User Guide for NEXUS

-The Cosmic Web Identification Tool-

version 1.0

Marius Cautun*
Kapteyn Astronomical Institute,
University of Groningen, Netherlands

January 10, 2013

*e-mail: cautun@astro.rug.nl

Contents

1 Introduction

For a description of the NEXUS and NEXUS+ methods see Cautun et al. (2012). The program presented here implements the algorithm described in the paper.

1.1 RAM requirements and CPU time

The `NEXUS` software does not have any formal requirements than the ones necessary to install the libraries on which this program depends on. For the external library requirements you have to check the library documentation pages.

Running `NEXUS` is not very demanding, since the code only needs to store about 12 scalar quantities for each grid cell. The most memory intensive and CPU demanding part of the method is running the `MMF_respone` program which computes the environment signature over a set of scales. This means that for a 256^3 grid the program needs 0.8GB of RAM while for a 512^3 grid it needs 6.5GB. The `MMF_respone` program needs for each smoothing radius about 30s and 500s CPU time for a grid of 256^3 and 512^3 cells, respectively. While this code makes use of the `OPENMP` parallelisation procedure, most of the time is taken by the Fourier Transform computation and hence going to multiple cores brings only a limited speed up, especially after 12 cores.

2 Installation

This section gives information about the external libraries needed by the `NEXUS` code and the steps necessary to compile and run the test examples. For most users, these steps should go pretty fast and without problems. In case that you do get errors, I can offer help regarding the compilation and running of the `NEXUS` code, but not for problems related to installing the external libraries. For that I recommend that you contact the specific library help pages or googling for the solution in the hope that other users had the same problem (this works in most cases).

2.1 External libraries

The `NEXUS` code depends on 3 external libraries: **GSL**, **Boost C++** and **FFTW** - these libraries are usually already installed on most systems. In the case of the **FFTW** library it needs to be installed with support for both double and single precision floats, which is not always the case. The following will give a very short description on where one can download the libraries and how they can be installed on your system. All the instruction presented below are for the Linux operating system, but similarly straightforward instruction can be found in the library packages for Mac or Windows.

Before compiling the program one needs to have installed the following libraries:

- **GNU Scientific Library (GSL)** which can be downloaded at `http://www.gnu.org/software/gsl/` (usually this library is already installed on Linux systems). To build the

library one needs to do the following easy steps (for additional information and options check the file `INSTALL` that comes with the library distribution):

1. Go to the directory where you untared the GSL library.
2. Type the configure command `./configure` . This will build the library in the default system library path, so you need root privileges for this. If you don't have a root account, use `./configure --prefix=FOLDER_PATH` where `FOLDER_PATH` is the path to the directory of choice where to install the library.
3. Type `make` and after that `make install`. Now you have successfully built the **GSL** library.

- **Boost C++ Libraries** which can be found at `http://www.boost.org/users/download/` (usually this library is already installed on Linux systems) The code was successfully build using Boost version 1.44, but should work with higher versions too. To build the library on a Linux system, on must follow the steps (more detailed information can be found in the `INSTALL` file or in the online documentation):

  1. Go to the directory where you untared the library.
  2. Type the configure command `./bootstrap.sh` . This will build the library in the default system library path, so you need root privileges for this. If you don't have a root account, use `./bootstrap.sh --prefix=FOLDER_PATH` where `FOLDER_PATH` is the path to the directory of choice where to install the library.
  3. Type `./bjam install`. Now you have successfully built the **Boost C++ Libraries**.

- **FFTW (Fastest Fourier Transform in the West)** which can be found at `http://www.fftw.org/download.html`. The code was succesfully build using FFTW version 3.2.2 and it should work successfully with higher versions too. For a successful build one must do the following (more detailed information can be found in the `INSTALL` file):

  1. Go to the directory where you untared the library.
  2. First install the library for double precision floats. Type `./configure --prefix=FOLDER_PATH --enable-type-prefix --enable-threads && make && make install` to build the library for doubles. `FOLDER_PATH` is the path to the directory of choice where to install the library.
  3. In this step you build the library for floats. Start by cleaning the installation from the previous step: `make clean`. After which type: `./configure --prefix=FOLDER_PATH --enable-single --enable-threads && make && make install` to build the library for floats.

In case that you did not install the libraries in the default system path, one must add the location of the libraries and include files to the system environments. Below are two examples of how one can do so for two common shells (`LIB_DIR` denotes the directory where the library build is located):

⋆ For the **tcsh** shells one needs to add the following lines in the `~/.cshrc` file:

```
setenv CPPFLAGS -I/LIB_DIR/include:$CPPFLAGS
```

```
setenv LD_LIBRARY_PATH /LIB_DIR/lib:$LD_LIBRARY_PATH

setenv LDFLAGS -L/LIB_DIR/lib:$LDFLAGS
```

⋆ For the **bash** shells one needs to add the following lines in the ∼/.profile file:

```
CPPFLAGS=-I/LIB_DIR/include:$CPPFLAGS

LD_LIBRARY_PATH=/LIB_DIR/lib:$LD_LIBRARY_PATH

LDFLAGS=-L/LIB_DIR/lib:$LDFLAGS
```

NOTE: In case one or more of the above system variables (i.e. CPPFLAGS, LD_LIBRARY_PATH or LDFLAGS) are not defined, than one should drop the variable name on the right side of the definition (e.g. if CPPFLAGS is not defined, than one should use setenv CPPFLAGS -I/LIB_DIR/include for the tcsh shell and export CPPFLAGS; CPPFLAGS=-I/LIB_DIR/include for the bash shell).

2.2 Python files

Some of the codes in the NEXUS package are written in Python. These codes are located in the python directory in the installation package. The main codes that compute the Cosmic Web environments are implement in C++, just some additional codes that facilitate the use of the data were written in Python. Therefore you don't absolutely need Python to identify the Cosmic Web environments, but having access to these additional programs is helpful to interact more easily with the output data. Moreover, you need Python to be able to run the example script files.

To be able to use the additional Python programs (these are the ones ending with the extension '.py') you need a working installation of Python version 2.6 or higher and the numpy Python library. Normally these are installed by default on most Linux systems, but in case you don't have them than download and install the program and libraries.

Before being able to use the Python codes, you need to set the following system path: setenv PYTHONPATH NEXUS_DIR/python/python_import:$PYTHONPATH if PYTHONPATH is already defined or using setenv PYTHONPATH NEXUS_DIR/python/python_import if the system variable is not defined. The variable NEXUS_DIR denotes the directory where you have the NEXUS package.

