

How to Simulate: A Beginner's Rundown

Shea Garrison-Kimmel, Miguel Rocha, Oliver Elbert

January 27, 2016

1 Introduction

This is *NOT* intended to be a replacement for the user's guides of the various codes that will be discussed here, but hopefully it will give some helpful hints and help you understand the user's guides more effectively.

This guide is going to use the following codes, with the dependencies listed:

1. MUSIC (<http://www.phys.ethz.ch/~hahn/MUSIC/>)
2. Gadget (<http://www.mpa-garching.mpg.de/gadget/>)
3. AHF (<http://popia.ft.uam.es/AHF/Download.html>)
4. rockstar (<https://code.google.com/p/rockstar/>)
5. consistent-trees (<https://code.google.com/p/consistent-trees/>)

To compile these codes, you'll first need to compile the following dependencies. Hopefully, these packages don't have unmet dependencies of their own – typically they don't, but if they do... good luck!

1. an MPI; I recommend MPICH if there isn't a satisfactory one installed already (<http://www.mpich.org/downloads/> – grab the top one)
2. FFTW, versions 2 **and** 3 (<http://www.fftw.org/>)
3. GSL (<http://www.gnu.org/software/gsl/>)
4. optionally, HDF5 version 1.6 (version 1.8 does **not** work with Gadget) (<http://www.hdfgroup.org/HDF5/>)

The basic steps of running a cosmological zoom-in simulation from start to finish are listed below. I'll describe them all in the sections that follow.

1. Choose a problem of interest (mass range, environment, etc.)
2. Select your box size and starting redshift
3. Create unigrid initial conditions at medium resolution (probably 512^3 particles) with MUSIC – ~ 1 hour anywhere
4. Run said ICs with Gadget, probably saving only a few timesteps – ~ 1 week on Greenplanet or less on NASA/XSEDE

5. Halo find on that box with **AHF** (because **rockstar** is non-periodic by default) ~ 1 hour anywhere
6. Make cuts on your halo catalog to find an object that meets the criteria set out in Step 1. If there aren't any, go back to step 3 and repeat with a new random seed.
7. Select the Lagrangian volume of that object – ~ 5 minutes anywhere
8. Create zoom-in initial conditions at higher resolution (probably not yet at the full resolution you're aiming for) using the same random seed with **MUSIC** – $\sim 1 - 2$ days on Blacklight
9. Run those initial conditions with **Gadget** – could take any amount of time
10. Run **rockstar** on the output snapshots
11. If everything went ok and you're not yet at the targeted resolution, go back and recreate your ICs at even higher resolution.
12. Run **consistent-trees** to create a merger tree
13. Do your analysis – turn the merger tree into something you can use and make some science!

2 Prerequisites: Compiling Dependencies and Setting Environment Variables

Here I'll give brief instructions for how to compile the dependencies for **MUSIC** and **Gadget**. **rockstar**, **consistent-trees**, and **AHF** don't really have any dependencies and should compile out of the box.

Throughout this document, I'm going to assume that you do not have root permissions and thus have to compile your codes from scratch and can't put them in `/`. Instead, we'll be directing all of our compiled binaries and libraries to live in `$HOME/code/compiled/`.

2.1 MPI

If there's an MPI installed already, you might want to give it a shot. That said, in 2010, I had difficulties with the MPI's pre-installed on Greenplanet. If you use a pre-installed MPI, then everywhere that you see either

```
-I/$(HOME)/code/compiled/include
```

or

```
-L/$(HOME)/code/compiled/lib
```

below, you should add

```
-I/path/to/mpi/include
```

and

```
-L/path/to/mpi/lib
```

Now, if you need to compile your own MPI, don't be scared — it's not actually that hard...assuming the system that you're working on has a good version of GCC installed.

1. Download and unpack the code with `wget <url>` and `tar -xvf <filename>`
2. Configure the code by `cd`-ing into the directory and running
`./configure --prefix=$HOME/code/compiled`
3. Make and install the binaries with `make && make install`.

2.2 Environment Variables

Now that MPI is installed, we need to tell your system (bash) to use that installed package. Specifically, that means that we have to change two environment variable, `PATH` and `LD_LIBRARY_PATH`. The former tells bash which folders to search for executables (e.g. `cd`, `make`, `python`, `mpicc`); the latter tells the LD library linker which folders to search for libraries at compile and run time.

You can edit your variables for a single session via the command line, but we want these definitions to stick when we open a new command line, so put them in either your `$HOME/.bashrc` or `$HOME/.bash_profile`. We also don't want to eliminate what's already in those paths; instead we want to prepend to those variables. So, add the following lines to one of those files:

```
export PATH=$HOME/code/compiled/bin:$PATH
export LD_LIBRARY_PATH=$HOME/code/compiled/lib:$LD_LIBRARY_PATH
```

and run `source $HOME/.bashrc` to update your variables.

