figure 2 shows the spacial decomposition in a 2-dimensional analogy, with $\texttt{nn} = 2$ and $\texttt{nnt} = 2$.

```
program CUBE
  call initialize
  call particle_initialize
  sync all
  call buffer_density
  call buffer_x
  call buffer_v
  do
    call timestep
    call update_x
    call buffer_density
    call buffer_x
    call PM ! update_v
    call buffer_v
    if(checkpoint_step) then
       call checkpoint
       if (final_step) exit
    endif
  enddo
  call finalize
endprogram
```

FIG. 3.— Overall structure of `CUBE`.

2.4. *Code overview*

By using the previous compact format storage, we modified the `CUBEP3M` and utilized it in the outputs TianNu simulation. On average, 9.125 bpp is used for all outputs and saves 62 % hard drive space on Tianhe-2. The compact version of `CUBEP3M`, named `CUBE`, accomplished the compact format in memory, and the code keeps only integer format for storing particles' phase space coordinates. `CUBE` is written in Coarray Fortran, where the coarray features are used to do MPI communications. Here we review the methodology of solving this integer based N-body problem. Like its predecessor `CUBEP3M`, `CUBE` uses two-level mesh grids for solving the Poisson equations. The advantages of this are described in Harnois-Déraps et al. (2013).

Figure 3 shows the overall structure of the code. Subroutine `initialize` creates necessary FFT plans and read in configuration files telling the program at which redshifts we need to do checkpoints, halofinds, or stop the simulation. Force kernels are also created or read in here. In `particle_initialize`, for each image, we read in initial positions and velocities in the compressed format. A initial condition generator is also written in the compressed format (see Appendix B). When this step is done by all images (synchronized by "`sync all`", which is equivalent to a `mpi_barrier`), the density fields `rho_c` are buffered between tiles (communications between images are needed) by subroutine `buffer_density`. Then, particles' phase space coordinates `x` and `v` are also buffered, by `buffer_x` and `buffer_v` respectively. In these steps, `x` and `v`'s ordering is changed according to the updated `rho_c`.

In each iteration of the main loop, we firstly call `timestep`, where a increment of time `dt` is controlled by particles' maximum velocities, accelerations, and cosmic expansions. According to `dt`, subroutine `update_x` up-

[1] Images are the concept of computing nodes or MPI tasks in Coarray Fortran. We use this terminology in this paper.

[2] Coarray Fortran concept. Codimensions can do communications between images.

dates the positions of particles in a "*gather*" algorithm. `x = x + v * dt` is executed twice, first time to determine a updated density field on the tile, `rho_c_new`, and second time to generate a new particle list `x_new` and `v_new` on the tile. The reason for this repeated execution is the dependence of `x_new` and `v_new`'s value and ordering on both the old `x`, `v` and the new `rho_c_new`. However, `x = x + v * dt` scales as $o(N)$ and is computational inexpensive. Although `x_new` and `v_new` arrays take extra memory, they do not appear simultaneously for multiple tiles, and compared to `x` and `v`, the extra memory overhead is by factor of $1/\texttt{nnt}^3$. This requires the "gather" algorithm to move particles[3] – as we synchronize the correct {`rho_c`, `x`, `v`} in the buffer region of each tile, and the particles in the extended (physical+buffer) region are a superset of particles locating in the physical region in the next time step, given that `v * dt` is not greater than the buffer depth. In `update_x`, {`rho_c_new`, `x_new`, `v_new`} can be set to collect only the physical region of the tile, with the rest of this subset discarded. At the end, {`rho_c`, `x`, `v`} is replaced by {`rho_c_new`, `x_new`, `v_new`}, and the memory of latter is freed up.

Next, `buffer_density` and `buffer_x` are called again to synchronize the updated particle positions in order that tile-based local fine forces are computed correctly.

