21cmFAST

Release 3.0.2

The 21cmFAST collaboration

Sep 29, 2020

CONTENTS

A semi-numerical cosmological simulation code for the radio 21cm signal.

This is the official repository for 21cmFAST. As of *v3.0.0*, it is conveniently wrapped in Python to enable more dynamic code.

CHAPTER

ONE

NEW FEATURES IN 3.0.0+

- Robust on-disk caching/writing both for efficiency and simplified reading of previously processed data (using HDF5).
- Convenient data objects which simplify access to and processing of the various density and ionization fields.
- De-coupled functions mean that arbitrary functionality can be injected into the process.
- Improved exception handling and debugging
- Comprehensive documentation
- Comprehensive test suite.
- Strict semantic versioning.

CHAPTER

TWO

DOCUMENTATION

Full documentation (with examples, installation instructions and full API reference) found at https://21cmfast.readthedocs.org.

CHAPTER

THREE

ACKNOWLEDGING

If you find *21cmFAST* useful in your research please cite at least one of the following (whichever is most suitable to you):

> Andrei Mesinger and Steven Furlanetto, "Efficient Simulations of Early Structure Formation and Reionization", The Astrophysical Journal, Volume 669, Issue 2, pp. 663-675 (2007), https://ui.adsabs.harvard.edu/link_gateway/2007ApJ...669..663M/doi:10.1086/521806
>
> Andrei Mesinger, Steven Furlanetto and Renyue Cen, "21CMFAST: a fast, seminumerical simulation of the high-redshift 21-cm signal", Monthly Notices of the Royal Astronomical Society, Volume 411, Issue 2, pp. 955-972 (2011), https://ui.adsabs.harvard.edu/link_gateway/2011MNRAS.411..955M/doi:10.1111/j.1365-2966.2010.17731.x

CHAPTER

FOUR

CONTENTS

4.1 Installation

The easiest way to install *21cmFAST* is to use *conda*. Simply use *conda install -c conda-forge 21cmFAST*. With this method, all dependencies are taken care of, and it should work on either Linux or MacOS. If for some reason this is not possible for you, read on.

4.1.1 Dependencies

We try to have as many of the dependencies automatically installed as possible. However, since *21cmFAST* relies on some C libraries, this is not always possible.

The C libraries required are:

- *gsl*
- *fftw* (compiled with floating-point enabled, and *–enable-shared*)
- *openmp*
- A C-compiler with compatibility with the *-fopenmp* flag. **Note:** it seems that on OSX, if using *gcc*, you will need *v4.9.4+*.

As it turns out, though these are fairly common libraries, getting them installed in a way that *21cmFAST* understands on various operating systems can be slightly non-trivial.

Linux

Most linux distros come with packages for the requirements, and also *gcc* by default, which supports *-fopenmp*. As long as these packages install into the standard location, a standard installation of *21cmFAST* will be automatically possible (see below). If they are installed to a place not on the *LD_LIBRARY/INCLUDE* paths, then you must use the compilation options (see below) to specify where they are.

Note: there exists the option of installing *gsl*, *fftw* and *gcc* using *conda*. This is discussed below in the context of MacOSX, where it is often the easiest way to get the dependencies, but it is equally applicable to linux.

MacOSX

On MacOSX, obtaining *gsl* and *fftw* is typically more difficult, and in addition, the newer native *clang* does not offer *-fopenmp* support.

For *conda* users (which we recommend using), the easiest way to get *gsl* and *fftw* is by doing *conda install -c conda-forge gsl fftw* in your environment.

Note: if you use *conda* to install *gsl* and *fftw*, then you will need to point at their location when installing *21cmFAST* (see compiler options below for details). In this case, the installation command should simply be *prepended* with:

```
LIB=/path/to/conda/env/lib INC=/path/to/conda/env/include
```

To get *gcc*, either use *homebrew*, or again, *conda*: *conda install -c anaconda gcc*. If you get the *conda* version, you still need to install the headers:

```
xcode-select --install
```

On older versions then you need to do:

```
open /Library/Developer/CommandLineTools/Packages/macOS_SDK_headers_for_macOS_<input␣
↪version>.pkg
```

For newer versions, you may need to prepend the following command to your *pip install* command when installing *21cmFAST* (see later instructions):

```
CFLAGS="-isysroot /Library/Developer/CommandLineTools/SDKs/MacOSX<input version>.sdk"
```

See faqs/installation_faq for more detailed questions on installation. If you are on MacOSX and are having trouble with installation (or would like to share a successful installation strategy!) please see the open issue.

With the dependencies installed, follow the instructions below, depending on whether you are a user or a developer.

4.1.2 For Users

Note: *conda* users may want to pre-install the following packages before running the below installation commands:

```
conda install numpy scipy click pyyaml cffi astropy h5py
```

Then, at the command line:

```
pip install git+git://github.com/21cmFAST/21cmFAST.git
```

If developing, from the top-level directory do:

```
pip install -e .
```

Note the compile options discussed below!

4.1.3 For Developers

If you are developing *21cmFAST*, we highly recommend using *conda* to manage your environment, and setting up an isolated environment. If this is the case, setting up a full environment (with all testing and documentation dependencies) should be as easy as (from top-level dir):

```
conda env create -f environment_dev.yml
```

Otherwise, if you are using *pip*:

```
pip install -e .[dev]
```

The `[dev]` "extra" here installs all development dependencies. You can instead use `[tests]` if you only want dependencies for testing, or `[docs]` to be able to compile the documentation.

4.1.4 Compile Options

Various options exist to manage compilation via environment variables. Basically, any variable with "INC" in its name will add to the includes directories, while any variable with "lib" in its name will add to the directories searched for libraries. To change the C compiler, use `CC`. Finally, if you want to compile the C-library in dev mode (so you can do stuff like valgrid and gdb with it), install with DEBUG=True. So for example:

```
CC=/usr/bin/gcc DEBUG=True GSL_LIB=/opt/local/lib FFTW_INC=/usr/local/include pip␣
↪install -e .
```

In addition, the `BOXDIR` variable specifies the *default* directory that any data produced by 21cmFAST will be cached. This value can be updated at any time by changing it in the `$CFGDIR/config.yml` file, and can be overwritten on a per-call basis.

While the `-e` option will keep your library up-to-date with any (Python) changes, this will *not* work when changing the C extension. If the C code changes, you need to manually run `rm -rf build/*` then re-install as above.

Logging in C-Code

By default, the C-code will only print to stderr when it encounters warnings or critical errors. However, there exist several levels of logging output that can be switched on, but only at compilation time. To enable these, use the following:

```
LOG_LEVEL=<log_level> pip install -e .
```

The `<log_level>` can be any non-negative integer, or one of the following (case-insensitive) identifiers:

```
NONE, ERROR, WARNING, INFO, DEBUG, SUPER_DEBUG, ULTRA_DEBUG
```

If an integer is passed, it corresponds to the above levels in order (starting from zero). Be careful if the level is set to 0 (or NONE), as useful error and warning messages will not be printed. By default, the log level is 2 (or WARNING), unless the DEBUG=1 environment variable is set, in which case the default is 4 (or DEBUG). Using very high levels (eg. ULTRA_DEBUG) can print out *a lot* of information and make the run time much longer, but may be useful in some specific cases.

4.2 Design Philosophy and Features for v3+

Here we describe in broad terms the design philosophy of the *new* `21cmFAST`, and some of its new features. This is useful to get an initial bearing of how to go about using `21cmFAST`, though most likely the *tutorials* will be better for that. It is also useful for those who have used the "old" `21cmFAST` (versions 2.1 and less) and want to know why they should use this new version (and how to convert). In doing so, we'll go over some of the key features of `21cmFAST` v3+. To get a more in-depth view of all the options and features available, look at the very thorough *API Reference*.

4.2.1 Design Philosophy

The goal of v3 of `21cmFAST` is to provide the same computational efficiency and scientific features of the previous generations, but packaged in a form that adopts the best modern programming standards, including:

- simple installation
- comprehensive documentation
- comprehensive test suite
- more modular code
- standardised code formatting
- truly open-source and collaborative design (via Github)

Partly to enable these standards, and partly due to the many *extra* benefits it brings, v3 also has the major distinction of being wrapped entirely in Python. The *extra* benefits brought by this include:

- a native python library interface (eg. get your output box directly as a `numpy` array).
- better file-writing, into the HDF5 format, which saves metadata along with the box data.
- a caching system so that the same data never has to be calculated twice.
- reproducibility: know which exact version of `21cmFAST`, with what parameters, produced a given dataset.
- significantly improved warnings and error checking/propagation.
- simplicity for adding new additional effects, and inserting them in the calculation pipeline.

We hope that additional features and benefits will be found by the growing community of `21cmFAST` developers and users.

4.2.2 How it Works

v3 is *not* a complete rewrite of `21cmFAST`. Most of the C-code of previous versions is kept, though it has been modularised and modified in many places. The fundamental routines are the same (barring bugfixes!).

The major programs of the original version (`init`, `perturb`, `ionize` etc.) have been converted into modular *functions* in C. Furthermore, most of the global parameters (and, more often than not, global `#define` options) have been modularised and converted into a series of input "parameter" `structs`. These get passed into the functions. Furthermore, each C function, instead of writing a bunch of files, returns an output `struct` containing all the stuff it computed.

Each of these functions and structs are wrapped in Python using the `cffi` package. CFFI compiles the C code once upon *installation*. Due to the fact that parameters are now passed around to the different functions, rather than being global defines, we no longer need to re-compile every time an option is changed. Python itself can handle changing the parameters, and can use the outputs in whatever way the user desires.

To maintain continuity with previous versions, a CLI interface is provided (see below) that acts in a similar fashion to previous versions.

When `21cmFAST` is installed, it automatically creates a configuration directory in the user's home: `~/.21cmfast`. This houses a number of important configuration options; usually default values of parameters. At this stage, the location of this directory is not itself configurable. The config directory contains example configuration files for the CLI interface (see below), which can also be copied anywhere on disk and modified. Importantly, the `config.yml` file in this directory specifies some of the more important options for how `21cmFAST` behaves by default. One such option is `boxdir`, which specifies the directory in which `21cmFAST` will cache results (see below for details). Finally, the config directory houses several data tables which are used to accelerate several calculations. In principle these files are over-writeable, but they should only be touched if one knows very well what they are doing.

Finally, `21cmFAST` contains a more robust cataloguing/caching method. Instead of saving data with a selection of the dependent parameters written into the filename – a method which is prone to error if a parameter which is not part of that selection is modified – `21cmFAST` writes all data into a configurable central directory with a hash filename unique to *all* parameters upon which the data depends. Each kind of dataset has attached methods which efficiently search this central directory for matching data to be read when necessary. Several arguments are available for all library functions which produce such datasets that control this output. In this way, the data that is being retrieved is always reliably produced with the desired parameters, and users need not concern themselves with how and where the data is saved – it can be retrieved merely by creating an empty object with the desired parameters and calling `.read()`, or even better, by calling the function to *produce* the given dataset, which will by default just read it in if available.

4.2.3 CLI

The CLI interface always starts with the command `21cmfast`, and has a number of subcommands. To list the available subcommands, use:

```
$ 21cmfast --help
```

To get help on any subcommand, simply use:

```
$ 21cmfast <subcommand> --help
```

Any subcommand which runs some aspect of `21cmFAST` will have a `--config` option, which specifies a configuration file (by default this is `~/.21cmfast/runconfig_example.yml`). This config file specifies the parameters of the run. Furthermore, any particular parameter that can be specified in the config file can be alternatively specified on the command line by appending the command with the parameter name, eg.:

```
$ 21cmfast init --config=my_config.yml --HII_DIM=40 hlittle=0.7 --DIM 100 SIGMA_8 0.9
```

The above command shows the numerous ways in which these parameters can be specified (with or without leading dashes, and with or without "=").

The CLI interface, while simple to use, does have the limitation that more complex arguments than can be passed to the library functions are disallowed. For example, one cannot pass a previously calculated initial conditions box to the `perturb` command. However, if such a box has been calculated with the default option to write it to disk, then it will automatically be found and used in such a situation, i.e. the following will not re-calculate the init box:

```
$ 21cmfast init
$ 21cmfast perturb redshift=8.0
```

This means that almost all of the functionality provided in the library is accessible via the CLI.

4.3 Tutorials and FAQs

The following introductory tutorials will help you get started with `21cmFAST`:

4.3.1 Running and Plotting Coeval Cubes

The aim of this tutorial is to introduce you to how `21cmFAST` does the most basic operations: producing single coeval cubes, and visually verifying them. It is a great place to get started with `21cmFAST`.

[1]:
```
%matplotlib inline
import matplotlib.pyplot as plt

# We change the default level of the logger so that
# we can see what's happening with caching.
import logging, sys, os
logger = logging.getLogger('21cmFAST')
logger.setLevel(logging.INFO)

import py21cmfast as p21c

# For plotting the cubes, we use the plotting submodule:
from py21cmfast import plotting

# For interacting with the cache
from py21cmfast import cache_tools
```

[2]:
```
print(f"Using 21cmFAST version {p21c.__version__}")
```

```
Using 21cmFAST version 3.0.0.dev2
```

Clear the cache so that we get the same results for the notebook every time (don't worry about this for now). Also, set the default output directory to `_cache/`:

[3]:
```
p21c.config['direc'] = '_cache'
cache_tools.clear_cache(direc="_cache")
```

```
2020-02-29 15:10:11,068 | INFO | Removing PerturbedField_
↪d3184aab64bde877b82cf02bb2993cd0_r12345.h5
2020-02-29 15:10:11,072 | INFO | Removing InitialConditions_
↪6f0eb48c62c36acef23416d5d0fbcf3b_r12345.h5
2020-02-29 15:10:11,102 | INFO | Removing BrightnessTemp_
↪e1bc72a4dbe403d8285dc7d5d9296033_r12345.h5
2020-02-29 15:10:11,106 | INFO | Removing PerturbedField_
↪a26a39863127d77625aa3cec5ea97941_r12345.h5
2020-02-29 15:10:11,111 | INFO | Removing IonizedBox_775ac06d5351d74136a42b5faacdc2d5_
↪r12345.h5
2020-02-29 15:10:11,117 | INFO | Removing BrightnessTemp_
↪9ad800878a70431b47d3ba2b5e6d5fd9_r12345.h5
2020-02-29 15:10:11,297 | INFO | Removing Coeval_z8.0_
↪a3c7dea665420ae9c872ba2fab1b3d7d_r12345.h5
2020-02-29 15:10:11,346 | INFO | Removing IonizedBox_cf2130a24099a2b4b0428560e1653efc_
↪r12345.h5
2020-02-29 15:10:11,352 | INFO | Removing IonizedBox_455465ec84d9bd239f55ab7325d3e9ea_
↪r12345.h5
```

(continues on next page)

(continued from previous page)

```
2020-02-29 15:10:11,357 | INFO | Removing PerturbedField_
↪be9da3b74bf987914d7fa6bae65e5e39_r12345.h5
2020-02-29 15:10:11,362 | INFO | Removing BrightnessTemp_
↪c46f2617c10930bff143cd51e7099f25_r12345.h5
2020-02-29 15:10:11,365 | INFO | Removed 11 files from cache.
```

Basic Usage

The simplest (and typically most efficient) way to produce a coeval cube is simply to use the `run_coeval` method. This consistently performs all steps of the calculation, re-using any data that it can without re-computation or increased memory overhead.

