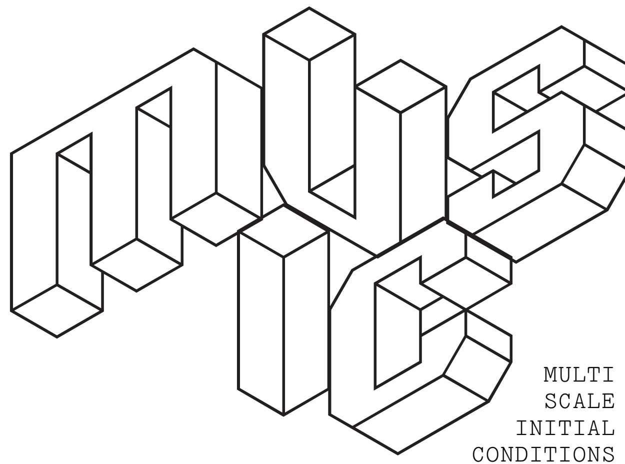


The MUSIC User's Manual

written by Oliver Hahn

version 1.0 - Tuesday, June 11, 2013

Table of Contents

Introduction 3

Description of the method 3

Cookbook for setting up a zoom simulation with MUSIC 3

Compiling MUSIC 4

Installing third-party libraries required by MUSIC 4

Installing FFTW (required) 4

Installing GSL (required) 5

Installing HDF5 (optional) 6

Adjusting the Makefile and Compiling MUSIC 6

Running MUSIC 8

The configuration file 8

The [setup] Section 8

The [cosmology] Section 10

The [random] Section 11

The [output] Section 12

The [poisson] Section 12

Output plug-ins 13

gadget 13

grafic2 13

enzo 14

tipsy 14

art 14

cart 14

nyx 14

generic 14

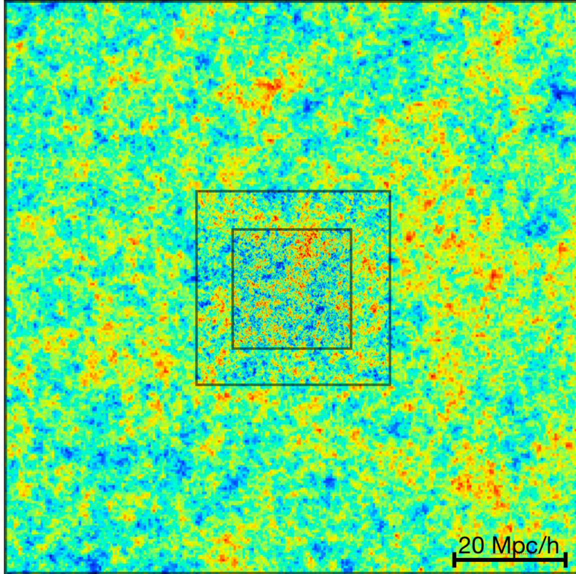
A sample configuration file 15

Extending MUSIC 16

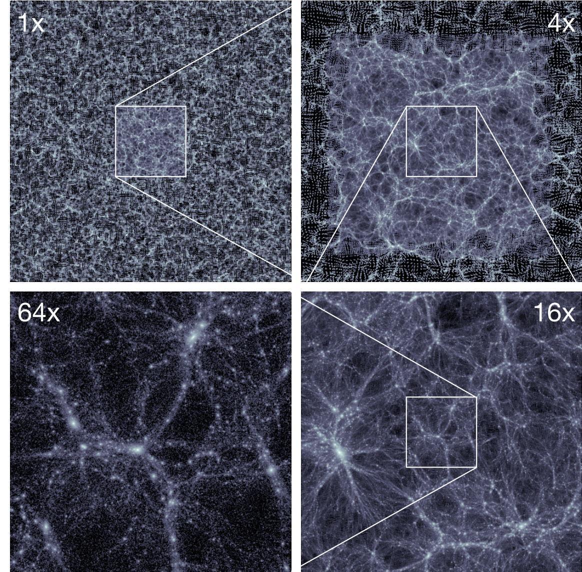
Adding a new output plug-in 16

Adding a new transfer function plug-in 16

Adding a new region generator plug-in 16



Density field in a 100 Mpc/h box with two initial levels of refinement generated with MUSIC.