2.3 FFTW

2.3.1 FFTW2 for Gadget

Gadget can only use version 2 of FFTW (currently 2.1.5), whereas **MUSIC** needs FFTW version 3 for multi-threading reasons. Let's start with FFTW2, which depends on your MPI installation and thus will tell you whether or not you did the previous step correctly.

Again, download and untar the code, then configure, compile, and install with

```
./configure --prefix=$HOME/code/compiled --enable-mpi --enable-  
type-prefix && make && make install
```

The second option tells FFTW to name the files to indicate that they're in double precision, which **Gadget** typically expects. If that worked correctly (which it should have), then awesome, and let's do it again with single precision, just to have it:

```
make clean && ./configure --prefix=$HOME/code/compiled --enable-  
mpi --enable-type-prefix --enable-float && make && make install
```

2.3.2 FFTW3 for MUSIC

Now let's get FFTW3 installed and compiled for MUSIC. We want to make sure that it has multithreading capabilities, and it won't hurt to get it working with MPI as well. Again, download and untar the code, `cd` into the directory, and do:

```
./configure --prefix=$HOME/code/compiled --enable-openmp && make  
&& make install
```

Don't worry – it won't overwrite your FFTW2 include files. FFTW3 files are named differently. You can also add `--enable-mpi` and/or `--enable-threads` if you'd like – if FFTW-2 compiled, these should compile without issues.

2.4 GSL

GSL is super easy and has no dependencies (that I know of). Simply download and untar as usual and run

```
./configure --prefix=$HOME/code/compiled && make && make install
```

3 Selecting the boxsize and starting redshift

When selecting the boxsize, you have to balance the computational costs with mass resolution gains. That is, the density in the box is fixed, so increasing the box size will add mass to the simulation and either push up the number or mass of particles. However, you don't want to go too small, because numerical issues will crop off and you won't get a fair sample of the universe. That specific “too small” is hard to calculate, but a rule of thumbs that I follow is $L_{\text{box}} \gtrsim 5 \text{ Mpc}/h$ for dwarfs and $L_{\text{box}} \gtrsim 25 \text{ Mpc}/h$ for MW-size galaxies. Of course, increasing your box size will also find more objects that might meet your criteria.

You also want to make sure that you will have enough particles in your object of interest at the unigrid resolution ($n_p = 512^3$) to accurately compute the Lagrange volumes. That means that you should

1. Calculate the particle mass for a few box sizes for a fix number of particles
2. Find out how many particles would be in a halo of your targeted mass for each box size, and eliminate any box sizes that have fewer than ~ 3000 particles

I won't go further into the various arguments for larger and smaller boxes here, because it can get very lengthy.

The starting redshift is a bit more concrete, and a cursory explanation is given in Section 6 of Jose's paper. It depends on the box size and resolution — higher mass particles means a lower starting redshift, essentially.

4 Creating unigrid ICs

Now you're ready to create initial conditions for your cosmological box, from which you're going to select zoom-in targets. We'll use MUSIC for this, so download the code and `cd` into the directory, replace the `Makefile` with the included

`Makefile.music`, run `make`, and you should get a `MUSIC` binary. The only major change I've made in the included Makefile is to add the `include` and `lib` directories that we installed FFTW into. That is, I've put

```
-I$(HOME)/code/compiled/include
-L$(HOME)/code/compiled/lib
```

at the start of `CPATHS` and `LPATHS`, respectively. This points `c++` to the proper header files and libraries for FFTW3 that we just installed.

You'll also need to create a `.conf` file, which is a `MUSIC` configuration file. There should be a sample included with this document named `fullbox.music.conf`, and there is also a sample in the `MUSIC` user's guide. I'll run through the options that I think are important for a fullbox simulation here and that you're likely to change, but I also want to note that you shouldn't move options between sections (which are delineated by bracketed keywords, e.g. "[setup]") and you **absolutely should** read the user's guide:

boxlength The size of the box. Assumed to be in `Mpc/h` but the distance unit can be changed to `kpc/h` by putting `gadget.usekpc = True` in the [output] section.

zstart The initial redshift.

levelmin Sets the number of particles that you want in your low resolution box. There will be $(2^{\text{levelmin}})^3$ particles in the box.

levelmax Set this equal to `levelmin` for now.