[4]:
```
coeval8, coeval9, coeval10 = p21c.run_coeval(
    redshift = [8.0, 9.0, 10.0],
    user_params = {"HII_DIM": 100, "BOX_LEN": 100},
    cosmo_params = p21c.CosmoParams(SIGMA_8=0.8),
    astro_params = p21c.AstroParams({"HII_EFF_FACTOR":20.0}),
    random_seed=12345
)
```

There are a number of possible inputs for `run_coeval`, which you can check out either in the API reference or by calling `help(p21c.run_coeval)`. Notably, the `redshift` must be given: it can be a single number, or a list of numbers, defining the redshift at which the output coeval cubes will be defined.

Other params we've given here are `user_params`, `cosmo_params` and `astro_params`. These are all used for defining input parameters into the backend C code (there's also another possible input of this kind; `flag_options`). These can be given either as a dictionary (as `user_params` has been), or directly as a relevant object (like `cosmo_params` and `astro_params`). If creating the object directly, the parameters can be passed individually or via a single dictionary. So there's a lot of flexibility there! Nevertheless we *encourage* you to use the basic dictionary. The other ways of passing the information are there so we can use pre-defined objects later on. For more information about these "input structs", see the API docs.

We've also given a `direc` option: this is the directory in which to search for cached data (and also where cached data should be written). Throughout this notebook we're going to set this directly to the `_cache` folder, which allows us to manage it directly. By default, the cache location is set in the global configuration in `~/.21cmfast/config.yml`. You'll learn more about caching further on in this tutorial.

Finally, we've given a random seed. This sets all the random phases for the simulation, and ensures that we can exactly reproduce the same results on every run.

The output of `run_coeval` is a list of `Coeval` instances, one for each input redshift (it's just a single object if a single redshift was passed, not a list). They store *everything* related to that simulation, so that it can be completely compared to other simulations.

For example, the input parameters:

[5]:
```
print("Random Seed: ", coeval8.random_seed)
print("Redshift: ", coeval8.redshift)
print(coeval8.user_params)
```

```
Random Seed:  12345
Redshift:  8.0
UserParams(BOX_LEN:100, DIM:300, HII_DIM:100, HMF:1, POWER_SPECTRUM:0, USE_FFTW_
↪WISDOM:False, USE_RELATIVE_VELOCITIES:False)
```

This is where the utility of being able to pass a *class instance* for the parameters arises: we could run another iteration of coeval cubes, with the same user parameters, simply by doing `p21c.run_coeval(user_params=coeval8.user_params, ...)`.

Also in the `Coeval` instance are the various outputs from the different steps of the computation. You'll see more about what these steps are further on in the tutorial. But for now, we show that various boxes are available:

[6]:
```
print(coeval8.hires_density.shape)
print(coeval8.brightness_temp.shape)
```

```
(300, 300, 300)
(100, 100, 100)
```

Along with these, full instances of the output from each step are available as attributes that end with "struct". These instances themselves contain the `numpy` arrays of the data cubes, and some other attributes that make them easier to work with:

[7]:
```
coeval8.brightness_temp_struct.global_Tb
```

[7]:
```
17.622644
```

By default, each of the components of the cube are cached to disk (in our `_cache/` folder) as we run it. However, the `Coeval` cube itself is *not* written to disk by default. Writing it to disk incurs some redundancy, since that data probably already exists in the cache directory in seperate files.

Let's save to disk. The save method by default writes in the current directory (not the cache!):

[8]:
```
filename = coeval8.save(direc='_cache')
```

The filename of the saved file is returned:

[9]:
```
print(os.path.basename(filename))
```

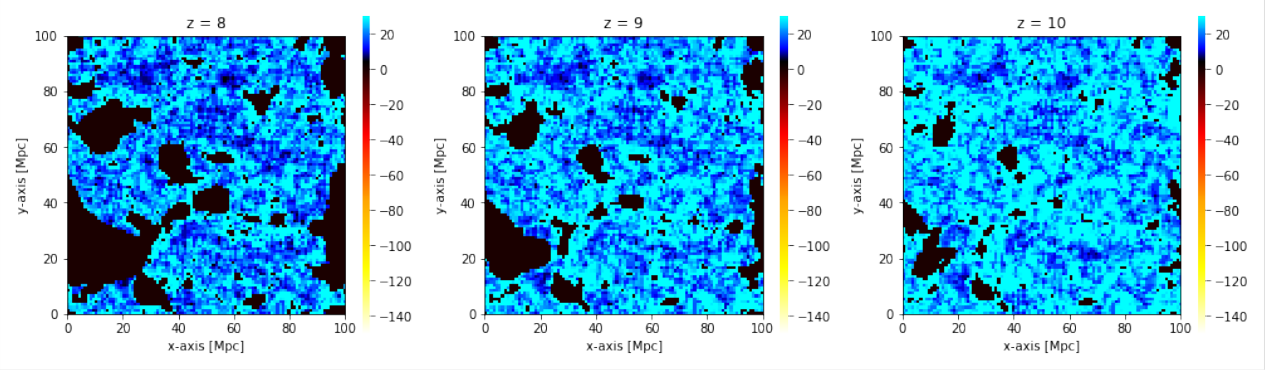
```
Coeval_z8.0_a3c7dea665420ae9c872ba2fab1b3d7d_r12345.h5
```

Such files can be read in:

[10]:
```
new_coeval8 = p21c.Coeval.read(filename, direc='.')
```

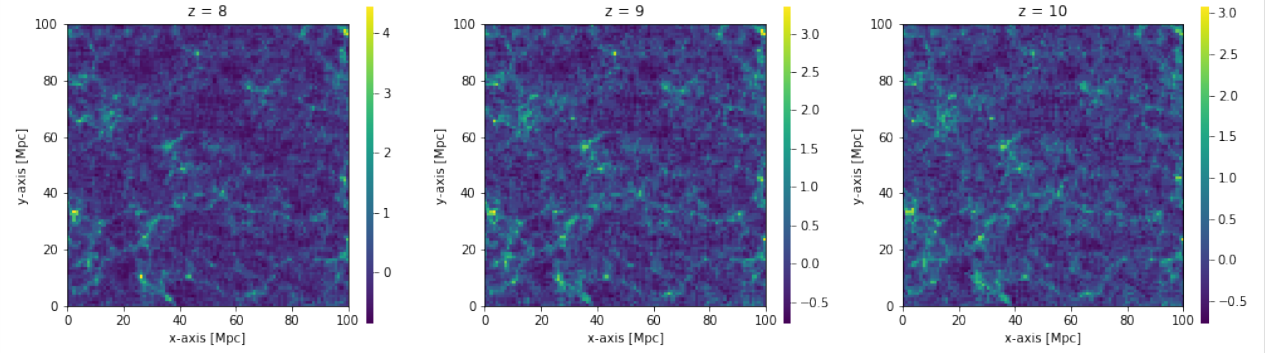
Some convenient plotting functions exist in the `plotting` module. These can work directly on `Coeval` objects, or any of the output structs (as we'll see further on in the tutorial). By default the `coeval_sliceplot` function will plot the `brightness_temp`, using the standard traditional colormap:

[11]:
```
fig, ax = plt.subplots(1,3, figsize=(14,4))
for i, (coeval, redshift) in enumerate(zip([coeval8, coeval9, coeval10], [8,9,10])):
    plotting.coeval_sliceplot(coeval, ax=ax[i], fig=fig);
    plt.title("z = %s"%redshift)
plt.tight_layout()
```



Any 3D field can be plotted, by setting the `kind` argument. For example, we could alternatively have plotted the dark matter density cubes perturbed to each redshift:

[12]:
```
fig, ax = plt.subplots(1,3, figsize=(14,4))
for i, (coeval, redshift) in enumerate(zip([coeval8, coeval9, coeval10], [8,9,10])):
    plotting.coeval_sliceplot(coeval, kind='density', ax=ax[i], fig=fig);
    plt.title("z = %s"%redshift)
plt.tight_layout()
```



To see more options for the plotting routines, see the API Documentation.

`Coeval` instances are not cached themselves – they are containers for data that is itself cached (i.e. each of the `_struct` attributes of `Coeval`). See the api docs for more detailed information on these.

You can see the filename of each of these structs (or the filename it would have if it were cached – you can opt to *not* write out any given dataset):

[13]:
```
coeval8.init_struct.filename
```

[13]:
```
'InitialConditions_6f0eb48c62c36acef23416d5d0fbcf3b_r12345.h5'
```

You can also write the struct anywhere you'd like on the filesystem. This will not be able to be automatically used as a cache, but it could be useful for sharing files with colleagues.

[14]:
```
coeval8.init_struct.save(fname='my_init_struct.h5')
```

This brief example covers most of the basic usage of `21cmFAST` (at least with `Coeval` objects – there are also `Lightcone` objects for which there is a separate tutorial).

For the rest of the tutorial, we'll cover a more advanced usage, in which each step of the calculation is done independently.

Advanced Step-by-Step Usage

Most users most of the time will want to use the high-level `run_coeval` function from the previous section. However, there are several independent steps when computing the brightness temperature field, and these can be performed one-by-one, adding any other effects between them if desired. This means that the new `21cmFAST` is much more flexible. In this section, we'll go through in more detail how to use the lower-level methods.

Each step in the chain will receive a number of input-parameter classes which define how the calculation should run. These are the `user_params`, `cosmo_params`, `astro_params` and `flag_options` that we saw in the previous section.

Conversely, each step is performed by running a function which will return a single object. Every major function returns an object of the same fundamental class (an `OutputStruct`) which has various methods for reading/writing the data, and ensuring that it's in the right state to receive/pass to and from C. These are the objects stored as `init_box_struct` etc. in the `Coeval` class.

As we move through each step, we'll outline some extra details, hints and tips about using these inputs and outputs.

Initial Conditions

The first step is to get the initial conditions, which defines the *cosmological* density field before any redshift evolution is applied.

[15]:
```
initial_conditions = p21c.initial_conditions(
    user_params = {"HII_DIM": 100, "BOX_LEN": 100},
    cosmo_params = p21c.CosmoParams(SIGMA_8=0.8),
    random_seed=54321
)
```

We've already come across all these parameters as inputs to the `run_coeval` function. Indeed, most of the steps have very similar interfaces, and are able to take a random seed and parameters for where to look for the cache. We use a different seed than in the previous section so that all our boxes are "fresh" (we'll show how the caching works in a later section).

These initial conditions have 100 cells per side, and a box length of 100 Mpc. Note again that they can either be passed as a dictionary containing the input parameters, or an actual instance of the class. While the former is the suggested way, one benefit of the latter is that it can be queried for the relevant parameters (by using `help` or a post-fixed `?`), or even queried for defaults:

[16]:
```
p21c.CosmoParams._defaults_
```

[16]:
```
{'SIGMA_8': 0.8102,
 'hlittle': 0.6766,
 'OMm': 0.30964144154550644,
 'OMb': 0.04897468161869667,
 'POWER_INDEX': 0.9665}
```

(these defaults correspond to the Planck15 cosmology contained in Astropy).

So what is in the `initial_conditions` object? It is what we call an `OutputStruct`, and we have seen it before, as the `init_box_struct` attribute of `Coeval`. It contains a number of arrays specifying the density and velocity fields of our initial conditions, as well as the defining parameters. For example, we can easily show the cosmology parameters that are used (note the non-default σ_8 that we passed):

[17]:
```
initial_conditions.cosmo_params
```

[17]:
```
CosmoParams(OMb:0.04897468161869667, OMm:0.30964144154550644, POWER_INDEX:0.9665,␣
↪SIGMA_8:0.8, hlittle:0.6766)
```

A handy tip is that the `CosmoParams` class also has a reference to a corresponding Astropy cosmology, which can be used more broadly:

[18]:
```
initial_conditions.cosmo_params.cosmo
```

[18]:
```
FlatLambdaCDM(name="Planck15", H0=67.7 km / (Mpc s), Om0=0.31, Tcmb0=2.725 K, Neff=3.
↪05, m_nu=[0.   0.   0.06] eV, Ob0=0.049)
```

Merely printing the initial conditions object gives a useful representation of its dependent parameters:

[19]:
```
print(initial_conditions)
```

```
InitialConditions(UserParams(BOX_LEN:100, DIM:300, HII_DIM:100, HMF:1, POWER_SPECTRUM:
↪0, USE_FFTW_WISDOM:False, USE_RELATIVE_VELOCITIES:False);
        CosmoParams(OMb:0.04897468161869667, OMm:0.30964144154550644, POWER_INDEX:0.
↪9665, SIGMA_8:0.8, hlittle:0.6766);
        random_seed:54321)
```

(side-note: the string representation of the object is used to uniquely define it in order to save it to the cache... which we'll explore soon!).

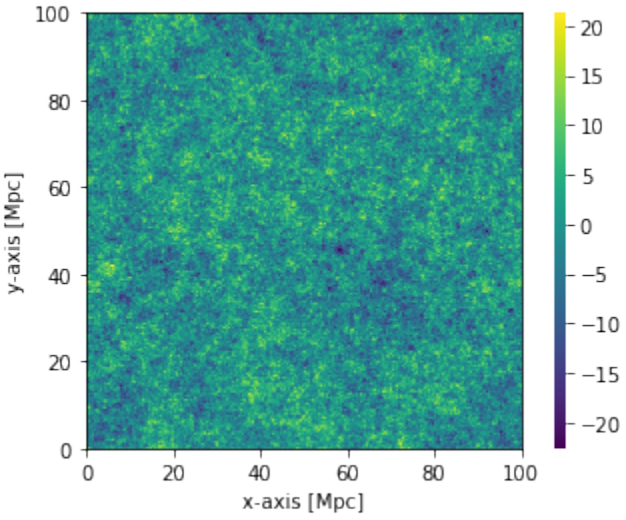
To see which arrays are defined in the object, access the `fieldnames` (this is true for *all* `OutputStruct` objects):

[20]:
```
initial_conditions.fieldnames
```

[20]:
```
['lowres_density',
 'lowres_vx',
 'lowres_vy',
 'lowres_vz',
 'lowres_vx_2LPT',
 'lowres_vy_2LPT',
 'lowres_vz_2LPT',
 'hires_density',
 'lowres_vcb',
 'hires_vcb']
```

The `coeval_sliceplot` function also works on `OutputStruct` objects (as well as the `Coeval` object as we've already seen). It takes the object, and a specific field name. By default, the field it plots is the *first* field in `fieldnames` (for any `OutputStruct`).

[21]:
```
plotting.coeval_sliceplot(initial_conditions, "hires_density");
```



Perturbed Field

After obtaining the initial conditions, we need to *perturb* the field to a given redshift (i.e. the redshift we care about). This step clearly requires the results of the previous step, which we can easily just pass in. Let's do that:

[22]:
```
perturbed_field = p21c.perturb_field(
    redshift = 8.0,
    init_boxes = initial_conditions
)
```

Note that we didn't need to pass in any input parameters, because they are all contained in the `initial_conditions` object itself. The random seed is also taken from this object.