Example of an N-body simulation of a deeply nested region of 6 initial levels generated with MUSIC and evolved with Gadget-2 to achieve an effective resolution of 8192^3 with 1160^3 particles in the high-res region.

Introduction

Description of the method

MUSIC (MUlti-Scale Initial Conditions) generates cosmological initial conditions for a hierarchical set of nested regions.

A detailed description of the method can be found in the code paper Hahn&Abel (2011), http://arxiv.org/abs/1103.6031. We kindly refer the reader to that paper for all technical aspects as well as performance and validation of the code.

Cookbook for setting up a zoom simulation with MUSIC

The procedure for setting up a zoom simulation follows typically the procedure of 4 steps, given below. Note that resolution levels in MUSIC are specified by their linear power-2 exponent, i.e. a resolution of 128^3 cells or particles corresponds to level 7 ($\log_2$ 128=7). We use the term “lower” for levels synonymously with “coarser” and “higher” with “finer”.

Run a unigrid dark matter-only pre-flight simulation

In order to set up unigrid initial conditions with MUSIC, select first the desired resolution for this pre-flight simulation. Assume we want to run a 128^3 simulation, the coarse grid level has to be set to $\log_2$ 128=7. Since we want to run a unigrid simulation, both `levelmin` and `levelmax` in section `[setup]` should be set to 7. Also the coarse grid seed needs to be chosen now and must not be changed afterwards. To do this, we set `seed[7]` in section `[random]` to the desired random seed. This seed determines the large scale structure and we will only add subgrid noise when performing refinement later. Now, set the box size, starting redshift, all the cosmological parameters and the input transfer function in the respective sections. Finally, select the output plugin in section `[output]` for the code with which you wish to perform this pre-flight simulation. Finally run MUSIC with the configuration file that contains all your settings and start your simulation. Note that you also have to explicitly specify a redshift at which to generate the initial conditions.

Identify a region of interest (e.g. a halo) and trace back the region to the initial conditions in order to obtain the extent of its Lagrangian patch.

Once the simulation has finished, the region of interest has to be identified. Save all dark matter particle IDs contained in the volume. In the next step, identify the positions of these particles in the initial conditions file as output by MUSIC. Compute the bounding box of these particles and allow for some safety boundary.

Set up MUSIC to resample this Lagrangian patch at the desired resolution and possibly with baryons.

Insert the coordinates of the lower-left corner of this bounding box into `ref_offset` in section `[setup]`, the lengths into ref_extent. Set the maximum refinement level in `levelmax`. **Do not change the seed you had specified for the full box simulation nor add new seeds for lower levels!** You can adjust `levelmin` if you do not specify seeds for these new coarser levels. Finally, add seeds for all additional finer levels including `levelmax` as `seed[...]` to section `[random]`. Add baryons, if desired, by saying `baryons=yes` in section `[setup]` and run MUSIC with this new configuration file.

Compiling MUSIC

Installing third-party libraries required by MUSIC

MUSIC requires a local installation of the FFTW and GSL libraries for its basic functionality. These are very common libraries and are usually found pre-installed at many scientific computing facilities. MUSIC makes use of the shared memory parallelization paradigm and is itself OpenMP parallelized. In order to also perform the FFTs in parallel, the multithread version of FFTW needs to be installed. In addition, the HDF5 (Hierarchical Data Format Library v5) can be installed. Note that some output formats (ENZO & the generic format) will be unavailable if the HDF5 library is not installed. The user will be guided through the necessary steps to perform a user-level installation of all these libraries in what follows.

