ABACUS User Guide

December 9, 2017

1. Introduction to Abacus

Abacus is a cosmology N-body code intended for large cosmological simulations with high-force accuracy. It utilizes fast near-field computations using GPUs and AVX instructions, as well as a novel method for solving the far field.

The Abacus computation is organized into a 3-dimensional grid of cells. Particles belong to cells. The near-field force is computed within a cell and its near neighbors. Set finding also happens within a fixed maximum number of cells. The far-field force is computed from the multipole moments of particles in the cells.

Abacus is built to run out-of-core, so that the particles are read only once per full time step. The computation occurs through a pipeline that takes a 1-dimensional sweep through the simulation volume. The plane of cells perpendicular to the sweep direction are called a slab. This is the x direction in the simulation units.

Abacus evolves through a series of states. Every full time step reads a state, evolves the particles, and then writes a new state. The states are the restart files. Because of the leapfrog time evolution, the states are generally not time-synchronous between the positions and velocities: the velocities will require an additional kick by the listed kick factor to be synchronous with the positions.

Because of this leapfrogging, the state files are generally not intended for long-term storage of the outputs. Instead, Abacus generates output files during full steps. Time slice outputs will occur at the redshift of the "read" state (so the full time step will be shortened so as not to overshoot the requested time slice). Light cone outputs are interpolated to the time when the past light cone from an observer crosses the particle or the coevolution set.

Within a full step, microstepping may occur. This means that particles in higher density regions with shorter dynamical times will use shorter leapfrog steps. However, to avoid needing to load the particles more than once from disk, the regions requiring finer time steps are identified on the fly so that all of the microsteps occur at once. The regions requiring mutually finer time resolution of the forces are known as coevolution sets. By definition, all particles outside of a given coevolution set have forces that are sufficiently constant over the full step as to be applied with only the top-level leapfrog. Note that one can always pick a short enough full time-step to satisfy one's accuracy criterion for this. Inside the coevolution set, a variety of time steps may be used.

In addition to their utility in time stepping, co-evolution sets also define the maximum boundaries of our halo and subhalo finding. In other words, halo finding never extends beyond the set

boundary. Finally, co-evolution sets are cohesive for light-cone outputs: we will always output a snapshot of a co-evolution set at the "read" state epoch, but shifting the center of mass position and velocity drift to the exact epoch when the past light cone sweeps by.

Coevolution sets are found by performing friends-of-friends with a fairly large linking length.

2. Zel'dovich Initial Conditions

The zeldovich code creates displacement vectors appropriate to the given power spectrum. It normalizes and scales the input power spectrum as instructed. Interpolation to the needed wavenumbers is done by spline, so one should input a reasonably fine grid. Modes with wavenumbers $|\vec{k}|$ above the Nyquist frequency are set to zero to avoid grid directionality artifacts. The code is available as a separate module, along with extensive documentation, at https://github.com/lgarrison/zeldovich-PLT.

The zeldovich code can also use the particle lattice eigenmodes instead of the usual continuum $\hat{k}$ modes to eliminiate the vorticity and decaying modes that usually enter on small scales. The code can also fudge the initial mode amplitudes to account for different growth rates as a function of $\vec{k}$, with the hope of seeding non-linear evolution with the correct linear displacements. These are the PLT ("Particle Linear Theory") features of the code.

Displacement vectors are output along with the grid position (as integers). The abacus IC loader maps this information to positions and velocities. Recall that in $\Omega_m = 1$, the comoving displacement vector is the same as the velocity in redshift-space displacement units. In other cosmologies, the scaling is $f = d\ln D/d\ln a$.

Displacement vectors are computed by multiplying by $i\vec{k}$ in Fourier space, rather by differencing in configuration space. In addition, we compute the density field grid itself, although this is not output by default. The 4 real FFTs are done by pairing into 2 complex FFTs.

When using PLT features, the velocities are not simply a rescaling of the displacements and have to be calculated in Fourier space. We pack the three velocity components into another 2 complex FFTs. In this case, the outputs will include the displacement and velocity.

Unfortunately, the zeldovich code labels the slab direction by z. It flips zyx to xyz as it outputs.

The zeldovich code is built to work out-of-core. It splits the 3-d computational volume in two directions, y and z, into NumBlocks each. It then performs the 3-d FFT in two parts, first loading slabs in y and doing the z FFT (and establishing the Hermitian symmetries), and then considering slabs in z, doing the xy FFTs, and outputing in z order (which becomes the abacus slab order). Between the two parts, we write the slabs to disk in yz blocks. In this way, we require only 2/NumBlocks fraction of the problem to be in memory.

Currently a fast 1-d FFT routine computes of order 500 Mcells/sec, which implies a memory

access requirement of 8 GB/sec. Hence, the zeldovich code is likely to be I/O limited. However, total I/O is only 32+32+18 bytes per particle. This is comparable to the I/O of a single abacus step. So unless the disk latency is noticeable, we expect the run-time to be comparable to one abacus step.

3. Parameter File Description

The parameter file holds values that should be constant throughout the simulation. In principle, one could change the file when restarting from a state, but behavior is not guaranteed. Still, this could be useful for adjusting output parameters, for example.

Values that change with time as the simulation evolves or that are calculated by the abacus code or cosmology module are in the state file.

Both headers are incorporated into output file headers.

3.1. Basic Simulation Parameters

`NP`: (`long long int`) The number of particles in the simulation. Note that math is allowed in inputs to GenParam, so you might opt to set this to a perfect cube, e.g., to 1024**3.

`CPD`: (`int`) The cells per dimension. This number must be odd. Should be a number with small prime factors. Small numbers of distinct factors are good too. See the `python/choose_cpd.py` utility for guidance on choosing this.

`BoxSize`: (`float`) The size of the box in comoving physical units.

`hMpc`: (`int`) =1 if we're using Mpc/h units. =0 if Mpc units.

