User's Guide for PICOLA

Cullan Howlett

cullan.howlett@port.ac.uk

Institute of Cosmology and Gravitation, University of Portsmouth

February 7, 2014

Contents

1 Introduction

PICOLA (**P**arallel **I**mplementation of **COLA**code) is a fast, planar-parallel code for creating and evolving a dark matter field, starting from the creation of a suitable set of initial conditions, using a combination of 2^{nd} order Langrangian Perturbation Theory and Particle-Mesh N-Body solutions to the dark matter equation of state. This allows us to generate large ensembles of dark matter simulations that, whilst lacking the accuracy and fidelity of N-Body simulations, can be created orders of magnitude faster.

There are two main types of simulation that PICOLA can do, snapshots and lightcones, each of which has a myriad of different options in terms of outputting and generating the initial conditions. More information on these is included in Sections 3 and 4. However, the main differences between these two types are that snapshot simulations simply evolve the dark matter field and output the whole field at user-specified redshifts. Lightcone simulations on the other hand only output a particle once it has exited the lightcone, with the lightcone is shrinking as the simulation steps towards lower redshifts. For each particle that exits we interpolate the time and hence the exact position of the particle when it left the lightcone. This creates a smoothly varying dark matter field over the full redshift range, whilst a snapshot simulation creates a more discrete realisation assuming all points in the simulation box are at the same redshift.

This document serves only as a guide on the usage of PICOLA and hopefully should contain all the necessary information for those wanting the run the code independently (if this is not the case then please let me know). In terms of the science and algorithms behind PICOLA we refer the reader to the accompanying paper *'PICOLA: A fast parallel code for generating mock catalogues'* (Howlett et al. in prep) and references therein. It is highly recommended that this paper be read before attempting to use the code.

Now for the technicalities. PICOLA is written in ANSI C and is compatible with GNU and Intel compilers. Parallelisation is achieved using the Message Passing Interface (MPI). On top of this the code requires the GSL and FFTW3 libraries, both of which should be easily obtainable. PICOLA, like the foundations on which it is built, is made publicly available under the GNU general public license. Feel free to modify it and distribute it as you please however the copyright for the original code remains with the authors and as such we knidly ask that if you find the code useful for any scientific work you reference the accompanying paper stated above.

2 Running PICOLA

2.1 Compilation

It is my hope that PICOLA prove relatively easy to compile, however past experience suggests that with any new code this is rarely the case. PICOLA requires three non-standard libraries that you may not have already:

- **MPI** - The 'Message Passing Interface' is essential for all the paralllelisation within the code and many versions exist. For example ,see MPICH (`http://www.mpich.org/`).

- **GSL** - The 'GNU Scientific Library'. This is an open source package containing a large number of numerical tools. PICOLA requires this library for a range of operations including random number generation and numerical integration. It can be found at `http://www.gnu.org/software/gsl/`

- **FFTW3** - The 'Fastest Fourier Transform in the West'. This performs the discrete Fourier transforms required to solve for the displacement of the particles during the creation of the initial conditions and timestepping. It is very important that FFTW3 and *not* FFTW2 is used as these two libraries are incompatible and whilst PICOLA could be changed to accomodate such a switch over, it is most definitely not recommended. On top of this the FFTW3 library must be compiled with parallel support enabled and, depending on how PICOLA will be used, both the single- and double-precision versions should be installed. The documentation associated with the library should enable a relatively pain-free compilation. The library itself can be found at `http://www.fftw.org/`

After unpacking the tar-file you should see two folders, one containing the source code (`/src/`) and another containing additional, useful utility codes and files (`/files/`). Alongside this there should be a `Makefile`. The `Makefile` contains a range of different C preprocessor options that when turned on and off change the way the code is compiled (further details on these are included in Section 3.). At the bottom of the file are the paths to the libraries mentioned above. **These *will* have to be changed based on the machine you are running on**.

To make the code a simple call to `make` or `gmake` should suffice. This will convert all the source code files into seperate object files before combining them into a single executable, the name of which can be changed at the top of the `Makefile`. The `make clean` command will remove the executable, all object files and any temporary files.