ref_extent and **ref_offset** Set these to "0,0,0" for now

cosmology The cosmology section is largely self-explanatory. The transfer function, however, must either be selected from one of the options built into `MUSIC`, which are listed in the user's guide (I've found `eisenstein` to work fine), or you can set

```
transfer = camb_file
transfer_file = path/to/file.dat
```

and use a file produced by CAMB for your specific cosmology and redshift. There should be a set of files for the WMAP-7 cosmology included with this document, but if you're using another cosmology, you're on your own. That said, `eisenstein` will probably work fine.

random section The random section essentially determines the box. Choose a six digit seed for the level that you've initialized the box at, and change it if you want to try another box. That is, if you've set

```
levelmin = X
```

then you should set

```
seed[X] = any six digit number
```

disk_cached If you have plenty of RAM available, set this to no. If you're nearing your limit, though, set it to yes. Essentially, this stores the random numbers generated to a file, rather than keeping them in RAM forever. However, it slows your IC generation down significantly because MUSIC will spend a lot of time reading from and writing to the disk.

output section Set `format = gadget2` and choose a filename.

I've always left the poisson section identical to that in the sample parameter file.

Now you can run MUSIC. If you're in an interactive session, that means simply running

```
export OMP_NUM_THREADS=<number of processors to run on>
./music/music <configuration file>
```

Otherwise, you should wrap those command in a PBS file; there should be a sample provided with this document called music.pbs.

5 Running the unigrid ICs

Now you're ready to run those initial conditions with **Gadget**. Let's start by compiling the code, so let's again `cd` into the directory replace the `Makefile` with the one included, `Makefile.GadFullBox`. There are a number of compile time options that I've set at the top that should be a working starting point for most simulations. Specifically, I've set the following options on – you shouldn't have to change anything except for possibly the value that `-DPMGRID` takes.

- `-DPERIODIC`
- `-DPEANOHILBERT`
- `-DWALLCLOCK`
- `-DPMGRID=XXX`, where $XXX = 2^{\text{levelmin}}$
- `-DDOUBLEPRECISION`
- `-DDOUBLEPRECISION_FFTW`
- `-DSYNCHRONIZATION`

I've also created a new `SYSTYPE`, which sets the include and library paths to point to `$HOME/code/compiled/include` and `$HOME/code/compiled/lib`. Now run `make` and you should get a **Gadget2** binary.

Next we need to set up a **Gadget** parameter file. Again, there should be a sample included with this document, but you should **100% absolutely** read the user's guide. I'm not going to run through most of the options; I'll instead focus on those that you're likely to have to change. Do NOT take this as ringing endorsement of the values for the other parameters (though they seem to work ok).

InitCondFile Set as either the relative (from where the run is initiated) or absolute path to the initial conditions file produced by MUSIC above.

OutputDir Set as either the relative or absolute path to an existing directory that you want snapshots and restart files to be saved in.

SnapshotFileBase The name of your output snapshots, so files will be named OutputDir/snapshot_XXX.

OutputListFilename The path to a file that contains a list, one per line, of scale factors at which you want outputs. An example is included as a_out_short.txt.

TimeLimitCPU The length of time the run can go for, in seconds.

TimeBegin The scale factor at which you're starting the simulation

Omega0, OmegaLambda, OmegaBaryon, HubbleParam The cosmology of the simulation; ought to match what you used in the .conf file, but here HubbleParam is h .

BoxSize The size of the box, in the length units of the simulations.

UnitLength_in_cm The length units that you want to use; defaults to Mpc/h, but you can set it to kpc/h if you like.

SofteningHalo The softening length that you want to use for the run. Typically you want to set it to ~ 4 times the Power (2003) radius for the mass of halo that you're targeting.

As an aside, the UnitLength and UnitVelocity together define the time unit for isolated simulations (in cosmological simulations, Time is a , the scale factor).

Now you can finally run the simulation. If you're in an interactive session, you can do

```
mpirun -np 4 ./Gadget-2.0.7/Gadget2/Gadget2 <num procs> gadget.param
```

Otherwise, you should wrap that in a PBS script like the included gadget.pbs.

6 Halo Finding on the Full Box

I recommend running AHF on the $z = 0$ snapshot – it is very easy to run AHF with periodic boundary conditions, whereas doing so with **rockstar** is annoying. The main files that you will edit when using AHF are:

Makefile.config The file where you set your defineflags and tell AHF to run in OpenMP or MPI

AHF.input The file that tells AHF where to look for the inputs and where to save the outputs (among other things)

DO NOT mess with the AHF Makefile proper. They warn you about this in the user guide, and with good reason.