`TimeStepAccel`: (`float`) Time-step parameter based on accelerations.

`TimeStepDlna`: (`float`) Maximum $\Delta(\ln a)$ allowed for a full step. For example, 0.02 means at least 50 steps per e-folding of the scale factor. DJE recommends at least 20 steps per e-fold for most applications, or .05.

`FinalRedshift`: (`float`) The final redshift of the simulation. If not supplied, then we use the lowest redshift TimeSlice output. However, one might want to run further to complete light cones.

`OverwriteState`: (`int`) If ≥ 0, ReadDirectory and WriteDirectory are internally set to the ReadDirectory. This saves space, and can be particularly useful for a ramdisk.

`RamDisk`: (`int`) If ≥ 0, let the IO code know that we can't use DIO (which ramdisks don't support).

3.2. Memory Allocation and other Execution Parameters

`NumSlabsInsertList`: (`float`) The amount of space to allocate for the insert list. This is supplied as a multiple (not necessarily integral) of np/cpd particles, i.e., the average number of particles per slab. The default is 2.

`NumSlabsInsertListIC`: (`float`) The amount of space to allocate for the insert list in the initial ingestion of IC particles. Here we typically have much less memory required elsewhere in the code, so this number can be larger than NumSlabsInsertList. Again, this is supplied as a multiple of np/cpd particles, i.e., the average number of particles per slab. Choosing 0 causes this parameter to be set to cpd, so that the insert list can contain all of the particles in the simulation. This is appropriate for small problems where the particles aren't sorted into slabs in the IC file. For large problems, one should use at least 4. Default is 4.

`MAXRAMMB`: (`int`) The maximum amount of RAM to use for internal slab allocations. Default is to detect the total amount of RAM. Internally, the code will load slabs until it reaches half this amount. The other half is saved for incidental allocations, like source/sink pencils.

`OMP_NUM_THREADS`: (`int`) Number of OpenMP threads. The default, 0, does not modify the system value (set by the $OMP_NUM_THREADS environment variable; if not given, default is to use all cores). Negative values use that many fewer than the max. For example, a value of -4 will use all but 4 cores. If using OMP_PLACES, this number should be equal to the total number of "places" (i.e. cores) specified.

`OMP_PLACES`: (`string`) Specifies on which cores to place OpenMP threads. This is useful for keeping the OpenMP threads on separate cores from the non-OpenMP threads (like the IO and GPU threads). For example, to use the first 8 cores on each of two sockets that have 12 cores each, use "{0}:8,{12}:8". Syntax is "{start}:count". The total number of places (16 in this example) should equal the OMP_NUM_THREADS value. To have any effect, this parameter must be used in conjunction with OMP_PROC_BIND, which tells OpenMP how to assign threads to the "places" (cores) listed here. The default is an empty string, meaning no explicit core assignemnts. In this case, the kernel scheduler will just do its best to keep threads separate.

Note that OpenMP offers no C/C++ interfaces to this functionality, so OMP_PLACES is set as an environment variable by the Python wrapper. This means that these parameters will have no effect if the singlestep binary is invoked outside the Python wrapper. The resulting thread-to-core assignments can be seen at the top of the log file. These are the only parameters need the Python wrapper to work. Can we change that?

`OMP_PROC_BIND`: (`string`) Specifies how to assign OpenMP threads to the "places" (cores) that were specified in OMP_PLACES. A value of "spread" seems to have the desired behavior of dividing

threads betweeen sockets when on a NUMA machine. The default (empty string) means do not bind threads to places. The resulting placements can be seen at the top of the log file.

IOCores: (`list of two ints`) The cores to which to bind the IO thread(s). The default ([-1, -1]) means do not bind to a core. This list should always have two elements (one for each IO thread); this does not mean that two IO threads will be started though. That is determined by SecondIOThreadDirs. It usually seems advantageous to give each IO thread its own core.

GPUThreadCoreStart: (`list of ints`) The cores on which to start placing the threads that dispatch work to the GPUs. NGPUThreadCores sets the count of cores to use immediately following these "start cores" to use. There are DirectBPD*NGPU GPU threads total. "DirectBPD" is "buffers-per-device" and is set by "./configure –with-direct-bpd=BPD" (default 3). The GPU threads are in charge of copying data to and from pinned memory (the GPU staging area); usually three threads per GPU can keep the GPUs fed. These threads can be on the same core, but it usually incurrs 20% performance hit to the effective GPU rate (because the copies get slower). This list of cores should be NGPU long; the idea is to keep all the threads that communicate with a given GPU on the same socket. On a 24 core system divided into two sockets, something like "[9, 21]" and NGPUThreadCores = 3 will give each GPU thread its own core (assuming DirectBPD = 3 and NGPU = 2). "[11, 23]" and NGPUThreadCores = 1 would give 3 threads to each core. In both cases, threads belonging to the same GPU will be on the same socket. These cores should be separate from the OpenMP and IO cores specified by OMP_PLACES and IOCores. The default of all -1 means do not bind GPU threads to cores.

NGPUThreadCores: (`int`) The number of cores following each core specified in GPUThreadCoreStart that will be available for GPU threads. Default of -1 means do not bind threads to cores.

Conv_OMP_NUM_THREADS: (`int`) Same as OMP_NUM_THREADS, but for the Convolution step. A value of -1 is usually appropriate, since we only need to reserve one core for the IO thread.

Conv_OMP_PLACES: (`string`) Same as OMP_PLACES, but for the Convolution step. You don't need to pay much attention to sockets here; the main thing is to make sure the OpenMP places don't overlap with Conv_IOCore. Something like "{0}:23" is fine if Conv_IOCore is 23.

Conv_OMP_PROC_BIND: (`string`) Same as OMP_PROC_BIND, but for the Convolution step. Either "spread" or "close" should be fine.

Conv_IOCore: (`int`) Same as IOCores, but for the Convolution step. Also, we only have one convolution IO thread presently, so this is just an int (not a list). The important thing is to keep this separate from the cores specified in Conv_OMP_PLACES.