2.2 Starting the code

After compilation the code can be run from the directory containing the executable using a command like

```
mpirun -np 64 PICOLA /pathtoparameterfile/parameterfile.param
```

This will run PICOLA on 64 processors with simulation parameters specified by the file `parameterfile.param`. This parameter file is very important and requires a strict set of parameters to be specified. If a single parameter is missing the code will not run. An example parameter file is included in the `/files/` directory. For an explanation of the required input parameters see Section 4. There is no limit (at least in the code) on how many processors can be used at once, however be mindful of memory usage and the fact that the more processors there are the more overheads we have associated with MPI. See Section 5 for some notes on PICOLA's memory consumption.

During runtime PICOLA will output logistical information either to the screen or to a file depending on whether the job is being run iteractively or not. Basic output will merely tell you the stage the code is at and useful parameters that have been calculated or read in, however in the case of an error/warning the code will tell the user why it occured and where in the source code the exact line can be found.

2.3 Required Input Files

In order for PICOLA to work several input files are required. The first of these is the parameter file mentioned above and this is the only one which must be passed as an argument to the executable. All the other files are passed via the parameters file itself. Some of these files *may* not actually be read in or used by the code depending on the compilation options and other input parameters, however a filename must still be supplied in the parameters file. See the directory `/files/` for some example input files

Output Redshift File: This file is essential for all compiled versions of PICOLA. The way it is used however varies depending on whether we are producing a snapshot or lightcone. For snapshot simulations the output redshift file must contain a list of the redshifts we wish to output a snapshot at (each of which acts as a kind of checkpoint) and the number of timesteps we want to take between this output redshift and the previous checkpoint. The previous checkpoint is either the previous redshift at which we output a snapshot or, in the case of the first output redshift, the user-specified initial redshift. This list does not need to be sorted and there is no limit on the number of

redshifts we can output at. There are a few caveats however, each of which will cause the code to return an error:

- The minimum number of timesteps between two checkpoints must be greater than 0.
- There must be at least one output redshift (and a corresponding value for the number of steps) in the output redshift file.
- All the output redshifts must be less than the user-specified initial redshift. If any of the output redshifts are equal (to within $1\text{x}10^{-6}\%$) to the initial redshift of the simulation then we will output the initial conditions before timestepping commences. In this particular case the corresponding value for the number of timesteps to take is irrelevant, though a value must still be supplied.

For lightcone simulations we, by definition, output every timestep. For this reason the output redshift file *must* contain only two output redshifts (and again their corresponding values for the number of timesteps). The first pair is the redshift at which to begin the lightcone and the number of steps between the initial redshift and this redshift. This output redshift may be equal to the initial redshift, in which case the whole simulation will be lightcone. The second pair is the final redshift and the total number of timesteps to take during the lightcone part of the simulation.

Whilst there are a lot of rules to remember for this particular input file, they should hopefully be quite self-explanatory and easy to remember/understand. On top of this the code performs many checks on the list within this file and will tell the user if anything seems amiss.

Power spectrum file: This input file contains the initial power spectrum used to generate the initial conditions of the simulation. The format of this file must be two columns, with each row containing a single pair of values for k and $P(k)$ respectively. This file will only be used if the value of the input parameter `WhichSpectrum` = 1. One very important thing to remember is that the input power spectrum must contain the required value for the scalar index, n_s, where $P(k) \propto k^{n_s} T(k)^2$ and $T(k)$ is the transfer function. Unlike when using the transfer function as input, the code assumes the scalar index has already been absorbed into the input power spectrum and will not apply the value given in the input parameters file. On the other hand, it is not necessary (but is recommended) that the input power spectrum returns the correct value for σ_8, the linear matter variance at 8Mpc, used for the normalisation of the power spectrum. This is because the code will calculate

the normalisation at run time and compare this to the user-specified value for σ_8 in the input parameters file, normalising the input power if neccesary.

Transfer Function File: This input file contains the input transfer function that may be used to create the initial conditions. This file will only be used if the value of the input parameter `WhichTransfer` = 1. As mentioned in Section 4, for non-Gaussian initial conditions we have to use the transfer function as opposed to the power spectrum. The format of the file must be two columns, with each row containing a single pair of values for k and $T(k)$ respectively. Unlike when using the input power spectrum file we assume the scalar index is not included in the transfer function and apply the user-specified value from the input parameters file. Again, as before, the normalisation will be checked and applied if necessary by the code at run time.