Installing FFTW (required)

MUSIC uses the FFTW (Fastest Fourier Transform in the West) to perform the FFT-based convolution of white noise with the matter transfer function. MUSIC supports both FFTW2 and FFTW3 for multi-threaded shared memory machines. Note that FFTW2 does not support transforms of very large arrays (>32bit), so that for very large refinement volumes, FFTW3 needs to be used.

FFTW version 2

The library should be installed for multi-threaded machines to allow highest performance on multi-core processors. In order to download and install the library follow these steps:

1. Download the source code of the library in version 2.1.5 from the website (http://www.fftw.org) - or follow this direct link which may however be outdated in the future.
2. Open a terminal and expand the gzipped tar archive.
3. Run the configure script with the following parameters:
   ```
   ./configure --prefix=$HOME/local/ --enable-type-prefix --enable-threads --enable-float
   ```
   where $HOME/local will install the library into a directory “local” inside of your home directory. You should set this to a directory that exists and where you are allowed to write files.

Add “--enable-float” if you want to compile for single precision; omit it if you want to compile for double precision. Omit “--enable-threads” if you do not want to compile the multi-threaded version of the library.

4. Compile and install the library
   ```
   make install
   ```

5. If you want to install both the single and double precision versions, repeat from step 3 and omit/add --enable-float when configuring in step 3 in order to compile also for double/single precision.

FFTW version 3

The library should be installed for multi-threaded machines to allow highest performance on multi-core processors. In order to download and install the library follow these steps:

6. Download the source code of the library in version 3.x from the website (http://www.fftw.org) - or follow this direct link which may however be outdated in the future.

7. Open a terminal and expand the gzipped tar archive.

8. Run the configure script with the following parameters:
   ```
   ./configure --prefix=$HOME/local/ --enable-openmp --enable-float
   ```
   where $HOME/local will install the library into a directory “local” inside of your home directory. You should set this to a directory that exists and where you are allowed to write files.
   Add “--enable-float” if you want to compile for single precision; omit it if you want to compile for double precision.
   Omit “--enable-openmp” if you do not want to compile the multi-threaded version of the library.

9. Compile and install the library
   ```
   make install
   ```

10. If you want to install both the single and double precision versions, repeat from step 3 and omit/add --enable-float when configuring in step 3 in order to compile also for double/single precision.

Installing GSL (required)

MUSIC requires the GNU scientific library (GSL) to compute integrals and generate random numbers. To download and install the library, follow these steps:

1. Download the source code of the library from the website (lhttp://www.gnu.org/software/gsl/) - or follow this direct link to version 1.9 which may however be outdated in the future.

2. Open a terminal and expand the gzipped tar archive.

3. Run the configure script with the following parameters:
   ```
   ./configure --prefix=$HOME/local/
   ```
   where $HOME/local will install the library into a directory “local” inside of your home directory. You should set this to a directory that exists and where you are allowed to write files.

4. Compile and install the library
   ```
   make install
   ```

Installing HDF5 (optional)

Some MUSIC output plug-ins - such as ENZO and the MUSIC generic format - require the Hierarchical Data Format v.5 (HDF5). If you want to use any of these plug-ins, you need to install HDF5. Follow these steps to install:

1. Download the source code of the library from the website (lhttp://www.hdfgroup.org/HDF5/) - or follow this direct link to the most recent version which may however be outdated in the future.

2. Open a terminal and expand the gzipped tar archive.

3. Run the configure script with the following parameters:
   `./configure --prefix=$HOME/local/`
   where $HOME/local will install the library into a directory "local" inside of your home directory. You should set this to a directory that exists and where you are allowed to write files.

4. Compile and install the library
   `make install`

Remark: Boxlib and Nyx

Nyx requires boxlib to be installed. The path to the installed version should be given in the MUSIC Makefile. Further details are however beyond the scope of this manual.