3.3. Far-field Parameters

`Order`: (`int`) The multipole order to use for the far-field. Typical choices would be 8 or 16. One really should not use low orders such as 2! Any integer between 2 and 16 is allowed.

`DerivativeExpansionRadius`: (`int`) The number of periodic replicas to sum over explicitly before switching to the asymptotic value. Suggest a good value! Is this the right definition?

`ConvolutionCacheSizeMB`: (`int`) The size of the convolution cache. Should fit within L2 or L3, so this value will be around 10 MB. Default is to detect the L3 cache size.

`DerivativesDirectory`: (`string`) The directory where the derivative files are stored. One expects to reuse derivatives between simulations, rather than regenerating them each time. The derivative files are about 1/16 the size of the multipoles, so one would like the I/O rate on this device to be 1/32 of the rate of the device for the multipoles and Taylors. Because the derivative files are not slab-ordered, the latency issue is much reduced. So in many cases, the derivatives can be stored on a normal hard disk. However, the convolution is presently almost entirely IO bound, so it's not a bad idea to put the derivatives on the fastest device available. We could add a second IO thread to handle the derivatives and overlap their IO with the M/T.

3.4. Near-field Parameters

`NearFieldRadius`: (`int`) The radius of cells to be solved in the near-field. Radius 1 means that each cell is acted on by itself and 26 neighbors. Radius 2 means 124 neighbors. One really should not use radius 0!

`SofteningLength`: (`float`) The softening length used in the near-field force, in the same units as BoxSize. For example, .05 would yield a 50 kpc/h comoving softening length, if BoxSize is given in comoving Mpc/h. This is the Plummer-equivalent length and may be internally converted to a different value depending on the softening technique being used. That is stored in the state as SofteningLengthInternal. The conversion ensures that the minimum orbital period is always $\sqrt{GM/\epsilon}$ (i.e. the minimum Plummer orbital period) independent of the softening technqiue being used.

`DirectNewtonRaphson`: (`int`) Choice of 1 means that we use a Newton-Raphson step to improve the precision of the forces. Is this really a choice that we want to offer? This option appears to have no effect in the current code.

`GPUMinCellSinks`: (`int`) Number of particles in a cell below which its forces will be computed on the CPU, not GPU. Currently disabled. Check functionality in the group-finding code.

We will eventually have parameters for such choice as trees, etc.

3.5. State Directories

The following defaults are set in the `directory.def` file. You usually want to "`#include directory.def`" just after setting SimName in your `abacus.par2` file.

`WorkingDirectory`: (`string`) Unless countermanded by the choices below, this is where the states will be created. This should be a large disk capable of fast sequential IO, such as a large RAID array. The default is read from the $ABACUS_TMP environment variable.

`ReadStateDirectory`: (`string`) The name of the read state directory. This defaults to "WorkingDirectory/read".

`WriteStateDirectory`: (`string`) The name of the write state directory. This defaults to "WorkingDirectory/write".

`PastStateDirectory`: (`string`) The name of the past state directory. This defaults to "WorkingDirectory/past".

`MultipoleDirectory`: (`string`) The name of the multipole directory. This should be a fast, low-latency disk, like an SSD. The default is "$ABACUS_SSD/multipoles", where $ABACUS_SSD is an enviroment variable.

`TaylorDirectory`: (`string`) The name of the Taylors directory. This should be a fast, low-latency disk, like an SSD. The default is "$ABACUS_SSD/taylor", where $ABACUS_SSD is an enviroment variable.

3.6. Initial Conditions

`InitialRedshift`: (`float`) The initial redshift of the simulation.

`LagrangianPTOrder`: (`int`) Instruction for how to use the initial Zel'dovich dispacements. =1 for Zel'dovich, =2 for 2LPT, =3 for 3LPT. The higher-order Lagrangian Perturbation Theory is performed by using Abacus to compute forces, which then get scaled to the LPT displacements at the given redshift. See doc/lpt.pdf for more detail.

LagrangianPTOrder >= 2 may involve re-reading the IC files during the last LPT step (probably Abacus step 2), because the velocities are used as scratch space in our in-place LPT scheme. If the ICs included velocities (e.g. because ZD_qPLT was turned on the Zel'dovich code), then we re-read the IC files to recover these velocities.

2LPT is tested and works well; 3LPT is implemented but does not give the expected results.

`InitialConditionsDirectory`: (`string`) The directory where the initial condition files will be found. This is also where the zeldovich code with write its outputs. This defaults to "$ABACUS_PERSIST/ic", where $ABACUS_PERSIST is an environment variable. We try to avoid fragmentation of the filesystem where the main state is stored ($ABACUS_TMP), hence the use of $ABACUS_PERSIST.

These files must be named "IC_0", "IC_1", etc. The number indicates the slab number; files $N-1$, N, and $N+1$ will be read before slab N is finished. File $N=0$ is the first file read; slab 1 is the first slab finished. This means that the particles in slab N must appear no later than file $N+1$. However, if the particles appear in a much earlier file, then the insert list will become very large. This is acceptable if the simulation fits entirely into memory. For large sims, we expect that the particles will be sorted into slabs, with no more than ± 1 tolerance. This tolerance is likely sufficient, e.g., for Zel'dovich displacements, so that particles can be placed in slab files by their grid position rather than their true initial position.

If a larger tolerance is needed, one can give FINISH_WAIT_RADIUS a larger value, like 2, inside Abacus.

`ICFormat`: (`string`) The format of the IC files. Current formats are: 'RVdouble', 'RVZel', 'RVdoubleZel', 'Heitmann', and 'Zeldovich'. The binary formats are documented at https://github.com/lgarrison/zeldovich-PLT.

`ICPositionRange`: (`float`) The size of the periodic box for the IC positions, in the units supplied in the IC file. For example, ICPositionRange of 1 means that the positions are all 0..1. A choice of 0 causes this parameter to be set equal to BoxSize.