Non-Gaussian Kernel File WTF is going on here???

2.4 Output Files

PICOLA can output the simulation in either ASCII or unformatted binary depending on whether the `UNFORMATTED` compilation option has been turned on in the `Makefile` (See Section 3). Each processor in the simulation outputs its own file, suffixed by the number of the processor, with a filepath and filename that must be specified in the input parameters file. In the case of lightcone simulations the word 'lightcone' is appended to the end of the simulation before the processor number.

For both snapshot and lightcone simulations the ASCII output format remains the same: particle ID (if turned on in the `Makefile`, see Section 3), x-position, y-position, z-position, x-velocity, y-velocity, z-velocity. The unformatted output is arranged differently for snapshots and lightcones due to the fact that lightcone simulations do not write out all the particle data at once.

One very important thing to remember is the units in which we output the particle velocities and how they differ between snapshots and lightcones. For lightcone simulations the particle velocity, v, is equal to the peculiar velocity of the particle, v_p, however in snapshot simulations the particle velocity is given by $v = v_p/\sqrt{a}$, where a is the scale factor of the snapshot. The reason for this is that it allows PICOLA to be used to generate the initial conditions for true N-Body simulations, conforming to the conventional units. As we don't know the scale factor at which a particle exited the lightcone in a lightcone simulation after the simulation is finished, we remove the scale factor dependence.

Example Fortran and C routines for reading in the unformatted binary output from PICOLA are included in `/files/prepare_PICOLA.f90` and `/files/prepare_PICOLA.c` respectively. For snapshot simulations the unformatted output contains a header,

followed by the x y and z positions for *all* the particles, then the x, y and z velocities and finally the ID's (if requested). For lightcones the format is a little more complicated and contains chunks of particles, where prior to each chunk we have written the number of particles in the chunk. The data within the chunk is written like the ASCII output, with ID's, positions and velocities for each individual particle written contiguously.

It is recommended that unformatted output is used, even though the format is harder to interpret, as this allows the code to write out large groups of particle at once as opposed to individually, and greatly decreases the runtime, especially for lightcone simulations.

3 Compilation Options

PICOLA has a range of preprocessor options that affect the way the code is compiled and hence how it runs. These are all listed in the `Makefile` and can be turned on and off by commenting out both the lines associated with a particular option. Once a compilation option has been changed the code must be completely recompiled by typing `make clean` followed by `make`. Here we list the various compilation options and their effects.

SINGLE_PRECISION: This option, when uncommented, forces the code to use single-precision floating point numbers and FFTW routines. By default all floating point numbers and FFTW routines are double precision. This can be useful when running smaller simulations as it will nearly half the memory required for the code, however be careful when dealing with large simulations as single-precision floating point accuracy may not be accurate enough for the Fourier transforms. A good rule of thumb is to assume single-precision is accurate to 6 decimal places, so for a simulation box of size 1000Mpch^{-1} we can get displacements accurate to 1kpch^{-1}. Similarly, for the FFTW routines we effectively invert the density, so if the number of particles is too large we risk losing precision during the transforms.

MEMORY_MODE: This option is designed to use as little memory as possible whilst still retaining double precision Fourier transforms. At the expense of speed we allocate and deallocate certain portions of memory every timestep, which means that we have to recalculate the FFTW plans every timestep. On top of this we set all particle data to single-precision. Hence when PICOLA is compiled with this option it is slower than the default single- and double-precision versions of the code, however it creates a compromise for large simulations, retaining some of the accuracy of double precision, at a

greatly reduced memory requirement. In fact for most uses of PICOLA it is recommended that this option remains on.

PARTICLE_ID: This assigns a unique number to each particle in the simulation to allow for quick identification and tracking of particles. The ID's themselves are 64-bit unsigned integers, which means that they add an extra 8 bytes to the storage required for each particle. This can quickly add up for large numbers of particles, however the reason behind this is that it allows simulations with more than $2^{32} - 1$ particles to be given ID's, whereas overflows would occur in this situation if we were to use 32-bit unsigned integers.