In **Makefile.config** the main thing to do is set the defineflags. For a fullbox like this the only flag needed is **DPERIODIC** to set periodic boundary conditions. The **Makefile.config** included here has sets of defineflags what they're used for. The defineflags under **DM Zooms** are used for running AHF on a snapshot from

a DM Zoom-in simulation. Those under `Halo`sGoingMad were used for snapshots in the Halo

s Going Mad Project. You can simply uncomment the set under whichever section is relevant to what you’re doing (here that would be `DM Fullbox`). The rest should be set for AHF to run on greenplanet.

The `AHF.input` file is slightly more interesting. The main things you’ll want to edit are:

ic_filename The location of the snapshot you want to run on. Should be of the form: `path/to/snap/snapname`

ic_filetype An integer that tells AHF what kind of file it’s looking at. Should be 60 for a Gadget snapshot.

outfile_prefix The first part of the name of for the files AHF outputs. Should be `/desired/location/name`

Other important items are: `NminPerHalo`, the minimum number of particles for a halo, `RhoVir`, which sets the normalisation for AHF’s densities, and `Dvir`, which tells AHF the overdensity to use to define the virial radius.

Because AHF normalizes densities it is important to keep track of whether `RhoVir` is set to 0 or 1. If set to 0, densities will be normalized to the critical density at the snapshot’s redshift. If set to 1 the background density will be used instead. If you want to use AHF’s density information here or on the zoom you’ll have to multiply by whichever you chose to normalize by. Either works, but do be sure to keep track of which you used.

Once you have your files set, enter `make` to compile AHF, and then you simply need to enter `./bin/AHF-v1.0-043 AHF.input` (preferably in a pbs script) in the AHF directory for it to run. On greenplanet this should take a few hours using an 8-core node, at which point you will have a bunch of output files to add to the IRATE file using `ahf2irate`.

Alternatively, you can use the method described below to run `rockstar`, but set the first snapshot analyzed to the $z = 0$ particle data.

Either way, you’ll end up with a halo catalog that you can search for an object that meets your criteria. You’ll probably end up with hundreds. Unless your problem has an environmental dependence, you probably want one of those criteria to be isolation, and it’s probably a good idea to sort by that measure, then take the best ~ 5 onto the next section.

7 Lagrange Volumes

A Lagrange volume is the volume that a group of particles occupied at an earlier time; in zoom-in simulations, we generally mean the volume occupied by the particles around a halo at $z = 0$ in the initial conditions so that we can put more particles in that part of the box and lower the resolution elsewhere.

Included with this document is a script, `calc_lagrange_vol.py`, that implements this idea. It is tailored to an IRATE file that contains a halo catalog, the $z = 0$ particle data, and the particle data of the ICs, but you should be able to apply the ideas in it to another setup if need be.

I recommend calculating the size of the volumes for the halos selected earlier and finding the smallest. Specifically, (barring environmental considerations) I wouldn’t stop until you find a halo that’s smaller than relation given in Halo 1

in Jose’s paper with the parameters in Table 2. This won’t necessarily speed up your run, but it will reduce the memory and storage constraints.

8 Creating Zoom-in ICs

Once you’ve picked the smallest Lagrange volume, it’s time to run a zoom-in simulation. Start by making a copy of your `.conf` file, and then let’s start editing it (the file `zoom_music.conf` is an example of an edited `.conf` file). Specifically, you’ll want to change the following parameters:

levelmin You’ll probably want to lower this from the resolution of the fullbox. How much varies; we’ve found that in a 50 Mpc/h box, setting it to 7 works well.

levelmax You’ll want to raise this to the resolution that you’re aiming for. You probably don’t want to jump directly to your production resolution, but instead run a test at an intermediate resolution.

red_offset The position of the corner of the zoom-in region, in units of 0 to 1. The Lagrange volume script will spit out a set of three numbers for this – just paste them in.

ref_extent The size of the zoom-in region, again in units of 0 to 1. You can again paste this from the output of Lagrange volume script.

Basically, **levelmax** set the resolution of the zoom-region and **ref_extent** and **ref_offset** set the position and size of the box. Another parameter, **padding** controls the size of the intermediate regions, but I recommend leaving it at 6 — we found that worked best in Jose’s paper. Do **NOT** change the seeds that you’ve put in already – if you do, you’ll get a completely different box. You can add seeds at higher levels if you’d like – read about the benefits in the user’s guide.

There are other ways that you can initialize the zoom-in region, such as with an ellipsoid, but we haven’t tested those methods and I’ve never used them. You should read the **MUSIC** user’s guide if you want to try them – I believe they basically work by reading in an ASCII file with positions that must be in the zoom-in region.