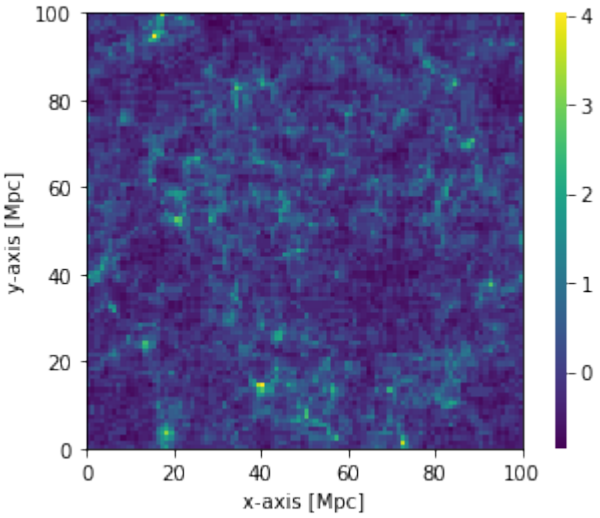
Again, the output is an `OutputStruct`, so we can view its fields:

[23]:
```
perturbed_field.fieldnames
```

[23]:
```
['density', 'velocity']
```

This time, it has only density and velocity (the velocity direction is chosen without loss of generality). Let's view the perturbed density field:

[24]:
```
plotting.coeval_sliceplot(perturbed_field, "density");
```



It is clear here that the density used is the *low*-res density, but the overall structure of the field looks very similar.

Ionization Field

Next, we need to ionize the box. This is where things get a little more tricky. In the simplest case (which, let's be clear, is what we're going to do here) the ionization occurs at the *saturated limit*, which means we can safely ignore the contribution of the spin temperature. This means we can directly calculate the ionization on the density/velocity fields that we already have. A few more parameters are needed here, and so two more "input parameter dictionaries" are available, `astro_params` and `flag_options`. Again, a reminder that their parameters can be viewed by using eg. `help(p21c.AstroParams)`, or by looking at the API docs.

For now, let's leave everything as default. In that case, we can just do:

[25]:
```
ionized_field = p21c.ionize_box(
    perturbed_field = perturbed_field
)
```

```
2020-02-29 15:10:43,902 | INFO | Existing init_boxes found and read in (seed=54321).
```

That was easy! All the information required by `ionize_box` was given directly by the `perturbed_field` object. If we had *also* passed a redshift explicitly, this redshift would be checked against that from the `perturbed_field` and an error raised if they were incompatible:

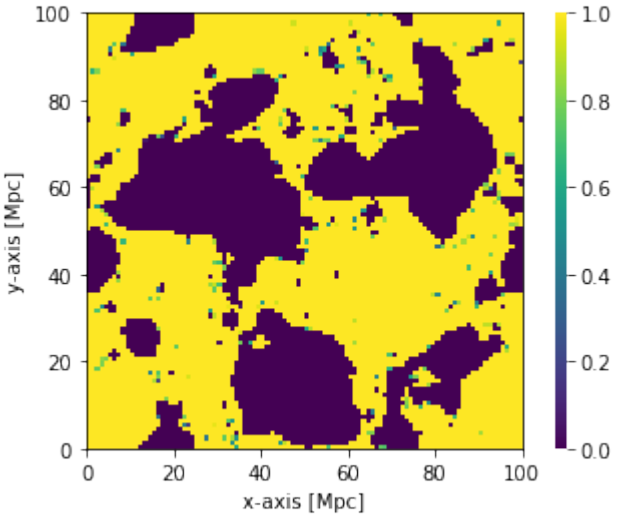
Let's see the fieldnames:

[26]:
```
ionized_field.fieldnames
```

[26]:
```
['first_box', 'xH_box', 'Gamma12_box', 'z_re_box', 'dNrec_box']
```

Here the `first_box` field is actually just a flag to tell the C code whether this has been *evolved* or not. Here, it hasn't been, it's the "first box" of an evolutionary chain. Let's plot the neutral fraction:

[27]:
```
plotting.coeval_sliceplot(ionized_field, "xH_box");
```



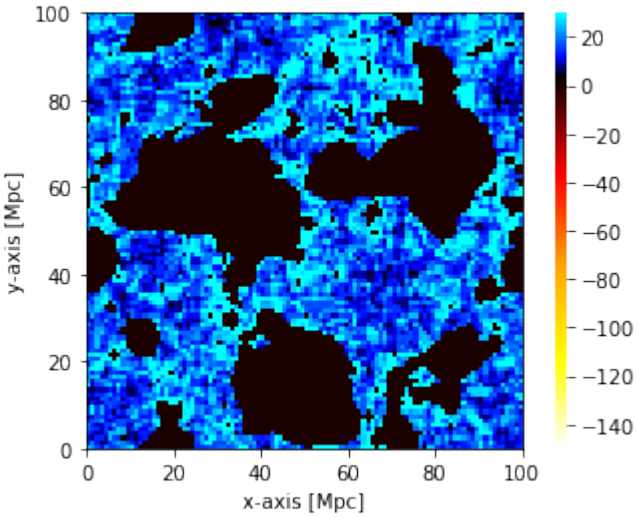
Brightness Temperature

Now we can use what we have to get the brightness temperature:

[28]:
```
brightness_temp = p21c.brightness_temperature(ionized_box=ionized_field, perturbed_
↪field=perturbed_field)
```

This has only a single field, `brightness_temp`:

[29]:
```
plotting.coeval_sliceplot(brightness_temp);
```



The Problem

And there you have it – you've computed each of the four steps (there's actually another, `spin_temperature`, that you require if you don't assume the saturated limit) individually.

However, some problems quickly arise. What if you want the `perturb_field`, but don't care about the initial conditions? We know how to get the full `Coeval` object in one go, but it would seem that the sub-boxes have to *each* be computed as the input to the next.

A perhaps more interesting problem is that some quantities require *evolution*: i.e. a whole bunch of simulations at a string of redshifts must be performed in order to obtain the current redshift. This is true when not in the saturated limit, for example. That means you'd have to manually compute each redshift in turn, and pass it to the computation at the next redshift. While this is definitely possible, it becomes difficult to set up manually when all you care about is the box at the final redshift.

`py21cmfast` solves this by making each of the functions recursive: if `perturb_field` is not passed the `init_boxes` that it needs, it will go and compute them, based on the parameters that you've passed it. If the previous `spin_temp` box required for the current redshift is not passed – it will be computed (and if it doesn't have a previous `spin_temp` *it* will be computed, and so on).

That's all good, but what if you now want to compute another `perturb_field`, with the same fundamental parameters (but at a different redshift)? Since you didn't ever see the `init_boxes`, they'll have to be computed all over again. That's where the automatic caching comes in, which is where we turn now…

Using the Automatic Cache

To solve all this, `21cmFAST` uses an on-disk caching mechanism, where all boxes are saved in HDF5 format in a default location. The cache allows for reading in previously-calculated boxes automatically if they match the parameters that are input. The functions used at every step (in the previous section) will try to use a cached box instead of calculating a new one, unless its explicitly asked *not* to.

Thus, we could do this:

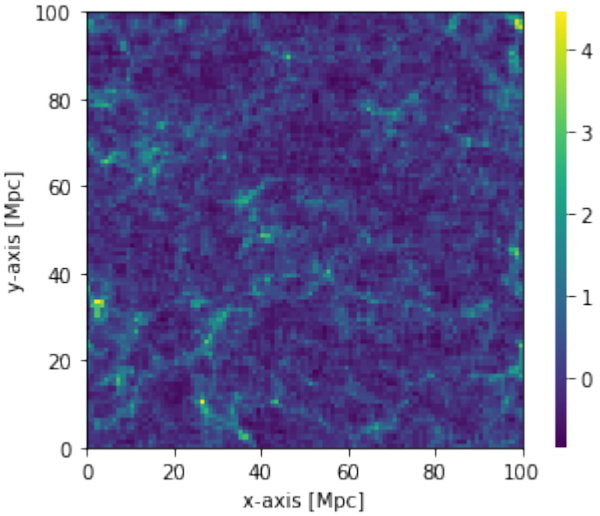
[30]:
```
perturbed_field = p21c.perturb_field(
    redshift = 8.0,
    user_params = {"HII_DIM": 100, "BOX_LEN": 100},
```

(continues on next page)

(continued from previous page)

```
    cosmo_params = p21c.CosmoParams(SIGMA_8=0.8),
)
plotting.coeval_sliceplot(perturbed_field, "density");
```

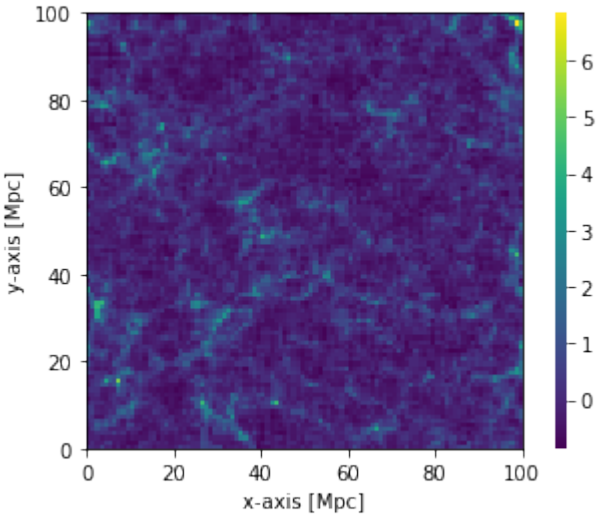
```
2020-02-29 15:10:45,367 | INFO | Existing z=8.0 perturb_field boxes found and read in␣
↪(seed=12345).
```



Note that here we pass exactly the same parameters as were used in the previous section. It gives a message that the full box was found in the cache and immediately returns. However, if we change the redshift:

[31]:
```
perturbed_field = p21c.perturb_field(
    redshift = 7.0,
    user_params = {"HII_DIM": 100, "BOX_LEN": 100},
    cosmo_params = p21c.CosmoParams(SIGMA_8=0.8),
)
plotting.coeval_sliceplot(perturbed_field, "density");
```

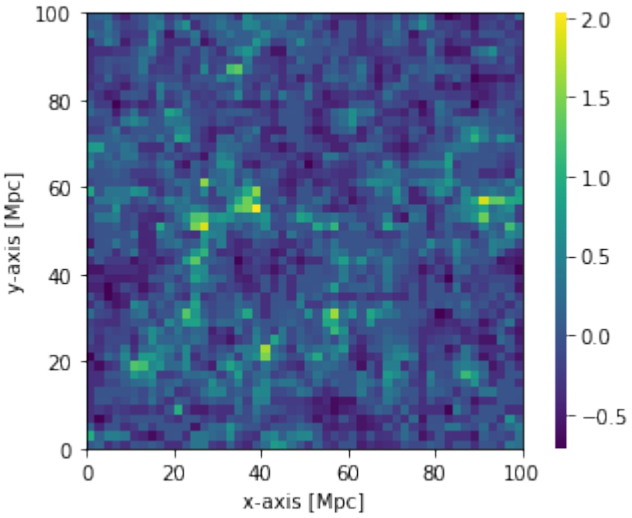
```
2020-02-29 15:10:45,748 | INFO | Existing init_boxes found and read in (seed=12345).
```



Now it finds the initial conditions, but it must compute the perturbed field at the new redshift. If we had changed the initial parameters as well, it would have to calculate everything:

[32]:
```
perturbed_field = p21c.perturb_field(
    redshift = 8.0,
    user_params = {"HII_DIM": 50, "BOX_LEN": 100},
    cosmo_params = p21c.CosmoParams(SIGMA_8=0.8),
)

plotting.coeval_sliceplot(perturbed_field, "density");
```



This shows that we don't need to perform the *previous* step to do any of the steps, they will be calculated automatically.

Now, let's get an ionized box, but this time we won't assume the saturated limit, so we need to use the spin temperature. We can do this directly in the `ionize_box` function, but let's do it explicitly. We will use the auto-generation of the initial conditions and perturbed field. However, the spin temperature is an *evolved* field, i.e. to compute the field at z, we need to know the field at $z + \Delta z$. This continues up to some redshift, labelled `z_heat_max`, above which the spin temperature can be defined directly from the perturbed field.

Thus, one option is to pass to the function a *previous* spin temperature box, to evolve to *this* redshift. However, we don't have a previous spin temperature box yet. Of course, the function itself will go and calculate that box if it's not given (or read it from cache if it's been calculated before!). When it tries to do that, it will go to the one before, and so on until it reaches `z_heat_max`, at which point it will calculate it directly.

To facilitate this recursive progression up the redshift ladder, there is a parameter, `z_step_factor`, which is a multiplicate factor that determines the previous redshift at each step.

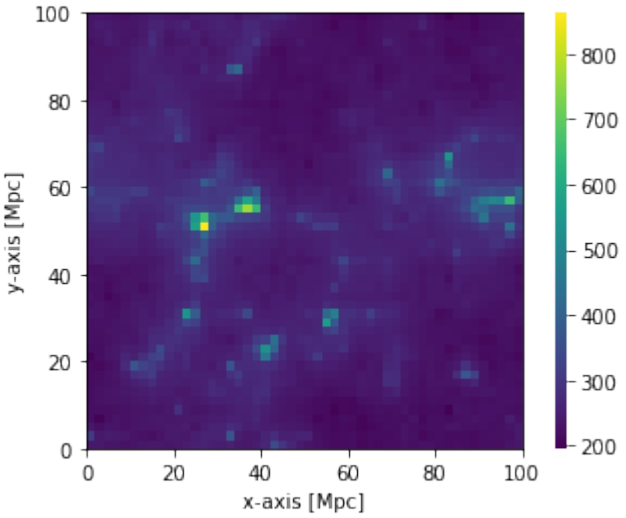
We can also pass the dependent boxes explicitly, which provides the parameters necessary.

WARNING: THIS IS THE MOST TIME-CONSUMING STEP OF THE CALCULATION!

[34]:
```
spin_temp = p21c.spin_temperature(
    perturbed_field = perturbed_field,
    zprime_step_factor=1.05,
)
```

```
2020-02-29 15:11:38,347 | INFO | Existing init_boxes found and read in␣
↪(seed=521414794440).
```

[35]:
```
plotting.coeval_sliceplot(spin_temp, "Ts_box");
```



Let's note here that each of the functions accepts a few of the same arguments that modifies how the boxes are cached. There is a `write` argument, which if set to `False`, will disable writing that box to cache (and it is passed through the recursive heirarchy). There is also `regenerate`, which if `True`, forces this box and all its predecessors to be re-calculated even if they exist in the cache. Then there is `direc`, which we have seen before.

Finally note that by default, `random_seed` is set to `None`. If this is the case, then any cached dataset matching all other parameters will be read in, and the `random_seed` will be set based on the file read in. If it is set to an integer number, then the cached dataset must also match the seed. If it is `None`, and no matching dataset is found, a random seed will be autogenerated.