LIGHTCONE: When this option is switched on PICOLA is converted to lightcone mode. In this mode the input redshift file must now contain the redshift at which to begin the lightcone and the redshift to end the simulation at and nothing else. The code runs as normal, generating the initial conditions and timestepping, until the first output redshift is reached. From then onwards when we move the particles we also check to see which of those have exited a lightcone, centred on the user-specified origin, with a comoving radius which has decreased over the course of the timestep. If a particle has exited, we then interpolate the exact time it left the lightcone and use this time to calculate its exact position when it exited the lightcone. We output the interpolated position and velocity of all the particle's that leave the lightcone each timestep.

Generally, the lightcone exists as a sphere, centred on the origin, with a comoving radius based on the current redshift. However, we can shape the lightcone somewhat by specifying, in the input parameters file, the number of times to replicate the simulation box in all 6 directions along the x, y, and z axes.. As a result of this, the lightcone simulation may be much slower than the a snapshot simulation if there are a large number of replicates to loop over. Also note that the number of replicates specified by the user is the *maximum* number of replicates. Any replicates that are unnecessary as they lie completely outside the lightcone will be forgotten. Conversely, and more importantly, if the redshift at which the lightcone begins is outside the furthest replicate (from the origin) then the code will *not* increase the number of replicates, assuming instead that this is due to the shape of the field that the user has requested. The code however does output a warning in such a case, as well as the maximum comoving distance the replicates and lightcone reach. It is then up to the user to check that these are what is required.

GAUSSIAN: This tells the code to create Gaussian initial conditions, otherwise the user must select one of the non-Gaussian initial conditions types. One

of the options amongst the Gaussian/Non-Gaussian initial conditions *must* be selected (there is no 'default' option).

LOCAL_FNL: Creates non-Gaussian initial conditions using $F_{nl,local}$, where the exact value of F_{nl} and the redshift at which the non-Gaussianity is applied are specified by the user in the input prameters file (See Section 4.). This option is only available for scale invariant, $n_s = 1$, initial power spectra, and, as with all types of non-Gaussianity in PICOLA, requires the transfer function as input. For local non-Gaussianity with $n_s \neq 1$ we must select the `GENERIC_FNL` compilation option and use the file `/files/input_kernel_local.txt` as input in the input parameters file.

EQUIL_FNL: Creates non-Gaussian initial conditions using $F_{nl,equil}$. The same rules apply as for local non-Gaussianity. For equilateral non-Gaussianity with $n_s \neq 1$ we must select the `GENERIC_FNL` compilation option and use the file `/files/input_kernel_equil.txt` as input in the input parameters file.

ORTHO_FNL: Creates non-Gaussian initial conditions using $F_{nl,ortho}$. The same rules apply as for local and equilateral non-Gaussianity. For orthogonal non-Gaussianity with $n_s \neq 1$ we must select the `GENERIC_FNL` compilation option and use the file `/files/input_kernel_ortho.txt` as input in the input parameters file.

GENERIC_FNL: Creates non-Gaussian initial conditions with a general F_{nl}. This implementation allows for $n_s \neq 1$ but requires a input kernel, containing the coefficients for the generic kernel, to work (see Section 2.3). Example kernels for local, equilateral and orthogonal non-Gaussianity are included in the `/files/` directory.

UNFORMATTED: This option makes the code output the particle data in unformatted binary as opposed to ASCII. To decrease the runtime of the code, especially when performing lightcone simulations, it is recommended that this option is turned on. For a description of the output style for both ASCII and unformatted options see Section 2.4.

4 Input Parameters

In this section we detail the various input parameters that PICOLA requires to work. Some of these depend on the exact compilation options we are running and must be removed or commented out when not necessary. Some also depend on other input parameters and so must be included even if the simulation will not require them, as the code does not know until runtime that they will not be necessary.

Which parameters these caveats apply to will also be detailed below. An example input parameters file can be found in `/files/run_parameters.param`

4.1 Simulation Outputs

OutputDir: This is the directory that PICOLA will write the output to.

FileBase: This is the base filename of the output from PICOLA. Every processor, each of which has a unique processor number, n, will write out its own file, the name of which will be the given by `FileBase.n`. In the case of lightcone simulations we also append the word 'lightcone' to the end of the base file name, i.e., `FileBase_lightcone.n`.

OutputRedshiftFile: This is the name of the file containing the list of outputs redshifts and the corresponding number of steps up to that redshift as detailed in Section 2.3. An example output redshift file is given in `/files/output_redshifts.dat`.