You’re now almost ready to run your zoom-in simulation, but first we want to separate the particles into different groups (halo, disk, bulge, star, and boundary) in the **Gadget** IC file so that we can give each particle mass a different softening length. For that, you’ll want to run your initial conditions through one of the `split_gb.py` scripts included with this document. Unfortunately, each is hard-coded to the number of resolution steps in the file. Each is named according to the number of resolution levels that there are, which is equal to $(\text{levelmax} - \text{levelmin}) + 1$, since it’s inclusive. We’ll be putting that number of resolution levels into four particle groups, since we want to leave star empty in case we want to rerun with star formation later.

So, as an example, if you set `levelmin = 7` and `levelmax = 12`, then you should use `split_gb_6to4.py`. There are six particle masses in the file and you want to put them into the four available groups. If there’s no pre-written script available for what you want to do, I recommend tweaking one of the given scripts to handle your problem.

9 Running the Zoom-in Simulation

Now you're finally ready to run your zoom-in simulation! You'll want to start by recompiling **Gadget** – there are a few compile time options that we'll want to change. I recommend creating a new folder and copying over the **Gadget** folder wholesale, then editing the Makefile to enable the following options:

-DUNEQUALSOFTENINGS

-DPLACEHIGHRESREGION=3

-DENLARGERECTION=1.2

Save and run `make clean && make`.

Now let's edit our parameter file. All we need to do is edit the name of our initial conditions file, probably the output file, and the softening lengths. Specifically, you need to set a softening length for each group that contain particles. There's again an example provided as `zoom_gadget.param`.

Lastly, just edit your PBS script to run with more processors, and you're now running a zoom-in simulation (from scratch)!

10 Post Processing: rockstar and consistent-trees

You're done now, but you probably want to make another halo catalog and likely a merger tree. You can run **AHF** again if you'd like, but here I'll talk you through how to run **rockstar** on all of the snapshots. Note that **rockstar** only halo finds on the high resolution particles, so the halo catalog will be incomplete and/or wrong in regions where low-resolution particles exist.

rockstar again has a configuration file that you'll have to make some a couple changes to, but the provided `rockstar_parallel.cfg` should give you a good starting point. You'll need to change

INBASE Path to the folder that holds the snapshots

OUTBASE Path to a folder that you want **rockstar** to write to

NUM_SNAPS The total number of snapshots, assumed to include 0, that your simulation contains.

STARTING_SNAP The first snapshot that you want **rockstar** to analyze.

NUM_BLOCKS Number of files per snapshot.

NUM_READER Number of reading processes. Best to set this the same as **NUM_BLOCKS**.

FILENAME The name of each snapshot. `<snap>` means the snapshot number and `<block>` means the number of the file in that snapshot. So, `snapshot_<snap>.<block>` will look for files named `INBASE/snapshot_0.0` through `INBASE/snapshot_0.7`, if **NUM_BLOCKS** = 8.

NUM_WRITERS The number of processors that you want to analyze the data for halos.

FORCE_RES The force softening in your high-res region.

You will also have to update the cosmology if you're not using WMAP-7.

Running **rockstar** means starting a master processes, the reader processes, and the writer processes. The included `rockstar.pbs` should give an idea of how to do that – basically, you run a single **rockstar** process with the `rockstar_parallel.cfg` file that you created above, then start `NUM_READERS` and `NUM_WRITERS` processes with the `OUTBASE/auto-rockstar.cfg` configuration file that **rockstar** will automatically create.

You can also use **AHF** for halo finding on the zoom. It performs some useful calculations for you in addition to halo-finding (e.g. density profiles and circular velocities), and is essentially the same as running on the fullbox. You only have to edit `Makefile.config` and set the defineflags to:

-DPERIODIC

-DMULTIMASS

as now we have particles of multiple masses. Again, the easiest way to do this is to comment the defineflags under **DM Fullbox** and uncomment the ones under **DM Zooms**.

Running **rockstar** can take a pretty long time, though usually not as long as running **Gadget**. Once you're done, you'll probably want to put the snapshots together into a merger tree. For this, start by running, e.g.,

```
./Rockstar-0.99.9/scripts/gen_merger_cfg.pl rockstar_parallel.cfg
```

This will initiate a perl script that will rework the parameters in your **rockstar** configuration file into a consistent-trees configuration file. It'll also give you instructions on how to run **consistent-trees**, which basically amounts to compiling it and running another perl script.

Once you have the merger tree, it's time to do science! Unfortunately, you're also largely on your own at this point. I have some scripts to convert said merger trees into an **IRATE** file, but they're poorly written and I'd rather not generally share them (but can make them available to people individually). Instead, you should think carefully about what data you need from the simulation, and work to extract that.