`ICVelocity2Displacement`: (`float`) The conversion factor by which to multiply the supplied velocities so as to convert them to redshift-space comoving displacements, in the same units as the supplied positions (see ICPositionRange) and at the initial redshift. A factor of 1.0 is appropriate for ICs received from the zeldovich code. A factor of -1 will automatically handle proper km/s.

`FlipZelDisp`: (`int`) If ≥ 0, reverse the sign of the displacement, if we're using a Zel'dovich-style IC format. Useful for debugging the second-order forces used in 2LPT.

3.7. Output Parameters

`SimName`: (`string`) The name of the simulation, intended to be unique for each simulation. This is used both as the primary path of the directory where output will occur, but also in filenames.

`OutputDirectory`: (`string`) This is where the outputs will be placed; in particular, the time slices

will be here. The default is "$ABACUS_PERSIST/SimName", where $ABACUS_PERSIST is an environment variable.

`LogDirectory`: (`string`) The directory where the logs are to be stored. The default is "$ABACUS_PERSIST/SimName/log". Each time step generates several log files, but these all go in the same directory.

`LightConeDirectory`: (`string`) The directory where the light cones should get written. The default is "$ABACUS_SSD/SimName/". The individual LCs will be organized into directorys called "lc_rawN", where N is the number of the light cone 0..7. Inside this directory, the outputs will be further divided with one subdirectory per time step; these names are fixed to be "stepNNNN". $ABACUS_SSD is an environment variable; we default to this fast directory because we expect to do a merge of these many small files which is hard on the filesystem. The "Analysis/LightCones/merge_lightcones.py" script will take the "lc_rawN" directory and create a "lcN" directory.

`GroupDirectory`: (`string`) The directory where group information will be written. The default is "$ABACUS_PERSIST/SimName/groups". Inside this directory, the outputs will be further divided with one subdirectory per time step; these names are fixed to be "stepNNNN". Not used currently. Check this after we implement OTF group finding.

`TimeSliceRedshift`: (`float vector`) The redshift of the requested time slices. When redshifts appear in file names, it will be as "z%5.3f".

`nTimeSlice`: (`int`) The number of time slice outputs.

`NLightCones`: (`int`) The number of light cones. We support up to 8.

`LightConeOrigins`: (`float vector`) The observation point of the light cones, in triples (x, y, z). Up to eight light cones can be supplied. The position need not be inside the primary wrapping of the box. The units are the same units as BoxSize.

`OutputFormat`: (`string`) Format for the time slice outputs. Options are: RVdouble, Packed, Heitmann, RVdoubleTag. Packed outputs are described in §7. The other formats are mostly useful for debugging, and their binary formats can be found in "DataModel/appendarena.cpp".

`BackupDirectory`: (`string`) Location to write intermittent state backups. Note that this does not need to include the multipoles, since a reliable "multipole recovery" utility is available as "make_multipoles". Should this also be an environment variable ($ABACUS_BACKUP)?

`BackupStepInterval`: (`int`) Number of steps between backups.

`SecondIOThreadDirs`: (`list of string`) The directory names whose files will be read/written by the second IO thread. A typical value would be `["multipole/", "taylor/"]` if the multipoles and Taylors are on a separate disk from the state and would thus benefit from having their IO overlapped with the primary IO thread. The timings file will identify which directories were assigned to which threads. The IO logs for each thread will have a ".1" or ".2" appended to them as appropriate.

`nSecondIOThreadDirs`: (`int`) The number of directories in `SecondIOThreadDirs`.

`PowerSpectrumStepInterval`: (`int`) The number of steps between computing on-the-fly power spectra. The 3D density field will be saved in the log directory and then converted to a 1D power spectrum after the step has finished. The `Analysis/PlotLinearTheory` contains some utilities to make plots from these spectra.

`PowerSpectrumN1d`: (`int`) The number of grid cells per dimension to use when computing the OTF density field.

These choices produce the following hierarchy of files:

$ABACUS_PERSIST/SimName ($ABACUS_PERSIST is an environment variable)

$ABACUS_PERSIST/SimName (aka OutputDirectory)
$ABACUS_PERSIST/SimName/sliceZ.ZZZ/SimName.zZ.ZZZ.slabNNNN.php14

$ABACUS_PERSIST/SimName/log (aka LogDirectory)
$ABACUS_PERSIST/SimName/log/SimName.stepNNNN.log
$ABACUS_PERSIST/SimName/log/SimName.stepNNNN.iolog[.1|.2]
$ABACUS_PERSIST/SimName/log/SimName.stepNNNN.time
$ABACUS_PERSIST/SimName/log/SimName.stepNNNN.convtime
$ABACUS_PERSIST/SimName/log/SimName.stepNNNN.convlog

$ABACUS_PERSIST/SimName/group (aka GroupDirectory)
$ABACUS_PERSIST/SimName/group/stepNNNN/SimName.stepNNNN.slabNNNN.grp

$ABACUS_PERSIST/SimName/lc_rawN (aka LightConeDirectory + lc_rawN)
$ABACUS_PERSIST/SimName/lc_rawN/stepNNNN/SimName.lcN.stepNNNN.slabNNNN.php14

$ABACUS_PERSIST/SimName/info (misc. important files)

The SimName is used both in the output path and as the file name prefix. Redshifts are written in names with format %5.3f. Slabs and Steps are written with %04d. Light cones are written with %1d.

The php14 suffix is just a place-holder, but this is to indicate the file format.

3.8. Environment Variables

Some of the above default directories are set by enviroment variables. These enviroment variables all begin with "ABACUS_" and are optional if you manually specify all directory paths. The `install.py` script will help you install these to `/.bashrc` or some other suitable location (or set up an Abacus module if on a system that uses modules, like many HPC clusters). N.B.: these are not parameter file parameters, but instead shell environment variables!