NumFilesWrittenInParallel: This tells the code the maximum number of processors that can output simultaneously, as there may be machine-dependent limitations on this. The code creates an artificial bottleneck when writing out so the fewer processors that can write out at once, the slower PICOLA will be. Obviously the value of this parameter must be larger than 0, and in the case where a value is entered that is greater than the number of processors preforming the simulation all processors are allowed to write out simultaneously.

4.2 Simulation Specifications

UseCOLA: Set to 1 (or in fact any non-zero value) to use the COLA method within the simulation. This modifies the way the particle's velocities and positions are calculated each timestep, See Howlett et al. (in prep.) for a complete description of the COLA method and why it is recommended. If this parameter is set to 0 then the PICOLA functions as a standard Particle-Mesh code.

Buffer: This parameter acts as a multiplicative factor to increase the amount of memory that each processor allocates to store the particle data. This is necessary because, by default, the code says that each processor stores equal numbers of particles, however this will not be the case as the dark matter field will not stay completely homogenous throughout the simulation. The buffer memory also acts as temporary memory for particles that move between

processors each timestep. Typical values for this are ~ 1.25, however this may need to be larger for simulations where the each processor only has a very thin slice of the simulation, i.e., large number of processors/small simulation box. If we are at risk of running out of memory, the code will stop and ask you to increase this input parameter.

Nmesh: The number of mesh cells per side in the simulation. The total number of mesh cells in the simulation is then given by $\texttt{Nmesh}^3 + 2$, to accomodate the FFT's. It is not necessary that this be a power of 2 or even, however such simulations will run faster and it is unwise to use odd numbers due to possible difficulties with the Fourier transforms. It is recommended that a value close to the number of particles per side is chosen, .i.e., so that there is ~ 1 particle per mesh cell. Values lower than this may result in poor force resolution, whilst values significantly larger can, on top of the large increase in the code's memory consumption, lead to spurious clustering due to discreteness effects. It is also recommended that `Nmesh` is easily divisible by the total number of processors to be used as it is required that there are an integer number of mesh cells per processor, even if that means that some processors will not get any mesh cells and hence will not be used for the entire simulation.

Nsample: The number of particles per side in the simulation box. The total number of particles in the simulation is given by $N_{tot} = \texttt{Nsample}^3$. Again it is not necessary that this is a power of 2 or even, however it is recommended that `Nsample` is chosen such that $\mathrm{N_{tot}}$ is easily divisible by the number of processors that the simulation is run on. This is because the particles are assigned to each processor based on the mesh cell they occupy and the number of mesh cells per processor must be an integer.

Box: This is the edge length of the simulation in box units (which are given physical dimensions by the `UnitLength_in_cm` input parameter). The box is periodic, and so any particles that leave the edge of the box get wrapped around to the opposite edge.

Init_Redshift: This is the initial redshift to start the simulation at. The initial conditions are computed at this redshift and it is from here that timestepping commences.

Seed: This is the random seed used to generate the initial conditions. The code is designed such that two otherwise identical simulations using different initial seeds will generate completely independent realisations (at least as independent as two random numbers can be). To run two identical sets of initial conditions we can use the same random seed.

WhichSpectrum: This is a flag to tell the code what type of initial power spectrum we will use to generate to initial conditions. A value of '0' means we will not use the power spectrum, opting for the transfer function instead. '1' means we will use an input tabulated power spectrum, specified in the input parameters file and formatted as described in Section 2.3. Any other value means we will use an analytic form based on the cosmological parameters. Note that this must be set to '0' for non-Gaussian initial conditions

WhichTransfer: This is a flag to tell the code what type of initial transfer function we will use to generate the initial conditions. A value of '0' means we will not use the transfer function, opting for the power spectrum instead. '1' means we will use an input tabulated transfer function, specified in the input parameters file and formatted as described in Section 2.3. Any other value means we will use an analytic form. Note that this must *not* be set to '0' for non-Gaussian initial conditions and must logically cooperate with the value for `WhichSpectrum`, .i.e., they cannot both be '0' and if one is non-zero the other *must* be set to '0'.

FileWithInputSpectrum: This is the filepath and name for the file containing the input tabulated power spectrum as defined in Section 2.3. It is only used when `WhichSpectrum` = 1, however an argument must be supplied in all cases as the code does not know until runtime that it will not use it.