Now if we calculate the ionized box, ensuring that it uses the spin temperature, then it will also need to be evolved. However, due to the fact that we cached each of the spin temperature steps, these should be read in accordingly:

[36]:
```
ionized_box = p21c.ionize_box(
    spin_temp = spin_temp,
    zprime_step_factor=1.05,
)
```

```
2020-02-29 15:12:55,794 | INFO | Existing init_boxes found and read in␣
↪(seed=521414794440).
2020-02-29 15:12:55,814 | INFO | Existing z=34.2811622461279 perturb_field boxes␣
↪found and read in (seed=521414794440).
2020-02-29 15:12:55,827 | INFO | Existing z=34.2811622461279 spin_temp boxes found␣
↪and read in (seed=521414794440).
2020-02-29 15:12:55,865 | INFO | Existing z=32.60110690107419 perturb_field boxes␣
↪found and read in (seed=521414794440).
2020-02-29 15:12:55,880 | INFO | Existing z=32.60110690107419 spin_temp boxes found␣
↪and read in (seed=521414794440).
2020-02-29 15:12:55,906 | INFO | Existing z=31.00105419149923 perturb_field boxes␣
↪found and read in (seed=521414794440).
2020-02-29 15:12:55,919 | INFO | Existing z=31.00105419149923 spin_temp boxes found␣
↪and read in (seed=521414794440).
2020-02-29 15:12:55,948 | INFO | Existing z=29.4771944680945 perturb_field boxes␣
↪found and read in (seed=521414794440).
2020-02-29 15:12:55,963 | INFO | Existing z=29.4771944680945 spin_temp boxes found␣
↪and read in (seed=521414794440).
2020-02-29 15:12:55,991 | INFO | Existing z=28.02589949342333 perturb_field boxes␣
↪found and read in (seed=521414794440).
2020-02-29 15:12:56,005 | INFO | Existing z=28.02589949342333 spin_temp boxes found␣
↪and read in (seed=521414794440).
```

(continues on next page)

(continued from previous page)

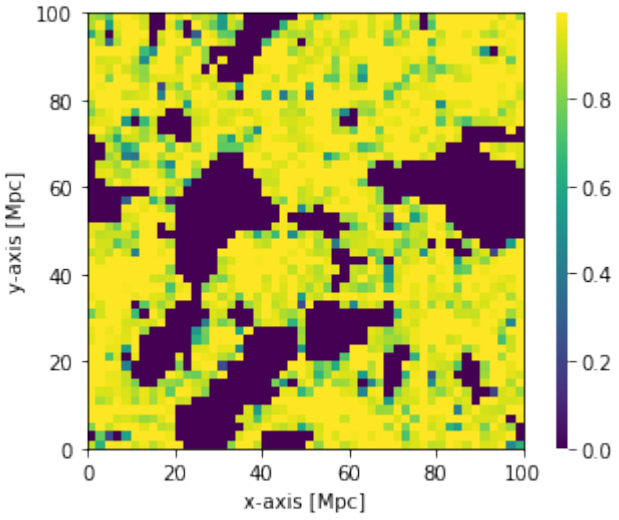
```
2020-02-29 15:12:56,033 | INFO | Existing z=26.643713803260315 perturb_field boxes␣
↪found and read in (seed=521414794440).
2020-02-29 15:12:56,051 | INFO | Existing z=26.643713803260315 spin_temp boxes found␣
↪and read in (seed=521414794440).
2020-02-29 15:12:56,079 | INFO | Existing z=25.32734647929554 perturb_field boxes␣
↪found and read in (seed=521414794440).
2020-02-29 15:12:56,094 | INFO | Existing z=25.32734647929554 spin_temp boxes found␣
↪and read in (seed=521414794440).
2020-02-29 15:12:56,127 | INFO | Existing z=24.073663313614798 perturb_field boxes␣
↪found and read in (seed=521414794440).
2020-02-29 15:12:56,141 | INFO | Existing z=24.073663313614798 spin_temp boxes found␣
↪and read in (seed=521414794440).
2020-02-29 15:12:56,168 | INFO | Existing z=22.879679346299806 perturb_field boxes␣
↪found and read in (seed=521414794440).
2020-02-29 15:12:56,182 | INFO | Existing z=22.879679346299806 spin_temp boxes found␣
↪and read in (seed=521414794440).
2020-02-29 15:12:56,205 | INFO | Existing z=21.742551758380767 perturb_field boxes␣
↪found and read in (seed=521414794440).
2020-02-29 15:12:56,219 | INFO | Existing z=21.742551758380767 spin_temp boxes found␣
↪and read in (seed=521414794440).
2020-02-29 15:12:56,403 | INFO | Existing z=20.659573103219778 perturb_field boxes␣
↪found and read in (seed=521414794440).
2020-02-29 15:12:56,418 | INFO | Existing z=20.659573103219778 spin_temp boxes found␣
↪and read in (seed=521414794440).
2020-02-29 15:12:56,620 | INFO | Existing z=19.62816486020931 perturb_field boxes␣
↪found and read in (seed=521414794440).
2020-02-29 15:12:56,635 | INFO | Existing z=19.62816486020931 spin_temp boxes found␣
↪and read in (seed=521414794440).
2020-02-29 15:12:56,784 | INFO | Existing z=18.645871295437438 perturb_field boxes␣
↪found and read in (seed=521414794440).
2020-02-29 15:12:56,793 | INFO | Existing z=18.645871295437438 spin_temp boxes found␣
↪and read in (seed=521414794440).
2020-02-29 15:12:56,931 | INFO | Existing z=17.71035361470232 perturb_field boxes␣
↪found and read in (seed=521414794440).
2020-02-29 15:12:56,941 | INFO | Existing z=17.71035361470232 spin_temp boxes found␣
↪and read in (seed=521414794440).
2020-02-29 15:12:57,085 | INFO | Existing z=16.81938439495459 perturb_field boxes␣
↪found and read in (seed=521414794440).
2020-02-29 15:12:57,095 | INFO | Existing z=16.81938439495459 spin_temp boxes found␣
↪and read in (seed=521414794440).
2020-02-29 15:12:57,243 | INFO | Existing z=15.970842280909132 perturb_field boxes␣
↪found and read in (seed=521414794440).
2020-02-29 15:12:57,254 | INFO | Existing z=15.970842280909132 spin_temp boxes found␣
↪and read in (seed=521414794440).
2020-02-29 15:12:57,399 | INFO | Existing z=15.162706934199171 perturb_field boxes␣
↪found and read in (seed=521414794440).
2020-02-29 15:12:57,408 | INFO | Existing z=15.162706934199171 spin_temp boxes found␣
↪and read in (seed=521414794440).
2020-02-29 15:12:57,544 | INFO | Existing z=14.393054223046828 perturb_field boxes␣
↪found and read in (seed=521414794440).
2020-02-29 15:12:57,554 | INFO | Existing z=14.393054223046828 spin_temp boxes found␣
↪and read in (seed=521414794440).
2020-02-29 15:12:57,691 | INFO | Existing z=13.66005164099698 perturb_field boxes␣
↪found and read in (seed=521414794440).
2020-02-29 15:12:57,700 | INFO | Existing z=13.66005164099698 spin_temp boxes found␣
↪and read in (seed=521414794440).
2020-02-29 15:12:57,832 | INFO | Existing z=12.961953943806646 perturb_field boxes␣
↪found and read in (seed=521414794440).
```

(continues on next page)

(continued from previous page)

```
2020-02-29 15:12:57,840 | INFO | Existing z=12.961953943806646 spin_temp boxes found␣
↪and read in (seed=521414794440).
2020-02-29 15:12:57,970 | INFO | Existing z=12.297098994101567 perturb_field boxes␣
↪found and read in (seed=521414794440).
2020-02-29 15:12:57,978 | INFO | Existing z=12.297098994101567 spin_temp boxes found␣
↪and read in (seed=521414794440).
2020-02-29 15:12:58,106 | INFO | Existing z=11.663903803906255 perturb_field boxes␣
↪found and read in (seed=521414794440).
2020-02-29 15:12:58,114 | INFO | Existing z=11.663903803906255 spin_temp boxes found␣
↪and read in (seed=521414794440).
2020-02-29 15:12:58,244 | INFO | Existing z=11.060860765625003 perturb_field boxes␣
↪found and read in (seed=521414794440).
2020-02-29 15:12:58,254 | INFO | Existing z=11.060860765625003 spin_temp boxes found␣
↪and read in (seed=521414794440).
2020-02-29 15:12:58,394 | INFO | Existing z=10.486534062500002 perturb_field boxes␣
↪found and read in (seed=521414794440).
2020-02-29 15:12:58,402 | INFO | Existing z=10.486534062500002 spin_temp boxes found␣
↪and read in (seed=521414794440).
2020-02-29 15:12:58,529 | INFO | Existing z=9.939556250000003 perturb_field boxes␣
↪found and read in (seed=521414794440).
2020-02-29 15:12:58,538 | INFO | Existing z=9.939556250000003 spin_temp boxes found␣
↪and read in (seed=521414794440).
2020-02-29 15:12:58,674 | INFO | Existing z=9.418625000000002 perturb_field boxes␣
↪found and read in (seed=521414794440).
2020-02-29 15:12:58,682 | INFO | Existing z=9.418625000000002 spin_temp boxes found␣
↪and read in (seed=521414794440).
2020-02-29 15:12:58,810 | INFO | Existing z=8.922500000000001 perturb_field boxes␣
↪found and read in (seed=521414794440).
2020-02-29 15:12:58,819 | INFO | Existing z=8.922500000000001 spin_temp boxes found␣
↪and read in (seed=521414794440).
2020-02-29 15:12:58,947 | INFO | Existing z=8.450000000000001 perturb_field boxes␣
↪found and read in (seed=521414794440).
2020-02-29 15:12:58,956 | INFO | Existing z=8.450000000000001 spin_temp boxes found␣
↪and read in (seed=521414794440).
2020-02-29 15:12:59,086 | INFO | Existing z=8.0 perturb_field boxes found and read in␣
↪(seed=521414794440).
```

[37]:
```
plotting.coeval_sliceplot(ionized_box, "xH_box");
```

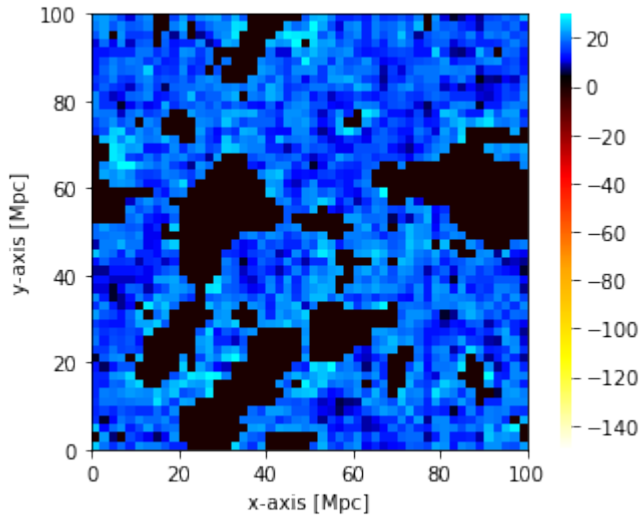


Great! So again, we can just get the brightness temp:

[38]:
```
brightness_temp = p21c.brightness_temperature(
    ionized_box = ionized_box,
    perturbed_field = perturbed_field,
    spin_temp = spin_temp
)
```

Now lets plot our brightness temperature, which has been evolved from high redshift with spin temperature fluctuations:

[39]:
```
plotting.coeval_sliceplot(brightness_temp);
```



We can also check what the result would have been if we had limited the maximum redshift of heating. Note that this *recalculates* all previous spin temperature and ionized boxes, because they depend on both `z_heat_max` and `zprime_step_factor`.

[40]:
```
ionized_box = p21c.ionize_box(
    spin_temp = spin_temp,
    zprime_step_factor=1.05,
    z_heat_max = 20.0
)

brightness_temp = p21c.brightness_temperature(
    ionized_box = ionized_box,
    perturbed_field = perturbed_field,
    spin_temp = spin_temp
)

plotting.coeval_sliceplot(brightness_temp);
```

```
2020-02-29 15:13:08,824 | INFO | Existing init_boxes found and read in␣
↪(seed=521414794440).
2020-02-29 15:13:08,840 | INFO | Existing z=19.62816486020931 perturb_field boxes␣
↪found and read in (seed=521414794440).
2020-02-29 15:13:11,438 | INFO | Existing z=18.645871295437438 perturb_field boxes␣
↪found and read in (seed=521414794440).
2020-02-29 15:13:11,447 | INFO | Existing z=19.62816486020931 spin_temp boxes found␣
↪and read in (seed=521414794440).
```

(continues on next page)