`ABACUS`: (`string`) Location of the Abacus source code. Not used by the main simulation code, only by some analysis utilities.

`ABACUS_TMP`: (`string`) Large disk with fast sequential IO for storing state.

`ABACUS_PERSIST`: (`string`) Large, stable disk for storing particle outputs and logs.

`ABACUS_SSD`: (`string`) Fast, low-latency disk like an SSD for storing multipoles/Taylors and light cones.

3.9. Cosmological Parameters

`H0`: (`float`) The Hubble constant in km/s/Mpc

`Omega_M`: (`float`) At $z = 0$.

`Omega_DE`: (`float`) At $z = 0$.

`Omega_K`: (`float`) At $z = 0$.

The sum of all the above Omegas must be 1.

`w0`: (`float`) Dark energy equation of state $w(z) = w_0 + (1 - a)w_a$.

`wa`: (`float`) See above.

3.10. Debugging and Special Cases

`StoreForces`: (`int`) If 1, store the accelerations in the OutputDirectory.

`ForceOutputDebug`: (`int`) If 1, output near and far forces separately. The time step is set to zero, so there is no other action and other outputs (e.g., group finding) may be garbled. Should only be set if StoreForces is not set.

In addition, using TimeStepDlna=0 forces the next time step to be 0, which implies that the particles won't advance (although forces and groups will still be computed).

`OutputEveryStep`: (`int`) If ≥ 0, produce a time slice output at every step.

3.11. Zel'dovich Initial Condition Generation Parameters

`ZD_Seed`: (`int`) The random number seed. This, and ZD_NumBlock, set the phases of the ICs.

`ZD_NumBlock`: (`int`) This is the number of blocks to break the FFT into, per linear dimension. This must be an even number; moreover, it must divide PPD ($NP^{1/3}$) evenly. The default is 2, but you may need a higher number. Changing ZD_NumBlock will change the phases of the ICs for a given ZD_Seed.Make phases independent of ZD_NumBlock

This is a key tuning parameter for the code. The full problem requires $32 \times NP$ bytes, which may exceed the amount of RAM. The zeldovich code holds 2/NumBlock of the full volume in memory, by splitting the problem in 2 dimensions into NumBlock^2 parts. Each block therefore is $32 \times NP$/NumBlock^2 bytes. It is important that these blocks be larger than the latency of the disk, so sizes of order 100 MB are useful. We are holding 2×NumBlock such blocks in memory.

Hence, for a computer with M bytes of available memory and a problem of NP particles, we need NumBlock $> 64 \times NP/M$ (preferably to be the next larger even number that divides evenly into $NP^{1/3}$) and we prefer that $32NP$/NumBlock^2 is larger than the latency.

For example, for a 4096^3 simulation, M is 2 TB. If we use NumBlock of 128, then we will need 32 GB of RAM and each block saved to disk will be 128 MB. For a 2048^3 simulation, M is 256 GB and NumBlock of 16 will require 32 GB of RAM and each block will be 1024 MB. For a 8192^3 simulation, M is 16 TB and NumBlock of 512 will require 64 GB of RAM and a block size of 256 MB.

If ZD_k_cutoff $\neq 1$, then the actual ZD_NumBlock will be ZD_NumBlock*ZD_k_cutoff. See ZD_k_cutoff for details.

`ZD_Pk_filename`: (`string`) The file name of the input power spectrum. This can be a CAMB power spectrum.

`ZD_Pk_scale`: (`double`) This is the quantity by which to multiply the wavenumbers in the input file to place them in the units that will be used in the zeldovich code, in which the fundamental wavenumber is 2π divided by BoxSize. Default value is 1.0.

As a common example, one might need to convert between Mpc^{-1} and h Mpc^{-1} units. The zeldovich code does not use the hMpc value, so it doesn't know what the units of BoxSize are. If

BoxSize is in h^{-1} Mpc units, so that hMpc=1, and if the $P(k)$ file had k in h Mpc^{-1} units, then all is well: use the value of 1.0.

However, if BoxSize were in Mpc units and the input power were in h Mpc^{-1} units, then we want to convert the wavenumbers to Mpc^{-1} units. That means multiplying by h, so we should use ZD_Pk_scale = h.

ZD_Pk_norm: (double) The scale at which to normalize the input $P(k)$, in the same units as BoxSize. For example, if BoxSize is given in h^{-1} Mpc, then one might choose 8 to select σ_8. If this value is 0, then the power spectrum will not be renormalized (but ZD_Pk_scale will be applied to the wavevectors, so beware that the power isn't thrown off, as it does have units of volume). The default is 0, but we recommend controlling the normalization.

ZD_Pk_sigma: (double) The amplitude to use to normalize the fluctuations of the density field, with a tophat of radius ZD_Pk_norm. This must be scaled to the initial redshift by the growth function; the zeldovich code does not know about cosmology. The default is 0, but one almost certainly wants to change this!

The Abacus Python wrapper will compute this value based on the cosmology and **sigma_8**. You almost never want to specify **ZD_Pk_sigma**; instead specify **sigma_8**.

Note that using this parameter means that the choice of unit of power in the input power spectrum file is irrelevant (but we do care about the unit of wavenumber, see above).

sigma_8: (double) The amplitude of present-day density fluctuations σ_8. This value is not read by the zeldovich code, but instead processed by the Python wrapper to produce **ZD_Pk_sigma** by scaling via the growth function.

ZD_Pk_smooth: (double) The length scale by which to smooth the input power spectrum before generating the density field. This is applied as a Gaussian smoothing as $\exp(-r^2/2a^2)$ on the density field, which is $\exp(-k^2a^2)$ on the power spectrum. Smoothing the power spectrum is useful for testing, as it reduces grid artifacts. The smooth occurs after the power spectrum has been normalized. Default is 0.

ZD_qoneslab: (int) If ≥ 0, output only one PPD slab. For debugging only. The default is -1.