FileWithInputTransfer: This is the filepath and name for the file containing the input tabulated transfer function as defined in Section 2.3. It is only used when `WhichTransfer` = 1, however an argument must be supplied in all cases, as with `FileWithInputSpectrum`.

4.3 Cosmology

Omega: The total cosmological matter density, Ω_m, at z = 0. This includes the contribution from both Cold Dark Matter and Baryons. At the moment PICOLA can only simulate flat ΛCDM simulations and so we calculate the dark energy density, Ω_Λ, via $\Omega_\Lambda = 1 - \Omega_m$.

OmegaBaryon: The total baryonic density, Ω_b, at $z = 0$. This is also included as a component in `Omega`.

HubbleParam: The dimensionless Hubble parameter 'little h' at $z = 0$.

Sigma8: The linear matter variance at 8Mpch^{-1}, defined as the integral of the power spectrum with a spherical tophat window smoothed over 8Mpch^{-1}.

This is used to normalise the power spectrum/transfer function, though these may already be normalised correctly if we are using a tabulated input.

PrimordialIndex: This is the scalar index, n_s, used to tilt the power spectrum. As described in Section 2.3 it is assumed that an input tabulated power spectrum will already have the correct scalar index included and so this is only used if we generate the initial conditions using the transfer function or the code computes its own power spectrum. Note that for non-Gaussian initial conditions this *must* be set to $n_s = 1.0$, unless we use a generic input kernel.

4.4 Units

UnitLength_in_cm: This defines the length unit, in cmh^{-1}, of the code at output. All coordinates during the simulation are unitless and the physical units are not added in till the end. The recommended value for this parameter is $3.085678\mathrm{x}10^{24}$, which puts all the particle positions in Mpch^{-1}.

Unitmass_in_g: This is the mass unit, in gh^{-1}, of the code. As with the coordinates, the mass remains unitless until output. The recommended value for this is $1.989\mathrm{x}10^{43}$, which puts the default mass unit at $10^{10}M_{\odot}$.

UnitVelocity_in_cm_per_s: This is the velocity unit, in cms^{-1}, of the code. Like the above choices this is not added in until output. The recommended value is $1.0\mathrm{x}10^{5}$, which means particle velocities on output are in units of kms^{-1}.

InputSpectrum_UnitLength_in_cm: This defines the unit length of any tabulated input spectra in cmh^{-1}.This will only be used when a tabulated input power spectrum/transfer function is supplied, however it must be supplied as an input parameter all times This does not have to be equal to `UnitLength_in_cm`, but this is recommended.

4.5 Optional Extras

All of these input parameters deal with either non-Gaussian initial conditions or lightcone simulations, which can be turned on with the `LOCAL_FNL`, `EQUIL_FNL`, `ORTHO_FNL`, `GENERIC_FNL` or `LIGHTCONE` options found in the `Makefile`. If none of these have been turned on before compilation then the corresponding input parameters *must* be commented out or removed from the input parameters file.

4.5.1 Non-Gaussianity

These must only be uncommented/added-in when the `LOCAL_FNL`, `EQUIL_FNL`, `ORTHO_FNL` or `GENERIC_FNL` preprocessor options have been turned on.

Fnl: This is the value of F_{nl} (local, equilateral, orthogonal or generic) used to generate the non-Gaussian initial conditions.

Fnl_Redshift: This is the redshift at which we apply the non-Gaussian potential. This redshift must be greater than or equal to the initial redshift at which we generate the initial conditions, however there is no other limit on when the non-Gaussian potential comes into play.

FileWithInputKernel: This is the filepath and filename for the input non-Gaussian kernel for generic non-Gaussianity. See Section 2.3 for further details on this. This file is only required if the *generic* non-Gaussianity option is turned on with the `GENERIC_FNL` preprocessor option, otherwise it must be commented out or removed. This is true even when we are using other forms of non-Gaussianity to generate the initial conditions. Some example kernels for local, equilateral and orthogonal non-Gaussianity are included in the `/files/` directory.

4.5.2 Lightcone

These must only be included when the `LIGHTCONE` preprocessor option has been turned on.