Adjusting the Makefile and Compiling MUSIC

Ideally you have to only adjust the paths to third-party libraries and your compiler in the Makefile. Below are the relevant sections at the beginning of the Makefile. You might need to adjust the bold expressions:

```
################################################################################
### compile time configuration options
FFTW3              = yes
MULTITHREADFFTW    = yes
SINGLEPRECISION    = no
HAVEHDF5           = yes
HAVEBOXLIB         = no

################################################################################
### compiler and path settings
CC       = g++
OPT      = -O3
CFLAGS   = -Wall -fopenmp
LFLAGS   = -fopenmp -lgsl -lgslcblas
CPATHS   = -I. -I$(HOME)/local/include -I/opt/local/include -I/usr/local/include
LPATHS   = -L$(HOME)/local/lib -L/opt/local/lib -L/usr/local/lib
```

The first three options configure basic functionality of the code:

FFTW3	set to **`yes`** if you are using FFTW3, **`no`** if you are using FFTW2
MULTITHREADFFTW	set to **`yes`** if you configured FFTW with the `--enable-threads` option
SINGLEPRECISION	set to **`yes`** if you want to run MUSIC in single precision mode and you have compiled a version of FFTWx with `--enable-float`
HAVEHDF5	set to **`yes`** if you have installed the HDF5 library and want to create output for ENZO
HAVEBOXLIB	set to **`yes`** if you have installed the boxlib library and want to create output for Nyx, needs the path in BOXLIB_HOME

You also need to add the paths where you installed the third-party libraries mentioned in the previous section to `CPATHS` and `LPATHS`, just replace the bold parts.

Running MUSIC

MUSIC is run by specifying the path/name of a configuration file as a command line parameter, e.g.

```
./MUSIC my_ics.conf
```

All specifics regarding the setup of the initial conditions to be generated and options to control the code behaviour are given in this file. The available options are detailed below.

MUSIC can be run using multiple threads on multi-core shared memory machines. The number of threads that MUSIC is allowed to use is taken from the OMP_NUM_THREADS environment variable. This variable can be set for bash shell users by executing

```
export OMP_NUM_THREADS=4
```

to allow a maximum of 4 threads for any subsequent call of MUSIC during the current shell session. For tcsh users, the respective command would be

```
setenv OMP_NUM_THREADS=4
```

The configuration file

The MUSIC configuration file is organized into sections, whose names appear in square brackets and options, which are assigned a value with the equal '=' operator. Currently, the following sections exist:

`[setup]`	General setup for the simulation box and the refinement hierarchy as well as code behavior control
`[cosmology]`	Cosmological parameters and the transfer function are defined here
`[random]`	The random seeds and the behavior of the random number generator
`[output]`	The ouput plug-in is selected and parameters for output
`[poisson]`	Behavior of the multi-grid Poisson solver is controlled

The [setup] Section

General setup for the simulation box and the refinement hierarchy as well as code behavior control is defined using the options available in this section. There is support for various region specifications via plug-ins that can be selected via the `region` parameter. Currently, MUSIC implements two region generator plugins that will be described below.

`boxlength`	The size of the full simulation box in comoving Mpc/h	real
`zstart`	The starting redshift for the simulation	real
`region`	The method used to generate the refinement hierarchy. Currently the methods `box` and `ellipsoid` are supported. If this parameter is not specified, it defaults to `box`.	string
`levelmin`	The level of the coarse grid which covers the full simulation box. The number specified is the 2-log of the number of grid cells per dimension, e.g. a level of 7 corresponds to 2^7=128 cells/dimension.	int