2.3 First compilation and running tests

This section describes the minimal steps necessary to compile the NEXUS package and be able to run the test examples.

If you installed any of the external libraries in a user specified directory and not the system default path, than you need to specify the library directories in the Makefile file that comes with the NEXUS code. Upon opening Makefile you will see in the first lines the variables: INC_PATHS and LIB_PATHS. These variables should store the directories where you build the corresponding libraries (i.e. the directories that you supplied to the --prefix option for **GSL**, **Boost** or **FFTW** libraries). For example if you installed **GSL** at GSL_PATH and **Boost** at BOOST_PATH than the two variables should read:

```
INC_PATHS = -I/GSL_PATH/include -I/BOOST_PATH/include
```

```
LIB_PATHS = -L/GSL_PATH/lib -L/BOOST_PATH/lib
```

Following the library paths, you can also choose the C++ compiler used to build `NEXUS` using the variable `CC`.

Now you should be able to compile the `NEXUS` codes by typing `make` in the `NEXUS` package directory. This will create the a set of executable in the `bin` directory that you can use to run a few tests.

The following examples are meant to test that the code compiled properly and also to show you the syntax and options of the `NEXUS` package. Run the following examples to check that everything is working properly:

1. `cd demo` to enter the demo directory. There you should see 2 files, `demo_density.a_den` is a density file on a 128^3 grid while `nexus_demo_script.py` is a `Python` script with the instructions necessary to run the `NEXUS` suite of programs. Type `./nexus_demo_script.py` and you should see the programs being executed. The output is a set of files with names 'node_*', 'fila_*' and 'wall_*' which give the environments detected in the input data. More on the file format later on.

2. `../bin/infoHeader.py demo_density.a_den` or `../bin/infoHeader.py all_clean.MMF` to see the header of the input and output files.

If you were able to run the above tests without an error it means that you have successfully installed the `NEXUS` package.

3 Programs and program options

This section describes some of the important programs included in the `NEXUS` package. For each of the program included in the package you can get a short description of the program and its options using the `-h, --help` options[1].

The `NEXUS` programs were designed to work on data in a periodic box. So all the programs assume periodic boundary conditions. If you want to deal with data in a non-periodic box, than you will have to add an additional buffer zone at the edges of your data box with values of 0. It is recommended that the size of the buffer zone along each dimension to be at least the largest filtering radius used to minimize boundary effects.

The `NEXUS` programs work on grids that can have different number of points along each dimension and also on data volumes that do not need to be spherical. While the examples use cubic volume with the same grid size along each dimension, this is not a limitation of the package.

3.1 `MMF_respone`

The `MMF_respone` code computes the environmental signature for a given set of scales. This is the most time and memory intensive part of the `NEXUS` program. Running this program corresponds to steps *(I)* to *(V)* as described in section 2 of the algorithm paper Cautun et al. (2012). The program syntax is as follows:

```
MMF_respone density_file scale_set (additional options)
```

Where the program options are given and described in Table 1. The filter radius for scale n is given by R_n which is computed using the equation:

$$R_n = R_0 b^n \tag{1}$$

where R_0 is the value of the smoothing scale at scale 0 while b is the base and typically has the value $b = \sqrt{2}$ - both values can be specified as program options. The program has an additional option called the *bias* parameter that influences the weight of different smoothing scales when computing the scale independent signature. In general you will not want to change the *bias* value unless you have a strong reason in doing so. The *bias* parameter can be understood in the context of the Hessian matrix equation (2) in Cautun et al. (2012) which reads:

$$H_{ij,R_n}(x) = R_n^{2bias} \frac{\partial^2 f_{R_n}(\mathbf{x})}{\partial x_i \partial x_j} \tag{2}$$

which gives the Hessian matrix for smoothing radius R_n. The term R_n^{2bias} involving the bias controls the weight of each scale and has the value $bias = 1$ for all the methods except for NEXUS_tidal and NEXUS_velshear for which $bias = 0$. The `MMF_respone` code automatically selects those values so you don't need to insert a value for this option.

[1] The `-h` option is always available for the C++ programs, but it may not be defined in the `Python` programs. But even in that case you get an error message if the number of argument is different than the expected number.

Table 1: Program options for the `MMF_respone` code of the `NEXUS` package.

Option	Details and description.
`density_file`	The name of the input density file (or velocity divergence file) used to compute the environmental signature. This needs to be a `DENSITY-FILE` - see section 4.1 for file format.
`scale_set`	Specify the scales set over which to smooth the input field and compute the signature. The scales needs to be inserted as a single string using numbers and the −(dash) and ,(comma) symbols. For example to use scales from 0 to 5 and than scales 8 and 12, insert $0-5, 8, 12$. The smoothing radius at scale n is given by $R_n = R_0 b^n$ where the values of R_0 and b can be specified using options given below.
`--node, --fila, --wall <arg>`	Specify for which Cosmic Web component to compute the signature. The option needs to be followed by the root name for the output file. E.g.: `--node nodes` outputs the node signature to a file called `nodes_maxResponse.MMF`. The output file is of type `NEXUS-FILE` - see section 4.2 for file format. If you specify `--node name1 --fila name2` the program will output the signature for both nodes and filaments.
`-f, --filter <arg>`	Specify the filter used to compute the signature. DEFAULT: the filters described in the paper with values 40, 30 and 20 for nodes, filaments and walls respectively. In the case of the NEXUS_tidal and NEXUS_velshear methods you need to change the filter values to 44, 34 and 24 for nodes, filaments and walls respectively.
`-r, --radius <arg>`	The value of R_0 in Mpc/h from Eq. (1). DEFAULT value: 0.5 Mpc/h.
`-b, --base <arg>`	The value of b from Eq. (1). DEFAULT value: $\sqrt{2}$ such that after every 2 scales the smoothing radius increases by a factor of 2.
`--bias <arg>`	Specify the bias — bias¿1 give larger weight to larger scales, while bias¡1 gives larger weight to smaller scales. DEFAULT: bias=1 - in general not important.
`--den, --logFilter, --log, --grav, --vel, --velPot`	Specify which method is used to compute the signature. The options correspond to: NEXUS_den, NEXUS+, NEXUS_denlog, NEXUS_tidal, NEXUS_veldiv and NEXUS_velshear as described in the methods paper. DEFAULT value: NEXUS_den.
`--mask`	Specify to use a mask for the filament and wall signature computations. This is useful for the NEXUS_den and NEXUS_tidal methods where the density in node (node & filaments) regions needs to be 0 for filament (wall) signature computation. DEFAULT: not enabled.
`-N, --nodeFile <arg>`	Specify the clean node environment file if the `--mask` option is set. This is needed for filament and wall signature computation.
`-F, --filaFile <arg>`	Specify the clean filament environment file if the `--mask` option is set. This is needed for wall signature computation.
`--allEigen`	Choose this option if to output an additional file with the eigenvalues and eigenvectors corresponding to the output signature.
`--eigenvector`	Choose this option if to output an additional file with the eigenvector giving the direction of the filament (eigenvector 3 along the filament) or wall (eigenvector 1 which gives the normal to the wall).