Origin_x: This is the position of the lightcone origin in the x-direction. The origin does not have to be within the simulation box or indeed even within any of the replicated boxes.

Origin_y: This is the position of the lightcone origin in the y-direction.

Origin_z: This is the position of the lightcone origin in the z-direction.

Nrep_neg_x: This is the maximum number of times to replicate the simulation box in the negative x-direction. For an explanation of what replicating the simulation box actually means see Howlett et al. (in prep.) and the information under the `LIGHTCONE` preprocessor option in Section 3.

Nrep_pos_x: This is the maximum number of times to replicate the simulation box in the positive x-direction.

Nrep_neg_y: This is the maximum number of times to replicate the simulation box in the negative y-direction.

Nrep_pos_y: This is the maximum number of times to replicate the simulation box in the positive y-direction.

Nrep_neg_z: This is the maximum number of times to replicate the simulation box in the negative z-direction.

Nrep_pos_z: This is the maximum number of times to replicate the simulation box in the positive z-direction.

5 Memory Consumption

As with any large-scale parallel code it is annoyingly easily to rack up large amounts of memory if runs are not planned with extreme care. On top of this it is rarely easy to determine how much memory a code will use. Fortunately, PICOLA, like any Particle-Mesh code, only has two main contributions to the total amount of memory required: The storage of the particles, and the storage of the mesh. As the size of these is defined by the user it is relatively easy to calculate the maximum amount of memory a code will use if one knows the correct formulae. To this end we have included a python routine under `/files/PICOLA_mem.py` that we hope will prove useful and would recommend be used by anyone planning large scale runs of PICOLA. The code has quite a few input parameters that depend on the run parameters of the planned simulation and the compilation options, but these should be adequately described in the code and so shall not be repeated here.

6 List of Source Code, Files and Utilities

Here is a list and brief description of all the source files, utilities and example files included with PICOLA.

Utilities	
PICOLA_mem.py	A short python code for calculating PICOLA's memory requirements.
prepare_PICOLA.c	A C code for reading in the unformatted snapshot and and lightcone outputs from PICOLA.
prepare_PICOLA.f90	A Fortran-90 code for reading in the unformatted snapshot and and lightcone outputs from PICOLA.

Example Input Files	
run_parameters.dat	An example input parameters file.
output_redshifts.dat	An example file containing output redshifts.
input_spectrum.dat	A file containing an example input power spectrum.
input_transfer.dat	A file containing an example input transfer function.
input_kernel_local.txt	Contains the kernel for local non-Gaussianity.
input_kernel_equil.txt	Contains the kernel for equilateral non-Gaussianity.
input_kernel_ortho.txt	Contains the kernel for orthogonal non-Gaussianity.

Source Files	
2LPT.c	Routines for initialising the particles and force mesh and for generating the initial conditions.
auxPM.c	Particle-Mesh routines for calculating the inter-particle forces and moving the particles between processors.
cosmo.c	All the cosmological equations including growth factors, etc.
kernel.c	Routines for reading in the kernel for generic non-Gaussianity.
lightcone.c	Sets up the lightcone simulations, deals with the replicates, and indentifies and outputs particles that leave the lightcone.
main.c	The main routine for the code. Also includes the main timestepping loop, routines to update the particle's velocities and positions each timestep and routines to output snapshots.
power.c	Prepares the initial power spectrum or transfer function.
proto.h	Contains all the function prototypes for the code.
read_param.c	Reads in and checks the input parameters and output redshifts.
vars.h & vars.c	Contains all the global variables required for PICOLA.

7 Disclaimer

The speed and accuracy of PICOLA will be highly dependent on the specific parameters used at run time. We provide no warranty nor do we provide any guarantees that the code produces the correct results, beyond those shown in Howlett et al. (in prep.). Whilst every effort has been made to run PICOLA under a wide variety of circumstances and for many different simulations, it is possible that there will be cases where PICOLA fails to perform correctly or give satisfactory results. If this is the case then please contact us and we will endeavour to come up with a solution, or, if you would like, have a look at the source code and add in your own improvements. On top of this any recommendations provided within this guide, or within PICOLA's source code and associated files are based entirely on the author's opinion and do not claim to produce the best results. We encourage all users of PICOLA to perform their own rigourous tests on the code and discover how it performs best for their particular implementation.