A standard particle-mesh scheme is then performed in the `PM` subroutine. First, by looping over tiles, fine force is calculated and velocities are updated on the physical region of each tile. The memory overhead of fine-mesh arrays are well controlled by $1/\texttt{nnt}^3$. Next, global coarse force is solved by using a coarser (usually by factor of 4) mesh by dimensional splitting – a distributed-memory cubic decomposed 3D coarse density field is interpolated by particles, and we Fourier transform data in consecutive three dimensions with global transposition in between (known as the pencil decomposition). After the multiplication of force kernels, the inverse transform takes place to get the cubic distributed coarse force field, upon which velocities are updated again. The memory usage of coarse arrays are negligible. An optional particle-particle (PP) force can be called to increase the force resolution and the velocities are updated last time according to this. As discussed in §2.2, updating velocities use model predicted velocity distribution, thus `v = v + dv` is executed only once, scaling directly to the velocity distribution of the next time step `d+dt`. During each of these force calculations, maximum accelerations are collected serving for the `timestep` in the next iteration, controlling next `dt`.

By far, particle velocities in the physical regions of each tile are updated. Particle locations in the buffer regions has updated before `PM` and remained unchanged. So we simply call `buffer_v` again, such that the `update_x` in the next iteration will be done correctly.

If a desired redshift is reached, we call `checkpoint` and/or exit the main loop. The corresponding logical variables are controlled in `timestep`. Finally, in `finalize` subroutine we destroy all the FFT plans and finish up any timing or statistics taken in the simulation.

2.5. *Memory layout*

[3] In contrast, `CUBEP3M` uses a "*scatter*" algorithm.

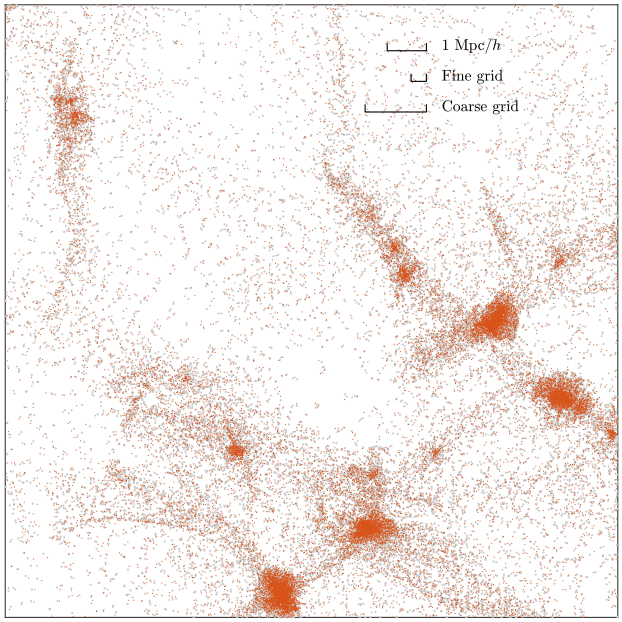

FIG. 4.— particles.

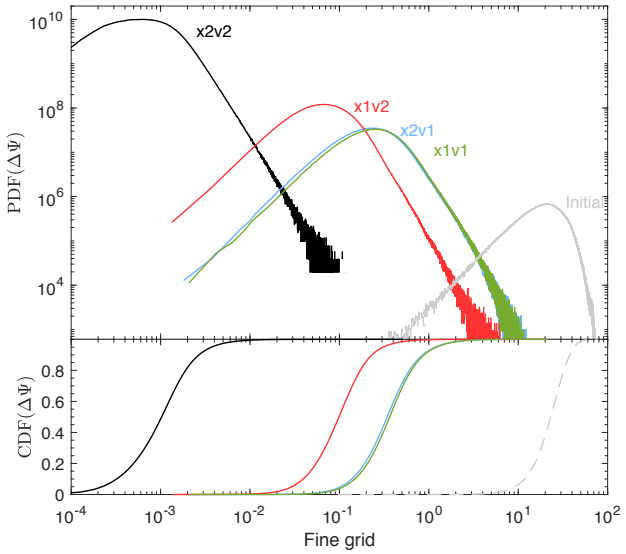

FIG. 5.— Displacement.