The output of the MMF_respone program is a NEXUS-FILE which contains for every grid point a 4 bytes (single precision) floating point value. This is the maximum environmental signature over all scales at that grid point.

The following are a few examples of using the MMF_respone program:

1. Compute the node signature using the NEXUS_tidal method:
   ```
   MMF_respone densityFile 0-6 --node nodeOut --grav -f 44
   ```

2. Compute both the filament and wall signature using NEXUS+, $R_0 = 0.25$ Mpc/h and $b = 1.2$:
   ```
   MMF_respone densityFile 0-8 --fila filaOut --wall wallOut --logFiltering --radius 0.25 --base 1.2
   ```

3. Compute the wall signature using NEXUS_den with node and filament masks:
   ```
   MMF_respone densityFile 0-6 --wall wallOut --den --mask --nodeFile nodeOut_clean.MMF --filaFile filaOut_clean.MMF
   ```

3.2 MMF_threshold

The MMF_threshold program computes the optimal signature threshold for the detection of the Cosmic Web components. This program corresponds to step *(VI)* as described in section 2 of the algorithm paper Cautun et al. (2012). The program syntax is as follows:

```
MMF_threshold signature_file output_file (additional options)
```

The additional options are specified in Table 2. The MMF_threshold program has two outputs. The first is a text file giving the optimal threshold value on the first line and the data used to compute that value in the rest of the file. If you use the --cleanFile option than the program will also write a clean environment file that gives the voxel values above and below the optimal threshold. These data is stored in a NEXUS-FILE type file that contains for each grid point a 2 byte integer with values of 0 (that voxel is not part of the current environment) and values of 1 corresponding to the voxels that form the current environment. The clean environment file can also be obtained using the MMF_clean program and the optimal threshold value given on the first line of the output text file.

The MMF_threshold program takes as input a density file giving the density in the volume of the computation. Normally this is the same density file that was used to compute the environment signature using the MMF_respone program[1]. While the physical units in which the density is expressed do not matter for the MMF_respone program, they do matter for the MMF_threshold input where the program expects that the density is given as the ratio $\rho/\rho_{background}$ between the density at the given point and the average background density. This density unit is needed for the optimal threshold computation for node environments where it is used to obtain the mass of each individual object. In this case the program also needs the value Ω_0 for the matter density; value that is stored inside the header of the density file. In the

[1]The only exception is for the NEXUS_veldiv and NEXUS_velshear methods that take a velocity divergence as input to the MMF_respone code. In that case the MMF_threshold program still needs a density filed, the density on the same grid as the velocity divergence.

Table 2: Program options for the `MMF_threshold` code of the `NEXUS` package.

Option	Details and description.
`signature_file`	This is the output file of the `MMF_respone` code that gives the signature of a given environment. This file is of type `NEXUS-FILE` - see section 4.2 for file format.
`output_file`	A text file that outputs the value of the optimal signature threshold (first line of the file) and the data used to compute that value.
`--node, --fila, --wall`	Specify the environment for which to compute the optimal threshold. You can specify at most one environment at a time. The environment you specify needs to match with the data stored in the `signature_file` input file.
`--cleanFile <arg>`	Give the name of the output file that stores 1 for voxels which are valid environments and 0 otherwise. This file is of type `NEXUS-FILE` storing 2 bytes integers (0 or 1).
`--densityFile, -D <arg>`	The name of the density file used to compute mass inside the environments.
`--minSize <arg>`	Specify the minimum size of an object to be considered valid. For nodes this option specifies the lowest mass (in Msolar/h units) of a object that is a valid node. For filaments and wall it specifies the lowest volume (in $(\text{Mpc/h})^3$ units) of distinct filaments and walls to be considered as significant features.
`--virDen <arg>`	The value of the virial density, as ratio with background density, at the redshift of the snapshot. This is need only for node computations.
`-N, --nodeFile <arg>`	Specify a node clean file that will be used as mask for filament and wall computations.
`-F, --filaFile <arg>`	Specify a filament clean file that will be used as mask for wall computations.
`--noMask`	Don't use node and filament masks.
`--neighbor <arg>`	Specify what constitutes a neighbour for a voxel. A value of 1 corresponds to using only the 6 closest grid points as neighbours while a value of 2 corresponds to using the closest 26 points. DEFAULT value: 1
`-r, --range <3 args>`	Insert space separated real values giving the minimum and maximum range of values in which to search for the optimal threshold. A third value gives the number of points in the range. DEFAULT value: `--range 1.e-5 .9 50` to compute the behaviour for thresholds from 10^{-5} to 0.9 of the maximum signature value using 50 points in that interval.

case of the optimal threshold computation for filament or wall environments the density unit or value of Ω_0 do not play a role.

Examples of running the `MMF_threshold` program:

1. Compute the optimal threshold for node environments with minimum object mass of $5 \times 10^{13} M_0/h$ and virial density at $z = 0$ of $\Delta = 350\rho_{background}$:
   ```
   MMF_threshold nodeOut_maxResponse.MMF nodeOut.threshold --densityFile density_file --virDen
   350 --minSize 5.e13 --node --cleanFile nodeOut_clean.MMF
   ```

2. Compute the optimal threshold for filaments using minimum valid filament volume of $10(\mathrm{Mpc/h})^3$ and using a node file mask:
   ```
   MMF_threshold filaOut_maxResponse.MMF filaOut.threshold --densityFile density_file --minSize
   10 --fila --cleanFile filaOut_clean.MMF --nodeFile nodeOut_clean.MMF
   ```

3.3 MMF_clean

The `MMF_clean` program takes as input a signature threshold value and computes all the voxels above and below that signature value. This program outputs the same environmental clean file as `MMF_threshold` , but in this case the threshold is not computed automatically but is inserted as a program argument. The `MMF_clean` program can be used to compute the environments when using a different value of the threshold than that specified by the `MMF_threshold` code. The `MMF_clean` program syntax is as follows:

```
MMF_clean signature_file output_file threshold_value (additional options)
```

The additional options are specified in Table 3 and most of them are the same as in the case of the `MMF_threshold` code since both programs share common tasks. The output of the `MMF_clean` code is a `NEXUS-FILE` with values of 1 corresponding to voxels in valid environments and values of 0 corresponding to non-detections. See the `MMF_threshold` description for additional details.