`levelmax`	The maximum refinement level in the simulation. This value has to be larger or equal to levelmin. This sets the effective resolution in the refinement region to $2^{levelmax}$ cells/dimension	int
`levelmin_TF`	The level of the coarse grid when the density grid is computed (the convolution with the transfer function is performed). This can be set to a number larger than levelmin, the density field will be averaged down after the convolution has been performed and the Poisson solver is invoked. Essentially, this improves the accuracy of the coarser density modes when a low resolution in the coarsest level is desired in the final simulation.	int
`force_equal_extent`	Forces the refinement region to be a cube with edge length equal to the largest length determined from the region parameters.	bool
`padding`	Number of grid cells in intermediate levels (when using levelmax>levelmin+1) surrounding the nested grids. The extent of an intermediate level is N/2+2*padding if N is the number of cells in the next finer level.	int
`overlap`	Number of extra padding cells for subgrids when computing the transfer function convolutions. These are discarded when computing the displacements but greatly reduce errors due to boundary effects	int
`blocking_factor`	This parameter can be used to require the dimensions of initial grids to be multiples of `blocking_factor`. Not used if parameter is not present. This is mainly necessary to optimize for block based AMR.	int
`align_top`	Require subgrids to be always aligned with the coarsest grid? This is necessary for some codes (ENZO) but not for others (Gadget).	bool
`periodic_TF`	This controls whether the transfer function kernel is periodic or not. The convolution is always periodic. Should be set to yes.	bool
`baryons`	Set to yes if also initial conditions for baryons shall be generated	bool
`use_2LPT`	Set to yes if 2nd order Lagrangian perturbation theory shall be used to compute particle displacements and velocities	bool
`use_LLA`	Set to yes if the baryonic density field shall be computed using a second order expansion of the local Lagrangian approximation (LLA). See Section 5.3 in Hahn & Abel (2011).	bool
`center_vel`	*Experimental feature* to give the subvolume a kick opposite to its predicted motion over time to minimize movement of the high-resolution region with respect to the grid rest frame.	bool

Region 'box' (default)

The default region generator is the cuboid 'box' region, specified by giving the coordinates of the bounding box of the high-resolution region.

`ref_offset`	Offset of the refinement region. Given as three comma-separated real values in units of the box length, e.g. `0.3, 0.4, 0.14`. *You can only specify ref_offset or ref_center, not both.*	3*real

`ref_center`	Center of the refinement region. Given as three comma-separated real values in units of the box length, e.g. `0.3, 0.4, 0.14.` *You can only specify ref_offset or ref_center, not both.*	3*real
`ref_extent`	Extent of the refinement region. Given as three comma-separated real values in units of the box length, e.g. `0.3, 0.4, 0.14` *You can only specify ref_extent or ref_dims, not both.*	3*real
`ref_dims`	Instead of ref_extent, also the resolution of the high resolution region in grid cells can be specified directly. *You can only specify ref_extent or ref_dims, not both.*	3*int

Region 'ellipsoid'

Instead of the default cuboid regions, a set of points can be specified that define the high-resolution region. When using 'ellipsoid', a minimum volume bounding ellipsoid will be fit to these points. For particle codes, only high-resolution particles in this region will be written out, for grid codes, masks will be written out that reflect the shape of the ellipsoid.

`region_point_file`	ASCII file containing point coordinates of the particles in the Lagrangian patch of the region of interest. These need to be given in box coordinates, i.e. [0..1], one point per line.	3*real
`region_point_shift`	[optional] If the simulation from which the coordinates in region_point_file have been determined was already a zoom simulation, the coordinates in the IC file need to be shifted back to their original position (the zoom region had been centered). Specify here the shift that was applied in the centering of the zoom region, as has been output by MUSIC in the line, e.g. - Domain will be shifted by (-12, -12, -12) *If you specify the shift, also the levelmin of that simulation needs to be specified as below:*	3*int
`region_point_levelmin`	[optional, but req. if `region_point_shift` is specified] The `levelmin` of the zoom-in simulation that has been run to determine the points that determine the region.	3*real

The [cosmology] Section

Specific settings for cosmological parameters and physical properties are set in this section.