ZD_qonemode: (int) If ≥ 0, zero out all modes except the one with the wavevector specified in ZD_one_mode.

ZD_one_mode: (three ints) This is the one wavevector that will be inserted into the box if ZD_qonemode ≥ 0. This is useful for automatically iterating through a series of wavevectors for examining isotropy or Nyquist effects, for example. Each component can be an integer in the range [-ppd/2,ppd/2].

ZD_qPLT: (**int**) If ≥ 0, turn on particle linear theory corrections. This tweaks the displacements and velocities, mostly near $k_{\rm Nyquist}$, to ensure everything starts in the growing mode. The output format will include velocities if you turn this on, either in the RVZel or RVdoubleZel format (set by a Makefile flag). PLT features are described in Garrison, et al. (2016).

ZD_PLT_filename: (**string**) The file containing the PLT eigenmodes; i.e. the true growing modes for the grid. We generate this on something like a 128^3 grid and linearly interpolate the eigenmodes and eigenvalues as needed.

ZD_qPLT_rescale: (**int**) If ≥ 0, increase the amplitude of the displacements on small scales (near $k_{\rm Nyquist}$) to compensate preemptively for future undergrowth that we know happens on a grid.

ZD_PLT_target_z: (**int**) If ZD_qPLT_rescale ≥ 0, then increase the initial displacements such that they will match the linear theory prediction at this redshift. Recall that modes on the grid (mostly) grow more slowly than linear theory, which is why we increase the initial displacements. This redshift should be before shell-crossing while linear theory is still mostly valid, e.g. $z \sim 5$. Provide a heuristic for calculating this number based on pixel-level density fluctuations

ZD_k_cutoff: (**double**) The wavenumber above which not to input any power, expressed such that $k_{\rm max} = k_{\rm Nyquist}/k_{\rm cutoff}$, e.g. ZD_k_cutoff = 2 means we null out modes above half-Nyquist. Non-whole numbers like 1.5 are allowed. This is useful for doing convergence tests, e.g. run once with PPD=64 and ZD_k_cutoff = 1, and again with PPD=128 and ZD_k_cutoff = 2. This will produce two boxes with the exact same modes (although the PLT corrections will be slightly different), but the second box's modes are oversampled by a factor of two. To keep the random number generation synchronized between the two boxes (fixed number of particle planes per block), ZD_NumBlock is increased by a factor of ZD_k_cutoff.

In addition, the Zel'dovich code expects the following parameters, defined above.

BoxSize: (**double**) This is the box size, either in Mpc or h^{-1} Mpc, depending on the setting of the hMpc key. However, the zeldovich code does not use the hMpc value. See ZD_Pk_scale for further discussion.

NP: (**long long int**) This must be a perfect cube for the zeldovich code to work.

CPD: (**int**) Zeldovich will output in slabs. These slabs are only defined by the initial grid position; we do not guarantee that the initial displacement may not have taken the particle out of the slab. Usually the deviations will be small enough that particles move by at most 1 slab.

InitialConditionsDirectory: (**string**) In addition to the usual usage, the zeldovich code will use this for the swap space for the block transpose. This will generate files of the name 'zeldovich.%d.%d'. These will be deleted after the code has run.

InitialRedshift: (double) Starting redshift of the sim; i.e. the output redshift for the zeldovich code. Only used by the zeldovich code for PLT rescaling, although the Python wrapper will use it to compute the growth function.

The following options have been disabled in the current code:

ZD_density_filename: (string) The file name where to write the density field. This option may be disabled.

ZD_qdensity: (int) If non-zero, output the grid densities that generate the Zel'dovich displacements. Default is 0. This option may be disabled, but I think that we should eventually restore this feature.

Perhaps we make the output of densities standard? E.g., four floats?

4. State file formats

A State is actually a full sub-directory, with all of the files describing the simulation's particles.

The file called 'state' is an ASCII file in ParseHeader format. It is described in the next section. This file is created last: if it exists, then the code singlestep completed properly and the state is ready to be used (read-only) by the next time step of abacus. If it does not exist, then the state is invalid. singlestep will not proceed if the WriteState state file exists, since it presumes it would overwriting valid data. However, deleting only the state file will allow singlestep to run, using the read state and the write state Taylors, but overwriting all else.

Many quantities are stored in slabs, implying CPD files each. These include the positions, velocities, auxillary variables, cellinfo structures, multipoles, and Taylors. Notably positions and velocities are stored as float[3] or double[3] arrays (so one particle's x, y, and z, then the next, etc.). Multipoles and Taylors are stored in YZ Fourier space, so these rather cryptic. Auxillary and cellinfo structures are defined in the code.

There are also a few smaller files. 'slabsize' is ASCII and holds the number of particles in each slab (after opening with CPD). The 'redlack_{x,y,z}' and globaldipole files contain information for the Redlack-Grindlay term.

Accelerations can also be written on request (see §3.10).

5. State File Description

The State file contains information that is computed by the Abacus code, particularly values that are epoch-dependent.

5.1. Properties of the Simulation

`np_state, cpd_state, order_state`: (`int`) These are just copied from the parameter file, so that we can be sure to be able to read the state files.

`ParameterFileName`: (`string`) The name of the Parameter file on which the state was invoked.

`CodeVersion`: (`string`) The git hash label of the executable.

`RunTime`: (`string`) The time stamp of the start of this invocation of singlestep().

`MachineName`: (`string`) The machine name where this step is being run.

DoublePrecision: (int) Whether the internal positions, velocities, and forces are being computed in single precision (==0) or double precision (==1).

ppd: (double) The cube root of the number of particles. We take care to round this off to an integer if it is very close.

FullStepNumber: (int) The full step number. The initial conditions get translated into a state (with multipoles) that is defined as step number 0. So the full step number is also the number of times that full forces have been computed.

LPTStepNumber: (int) How many steps into our Lagrangian Perturbation Theory computation we are. 0 if this is a normal time step; non-zero if this is an LPT step.