Examples of running the `MMF_clean` program:

1. Compute the valid node environments with minimum object mass of $5 \times 10^{13} M_0/h$ using signature threshold of 0.1:
   ```
   MMF_clean nodeOut_maxResponse.MMF nodeOut_clean.MMF 0.1 --densityFile density_file --minSize
   5.e13 --node
   ```

2. Compute the valid filament environments using minimum valid filament volume of $10(\mathrm{Mpc/h})^3$, using a node file mask and signature threshold of 0.01:
   ```
   MMF_clean filaOut_maxResponse.MMF filaOut_clean.MMF 0.01 --densityFile density_file --minSize
   10 --fila --nodeFile nodeOut_clean.MMF
   ```

3.4 MMF_combine.py

The `MMF_combine.py` program combines the 3 files that give the grid points assigned to node, filament and wall environments. These files are the output of the `MMF_threshold` or `MMF_clean` codes

Table 3: Program options for the `MMF_clean` code of the `NEXUS` package.

Option	Details and description.
`signature_file`	This is the output file of the `MMF_respone` code that gives the signature of a given environment. This file is of type `NEXUS-FILE` - see section 4.2 for file format.
`output_file`	Give the name of the output file that stores 1 for voxels which are valid environments and 0 otherwise. This file is of type `NEXUS-FILE` storing 2 bytes integers (0 or 1).
`threshold_value`	Give the signature threshold that discriminates between valid environments (environmental signature larger than the threshold) and spurious detections (environmental signature smaller than the threshold).
`--node, --fila, --wall`	Specify the environment for which to compute the valid environments. You can specify at most one environment at a time. The environment you specify needs to match with the data stored in the `signature_file` input file.
`--cleanFile <arg>`	Give the name of the output file that stores 1 for voxels which are valid environments and 0 otherwise. This file is of type `NEXUS-FILE` storing 2 bytes integers (0 or 1).
`--densityFile, -D <arg>`	The name of the density file used to compute mass inside the environments.
`--minSize <arg>`	Specify the minimum size of an object to be considered valid. For nodes this option specifies the lowest mass (in Msolar/h units) of a object that is a valid node. For filaments and wall it specifies the lowest volume (in $(\mathrm{Mpc/h})^3$ units) of distinct filaments and walls to be considered as significant features.
`-N, --nodeFile <arg>`	Specify a node clean file that will be used as mask for filament and wall computations.
`-F, --filaFile <arg>`	Specify a filament clean file that will be used as mask for wall computations.
`--noMask`	Don't use node and filament masks.
`--neighbor <arg>`	Specify what constitutes a neighbour for a voxel. A value of 1 corresponds to using only the 6 closest grid points as neighbours while a value of 2 corresponds to using the closest 26 points. DEFAULT value: 1.

Table 4: Program options for the `MMF_spineDirection` code of the `NEXUS` package. There are a few additional options, but they are not important - for those check the help message or the source code of the program.

Option	Details and description.
`environment_file`	The clean environment file which is the output of the `MMF_threshold` or `MMF_clean` programs.
`output_file`	A `NEXUS-FILE` type file which gives for every valid voxel a 3 component vector giving the direction of filaments and walls (see text for details).
`--nodeFile <arg>`	The file containing the clean node environment. This is needed for both filaments and wall computations.
`--filaFile <arg>`	The file containing the clean filament environment. This is needed for wall computations.
`--radius <arg>`	The smoothing radius R in Mpc/h over which to average the point distribution. DEFAULT value: 1. Mpc/h.
`--convergenceFraction <arg>`	Specify the tolerance $\epsilon_{fraction}$ (see text for details). DEFAULT value: 0.01 corresponding to less than 1% of points not converged.
`--eigenvalueThreshold <arg>`	Specify the tolerance ϵ_λ (see text for details). DEFAULT value: 0.3
`--distanceThreshold <arg>`	Specify the tolerance $\epsilon_{distance}$ in Mpc/h (see text for details). DEFAULT value: 0.05 Mpc/h
`--directionThreshold <arg>`	Specify the tolerance ϵ_θ (see text for details). DEFAULT value: 0.996 corresponding to cos(5°)

for the node, filament and wall computations. The program syntax is as follows:

```
MMF_combine.py node_file filament_file wall_file output_file
```

The resulting Cosmic Web environments are written to a `NEXUS-FILE` which contains the values: 0=voids, 2=walls, 3=filaments and 4=nodes. Each voxel has assigned one of the 4 values and the results are saved in the output file using a 2 bytes integer to store the environment data for each grid point.

3.5 `MMF_spineDirection`

The `MMF_spineDirection` code is used to compute the geometrical direction of filaments and walls. For filaments it outputs the direction along the filaments while for walls it outputs the normal to the plane of the wall. The program syntax is as follows:

```
MMF_spineDirection environment_file output_file (additional options)
```

The additional `MMF_spineDirection` program options are shown in Table 4.

To understand the program options one needs to understand how the code computes the geometrical spine of filaments and central plane of walls. I will discuss the method for fila-

ments and I will focus on the difference for walls only on there is a difference between the two environments. The `MMF_spineDirection` program executes the following steps:

1. Classify as filaments all the grid points that are assigned to filaments and nodes. NOTE that in the following both node and filament volumes are classified as filaments. We need to include the node regions too since in their absence there is a spurious fragmentation of the filaments around the nodes since the nodes are located inside the filamentary regions.
2. Replace the voxels found at the previous steps with points located at the center of each valid voxel. Assign to each point the same mass to obtain geometrical quantities.
3. Obtain a rough approximation for the direction of the filament by iteratively contracting the point cloud along the direction of maximum mass gradient. The steps necessary to do so are:
   i. For each point find the center of mass[1] of the cloud of points within distance R from the point in consideration.
   ii. Move each point to the center of mass of its associated point cloud.
   iii. Find again for each point the center of mass of its point cloud using the updated point positions computed in step [ii.] [2].
   iv. Repeat steps [ii.] and [iii.] until the **displacement convergence criterion** is achieved. This will be described shortly.
   v. Once the point set converged to its final position, find for each point the inertia momentum of the point cloud within a distance R from the given point. The eigenvector corresponding to the *largest* eigenvalue of the inertia moment gives an initial guess for the filament direction.
4. Obtain a better approximation for the filament direction. Contract again the initial point cloud (found in step [2]) but now taking into account an estimate of the filament direction found in the previous iteration. The steps needed to do so are:
   i. For each point find the center of mass of the cloud of points within distance R from the point in consideration.
   ii. Move each point towards the center of mass of its associated point cloud, but only along the direction *perpendicular on the filament*. For example for point i we have its position vector $\mathbf{x_i}$, the filament direction vector at that point $\mathbf{v_i}$ and the position vector of the center of mass $\mathbf{x_{CM,i}}$ of the associated point cloud. This means that the position of the point after the move is given by:

$$\mathbf{x_{i,new}} = \mathbf{x_i} + (\mathbf{d_i} - (\mathbf{d_i} \cdot \mathbf{v_i})\mathbf{v_i}) \,, \qquad \text{with } \mathbf{d_i} = \mathbf{x_{CM,i}} - \mathbf{x_i} \tag{3}$$

   where $\mathbf{d_i}$ is the displacement between the current point position and the center of mass of the point cloud. For simplicity I took $\mathbf{v_i}$ as a length 1 vector. The scalar product $\mathbf{d_i} \cdot \mathbf{v_i}$ gives the projection of displacement along the direction of the filament. Subtracting this projection from the displacement vector leaves only the component of the displacement vector perpendicular on the filament.

[1] Since we are interested on geometrical quantities we assigned to each point

[2] NOTE that due to the fact that points change position, the algorithm needs to compute again for each point its associated point cloud within the distance R.

iii. Find again for each point the center of mass of its point cloud using the updated point positions computed in step [ii.].

iv. Repeat steps [ii.] and [iii.] until the **displacement convergence criterion** is achieved. This will be described shortly.

v. Once the point set converged to its final position, find for each point the inertia momentum of the point cloud within a distance R from the given point. The eigenvector corresponding to the *largest* eigenvalue of the inertia moment gives a new estimate for the filament direction. Update the filament directions at every point using the new estimate.

5. Repeat iteratively step [4.] until convergence is achieved. The code tests for convergence by comparing the filament direction obtained in the current and previous iterations. For every point it computes the angle θ between the current filament direction and that of the previous iteration. If this angle is smaller than a given threshold[1], than the program considers that the filament direction at this point has converged. The program stops only when most of the points have a converged filament direction. It tests for this requiring that the fraction of points with non-converged directions is less than $\epsilon_{fraction}$[2].

For the determination of the wall direction[3] there are a few small differences when compared to the same procedure for filaments. In the following I will present only the points where there are differences:

1. Classify as walls all the grid points that are found to be wall, filaments and nodes.
3. Obtain a rough approximation for the direction of the wall by iteratively contracting the point cloud along the direction of maximum mass gradient. All steps are the same as for filaments except:

   v. Compute for each point the inertia momentum of the point cloud within a distance R from the given point. The eigenvector corresponding to the *smallest* eigenvalue of the inertia moment gives an initial guess for the normal to the wall plane.

4. Obtain a better approximation for the wall direction. Contract again the initial point cloud (found in step [2]) but now taking into account an estimate of the wall direction found in the previous iteration. The steps different from the filament computation are:

   ii. Move each point towards the center of mass of its associated point cloud, but only along the *normal direction to the wall plan*. For example for point i we have its position vector $\mathbf{x_i}$, the wall direction vector at that point $\mathbf{v_i}$ and the position vector of the center of mass $\mathbf{x_{CM,i}}$ of the associated point cloud. This means that the position of the point after the move is given by:

$$\mathbf{x_{i,new}} = \mathbf{x_i} + (\mathbf{d_i} \cdot \mathbf{v_i})\mathbf{v_i}, \qquad \text{with } \mathbf{d_i} = \mathbf{x_{CM,i}} - \mathbf{x_i} \tag{4}$$

[1] In practice we test for small angle θ by requiring that $\cos\theta > \epsilon_\theta$ where ϵ_θ is a program option specified by the user.

[2] The non-converged points are typically at the intersection of filaments and usually don't have a well defined geometrical filament direction associated to them.

[3] The wall direction specifies the normal to the plane of the wall. In the text I use both expressions to mean the same thing.

where $\mathbf{d_i}$ is the displacement between the current point position and the center of mass of the point cloud. The scalar product $\mathbf{d_i} \cdot \mathbf{v_i}$ gives the projection of displacement along the normal to the wall, so the point is displaced only along the normal to the wall.

v. Compute for each point the inertia momentum of the point cloud within a distance R from the given point. The eigenvector corresponding to the *smallest* eigenvalue of the inertia moment gives a new estimate for the wall direction. Update the wall directions at every point using the new estimate.

An important point that was not addressed above deals with the conditions needed to test for the **displacement convergence criterion** that is needed in steps [3.iv.] and [4.iv.]. These are the conditions testing that the point distribution was contracted iteratively to the spine of the filaments and the central plane of the walls. I found that a good test to recognize the convergence to this central axis/plane is given by testing for 3 different quantities:

1. The distance a point is expect to move in step [3.ii.] or [4.ii.] is smaller than a certain distance threshold $\epsilon_{distance}$. This means that in any subsequent iterations points will move less and less.
2. The point cloud within distance R from the point in question is very anisotropic and has a strong filament/wall characteristic. This anisotropy is tested for by looking at the ratio of the eigenvalues λ_i of the inertia moment (for simplicity we take $\lambda_1 \leq \lambda_2 \leq \lambda_3$). For filaments we expect the point cloud to be along a line, this means that the ratio λ_2/λ_3 is very small (i.e. $\lambda_2/\lambda_3 < \epsilon_\lambda$). For walls the point cloud should be along a plane and therefore the ratio λ_1/λ_2 is very small (i.e. $\lambda_1/\lambda_2 < \epsilon_\lambda$).
3. Once most of the points satisfy the above 2 criteria, we consider that the iteration has converged to the current central axis/plane of the filament/wall. In practice we test for this requiring that the fraction of points not satisfying one of the two criteria is smaller than $\epsilon_{fraction}$[1].

The output of the `MMF_spineDirection` program is written to a `NEXUS-FILE` . To save space only the direction vector corresponding to valid voxels is written in the file. This means that every filament and node voxel will have assigned a filament direction. Also every wall, filament and node voxel will have assigned a wall direction. For an example of how to read in the output files of the `MMF_spineDirection` or `MMF_spineProperties` programs see the function `NEXUSEnvironmentProperties` in the python file `python/python_import/MMF.py`.