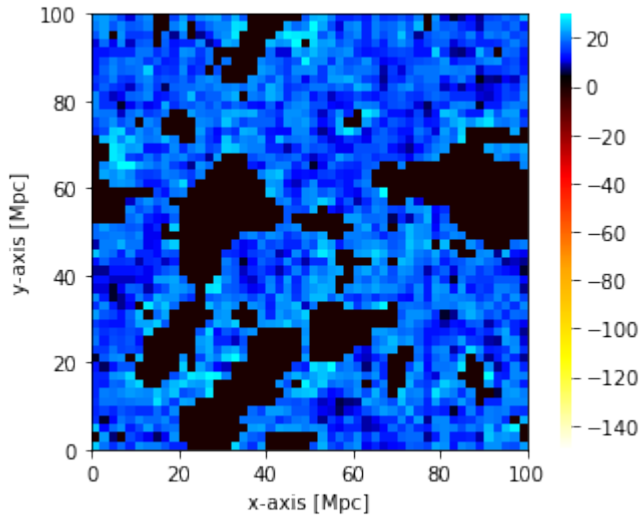
(continued from previous page)

```
2020-02-29 15:13:14,041 | INFO | Existing z=17.71035361470232 perturb_field boxes␣
↪found and read in (seed=521414794440).
2020-02-29 15:13:14,050 | INFO | Existing z=18.645871295437438 spin_temp boxes found␣
↪and read in (seed=521414794440).
2020-02-29 15:13:16,667 | INFO | Existing z=16.81938439495459 perturb_field boxes␣
↪found and read in (seed=521414794440).
2020-02-29 15:13:16,675 | INFO | Existing z=17.71035361470232 spin_temp boxes found␣
↪and read in (seed=521414794440).
2020-02-29 15:13:19,213 | INFO | Existing z=15.970842280909132 perturb_field boxes␣
↪found and read in (seed=521414794440).
2020-02-29 15:13:19,222 | INFO | Existing z=16.81938439495459 spin_temp boxes found␣
↪and read in (seed=521414794440).
2020-02-29 15:13:21,756 | INFO | Existing z=15.162706934199171 perturb_field boxes␣
↪found and read in (seed=521414794440).
2020-02-29 15:13:21,764 | INFO | Existing z=15.970842280909132 spin_temp boxes found␣
↪and read in (seed=521414794440).
2020-02-29 15:13:24,409 | INFO | Existing z=14.393054223046828 perturb_field boxes␣
↪found and read in (seed=521414794440).
2020-02-29 15:13:24,417 | INFO | Existing z=15.162706934199171 spin_temp boxes found␣
↪and read in (seed=521414794440).
2020-02-29 15:13:26,938 | INFO | Existing z=13.66005164099698 perturb_field boxes␣
↪found and read in (seed=521414794440).
2020-02-29 15:13:26,947 | INFO | Existing z=14.393054223046828 spin_temp boxes found␣
↪and read in (seed=521414794440).
2020-02-29 15:13:29,504 | INFO | Existing z=12.961953943806646 perturb_field boxes␣
↪found and read in (seed=521414794440).
2020-02-29 15:13:29,517 | INFO | Existing z=13.66005164099698 spin_temp boxes found␣
↪and read in (seed=521414794440).
2020-02-29 15:13:32,163 | INFO | Existing z=12.297098994101567 perturb_field boxes␣
↪found and read in (seed=521414794440).
2020-02-29 15:13:32,171 | INFO | Existing z=12.961953943806646 spin_temp boxes found␣
↪and read in (seed=521414794440).
2020-02-29 15:13:34,704 | INFO | Existing z=11.663903803906255 perturb_field boxes␣
↪found and read in (seed=521414794440).
2020-02-29 15:13:34,712 | INFO | Existing z=12.297098994101567 spin_temp boxes found␣
↪and read in (seed=521414794440).
2020-02-29 15:13:37,257 | INFO | Existing z=11.060860765625003 perturb_field boxes␣
↪found and read in (seed=521414794440).
2020-02-29 15:13:37,266 | INFO | Existing z=11.663903803906255 spin_temp boxes found␣
↪and read in (seed=521414794440).
2020-02-29 15:13:39,809 | INFO | Existing z=10.486534062500002 perturb_field boxes␣
↪found and read in (seed=521414794440).
2020-02-29 15:13:39,817 | INFO | Existing z=11.060860765625003 spin_temp boxes found␣
↪and read in (seed=521414794440).
2020-02-29 15:13:42,378 | INFO | Existing z=9.939556250000003 perturb_field boxes␣
↪found and read in (seed=521414794440).
2020-02-29 15:13:42,387 | INFO | Existing z=10.486534062500002 spin_temp boxes found␣
↪and read in (seed=521414794440).
2020-02-29 15:13:44,941 | INFO | Existing z=9.418625000000002 perturb_field boxes␣
↪found and read in (seed=521414794440).
2020-02-29 15:13:44,950 | INFO | Existing z=9.939556250000003 spin_temp boxes found␣
↪and read in (seed=521414794440).
2020-02-29 15:13:47,518 | INFO | Existing z=8.922500000000001 perturb_field boxes␣
↪found and read in (seed=521414794440).
2020-02-29 15:13:47,528 | INFO | Existing z=9.418625000000002 spin_temp boxes found␣
↪and read in (seed=521414794440).
2020-02-29 15:13:50,077 | INFO | Existing z=8.450000000000001 perturb_field boxes␣
↪found and read in (seed=521414794440).
```

(continues on next page)

(continued from previous page)

```
2020-02-29 15:13:50,086 | INFO | Existing z=8.922500000000001 spin_temp boxes found␣
↪and read in (seed=521414794440).
2020-02-29 15:13:52,626 | INFO | Existing z=8.0 perturb_field boxes found and read in␣
↪(seed=521414794440).
2020-02-29 15:13:52,762 | INFO | Existing brightness_temp box found and read in␣
↪(seed=521414794440).
```



As we can see, it's very similar!

4.3.2 Running and Plotting LightCones

This tutorial follows on from the coeval cube tutorial, and provides an introduction to creating lightcones with `21cmFAST`. If you are new to `21cmFAST` you should go through the coeval cube tutorial first.

There are two ways of creating lightcones in `21cmFAST`: manual and automatic. The manual way involves evolving a coeval simulation through redshift and saving slices of it into a lightcone array. The advantage of this method is that one can precisely choose the redshift nodes to simulate and decide on interpolation methods. However, in this tutorial, we will focus on the single function that is included to do this for you: `run_lightcone`.

The function takes a few different arguments, most of which will be familiar to you if you've gone through the coeval tutorial. All simulation parameters can be passed (i.e. `user_params`, `cosmo_params`, `flag_options` and `astro_params`). As an alternative to the first two, an `InitialConditions` and/or `PerturbField` box can be passed.

Furthermore, the evolution can be managed with the `zprime_step_factor` and `z_heat_max` arguments.

Finally, the final *minimum* redshift of the lightcone is set by the `redshift` argument, and the maximum redshift of the lightcone is defined by the `max_redshift` argument (note that this is not the maximum redshift evaluated, which is controlled by `z_heat_max`, merely the maximum saved into the returned lightcone).

You can specify which 3D quantities are interpolated as lightcones, and which should be saved as global parameters.

Let's see what it does. We won't use the spin temperature, just to get a simple toy model:

[1]:
```
import py21cmfast as p21c
from py21cmfast import plotting
import os
```

(continues on next page)

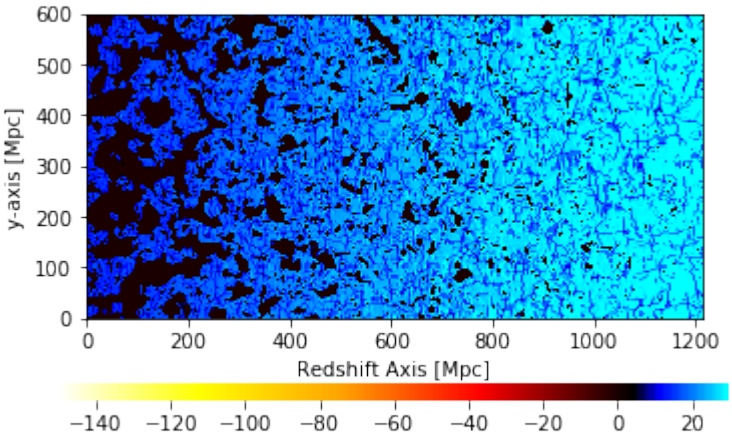
(continued from previous page)

```
print(f"21cmFAST version is {p21c.__version__}")
```

```
21cmFAST version is 3.0.0.dev2
```

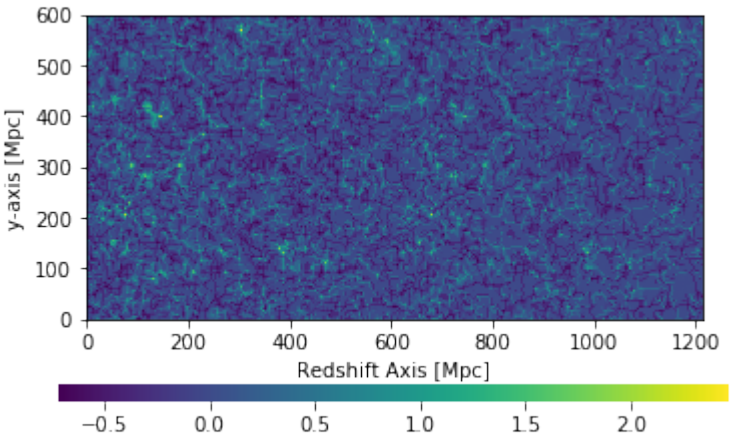
[2]:
```
lightcone = p21c.run_lightcone(
    redshift = 7.0,
    max_redshift = 12.0,
    user_params = {"HII_DIM":150, "BOX_LEN": 600},
    lightcone_quantities=("brightness_temp", 'density'),
    global_quantities=("brightness_temp", 'density', 'xH_box'),
    direc='_cache'
)
```

[3]:
```
plotting.lightcone_sliceplot(lightcone);
```



[4]:
```
plotting.lightcone_sliceplot(lightcone, "density")
```

[4]:
```
(<Figure size 432x288 with 2 Axes>,
 <matplotlib.axes._subplots.AxesSubplot at 0x7f5e93aaa4d0>)
```



Simple!

You can also save lightcones:

[5]:
```
filename = lightcone.save(direc='_cache')
```

[6]:
```
print(os.path.basename(filename))
```

```
LightCone_z7.0_da45d92043dfdc0c14f34f3ded434358_r287478667967.h5
```

[1]:
```
import numpy as np
import matplotlib
%matplotlib inline
import matplotlib.pyplot as plt
import py21cmfast as p21c

#import logging
#logger = logging.getLogger("21cmFAST")
#logger.setLevel(logging.INFO)

random_seed = 1993

EoR_colour = matplotlib.colors.LinearSegmentedColormap.from_list('mycmap',\
             [(0, 'white'),(0.33, 'yellow'),(0.5, 'orange'),(0.68, 'red'),\
              (0.83333, 'black'),(0.9, 'blue'),(1, 'cyan')])
plt.register_cmap(cmap=EoR_colour)
```

This result was obtained using 21cmFAST at commit 2bb4807c7ef1a41649188a3efc462084f2e3b2e0

4.3.3 Fiducial and lightcones

Let's fix the initial condition for this tutorial.

[2]:
```
output_dir = '/home/yqin/aida/hybrid/mini-halos/'
HII_DIM = 128
BOX_LEN = 250

# USE_FFTW_WISDOM make FFT faster
user_params = {"HII_DIM":HII_DIM, "BOX_LEN": BOX_LEN, "USE_FFTW_WISDOM": True}

initial_conditions = p21c.initial_conditions(user_params=user_params,random_
↪seed=random_seed, direc=output_dir)
```

Let's run a 'fiducial' model and see its lightcones

Note that the reference model has

```
pow(10, "F_STAR7_MINI") = pow(10, "F_STAR10") / pow(1000,ALPHA_STAR) * 10 # 10 times␣
↪enhancement
pow(10, "F_ESC7_MINI" ) = pow(10, "F_ESC10" ) / pow(1000,ALPHA_ESC ) / 10 # 0.1 times␣
↪enhancement to balance the 10 times enhanced Ngamma
pow(10, "L_X_MINI"    ) = pow(10, "L_X")
1 - "F_H2_SHIELD"  = 1
```

[3]:
```
# the lightcones we want to plot later together with their color maps and min/max
lightcone_quantities = ('brightness_temp','Ts_box','xH_box',"dNrec_box",'z_re_box',
↪'Gamma12_box','J_21_LW_box',"density")
cmaps = [EoR_colour,'Reds','magma','magma','magma','cubehelix','cubehelix','viridis']
vmins = [-150, 1e1, 0, 0, 5, 0, 0, -1]
vmaxs = [  30, 1e3, 1, 2, 9, 1,10,  1]
```

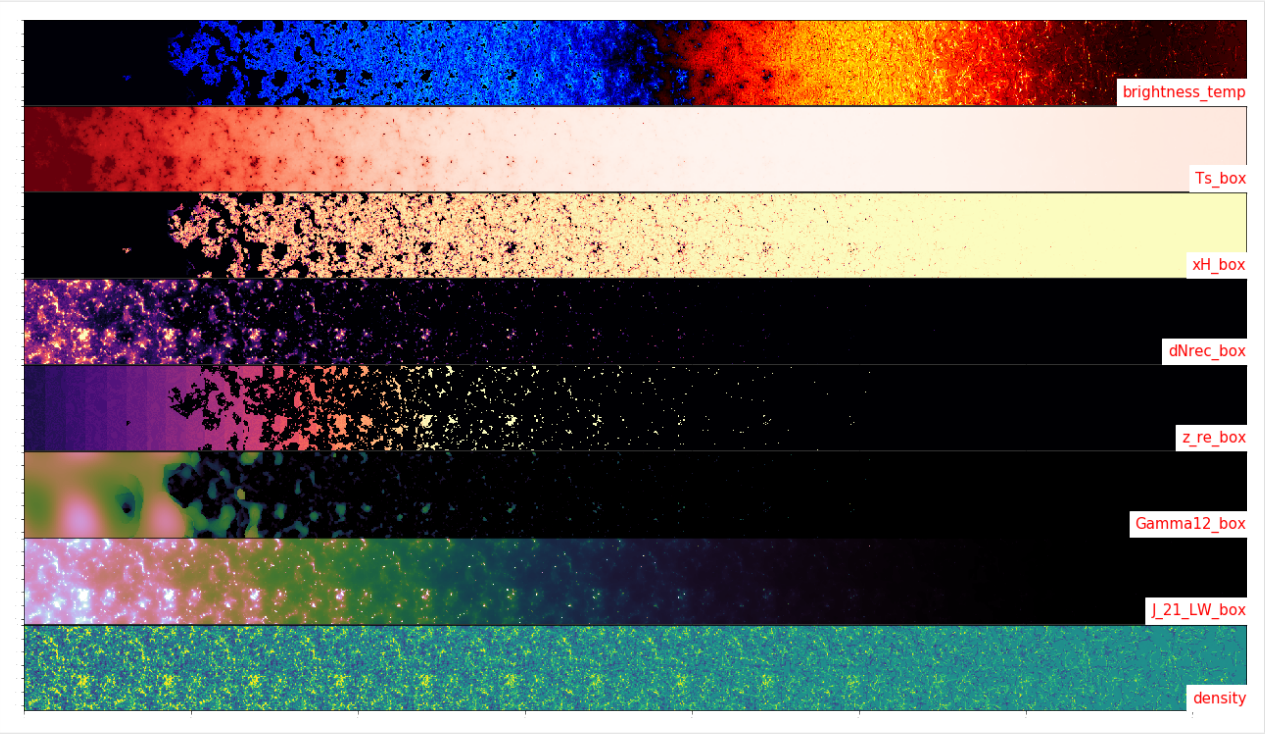
(continues on next page)

(continued from previous page)

```
# set necessary flags for using minihalos and astro parameter
astro_params_fid = {'ALPHA_ESC': 0.0, 'F_ESC10': -1.222, 'F_ESC7_MINI' : -2.222,
                    'ALPHA_STAR': 0.5,'F_STAR10': -1.25, 'F_STAR7_MINI': -1.75,
                    'L_X': 40.5, 'L_X_MINI': 40.5, 'NU_X_THRESH': 500.0, 'F_H2_SHIELD
↪': 0.0}
flag_options_fid = {"INHOMO_RECO":True, 'USE_MASS_DEPENDENT_ZETA':True, 'USE_TS_FLUCT
↪':True, 'USE_MINI_HALOS':True}

lightcone_fid = p21c.run_lightcone(
        redshift = 5.5,
        init_box = initial_conditions,
        flag_options = flag_options_fid,
        astro_params = astro_params_fid,
        lightcone_quantities=lightcone_quantities,
        global_quantities=lightcone_quantities,
        random_seed = random_seed,
        direc = output_dir
)

fig, axs = plt.subplots(len(lightcone_quantities),1,
            figsize=(getattr(lightcone_fid, lightcone_quantities[0]).shape[2]*0.01,
                     getattr(lightcone_fid, lightcone_quantities[0]).shape[1]*0.
↪01*len(lightcone_quantities)))
for ii, lightcone_quantity in enumerate(lightcone_quantities):
    axs[ii].imshow(getattr(lightcone_fid, lightcone_quantity)[1],
                   vmin=vmins[ii], vmax=vmaxs[ii],cmap=cmaps[ii])
    axs[ii].text(1, 0.05, lightcone_quantity,horizontalalignment='right',
↪verticalalignment='bottom',
            transform=axs[ii].transAxes,color = 'red',backgroundcolor='white',
↪fontsize = 15)
    axs[ii].xaxis.set_tick_params(labelsize=0)
    axs[ii].yaxis.set_tick_params(labelsize=0)
plt.tight_layout()
fig.subplots_adjust(hspace = 0.01)
```



varying parameters

let's vary paremeters that describe mini-halos and see the impact to the global signal