Do2LPTVelocityRereading: (int) Our 2LPT implementation overwrites the initial particle velocities which is fine if we are using the Zeldovich approximation (because they can be recomputed from the displacements). If we are using PLT corrections (or any IC format that specifies a velocity), then we have to reread the IC files to restore the overwritten velocities. This flag specifies whether that will happen during this step.

SofteningType: (string) The force law being employed. Possible values are: "plummer", "cubic_plummer", "cubic_spline", "single_spline".

SofteningLength: (double) The Plummer-equivalent softening length that the user inputs. See the description in the Parameters section. Same units as BoxSize.

SofteningLengthInternal: (double) The softening length scaled for this particular softening technique to produce a minimum orbital period that matches Plummer. Same units as BoxSize.

5.2. Computed Properties of the Cosmology

BoxSizeMpc, BoxSizeHMpc: (double) The comoving size of the box, in Mpc and h^{-1} Mpc.

HubbleTimeGyr, HubbleTimeHGyr: (double) The value of $1/H_0$ in Gyr and h^{-1} Gyr. This is important because all of the times below will be reported in units of $H_0 = 1$.

ParticleMassMsun, ParticleMassHMsun: (double) The mass of a single particle, in $M_\odot$ and $h^{-1}\ M_\odot$.

5.3. Properties of the Epoch

`ScaleFactor`: (`double`) The scale factor a, normalized to $a = 1$ at the present day.

`Redshift`: (`double`) The redshift $1 + z = 1/a$.

`VelZSpace_to_kms`: (`double`) The Hubble velocity across the full box in km/s. In other words, this is $H(z)L/(1 + z)$, in km/s. If one has velocities in redshift-space displacement unit-box units (our default), then multiplying by this factor will convert the velocity to km/s.

`VelZSpace_to_Canonical`: (`double`) The conversion from our basic output velocity unit of redshift-space displacement (in unit box length units) to the code canonical velocities. We have $v_{\rm canon} = av_{\rm proper} = a^2 v_{\rm comoving}$ and $v_{\rm proper} = H(z)av_{\rm zspace}$, so $v_{\rm canon} = v_{\rm zspace}a^2H(z)/H_0$ (remembering that the code uses $1/H_0$ time units).

`Time`: (`double`) The proper time, in $1/H_0$ units.

`etaK`: (`double`) The integral $\int_0^t dt/a$, which is used in kicking the particles. This is also the conformal time. In $1/H_0$ units.

`etaD`: (`double`) The integral $\int^t dt/a^2$, which is used in drifting the particles. In $1/H_0$ units. This integral diverges as $a \to 0$, but we initialize at an early time with the quantity $\eta_D = -2/a^2H(z)$. Although η_D is negative, it increases with time and our $\Delta\eta_D$ are positive.

`Growth`: (`double`) The linear growth function, normalized to a at early times.

`Growth_on_a`: (`double`) The linear growth function divided by a, which is unity in Einstein-de Sitter.

`f_growth`: (`double`) The growth rate $d\ln D/d\ln a$.

`w`: (`double`) The equation of state of dark energy at the present epoch.

`HubbleNow`: (`double`) The Hubble parameter at the current epoch, in H_0 units, i.e., $H(z)/H_0$.

`Htime`: (`double`) The product of the Hubble parameter and the current time, $H(z)t(z)$. This is 2/3 in Einstein-de Sitter.

`OmegaNow_m`: (`double`) The value of Ω_m that an observer at this epoch would measure.

`OmegaNow_K`: (`double`) The value of Ω_K that an observer at this epoch would measure.

`OmegaNow_DE`: (`double`) The value of Ω_{DE} that an observer at this epoch would measure.

5.4. Properties of the Timestep

`DeltaTime`: (`double`) The time difference between this state and the previous one.

`DeltaScaleFactor`: (`double`) The scale factor difference between this state and the previous one.

`DeltaRedshift`: (`double`) The redshift difference between this state and the previous one (flipped to be a positive number for increasing time).

`ScaleFactorHalf`: (`double`) The scale factor of the time halfway between this state and the previous one.

`LastHalfEtaKick`: (`double`) The $\Delta\eta_K$ kick that remains to be applied to the velocities in this state to bring them up to the epoch in this state. Recall that the velocities are stored a half-step before and will be leaped over to be a half-step ahead, before they are drifted again. Note that when the particles are in the Read state, this is the $\Delta\eta_K$ we will apply, yet we no longer have access to the state epoch *before* this Read state, hence the need to save this quantity.

`FirstHalfEtaKick`: (`double`) The $\Delta\eta_K$ kick that will be applied to move the velocities halfway from the previous epoch to the current epoch.

`DeltaEtaDrift`: (`double`) The $\Delta\eta_D$ drift that will be applied to move the positions from the previous epoch to the current epoch.

5.5. Statistics of the Particle Distribution

These are used to compute the time step and to monitor the code.

`RMS_Velocity`: (`double`) The rms velocity in the entire simulation ($\langle|\vec{v}|\rangle$), in simulation units (canonical velocities, unit box length, $1/H_0$ time unit). One can use the VelZSpace_to_Canonical and RedshiftSpaceConversion values to convert the units. Important: this is the rms velocity at the epoch in ReadState: the velocities in the ReadState are kicked to be synchronous with the positions, then this maximum is tracked and written into WriteState.

`MaxVelocity`: (`double`) The maximum velocity component ($v_{\{x,y,z\}}$), in simulation units (canonical velocities, unit box length, $1/H_0$ time unit). One can use the VelZSpace_to_Canonical and RedshiftSpaceConversion values to convert the units. Note that this is not the maximum $|\vec{v}|$. Important: this is the maximum velocity at the epoch in ReadState: the velocities in the ReadState are kicked to be synchronous with the positions, then this maximum is tracked and written into WriteState.