The following represent two examples of computing the filament and wall directions:

1. Compute the geometrical direction of filaments using a 1Mpc/h filter radius and with at most 1% non-converged filament directions:
   ```
   MMF_spineDirection filaOut_clean.MMF filaOut_spineDirection.MMF --nodeFile nodeOut_clean.MMF
   --radius 1 --convergenceFraction 0.01
   ```

[1] The points not satisfying these criteria are located at the intersection of filaments and walls. The convergence for these points is quite slow.

Table 5: Program options for the `MMF_spineProperties` code of the `NEXUS` package. There are a few additional options, but they are not important - for those check the help message or the source code of the program.

Option	Details and description.
`environment_direction_file`	The file giving the filament or wall direction obtained using the `MMF_spineDirection` program.
`output_file`	A `NEXUS-FILE` type file which gives for every valid voxel 2 scalar values giving the size and density of filaments and walls (see text for details).
`--densityFile <arg>`	The file containing the density field. This is needed to compute the mass density.
`--cleanFile <arg>`	The file containing the clean environment. This is the output file of the `MMF_threshold` or `MMF_clean` programs.
`--nodeFile <arg>`	The file containing the clean node environment. This is needed for both filaments and wall computations.
`--filaFile <arg>`	The file containing the clean filament environment. This is needed for wall computations.
`--radius <arg>`	The smoothing radius R in Mpc/h over which to average the point distribution. DEFAULT value: 1. Mpc/h.
`--convergenceFraction <arg>`	Specify the tolerance $\epsilon_{fraction}$ (see text for details). DEFAULT value: 0.005 corresponding to less than 0.5% of points not converged.
`--eigenvalueThreshold <arg>`	Specify the tolerance ϵ_λ (see text for details). DEFAULT value: 0.3
`--distanceThreshold <arg>`	Specify the tolerance $\epsilon_{distance}$ in Mpc/h (see text for details). DEFAULT value: 0.05 Mpc/h

2. Compute the geometrical direction of walls using a 1Mpc/h filter radius and with at most 1% non-converged wall directions:
```
MMF_spineDirection wallOut_clean.MMF wallOut_spineDirection.MMF --nodeFile nodeOut_clean.MMF
--filaFile filaOut_clean.MMF --radius 1 --convergenceFraction 0.01
```

3.6 `MMF_spineProperties`

The `MMF_spineProperties` program is used to compute the properties of filaments and walls. It computes the diameter and linear mass density of filaments and the height and surface mass density of walls. The program syntax is as follows:

```
MMF_spineProperties environment_direction_file output_file (additional options)
```

The additional `MMF_spineProperties` program options are shown in Table 5.

The program computes the environment properties by first contracting the filaments/walls to a central axis/plane. The contraction procedure is done using the following steps:

1. Classify as filaments (walls) all the grid points that are assigned to filaments and nodes (walls, filaments and nodes).
2. Replace the voxels found at the previous steps with points located at the center of each valid voxel. Assign to each point the same mass to obtain the geometrical central axis/plane and not the mass weighted one.
3. Contract the distribution of points to its central axis/plane for filaments/walls. The contraction is done using the filament/wall directions computed by the `MMF_spineDirection` program. The following presents the steps of the procedure:
    i. For each point find the center of mass of the cloud of points within distance R from the point in consideration.
    ii. Move each point towards the center of mass of its associated point cloud taking into account the environment direction. When computing the filament properties, move the points only *perpendicular on the filament direction*. The position of the point after the move is given by Eq. (3). When computing the wall properties, move the points only along the *normal direction to the wall plan* according to Eq. (4).
    iii. Find again for each point the center of mass of its point cloud using the updated point positions computed in step [ii.].
    iv. Repeat steps [ii.] and [iii.] until the **displacement convergence criterion** is achieved (see section 3.5 for a description of the convergence criteria). The converged results correspond to the central axis/plane of the filaments/walls.

Once the program found the filament spine and wall central plane it can continue and compute the size and densities of these objects. It computes these environment properties smoothed over a top-hat filter of radius R. The resulting properties are dependent on the filter radius R. For each point i in the point set it identifies all the N_i points that are within a distance R from the point i. These N_i points correspond to N_i grid cells and therefore each point represents the volume V_{cell} of a grid cell. Each of these points has associated to it a mass, the mass of the grid cell it represents which is given as the density in that grid cell times the grid cell volume V_{cell}. For simplicity I denote with $m_{i,j}$ the mass of the j-th neighbour of point i. The following gives the expression for the size and densities of filaments and walls:

1. The program approximates locally each filament region as a cylinder of diameter D_i and length $2R$. It than uses that the volume of the cylinder should correspond to the volume of the N_i points found with distance R from point i. This means that the **filament diameter** at point i is given by:

$$D_i = \sqrt{\frac{2N_i V_{cell}}{\pi R}} \tag{5}$$

   Similarly if locally the filament has linear mass density Λ_i than the mass contained by the filament in a length of $2R$ is $2R\Lambda_i$. This mass corresponds to the mass contained in the N_i points found around point i and therefore the local **linear mass density of the filament** is:

$$\Lambda_i = \frac{\sum_{j=0}^{N_i} m_{i,j}}{2R} \tag{6}$$

   where the sum is over all the neighbour points of point i.

2. The program approximates locally each wall region as a plane of constant height H_i over the area πR^2. It than uses that the volume of this should correspond to the volume of the N_i points found with distance R from point i. This means that the **wall height** at point i is given by:

$$H_i = \frac{N_i V_{cell}}{\pi R^2} \tag{7}$$

Similarly if locally the wall has surface mass density σ_i than the mass contained by the wall in an area of πR^2 is $\pi R^2 \sigma_i$. This mass corresponds to the mass contained in the N_i points found around point i and therefore the local **surface mass density of the wall** is:

$$\sigma_i = \frac{\sum_{j=0}^{N_i} m_{i,j}}{\pi R^2} \tag{8}$$

where the sum is over all the neighbour points of point i.

The output of the `MMF_spineProperties` program is written to a `NEXUS-FILE` . To save space only the properties corresponding to valid voxels is written in the output file. This means that every filament and node voxel will have assigned a filament diameter and linear mass density. Also every wall, filament and node voxel will have assigned a wall height and surface mass density. For an example of how to read in the output files of the `MMF_spineDirection` or `MMF_spineProperties` programs see the function `NEXUSEnvironmentProperties` in the python file `python/python_import/MMF.py`.