We keep other parameters fixed and vary one of following by a factor of 0.1, 0.5, 2 and 10:

```
pow(10, "F_STAR7_MINI")
pow(10, "F_ESC7_MINI")
pow(10, "L_X_MINI")
1 - "F_H2_SHIELD"
```

We also have a NOmini model where mini-halos are not included

[8]:
```
#defining those color, linstyle, blabla
linestyles = ['-', '-',':','-.','-.',':']
colors     = ['gray','black','#e41a1c','#377eb8','#e41a1c','#377eb8']
lws        = [1,3,2,2,2,2]

textss   = ['varying '+r'$f_{*,7}^{\rm mol}$',\
            'varying '+r'$f_{\rm esc}^{\rm mol}$',\
            'varying '+r'$L_{\rm x}^{\rm mol}$',\
            'varying '+r'$1-f_{\rm H_2}^{\rm shield}$']
factorss = [[0, 1, 0.1, 0.5, 2, 10],] * len(textss)
labelss  = [['NOmini', 'reference', 'x0.1', 'x0.5', 'x2', 'x10'],] * len(textss)
```

Note that I've run these simulations in parallel before this tutorial. With these setup, each took ~6h to finish. Here, running means read the cached outputs.

global properties

[86]:
```
global_quantities = ('brightness_temp','Ts_box','xH_box',"dNrec_box",'z_re_box',
↪'Gamma12_box','J_21_LW_box',"density")
#choose some to plot...
plot_quantities = ('brightness_temp','Ts_box','xH_box',"dNrec_box",'Gamma12_box','J_
↪21_LW_box')
ymins = [-120, 1e1, 0, 0, 0, 0]
ymaxs = [  30, 1e3, 1, 1, 1,10]

fig, axss = plt.subplots(len(plot_quantities), len(labelss),
                         sharex=True, figsize=(4*len(labelss),2*len(global_
↪quantities)))

for pp, texts in enumerate(textss):
    labels  = labelss[pp]
    factors = factorss[pp]
    axs     = axss[:,pp]
    for kk, label in enumerate(labels):
        flag_options = flag_options_fid.copy()
        astro_params = astro_params_fid.copy()
        factor = factors[kk]
        if label == 'NOmini':
            flag_options.update({'USE_MINI_HALOS': False})
        else:
            flag_options.update({'USE_MINI_HALOS': True})
            if pp == 0:
                astro_params.update({'F_STAR7_MINI': astro_params_fid['F_STAR7_MINI
↪']+np.log10(factor)})
            elif pp == 1:
                astro_params.update({'F_ESC7_MINI': astro_params_fid['F_ESC7_MINI
↪']+np.log10(factor)})
            elif pp == 2:
                astro_params.update({'L_X_MINI': astro_params_fid['L_X_MINI']+np.
↪log10(factor)})
            else:
                if factor > 1: continue # can't do negative F_H2_SHIELD
                astro_params.update({'F_H2_SHIELD': 1. - (1. - astro_params_fid['F_H2_
↪SHIELD']) * factor})
        if label == 'reference':
            lightcone = lightcone_fid
        else:
            lightcone = p21c.run_lightcone(
                redshift = 5.5,
                init_box = initial_conditions,
                flag_options = flag_options,
                astro_params = astro_params,
                global_quantities=global_quantities,
                random_seed = random_seed,
                direc = output_dir
            )

        freqs = 1420.4 / (np.array(lightcone.node_redshifts) + 1.)
        for jj, global_quantity in enumerate(plot_quantities):
            axs[jj].plot(freqs, getattr(lightcone, 'global_%s'%global_quantity.
↪replace('_box','')),
                         color=colors[kk],linestyle=linestyles[kk], label =␣
↪labels[kk],lw=lws[kk])
```

(continues on next page)

(continued from previous page)

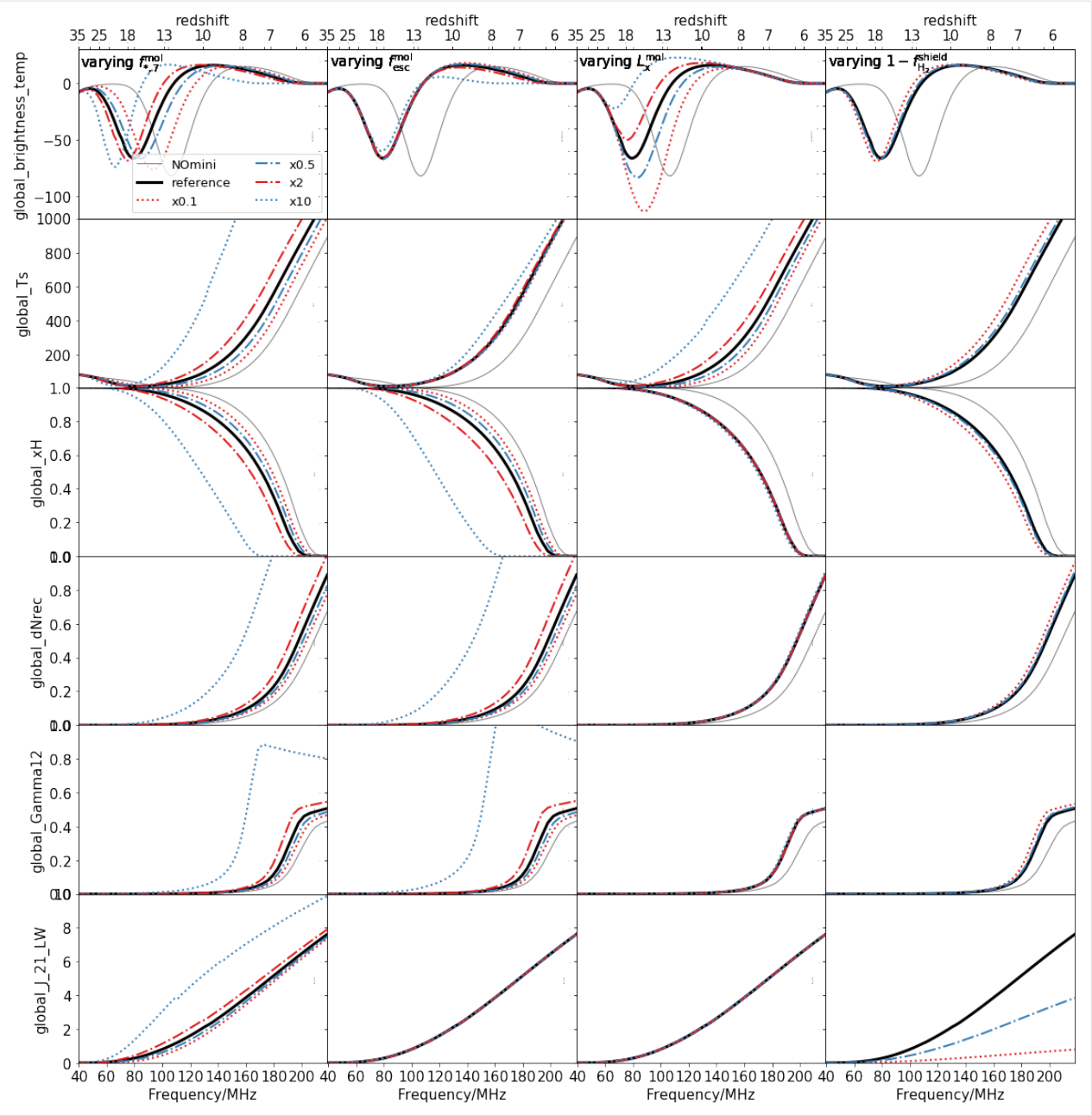
```
    axs[0].text(0.01, 0.99, texts,horizontalalignment='left',verticalalignment='top',
                        transform=axs[0].transAxes,fontsize = 15)
    for jj, global_quantity in enumerate(plot_quantities):
        axs[jj].set_ylim(ymins[jj], ymaxs[jj])
    axs[-1].set_xlabel('Frequency/MHz',fontsize=15)
    axs[-1].xaxis.set_tick_params(labelsize=15)

    axs[0].set_xlim(1420.4 / (35 + 1.), 1420.4 / (5.5 + 1.))
    zlabels = np.array([ 6,  7,  8,  10, 13, 18, 25, 35])
    ax2 = axs[0].twiny()
    ax2.set_xlim(axs[0].get_xlim())
    ax2.set_xticks(1420.4 / (zlabels + 1.))
    ax2.set_xticklabels(zlabels.astype(np.str))
    ax2.set_xlabel("redshift",fontsize=15)
    ax2.xaxis.set_tick_params(labelsize=15)
    ax2.grid(False)

    if pp == 0:
        axs[0].legend(loc='lower right', ncol=2,fontsize=13,fancybox=True,
↪frameon=True)
        for jj, global_quantity in enumerate(plot_quantities):
            axs[jj].set_ylabel('global_%s'%global_quantity.replace('_box',''),
↪fontsize=15)
            axs[jj].yaxis.set_tick_params(labelsize=15)
    else:
        for jj, global_quantity in enumerate(plot_quantities):
            axs[jj].set_ylabel('global_%s'%global_quantity.replace('_box',''),
↪fontsize=0)
            axs[jj].yaxis.set_tick_params(labelsize=0)

plt.tight_layout()
fig.subplots_adjust(hspace = 0.0,wspace=0.0)
```

[86]:



21-cm power spectra

[87]:
```
# define functions to calculate PS, following py21cmmc
from powerbox.tools import get_power

def compute_power(
    box,
    length,
    n_psbins,
    log_bins=True,
    ignore_kperp_zero=True,
```

(continues on next page)

(continued from previous page)

```
    ignore_kpar_zero=False,
    ignore_k_zero=False,
):
    # Determine the weighting function required from ignoring k's.
    k_weights = np.ones(box.shape, dtype=np.int)
    n0 = k_weights.shape[0]
    n1 = k_weights.shape[-1]

    if ignore_kperp_zero:
        k_weights[n0 // 2, n0 // 2, :] = 0
    if ignore_kpar_zero:
        k_weights[:, :, n1 // 2] = 0
    if ignore_k_zero:
        k_weights[n0 // 2, n0 // 2, n1 // 2] = 0

    res = get_power(
        box,
        boxlength=length,
        bins=n_psbins,
        bin_ave=False,
        get_variance=False,
        log_bins=log_bins,
        k_weights=k_weights,
    )

    res = list(res)
    k = res[1]
    if log_bins:
        k = np.exp((np.log(k[1:]) + np.log(k[:-1])) / 2)
    else:
        k = (k[1:] + k[:-1]) / 2

    res[1] = k
    return res

def powerspectra(brightness_temp, n_psbins=50, nchunks=10, min_k=0.1, max_k=1.0,␣
↪logk=True):
    data = []
    chunk_indices = list(range(0,brightness_temp.n_slices,round(brightness_temp.n_
↪slices / nchunks),))

    if len(chunk_indices) > nchunks:
        chunk_indices = chunk_indices[:-1]
    chunk_indices.append(brightness_temp.n_slices)

    for i in range(nchunks):
        start = chunk_indices[i]
        end = chunk_indices[i + 1]
        chunklen = (end - start) * brightness_temp.cell_size

        power, k = compute_power(
            brightness_temp.brightness_temp[:, :, start:end],
            (BOX_LEN, BOX_LEN, chunklen),
            n_psbins,
            log_bins=logk,
        )
        data.append({"k": k, "delta": power * k ** 3 / (2 * np.pi ** 2)})
```

(continues on next page)

(continued from previous page)

```
    return data
```

[96]:
```
# do 5 chunks but only plot 1 - 4, the 0th has no power for minihalo models where xH=0
nchunks = 4

fig, axss = plt.subplots(nchunks, len(labelss), sharex=True,sharey=True,
↪figsize=(4*len(labelss),3*(nchunks)),subplot_kw={"xscale":'log', "yscale":'log'})

for pp, texts in enumerate(textss):
    labels  = labelss[pp]
    factors = factorss[pp]
    axs     = axss[:,pp]
    for kk, label in enumerate(labels):
        flag_options = flag_options_fid.copy()
        astro_params = astro_params_fid.copy()
        factor = factors[kk]
        if label == 'NOmini':
            flag_options.update({'USE_MINI_HALOS': False})
        else:
            flag_options.update({'USE_MINI_HALOS': True})
            if pp == 0:
                astro_params.update({'F_STAR7_MINI': astro_params_fid['F_STAR7_MINI
↪']+np.log10(factor)})
            elif pp == 1:
                astro_params.update({'F_ESC7_MINI': astro_params_fid['F_ESC7_MINI
↪']+np.log10(factor)})
            elif pp == 2:
                astro_params.update({'L_X_MINI': astro_params_fid['L_X_MINI']+np.
↪log10(factor)})
            else:
                if factor > 1: continue # can't do negative F_H2_SHIELD
                astro_params.update({'F_H2_SHIELD': 1. - (1. - astro_params_fid['F_H2_
↪SHIELD']) * factor})
        if label == 'reference':
            lightcone = lightcone_fid
        else:
            lightcone = p21c.run_lightcone(
                redshift = 5.5,
                init_box = initial_conditions,
                flag_options = flag_options,
                astro_params = astro_params,
                global_quantities=global_quantities,
                random_seed = random_seed,
                direc = output_dir
            )

        PS = powerspectra(lightcone)
        for ii in range(nchunks):
            axs[ii].plot(PS[ii+1]['k'], PS[ii+1]['delta'], color=colors[kk],
↪linestyle=linestyles[kk], label = labels[kk],lw=lws[kk])

            if pp == len(textss)-1 and kk == 0:
                axs[ii].text(0.99, 0.01, 'Chunk-%02d'%(ii+1),horizontalalignment=
↪'right',verticalalignment='bottom',
                        transform=axs[ii].transAxes,fontsize = 15)
```

(continues on next page)

(continued from previous page)