Here we summarize the memory usage in unit of bpp (byte per particle).

3. RESULTS

We use the same initial conditions, and use `CUBEP3M` and `CUBE` to simulate the large scale structure formation separately and compare their results.

Results of `izipx=1`

Results of `izipx=2`

4. DISCUSSION AND CONCLUSION

Cosmological simulation, PM method.

PID.

Summarize memory.

Phi, GPU.

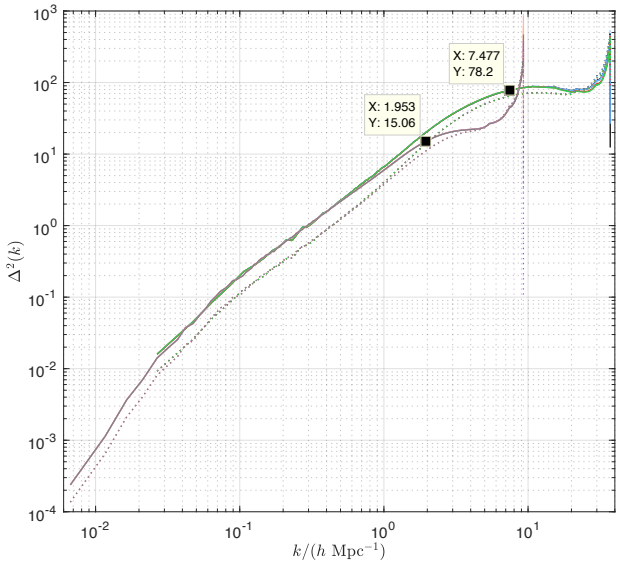

FIG. 6.— power spectrum.

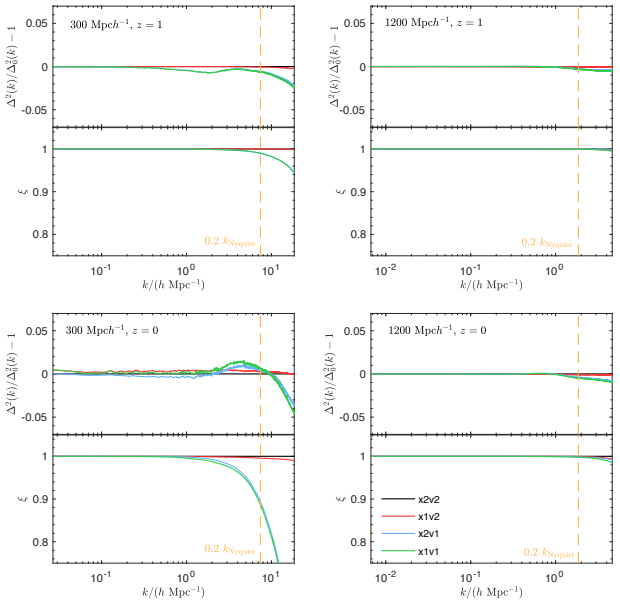

FIG. 7.— power spectrum and cross correlation.

APPENDIX

A. VELOCITY DISTRIBUTION FUNCTION

B. INITIAL CONDITION GENERATOR

Thanks.

REFERENCES

Emberson, J. D. et al. 2017, Research in Astronomy and Astrophysics, 17, 085, 1611.01545

Harnois-Déraps, J., Pen, U.-L., Iliev, I. T., Merz, H., Emberson, J. D., & Desjacques, V. 2013, MNRAS, 436, 540, 1208.5098

Hockney, R. W., & Eastwood, J. W. 1988, Computer simulation using particles

Yu, H.-R. et al. 2017, Nature Astronomy, 1, 0143, 1609.08968