The following represent two examples of computing the filament and wall properties:

1. Compute the geometrical direction of filaments using a 1Mpc/h filter radius and with at most 1% non-converged filament directions:
```
MMF_spineProperties filaOut_spineDirection.MMF filaOut_spineProperties.MMF --densityFile
density_file --cleanFile filaOut_clean.MMF --nodeFile nodeOut_clean.MMF --radius 1
--convergenceFraction 0.01
```

2. Compute the geometrical direction of walls using a 1Mpc/h filter radius and with at most 1% non-converged wall directions:
```
MMF_spineProperties wallOut_spineDirection.MMF wallOut_spineProperties.MMF --densityFile
density_file --cleanFile wallOut_clean.MMF --nodeFile nodeOut_clean.MMF
--filaFile filaOut_clean.MMF --radius 1 --convergenceFraction 0.01
```

3.7 `infoHeader.py`

The `infoHeader.py` code prints the header of a `DENSITY-FILE` or `NEXUS-FILE` file. For details on the headers of the two files see section 4. The program syntax is as follows:

```
infoHeader.py density(nexus)_file
```

Table 6: Program options for the `density_raw.py` code of the `NEXUS` package.

Option	Details and description.
`--raw`	Outputs only the raw data values without any header or wrapping around the data block.
`--grid`	Outputs 3 integers (4bytes) giving the grid size along each dimension followed by the raw data.
`--box`	Outputs 3 integers (4bytes) giving the grid size along each dimension followed by the box coordinates (xMin,xMax,yMin,yMax,zMin,zMax) in float format (4bytes). After this there is the raw data.
`--fortran`	Outputs any of the above using the FORTRAN way of reading/writing binary data. Each block of data (grid size, box coordinates and raw data) has a a 8 bytes integer preceding and following it giving the size of the data block in bytes.

3.8 density_raw.py

The `density_raw.py` code rewrites the data in a `DENSITY-FILE` or `NEXUS-FILE` file to a simple binary file containing only the grid data. The program syntax is as follows:

```
density_raw.py density(nexus)_file output_file (additional options)
```

Where the additional options are given in Table 6. Note that the data written to the output file has the same type as the data stored in the input file, so basically this program strips from the input file the header and the wrapping around the data block.

3.9 density_fromRaw.py

The `density_fromRaw.py` program reads the density data from a binary file and rewrites the data to a `DENSITY-FILE` that can later on be used as input to the `NEXUS` programs. The program syntax is as follows:

```
density_fromRaw.py input_binary_file output_density_file grid_size box_length
omega_matter
```

Where `grid_size` gives the size of the grid along one dimension and `box_length` gives the length of the periodic box in Mpc/h. The option `omega_matter` is the cosmological matter density Ω_0 at $z = 0$ (typically around 0.3) which is used for node detection. Note that this program assumes that the data was computed in a box with the same number of grid points along each direction and that the box is cubic (all sides have the same length). The `NEXUS` programs work for non-cubic box and for different grid sizes along each direction, so if you have a more complex box geometry you can modify this program to take into account non-cubic boxes and grids.

3.10 density_toVisit.py

The density_toVisit.py program writes a short text file that can be used later on to visualize the data contained in the file using the VisIt program (https://wci.llnl.gov/codes/visit/). The program syntax is as follows:

```
density_toVisit.py density(nexus)_file
```

The program writes a text file with the same name and ending in the .bov extension that tells VisIt the data format and properties. Note that for visualization you need both the initial density/nexus file as well as the descriptive .bov file. From the VisIt's GUI you should open the .bov file.

3.11 Scripts and examples

Identifying the Cosmic Web environments consists of several steps which are given in the python script file scripts/nexus_script.py. The information on how the run the script can be found in the file itself. The python script file executes the steps necessary to identify the nodes, filaments and walls:

1. Use MMF_respone to compute the **node** signature.
2. Use MMF_threshold to compute the optimal **node** threshold and output a clean file with the valid **node** environments.
3. Use MMF_respone to compute the **filament** signature.
4. Use MMF_threshold to compute the optimal **filament** threshold and output a clean file with the valid **filament** environments.
5. Use MMF_respone to compute the **wall** signature[1].
6. Use MMF_threshold to compute the optimal **wall** threshold and output a clean file with the valid **wall** environments.
7. Use MMF_combine.py to combine the individual files giving the node, filament and wall environments to obtain a file giving **all** the Cosmic Web components.

The example script file contains instruction on how to compute the geometrical direction and properties of filaments and walls, but the commands are commented out. They should be enabled only if they are needed. Especially the procedure to compute the geometrical direction of the environments can be CPU intensive since it needs to obtain the neighbours with a distance R for every point several tens to hundred times. The MMF_spineDirection program is especially slow if the ratio of the smoothing radius R to the size of the grid cell is large. Therefore these programs should be run only when needed.

3.12 Additional programs

On top of the programs described above, I have included a few additional python programs in the python directory. You will need to investigate those programs yourself to figure out

[1]Since the filament and wall signatures are computed using the same method NEXUS+, steps 3. and 5. are combined in a single step in the example script file scripts/nexus_script.py.

what they do. All the python modules needed by the programs are included in the directory `python/python_import/`.

4 Input and Output file format

This section present the format of the two file types used as input and output in the NEXUS package. The DENSITY-FILE files are used to store the initial density or velocity divergence field needed as input to the `MMF_respone` program as well as the density field needed for the `MMF_threshold` and `MMF_clean` programs. The NEXUS-FILE files are used to store the outputs of the NEXUS programs at each intermediate stage needed for the identification of the Cosmic Web.

Both the DENSITY-FILE and NEXUS-FILE have the same format:

```
8 bytes integer preceding the file header (value=1024)
    file header of size 1024 bytes
8 bytes integer succeeding the file header (value=1024)

8 bytes integer preceding the file header (value=data size in bytes)
    data
8 bytes integer succeeding the file header (value=data size in bytes)
```

Where the values of the preceding and succeeding 8 bytes integers can be used to check that the file is not corrupt and that the procedure to read data was correct. The description and structure of the DENSITY-FILE and NEXUS-FILE headers is given below.

4.1 DENSITY-FILE format

The header of the DENSITY-FILE file type is used to store information about the data in the file as well as the commands used to obtained that data. In fact most of the NEXUS programs will write the instructions used to obtain the current file into the header of the file such that the data can be easily reproduced in case there is a need to do so. The structure and the instructions to read/write the DENSITY-FILE format can be found in the `src/densityFile` directory for the C++ part of the program[1] and in the file `python/python_import/density.py` for the python programs.

The entries of the DENSITY-FILE header are shown in Table 7 in the same order as they are found in the file.