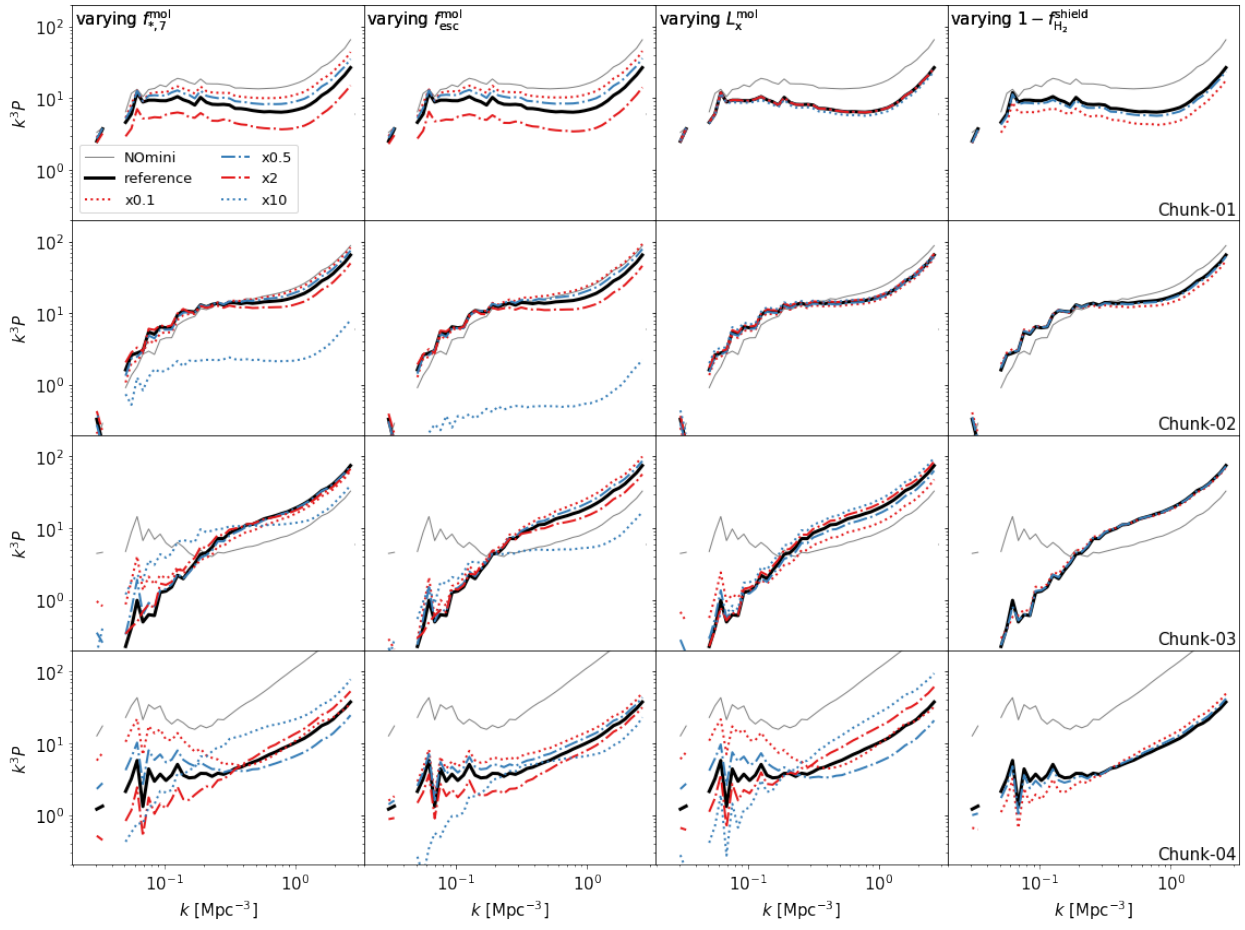
```
    axs[0].text(0.01, 0.99, texts,horizontalalignment='left',verticalalignment='top',
                        transform=axs[0].transAxes,fontsize = 15)

    axs[-1].set_xlabel("$k$ [Mpc$^{-3}$]",fontsize=15)
    axs[-1].xaxis.set_tick_params(labelsize=15)

    if pp == 0:
        for ii in range(nchunks):
            axs[ii].set_ylim(2e-1, 2e2)
            axs[ii].set_ylabel("$k^3 P$", fontsize=15)
            axs[ii].yaxis.set_tick_params(labelsize=15)
    else:
        for ii in range(nchunks-1):
            axs[ii].set_ylim(2e-1, 2e2)
            axs[ii].set_ylabel("$k^3 P$", fontsize=0)
            axs[ii].yaxis.set_tick_params(labelsize=0)

axss[0,0].legend(loc='lower left', ncol=2,fontsize=13,fancybox=True,frameon=True)
plt.tight_layout()
fig.subplots_adjust(hspace = 0.0,wspace=0.0)
```

[96]:



Now you know how minihalo can shape the 21-cm signal!

[ ]:

If you've covered the tutorials and still have questions about "how to do stuff" in `21cmFAST`, consult the FAQs:

4.3.4 Installation FAQ

Errors with "recompile with -fPIC" for FFTW

Make sure you have installed FFTW with `--enable-shared`. On Ubuntu, you will also need to have `libfftw3-dev` installed.

4.3.5 Miscellaneous FAQs

My run seg-faulted, what should I do?

Since `21cmFAST` is written in C, there is the off-chance that something catastrophic will happen, causing a seg-fault. Typically, if this happens, Python will not print a traceback where the error occurred, and finding the source of such errors can be difficult. However, one has the option of using the standard library faulthandler. Specifying `-X faulthandler` when invoking Python will cause a minimal traceback to be printed to `stderr` if a segfault occurs.

Configuring 21cmFAST

`21cmFAST` has a configuration file located at `~/.21cmfast/config.yml`. This file specifies some options to use in a rather global sense, especially to do with I/O. You can directly edit this file to change how `21cmFAST` behaves for you across sessions. For any particular function call, any of the options may be overwritten by supplying arguments to the function itself. To set the configuration for a particular session, you can also set the global *config* instance, for example:

```
>>> import py21cmfast as p21
>>> p21.config['regenerate'] = True
>>> p21.run_lightcone(...)
```

All functions that use the *regenerate* keyword will now use the value you've set in the config. Sometimes, you may want to be a little more careful – perhaps you want to change the configuration for a set of calls, but have it change back to the defaults after that. We provide a context manager to do this:

```
>>> with p21.config.use(regenerate=True):
>>>     p21.run_lightcone()
>>>     print(p21.config['regenerate'])  # prints "True"
>>> print(p21.config['regenerate'])  # prints "False"
```

To make the current configuration permanent, simply use the `write` method:

```
>>> p21.config['direc'] = 'my_own_cache'
>>> p21.config.write()
```

Global Parameters

There are a bunch of "global" parameters that are used throughout the C code. These are parameters that are deemed to be constant enough to not expose them through the regularly-used input structs, but nevertheless may necessitate modification from time-to-time. These are accessed through the `global_params` object:

```
>>> from py21cmfast import global_params
```

Help on the attributes can be obtained via `help(global_params)` or in the docs. Setting the attributes (which affects them everywhere throughout the code) is as simple as, eg:

```
>>> global_params.Z_HEAT_MAX = 30.0
```

If you wish to use a certain parameter for a fixed portion of your code (eg. for a single run), it is encouraged to use the context manager, eg.:

```
>>> with global_params.use(Z_HEAT_MAX=10):
>>>     run_lightcone(...)
```

4.4 API Reference

4.4.1 py21cmfast

py21cmfast.inputs	Input parameter classes.
py21cmfast.outputs	Output class objects.
py21cmfast.wrapper	The main wrapper for the underlying 21cmFAST C-code.
py21cmfast.plotting	Simple plotting functions for 21cmFAST objects.
py21cmfast.cache_tools	A set of tools for reading/writing/querying the in-built cache.

py21cmfast.inputs

Input parameter classes.

There are four input parameter/option classes, not all of which are required for any given function. They are *UserParams*, *CosmoParams*, *AstroParams* and *FlagOptions*. Each of them defines a number of variables, and all of these have default values, to minimize the burden on the user. These defaults are accessed via the `_defaults_` class attribute of each class. The available parameters for each are listed in the documentation for each class below.

Along with these, the module exposes `global_params`, a singleton object of type *GlobalParams*, which is a simple class providing read/write access to a number of parameters used throughout the computation which are very rarely varied.

Classes

AstroParams(*args[, INHOMO_RECO])	Astrophysical parameters.
CosmoParams(*args, **kwargs)	Cosmological parameters (with defaults) which translates to a C struct.
FlagOptions(*args, **kwargs)	Flag-style options for the ionization routines.
GlobalParams(wrapped, ffi)	Global parameters for 21cmFAST.
UserParams(*args, **kwargs)	Structure containing user parameters (with defaults).

py21cmfast.inputs.AstroParams

class py21cmfast.inputs.**AstroParams**(**args*, *INHOMO_RECO=False*, ***kwargs*)

Astrophysical parameters.

To see default values for each parameter, use `AstroParams._defaults_`. All parameters passed in the constructor are also saved as instance attributes which should be considered read-only. This is true of all input-parameter classes.

Parameters

- **INHOMO_RECO** (*bool, optional*) – Whether inhomogeneous recombinations are being calculated. This is not a part of the astro parameters structure, but is required by this class to set some default behaviour.
- **HII_EFF_FACTOR** (*float, optional*) – The ionizing efficiency of high-z galaxies (zeta, from Eq. 2 of Greig+2015). Higher values tend to speed up reionization.
- **F_STAR10** (*float, optional*) – The fraction of galactic gas in stars for 10^10 solar mass haloes. Only used in the "new" parameterization, i.e. when *USE_MASS_DEPENDENT_ZETA* is set to True (in FlagOptions). If so, this is used along with *F_ESC10* to determine *HII_EFF_FACTOR* (which is then unused). See Eq. 11 of Greig+2018 and Sec 2.1 of Park+2018. Given in log10 units.
- **F_STAR7_MINI** (*float, optional*) – The fraction of galactic gas in stars for 10^7 solar mass minihaloes. Only used in the "minihalo" parameterization, i.e. when *USE_MINI_HALOS* is set to True (in FlagOptions). If so, this is used along with *F_ESC7_MINI* to determine *HII_EFF_FACTOR_MINI* (which is then unused). See Eq. 8 of Qin+2020. Given in log10 units.
- **ALPHA_STAR** (*float, optional*) – Power-law index of fraction of galactic gas in stars as a function of halo mass. See Sec 2.1 of Park+2018.
- **F_ESC10** (*float, optional*) – The "escape fraction", i.e. the fraction of ionizing photons escaping into the IGM, for 10^10 solar mass haloes. Only used in the "new" parameterization, i.e. when *USE_MASS_DEPENDENT_ZETA* is set to True (in FlagOptions). If so, this is used along with *F_STAR10* to determine *HII_EFF_FACTOR* (which is then unused). See Eq. 11 of Greig+2018 and Sec 2.1 of Park+2018.
- **F_ESC7_MINI** (*float, optional*) – The "escape fraction for minihalos", i.e. the fraction of ionizing photons escaping into the IGM, for 10^7 solar mass minihaloes. Only used in the "minihalo" parameterization, i.e. when *USE_MINI_HALOS* is set to True (in FlagOptions). If so, this is used along with *F_ESC7_MINI* to determine *HII_EFF_FACTOR_MINI* (which is then unused). See Eq. 17 of Qin+2020. Given in log10 units.
- **ALPHA_ESC** (*float, optional*) – Power-law index of escape fraction as a function of halo mass. See Sec 2.1 of Park+2018.

- **M_TURN** (*float, optional*) – Turnover mass (in log10 solar mass units) for quenching of star formation in halos, due to SNe or photo-heating feedback, or inefficient gas accretion. Only used if *USE_MASS_DEPENDENT_ZETA* is set to True in `FlagOptions`. See Sec 2.1 of Park+2018.
- **R_BUBBLE_MAX** (*float, optional*) – Mean free path in Mpc of ionizing photons within ionizing regions (Sec. 2.1.2 of Greig+2015). Default is 50 if *INHOMO_RECO* is True, or 15.0 if not.
- **ION_Tvir_MIN** (*float, optional*) – Minimum virial temperature of star-forming haloes (Sec 2.1.3 of Greig+2015). Given in log10 units.
- **L_X** (*float, optional*) – The specific X-ray luminosity per unit star formation escaping host galaxies. Cf. Eq. 6 of Greig+2018. Given in log10 units.
- **L_X_MINI** (*float, optional*) – The specific X-ray luminosity per unit star formation escaping host galaxies for minihalos. Cf. Eq. 23 of Qin+2020. Given in log10 units.
- **NU_X_THRESH** (*float, optional*) – X-ray energy threshold for self-absorption by host galaxies (in eV). Also called E_0 (cf. Sec 4.1 of Greig+2018). Typical range is (100, 1500).
- **X_RAY_SPEC_INDEX** (*float, optional*) – X-ray spectral energy index (cf. Sec 4.1 of Greig+2018). Typical range is (-1, 3).
- **X_RAY_Tvir_MIN** (*float, optional*) – Minimum halo virial temperature in which X-rays are produced. Given in log10 units. Default is *ION_Tvir_MIN*.
- **F_H2_SHIELD** (*float, optional*) – Self-shielding factor of molecular hydrogen when experiencing LW suppression. Cf. Eq. 12 of Qin+2020
- **t_STAR** (*float, optional*) – Fractional characteristic time-scale (fraction of hubble time) defining the star-formation rate of galaxies. Only used if *USE_MASS_DEPENDENT_ZETA* is set to True in `FlagOptions`. See Sec 2.1, Eq. 3 of Park+2018.
- **N_RSD_STEPS** (*int, optional*) – Number of steps used in redshift-space-distortion algorithm. NOT A PHYSICAL PARAMETER.

Methods

`__init__`(*args[, INHOMO_RECO])	Initialize self.
`clone`(**kwargs)	Make a fresh copy of the instance with arbitrary parameters updated.
`convert`(key, val)	Convert a given attribute before saving it the instance.
`refresh_cstruct`()	Delete the underlying C object, forcing it to be rebuilt.
`update`(**kwargs)	Update the parameters of an existing class structure.

py21cmfast.inputs.AstroParams.__init__

AstroParams.**__init__**(**args*, *INHOMO_RECO=False*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

py21cmfast.inputs.AstroParams.clone

AstroParams.**clone**(***kwargs*)
Make a fresh copy of the instance with arbitrary parameters updated.

py21cmfast.inputs.AstroParams.convert

AstroParams.**convert**(*key*, *val*)
Convert a given attribute before saving it the instance.

py21cmfast.inputs.AstroParams.refresh_cstruct

AstroParams.**refresh_cstruct**()
Delete the underlying C object, forcing it to be rebuilt.

py21cmfast.inputs.AstroParams.update

AstroParams.**update**(***kwargs*)
Update the parameters of an existing class structure.

This should always be used instead of attempting to *assign* values to instance attributes. It consistently re-generates the underlying C memory space and sets some book-keeping variables.

Parameters **kwargs** – Any argument that may be passed to the class constructor.

Attributes

R_BUBBLE_MAX	Maximum radius of bubbles to be searched.
X_RAY_Tvir_MIN	Minimum virial temperature of X-ray emitting sources (unlogged and set dynamically).
defining_dict	Pure python dictionary representation of this class, as it would appear in C.
fieldnames	List names of fields of the underlying C struct.
fields	List of fields of the underlying C struct (a list of tuples of "name, type").
pointer_fields	List of names of fields which have pointer type in the C struct.
primitive_fields	List of names of fields which have primitive type in the C struct.
pystruct	A pure-python dictionary representation of the corresponding C structure.
self	Dictionary which if passed to its own constructor will yield an identical copy.

py21cmfast.inputs.AstroParams.R_BUBBLE_MAX

property AstroParams.**R_BUBBLE_MAX**
Maximum radius of bubbles to be searched. Set dynamically.

py21cmfast.inputs.AstroParams.X_RAY_Tvir_MIN

property AstroParams.**X_RAY_Tvir_MIN**
Minimum virial temperature of X-ray emitting sources (unlogged and set dynamically).

py21cmfast.inputs.AstroParams.defining_dict

property AstroParams.**defining_dict**
Pure python dictionary representation of this class, as it would appear in C.