`Omega_m`	The total matter density parameter (now)	real
`Omega_L`	The cosmological constant density parameter (now)	real
`Omega_b`	The baryon density parameter (now)	real
`H0`	The Hubble constant (now), in km/s/Mpc	real
`sigma_8`	Normalization of the power spectrum	real
`nspec`	Power law index of the density perturbation spectrum after inflation	real

`transfer`	Name of the transfer function plug-in to be used. Depending on the choice, there are different additional parameters that need to be set. MUSIC comes with the following default plug-ins: • `BBKS` - for the Bardeen... fit to the transfer function without baryon features. • `eisenstein` - for the Eisenstein & Hu (..) fit for the CDM transfer function with baryon features • `CAMB` - for CAMB (...) output transfer functions (tabulated). The filename of the additional option `transfer_file` has to indicate the file from which the tabulated transfer function shall be read.	string
`YHe`	Helium abundance. This is used to compute the initial gas temperature if baryons are present. (Will only be used if the simulation code supports reading temperature fields). Optional. Default value: 0.248.	real
`gamma`	Adiabatic exponent. This is used to compute the initial gas temperature if baryons are present. (Will only be used if the simulation code supports reading temperature fields). Optional. Default value: 5/3	real

The [random] Section

In this section, the random seeds for the various levels should be given in the form seed[level] = seedval, i.e. e.g.

```
seed[7]=152521
seed[8]=532211
```

In this example, the random values generated for level 7 (i.e. a grid with 128^3 cells) are used as constraints for the next finer values, i.e. the large scale modes are determined by it. This is always true, for the lowest seed given. It is also possible to provide only a seed for level 8, in which case the random values will be averaged down outside the refinement region. Note that in the current implementation, seed values are intimately tied to the level, i.e. the same seed number given for two levels does not generate the same phases. The seed for the lowest level specified (which does not have to coincide with `levelmin`) determines the phases. Higher level seeds are just used to refine the noise. ***Thus, if you want to keep phases identical between simulations, do not change any of the seeds for any of the levels that the two simulations have in common!!!***

How does the parallel random number generator work?

The implementation of the parallel random number generator is as follows: In order to be able to draw reproducible random numbers without the need to always draw all numbers at the highest resolution, the grids are broken down into *subcubes* with independent random seeds. These seeds are computed from the level seed and the position of the subcube and are thus unique. Only random numbers for those subcubes which intersect regions of interest need to be drawn those effectively allowing for efficient and parallelizable random number generation on large grids. The size of the subcubes can be chosen via the parameter `cubesize`, not that changing `cubesize` for two simulations with the same seeds specified leads to different random numbers.

Random numbers from file

Instead of a seed number (integer), also a string can be given which gives the path and location of a file containing the white noise for the level. The file needs to contain $2^{3*level}$ numbers and should be in GRAFIC format, i.e. a FORTRAN unformatted file written by the following FORTRAN in the GRAFIC2 source code file ic4.f:

```
print*,'Writing random numbers used in ic4 to ',filename
open(11,file=filename,form='unformatted')
rewind 11
write(11) np1,np2,np3,iseed
do i3=1,np3
     write(11) ((f(i1,i2,i3),i1=1,np1),i2=1,np2)
end do
close(11)
```

where `np1`, `np2`, `np3` have to be equal to 2^{level} and `iseed` is an integer containing the seed (not used further), f is an array of double or single precision real numbers with the random values. **NOTE THAT FOR HISTORICAL REASONS, THERE IS A FACTOR OF -1 CURRENTLY WITH WHICH GRAFIC2 NOISE HAS TO BE MULTIPLIED IN ORDER TO GET COMPATIBLE ICS. This can be achieved inside MUSIC by specifying `grafic_sign=yes` in section `[random]`.**

The [output] Section

Output in MUSIC is performed using output plug-ins. The parameters that are given in this section will be parsed by each plugin itself. Therefore the specific parameters available depend on the output plug-in chosen. However, all output-plugins are required to work in a minimal setting, when no options are given besides a filename. Then:

`format`	the string identifier that selects the plug-in. Standard plug-ins are `generic`, `gadget`, `grafic2` (for RAMSES e.g.) and `enzo`.	string
`filename`	name and path of the file (or directory - depending on the plug-in) where the output data will be stored	string

Further options specific to the chosen output plugin are possible. They are listed below.