[1] The `src/densityFile` contains 3 files: `density_header.h`, `densityHeader.cc` and `densityFile.h`. The first two define the header of the file and also contain the definition of the class functions. The last file contains the code to write and read the DENSITY-FILE files.

Table 7: The elements of the `DENSITY-FILE` header. The symbol `size_t` denotes an 8 bytes unsigned integer, `int` denotes a 4 bytes integer and `double` denotes a 8 bytes floating point number. If an entry name is followed by the symbol `[N]` it means that variable is an array with `N` elements. The rows of the table are split in 3 sections by the use of double horizontal lines. The middle section is a copy of the `GADGET` file header used to obtain the density and, with the exception of the `Omega0` entry, is not used for further computations.

Entry name	Type	Details and description.
`gridSize[3]`	`size_t`	3 element array that stores the grid size along each direction.
`totalGrid`	`size_t`	Stores the total number of cells in the grid. Is the product of the elements of the above array.
`fileType`	`int`	An id that keeps track of the field saved in the file. A value=1 means that the file stores a density field, a value=13 means the file stores a velocity divergence field. Note that the `MMF_respone` code checks this header value.
`noDensityFiles`	`int`	=1. $\neq 1$ if file is split into multiple subfiles (NOT USED).
`densityFileGrid[3]`	`int`	important only for split files (NOT USED).
`indexDensityFile`	`int`	important only for split files (NOT USED).
`box[6]`	`double`	the boundaries of the box $(x_{min}, x_{max}, y_{min}, y_{max}, z_{min}, z_{max})$.
`npartTotal[6]`	`size_t`	number of particles taken from `GADGET` file.
`mass[6]`	`double`	mass of `GADGET` particles.
`time`	`double`	scale factor a in `GADGET` simulation.
`redshift`	`double`	redshift in `GADGET` simulation.
`BoxSize`	`double`	box length in `GADGET` simulation.
`Omega0`	`double`	matter density Ω_0. This is needed in the `MMF_threshold` code for node detection.
`OmegaLambda`	`double`	dark energy density Ω_Λ.
`HubbleParam`	`double`	Hubble parameter h.
`method`	`size_t`	method used to compute the density (NOT USED).
`fill[760]`	`char`	string containing the commands used to obtain the current data stored in the file.
`FILE_ID`	`size_t`	an unique file ID (value=1) (NOT USED).

4.2 NEXUS-FILE format

The header of the NEXUS-FILE file type is used to store information about the data in the file as well as the commands used to obtained that data. In fact most of the NEXUS programs will write the instructions used to obtain the current file into the header of the file such that the data can be easily reproduced in case there is a need to do so. The structure and the instructions to read/write the NEXUS-FILE format can be found in the src/MMF_file directory for the C++ part of the program[1] and in the file python/python_import/MMF.py for the python programs.

The entries of the NEXUS-FILE header are shown in Table 8 in the same order as they are found in the file. The entry fileType can have the following values:

10= Stores the environmental signatures as a 4 bytes float, which is the output of the MMF_respone computation.

20= Stores the valid environment data as a 2 bytes integer with values of 0 or 1. This is the output of theMMF_threshold or MMF_clean computations.

21= Stores the full Cosmic Web environment data which is the output of the MMF_combine.py code. Each grid point has a 2 bytes integer with values: 0=voids, 2=walls, 3=filaments and 4=nodes.

40= Stores the geometrical direction of filaments and walls. This is the output of the MMF_spineDirection code. Each filament or wall grid point has a 3 component vector, each component a 4 bytes float. This vector gives the direction along the filament or the normal to the plane of the wall. Note that this file stores the direction vector only for the voxels found in filaments/walls.

50= Stores the properties of filaments and walls computed by the MMF_spineProperties code. Each filament or wall grid cells have two 4 bytes floats associated to them that give the properties of the environments. For filaments the two properties give the filament diameter and the linear mass density along the filament at that point. For walls the two properties give the wall height and the surface mass density along the wall at that point.

References

Cautun, M., van de Weygaert, R., & Jones, B. J. T. 2012, M.N.R.A.S., 402

[1]The src/MMF_file contains 3 files: MMF_header.h, MMF_header.cc and MMF_file.h. The first two define the header of the file and also contain the definition of the class functions. The last file contains the code to write and read the NEXUS-FILE files.

Table 8: The elements of the NEXUS-FILE header. The symbol size_t denotes an 8 bytes unsigned integer, int denotes a 4 bytes integer, float denotes a 4 bytes floating point number and double denotes a 8 bytes floating point number. If an entry name is followed by the symbol [N] it means that variable is an array with N elements. The rows of the table are split in 3 sections by the use of double horizontal lines. The middle section is a copy of the GADGET file header used to obtain the density and, with the exception of the Omega0 entry, is not used for further computations.

Entry name	Type	Details and description.
gridSize[3]	size_t	3 element array that stores the grid size along each direction.
totalGrid	size_t	Stores the total number of cells in the grid. Is the product of the elements of the above array.
feature	int	an environment tag: 4=nodes, 3=filaments, 2=walls and 5=all environments (node + filaments + walls + voids).
scale	int	the index of the smoothing scale used to obtain results. Has a value of -1 for the scale independent maximum signature.
radius	float	the filter radius for a single scale and -1. for the multiscale analysis.
bias	float	the *bias* parameter - see section 3.1.
filter	int	the filter type used to define the environmental signature (NOT USED).
fileType	int	An id that keeps track of the field saved in the file. More on that in the main text.
noMMF_files	int	=1. $\neq 1$ if file is split into multiple subfiles (NOT USED).
MMF_fileGrid[3]	int	important only for split files (NOT USED).
indexMMF_file	int	important only for split files (NOT USED).
method	int	the method used to obtain the environments: 1=NEXUS_DEN, 10=NEXUS_tidal, 5=NEXUS_denlog, 100=NEXUS+, 20=NEXUS_veldiv and 30=NEXUS_velshear.
box[6]	double	the boundaries of the box $(x_{min}, x_{max}, y_{min}, y_{max}, z_{min}, z_{max})$.
npartTotal[6]	size_t	number of particles taken from GADGET file.
mass[6]	double	mass of GADGET particles.
time	double	scale factor a in GADGET simulation.
redshift	double	redshift in GADGET simulation.
BoxSize	double	box length in GADGET simulation.
Omega0	double	matter density Ω_0. This is needed in the MMF_threshold code for node detection.
OmegaLambda	double	dark energy density Ω_Λ.
HubbleParam	double	Hubble parameter h.
fill[744]	char	string containing the commands used to obtain the current data stored in the file.
FILE_ID	size_t	an unique file ID (value=10) (NOT USED).