`MaxAcceleration`: (`double`) The maximum acceleration component ($a_{\{x,y,z\}}$), in simulation units (unit box length, $1/H_0$ time unit). Note that this is not the maximum $|\vec{a}|$. Important: this is the maximum acceleration at the epoch in ReadState: the positions in the ReadState are used to compute accelerations, on which this maximum is tracked and written into WriteState.

`MinVrmsOnAmax`: (`double`) In each cell, we compute the ratio of the rms velocity and the maximum acceleration. The rms velocity is $\langle|\vec{v}|^2\rangle$, i.e., the mean velocity has not been removed before computing the second moment, so this is not Galilean invariant. The maximum acceleration is done by component ($a_{\{x,y,z\}}$). After this ratio is computed for each cell, the minimum over all cells is computed and reported.

Reconsider whether computing the full norms or proper variances would actually be expensive.

`MaxCellSize, MinCellSize`: (`int`) The maximum and minumum number of particles in a cell. These are computed at the positions in WriteState.

`StdDevCellSize`: (`double`) The standard deviation of the density contrast (fractional overdensity $\rho/\bar{\rho} - 1$) within cells.

6. Output Files

Abacus can produce both time slice and light cone files.

Time slices are divided into CPD slab files, for reasonable portability. These files are all written into one directory, with a different directory for each time slice.

Light cones span many steps. To keep the files organized, we write one directory per time step, with CPD slab files in each directory. Of course, these files will be notably smaller than the time slice files (only a fraction of particles in each slab are output in a given time step). But since we are reading and writing a half-dozen large files per slab per step, writing one more, even if small, is negligible overhead.

For both types of output, we write a ParseHeader format header at the top of the file. The Parameter file is in this header, as is a large portion of the State variables (those not known at the time of output are omitted). After the ParseHeader end-of-header token, the particles, velocities, and perhaps auxillary variables are written in a binary format.

In addition, the power spectrum file used in the Zel'dovich code is copied into the Output and Log directory when the ICs are created.

Sometimes we have many auxillary files that we want to carry around with the outputs. An example would be CAMB parameter files or simple diagnostic plots like the power spectrum evolution. These can be placed in the `info` directory in the simulation configuration directory; the Python wrappers will copy this info directory to any relevant output or data product directories.

7. Packed Outputs (pack14)

The cell-oriented structure of abacus allows for an efficient way to compress the particle data. We output particles in sets for a given cell, and the positions are stored relative to the cell center. A typical cell size is a few Mpc. 12 bits of precision will yield a part in 4000 precision, which is about 1 kpc. This is adequate for all reasonable output analysis tasks. Note that abacus output files are not intended to be adequate for dynamical restarts; one always has the option to save states for restart.

Abacus always outputs its velocities in redshift-space units, i.e., the comoving displacement that the particle would appear to be in redshift space. This avoids a need to choose a second set of units for velocities. We have $v_{\rm proper} = Hx_{\rm proper} = Hax_{\rm comoving}$, so $v_{\rm zspace} = v_{\rm proper}(1+z)/H(z)$.

Velocities rarely get above 6000 km/s, so a part in 4000 means 1-2 km/s precision, which is 10 kpc in zspace displacement. Again, this is adequate for output analyses. We track the maximum velocity in unit-cell lengths and rescale the velocities by a factor A so that their displacements are always contained within the unit cell.

So our standard pack14 class stores 12 bits for each of the 6 phase space components, a total of 9 bytes. We then add 5 bytes of ID numbers or group IDs, for a total of 14 bytes per particle. We call this the pack14 format.

The 12 bits are stored as unsigned. However, phase space quantities are signed, so we add 2048 before storing. We scale the unit cell to a range of ± 2000. Phase space quantities are permitted to be in the range of ± 2030, to allow some small overflow beyond the cell.

The id numbers are stored explicitly as unsigned characters, so that the standard is endian-independent.

Cells are begun with a structure of the same 14-byte length. This structure can be identified because the first byte is 0xff, which makes the first position value 2032-2047. The remaining 5 12-bit numbers are filled with the cell i, j, k, the CPD of the calculation, and the velocity rescaling A (which must be a positive integer). Note that all of these quantities are notionally unsigned, but it might be convenient to allow the ijk to go slightly negative (e.g., for a periodic wrap). Therefore, we add 30 to these quantities before storing. Values outside of 16..4080 are still illegal, so these values must lie between -30 and 4020. Hence, the standard only works to CPD of 4020. The 5 bytes for ID are currently unused in the cell headers.

After a cell element, all elements are taken to be particles in that cell, until the next cell element is read. We do not insist on knowing the number of particles in the cell before writing the cell header. Since we are likely to have dozens to hundreds of particles per cell, the cost of the cell header is small.

The global position is then (cellx + particlex/2000.0)*B, where B is the box size divided by CPD.

The global velocity is (particlevx/2000.0)*A*B, where A is the scaling factor chosen when packing the velocities. We expect that good A's will be of order 30. There is not much value in making A smaller than 1, since this would mean more resolution in velocity than position, so we store A as an integer.

We provide routines to unpack these files to double-precision lists of positions and velocities. Note that 12 bits intra-cell, plus 10-12 bits of cell ids implies 22-24 bits of precision, comparable to single precision.

Note that were the data stored in single precision, we would require 24 bytes for the phase space data, plus 5-8 bytes for ids, a total of 29-32 bytes. So the packing saves a factor of 2 in disk space.

8. Logging

Abacus produces an extensive set of log files for each simulation. These are written into the log directory, with separate files for each time step.

In addition, there is a top-level summary log that is updated after each successful time step.

The singlestep code produces 2 to 4 logs: a primary log, a report on timings, and a log on the actions of the I/O thread(s), if those threads are running. The convolution code produces a primary log and a timing log.

The log contains a millisecond-level record of the elapsed time since the log was initiated. Log verbosity can be controlled with a parameter choice.

If the program has crashed, then the files laststep.{suffix} contains the logs.