The [poisson] Section

`laplace_order`	order of the finite difference approximation for the Laplacian operator	int
`grad_order`	order of the finite difference approximation for the gradient operator	int
`accuracy`	the residual norm required to establish convergence of the multigrid solver	real
`smoother`	name of the smoothing sweep method used in the multigrid method: `gs` (Gauss-Seidel), `jacobi` (Jacobi), `sor` (Successive Overrelaxation)	string
`pre_smooth`	number of pre-smoothing sweeps	int
`post_smooth`	number of post-smoothing sweeps	int
`fft_fine`	controls whether the hybrid Poisson solver shall be used (see paper for details)	bool

Output plug-ins

gadget

For Gadget, the high resolution particles are written out as Gadget type 0 for gas (if baryons are enabled), and as type 1 for dark matter. The coarse level particles are all of type 5 and correspond to “total matter” particles, no SPH particles will be present outside the high-resolution region.

By default, initial conditions for Gadget are generated with the following units:

```
UnitLength_in_cm         3.08568025e24      ;  1.0 Mpc
UnitMass_in_g            1.989e43           ;  1.0e10 solar masses
UnitVelocity_in_cm_per_s 1e5                ;  1 km/sec
```

Alternatively, in the MUSIC parameter file, it is possible to specify the following extra parameters in section `[output]` which only apply to the Gadget-2 output plugin:

`gadget_usekpc`	if set to `yes` , the length unit will be in kpc, i.e. UnitLength_in_cm becomes 3.08568025e21, the velocity unit remains the same.	bool
`gadget_usemsol`	if set to `yes`, the mass unit will be in solar masses, i.e. UnitMass_in_g becomes 1.989e33, the velocity unit remains the same.	bool
`gadget_num_files`	this will split the initial conditions file into several files, necessary for very large particle numbers (>2^{31}), or for convenience reasons	int
`gadget_coarsetype`	Gadget particle type to be used for coarse particles (2,3 or 5), default is 5	int
`gadget_longids`	Use 64bit integers for particle IDs. Default if the parameter is not given is 32bit if less than 2^{32} particles, 64bit if more.	bool

grafic2

This is the format for RAMSES. The specific options for the GRAFIC output plugin in section [output] are

`ramses_nml`	if set to `yes` , a (**partial!**) RAMSES namelist file will be generated. It can be used as a starting point for the simulation namelist file. A mask (‘ic_refmap’) will be output that tags the masked refinement region.	bool
`ramses_old_nml`	if set to `yes` , a RAMSES namelist file will be generated containing the base grid structure. Instead of a mask field, a geometric refinement hierarchy will be written out. It can be used as a starting point for the simulation namelist file.	bool
`ramses_pvar_idx`	writes a refinement mask out as a passive variable field, this parameter specifies the index of the variable, filename becomes ic_pvar_<idx>, default is 1	int
`ramses_pvar_val`	value to which the passive variable should be initialized, default is 0.5	real

Note that Gadget-2 allows for a zoom-region particle mesh that can be enabled in the Makefile. Please consult the Gadget-2 manual for this option.

enzo

The ENZO output plugin has no extra parameters at this time. It outputs by default a template parameter file `parameter_file.txt` in the output directory which contains the grid set-up etc and can be used as a starting point for setting up a simulation. **It is not a complete parameter file to run a simulation, other parameters need to be added.** The sample parameter file `AMRCosmologySimulation.enzo` that comes with ENZO should be used for guidance.

tipsy

The tipsy output plugin has the extra parameters listed below:

`tipsy_eps`	this specifies how the softening is calculated for the particles. You can specified one real number that specifies the force softening for each particle in units of the mean linear particle separation on each refinement level, i.e. a value of 0.05 corresponds to 1/20 of the mean linear separation. The default value if the parameter is not given is 0.05.	real
`tipsy_native`	if set to `yes`, native output will be written out, if set to `no`, all data will be piped through XDR (External Data Representation) encoding. Default is `no`, i.e. XDR encoded values.	bool

art

The ART output plugin has no extra parameters at this time.

cart

The Chicago-ART output plugin has no extra parameters at this time.

nyx

The nyx output plugin has no extra parameters at this time.

generic

The generic plugin can be used to output simple HDF5 data arrays for all fields that can be readily used for subsequent analysis/visualization with other tools.

A sample configuration file

This creates a 100Mpc/h box with a 128^3 base grid and a refinement region at 256^3 effective resolution with a corner point at (40,40,40) Mpc/h and a side length of 20 Mpc/h in all directions.

```
[setup]
boxlength            = 100
zstart               = 50
levelmin             = 7
levelmin_TF          = 7
levelmax             = 8
padding              = 8
overlap              = 4
ref_offset           = 0.4, 0.4, 0.4
ref_extent           = 0.2, 0.2, 0.2
align_top            = yes
baryons              = no
use_2LPT             = no
use_LLA              = no
periodic_TF          = yes

[cosmology]
Omega_m              = 0.276
Omega_L              = 0.724
Omega_b              = 0.045
H0                   = 70.3
sigma_8              = 0.811
nspec                = 0.961
transfer             = eisenstein

[random]
seed[7]              = 12345
seed[8]              = 23456

[output]
##generic FROLIC data format (used for testing)
#format              = generic
#filename            = debug.hdf5

##ENZO - also outputs the settings for the parameter file
#format              = enzo
#filename            = ic.enzo

##Gadget-2 (type=1: high-res particles, type=5: rest)
##no gas possible at the moment
format               = gadget2
filename             = ics_gadget.dat
shift_back           = yes


##Grafic2 compatible format for use with RAMSES
##option 'ramses_nml'=yes writes out a startup nml file
#format              = grafic2
#filename            = ics_ramses
#ramses_nml          = yes

[poisson]
fft_fine             = no
accuracy             = 1e-5
pre_smooth           = 3
post_smooth          = 3
smoother             = gs
laplace_order        = 6
grad_order           = 6
```

Extending MUSIC

Adding a new output plug-in

New output plugins can be added any time to the directory `plugins`. Note that the naming convention is that output plugins start with "`output_`". The output plugins need to derive from the class `output_plugin`, declared in file `output.hh`, and implement all the `write_...` functions. The implementation of the grafic2 plugin in file `plugins/output_grafic2.cc` can be taken as a simple example. Note, that the plugin has to register itself using the abstract factory pattern with the plugin creator by adding a line similar to this to the end of the new plugin file:

```
namespace{

output_plugin_creator_concrete<my_output_plugin> creator("myplugin_name");

}
```

When MUSIC is recompiled (using make), the plugin will be automatically available by specifying

```
[output]
  format = myplugin_name
```

in the parameter file, no other MUSIC source file needs to be touched.

Adding a new transfer function plug-in

New transfer functions, e.g. new fits, modified fits, or new reading modules for tabulated functions, can be added via the plug-in mechanism. These plugins need to derive from the class `transfer_function_plugin`, defined in `transfer_function.hh`. Examples can be found in the plugins directory.

Adding a new region generator plug-in

New region generators can be added via the plug-in mechanism. These generate a bounding box that initializes the MUSIC hierarchy. They are queried cell-by-cell upon output to provide support for masking. The region generator needs to derive from the class `region_generator_plugin`, defined in `region_generator.hh`. An example is the ellipsoid generator in `plugins/region_ellipsoid.cc`.