Note: This is not the same as *pystruct*, as it omits all variables that don't need to be passed to the constructor, but appear in the C struct (some can be calculated dynamically based on the inputs). It is also not the same as *self*, as it includes the 'converted' values for each variable, which are those actually passed to the C code.

---

py21cmfast.inputs.AstroParams.fieldnames

property AstroParams.**fieldnames**
List names of fields of the underlying C struct.

py21cmfast.inputs.AstroParams.fields

property AstroParams.**fields**
List of fields of the underlying C struct (a list of tuples of "name, type").

py21cmfast.inputs.AstroParams.pointer_fields

property AstroParams.**pointer_fields**
List of names of fields which have pointer type in the C struct.

py21cmfast.inputs.AstroParams.primitive_fields

property AstroParams.**primitive_fields**
List of names of fields which have primitive type in the C struct.

py21cmfast.inputs.AstroParams.pystruct

property AstroParams.**pystruct**
A pure-python dictionary representation of the corresponding C structure.

py21cmfast.inputs.AstroParams.self

property AstroParams.**self**
Dictionary which if passed to its own constructor will yield an identical copy.

Note: This differs from *pystruct* and *defining_dict* in that it uses the hidden variable value, if it exists, instead of the exposed one. This prevents from, for example, passing a value which is 10**10**val (and recurring!).

---

py21cmfast.inputs.CosmoParams

class py21cmfast.inputs.**CosmoParams**(**args*, ***kwargs*)
Cosmological parameters (with defaults) which translates to a C struct.

To see default values for each parameter, use CosmoParams._defaults_. All parameters passed in the constructor are also saved as instance attributes which should be considered read-only. This is true of all input-parameter classes.

Default parameters are based on Plank18, https://arxiv.org/pdf/1807.06209.pdf, Table 2, last column. [TT,TE,EE+lowE+lensing+BAO]

Parameters

- **SIGMA_8** (*float, optional*) – RMS mass variance (power spectrum normalisation).
- **hlittle** (*float, optional*) – The hubble parameter, H_0/100.
- **OMm** (*float, optional*) – Omega matter.
- **OMb** (*float, optional*) – Omega baryon, the baryon component.
- **POWER_INDEX** (*float, optional*) – Spectral index of the power spectrum.

Methods

__init__(*args, **kwargs)	Initialize self.
clone(**kwargs)	Make a fresh copy of the instance with arbitrary parameters updated.
convert(key, val)	Make any conversions of values before saving to the instance.
refresh_cstruct()	Delete the underlying C object, forcing it to be rebuilt.
update(**kwargs)	Update the parameters of an existing class structure.

py21cmfast.inputs.CosmoParams.__init__

CosmoParams.**__init__**(**args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

py21cmfast.inputs.CosmoParams.clone

CosmoParams.**clone**(***kwargs*)
Make a fresh copy of the instance with arbitrary parameters updated.

py21cmfast.inputs.CosmoParams.convert

CosmoParams.**convert**(*key*, *val*)
Make any conversions of values before saving to the instance.

py21cmfast.inputs.CosmoParams.refresh_cstruct

CosmoParams.**refresh_cstruct**()
Delete the underlying C object, forcing it to be rebuilt.

py21cmfast.inputs.CosmoParams.update

CosmoParams.**update**(***kwargs*)
Update the parameters of an existing class structure.

This should always be used instead of attempting to *assign* values to instance attributes. It consistently re-generates the underlying C memory space and sets some book-keeping variables.

Parameters **kwargs** – Any argument that may be passed to the class constructor.

Attributes

OMl	Omega lambda, dark energy density.
cosmo	Return an astropy cosmology object for this cosmology.
defining_dict	Pure python dictionary representation of this class, as it would appear in C.
fieldnames	List names of fields of the underlying C struct.
fields	List of fields of the underlying C struct (a list of tuples of "name, type").
pointer_fields	List of names of fields which have pointer type in the C struct.
primitive_fields	List of names of fields which have primitive type in the C struct.
pystruct	A pure-python dictionary representation of the corresponding C structure.
self	Dictionary which if passed to its own constructor will yield an identical copy.

py21cmfast.inputs.CosmoParams.OMl

property CosmoParams.**OMl**
Omega lambda, dark energy density.

py21cmfast.inputs.CosmoParams.cosmo

property CosmoParams.**cosmo**
Return an astropy cosmology object for this cosmology.

py21cmfast.inputs.CosmoParams.defining_dict

property CosmoParams.**defining_dict**
Pure python dictionary representation of this class, as it would appear in C.

Note: This is not the same as *pystruct*, as it omits all variables that don't need to be passed to the constructor, but appear in the C struct (some can be calculated dynamically based on the inputs). It is also not the same as *self*, as it includes the 'converted' values for each variable, which are those actually passed to the C code.

---

py21cmfast.inputs.CosmoParams.fieldnames

property CosmoParams.**fieldnames**
List names of fields of the underlying C struct.

py21cmfast.inputs.CosmoParams.fields

property CosmoParams.**fields**
List of fields of the underlying C struct (a list of tuples of "name, type").

py21cmfast.inputs.CosmoParams.pointer_fields

property CosmoParams.**pointer_fields**
List of names of fields which have pointer type in the C struct.

py21cmfast.inputs.CosmoParams.primitive_fields

property CosmoParams.**primitive_fields**
List of names of fields which have primitive type in the C struct.

py21cmfast.inputs.CosmoParams.pystruct

property CosmoParams.**pystruct**
A pure-python dictionary representation of the corresponding C structure.

py21cmfast.inputs.CosmoParams.self

property CosmoParams.**self**
Dictionary which if passed to its own constructor will yield an identical copy.

Note: This differs from *pystruct* and *defining_dict* in that it uses the hidden variable value, if it exists, instead of the exposed one. This prevents from, for example, passing a value which is 10**10**val (and recurring!).

---

py21cmfast.inputs.FlagOptions

class py21cmfast.inputs.**FlagOptions**(**args*, ***kwargs*)
Flag-style options for the ionization routines.

To see default values for each parameter, use `FlagOptions._defaults_`. All parameters passed in the constructor are also saved as instance attributes which should be considered read-only. This is true of all input-parameter classes.

Note that all flags are set to False by default, giving the simplest "vanilla" version of 21cmFAST.

Parameters

- **USE_HALO_FIELD** (*bool, optional*) – Set to True if intending to find and use the halo field. If False, uses the mean collapse fraction (which is considerably faster).
- **USE_MINI_HALOS** (*bool, optional*) – Set to True if using mini-halos parameterization. If True, USE_MASS_DEPENDENT_ZETA and INHOMO_RECO must be True.
- **USE_MASS_DEPENDENT_ZETA** (*bool, optional*) – Set to True if using new parameterization. Setting to True will automatically set *M_MIN_in_Mass* to True.
- **SUBCELL_RSDS** (*bool, optional*) – Add sub-cell redshift-space-distortions (cf Sec 2.2 of Greig+2018). Will only be effective if *USE_TS_FLUCT* is True.
- **INHOMO_RECO** (*bool, optional*) – Whether to perform inhomogeneous recombinations. Increases the computation time.
- **USE_TS_FLUCT** (*bool, optional*) – Whether to perform IGM spin temperature fluctuations (i.e. X-ray heating). Dramatically increases the computation time.
- **M_MIN_in_Mass** (*bool, optional*) – Whether the minimum halo mass (for ionization) is defined by mass or virial temperature. Automatically True if *USE_MASS_DEPENDENT_ZETA* is True.
- **PHOTON_CONS** (*bool, optional*) – Whether to perform a small correction to account for the inherent photon non-conservation.

Methods

__init__(*args, **kwargs)	Initialize self.
clone(**kwargs)	Make a fresh copy of the instance with arbitrary parameters updated.
convert(key, val)	Make any conversions of values before saving to the instance.
refresh_cstruct()	Delete the underlying C object, forcing it to be rebuilt.
update(**kwargs)	Update the parameters of an existing class structure.

py21cmfast.inputs.FlagOptions.__init__

FlagOptions.**__init__**(**args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

py21cmfast.inputs.FlagOptions.clone

FlagOptions.**clone**(***kwargs*)
Make a fresh copy of the instance with arbitrary parameters updated.

py21cmfast.inputs.FlagOptions.convert

FlagOptions.**convert**(*key*, *val*)
Make any conversions of values before saving to the instance.

py21cmfast.inputs.FlagOptions.refresh_cstruct

FlagOptions.**refresh_cstruct**()
Delete the underlying C object, forcing it to be rebuilt.

py21cmfast.inputs.FlagOptions.update

FlagOptions.**update**(***kwargs*)
Update the parameters of an existing class structure.

This should always be used instead of attempting to *assign* values to instance attributes. It consistently re-generates the underlying C memory space and sets some book-keeping variables.

Parameters **kwargs** – Any argument that may be passed to the class constructor.

Attributes

INHOMO_RECO	Automatically setting INHOMO_RECO to True if USE_MINI_HALOS.
M_MIN_in_Mass	Whether minimum halo mass is defined in mass or virial temperature.
PHOTON_CONS	Automatically setting PHOTON_CONS to False if USE_MINI_HALOS.
USE_HALO_FIELD	Automatically setting USE_MASS_DEPENDENT_ZETA to False if USE_MINI_HALOS.
USE_MASS_DEPENDENT_ZETA	Automatically setting USE_MASS_DEPENDENT_ZETA to True if USE_MINI_HALOS.
USE_TS_FLUCT	Automatically setting USE_TS_FLUCT to True if USE_MINI_HALOS.
defining_dict	Pure python dictionary representation of this class, as it would appear in C.
fieldnames	List names of fields of the underlying C struct.
fields	List of fields of the underlying C struct (a list of tuples of "name, type").
pointer_fields	List of names of fields which have pointer type in the C struct.
primitive_fields	List of names of fields which have primitive type in the C struct.
pystruct	A pure-python dictionary representation of the corresponding C structure.
self	Dictionary which if passed to its own constructor will yield an identical copy.

py21cmfast.inputs.FlagOptions.INHOMO_RECO

property FlagOptions.**INHOMO_RECO**
Automatically setting INHOMO_RECO to True if USE_MINI_HALOS.

py21cmfast.inputs.FlagOptions.M_MIN_in_Mass

property FlagOptions.**M_MIN_in_Mass**
Whether minimum halo mass is defined in mass or virial temperature.

py21cmfast.inputs.FlagOptions.PHOTON_CONS

property FlagOptions.**PHOTON_CONS**
Automatically setting PHOTON_CONS to False if USE_MINI_HALOS.

py21cmfast.inputs.FlagOptions.USE_HALO_FIELD

property FlagOptions.**USE_HALO_FIELD**
Automatically setting USE_MASS_DEPENDENT_ZETA to False if USE_MINI_HALOS.

py21cmfast.inputs.FlagOptions.USE_MASS_DEPENDENT_ZETA

property FlagOptions.**USE_MASS_DEPENDENT_ZETA**
Automatically setting USE_MASS_DEPENDENT_ZETA to True if USE_MINI_HALOS.

py21cmfast.inputs.FlagOptions.USE_TS_FLUCT

property FlagOptions.**USE_TS_FLUCT**
Automatically setting USE_TS_FLUCT to True if USE_MINI_HALOS.

py21cmfast.inputs.FlagOptions.defining_dict

property FlagOptions.**defining_dict**
Pure python dictionary representation of this class, as it would appear in C.

Note: This is not the same as *pystruct*, as it omits all variables that don't need to be passed to the constructor, but appear in the C struct (some can be calculated dynamically based on the inputs). It is also not the same as *self*, as it includes the 'converted' values for each variable, which are those actually passed to the C code.

---

py21cmfast.inputs.FlagOptions.fieldnames

property FlagOptions.**fieldnames**
List names of fields of the underlying C struct.

py21cmfast.inputs.FlagOptions.fields

property FlagOptions.**fields**
List of fields of the underlying C struct (a list of tuples of "name, type").

py21cmfast.inputs.FlagOptions.pointer_fields

property FlagOptions.**pointer_fields**
List of names of fields which have pointer type in the C struct.

py21cmfast.inputs.FlagOptions.primitive_fields

property FlagOptions.**primitive_fields**
List of names of fields which have primitive type in the C struct.

py21cmfast.inputs.FlagOptions.pystruct

property FlagOptions.**pystruct**
A pure-python dictionary representation of the corresponding C structure.

py21cmfast.inputs.FlagOptions.self

property FlagOptions.**self**
Dictionary which if passed to its own constructor will yield an identical copy.

Note: This differs from *pystruct* and *defining_dict* in that it uses the hidden variable value, if it exists, instead of the exposed one. This prevents from, for example, passing a value which is 10**10**val (and recurring!).

---

py21cmfast.inputs.GlobalParams

class py21cmfast.inputs.**GlobalParams**(*wrapped*, *ffi*)
Global parameters for 21cmFAST.

This is a thin wrapper over an allocated C struct, containing parameter values which are used throughout various computations within 21cmFAST. It is a singleton; that is, a single python (and C) object exists, and no others should be created. This object is not "passed around", rather its values are accessed throughout the code.

Parameters in this struct are considered to be options that should usually not have to be modified, and if so, typically once in any given script or session.

Values can be set in the normal way, eg.:

```
>>> global_params.ALPHA_UVB = 5.5
```

The class also provides a context manager for setting parameters for a well-defined portion of the code. For example, if you would like to set Z_HEAT_MAX for a given run:

```
>>> with global_params.use(Z_HEAT_MAX=25):
>>>     p21c.run_lightcone(...)  # uses Z_HEAT_MAX=25 for the entire run.
>>> print(global_params.Z_HEAT_MAX)
35.0
```

Variables

- **ALPHA_UVB** (*float*) – Power law index of the UVB during the EoR. This is only used if *INHOMO_RECO* is True (in *FlagOptions*), in order to compute the local mean free path inside the cosmic HII regions.
- **EVOLVE_DENSITY_LINEARLY** (*bool*) – Whether to evolve the density field with linear theory (instead of 1LPT or Zel'Dovich). If choosing this option, make sure that your cell size is in the linear regime at the redshift of interest. Otherwise, make sure you resolve small enough scales, roughly we find BOX_LEN/DIM should be < 1Mpc
- **SMOOTH_EVOLVED_DENSITY_FIELD** (*bool*) – If True, the zeldovich-approximation density field is additionally smoothed (aside from the implicit boxcar smoothing performed when re-binning the ICs from DIM to HII_DIM) with a Gaussian filter of width `R_smooth_density*BOX_LEN/HII_DIM`. The implicit boxcar smoothing in `perturb_field()` bins the density field on scale DIM/HII_DIM, similar to what Lagrangian codes do when constructing Eulerian grids. In other words, the density field is quantized into `(DIM/HII_DIM)^3` values. If your usage requires smooth density fields, it is recommended to set this to True. This also decreases the shot noise present in all grid based codes, though it overcompensates by an effective loss in resolution. **Added in 1.1.0**.
- **R_smooth_density** (*float*) – Determines the smoothing length to use if *SMOOTH_EVOLVED_DENSITY_FIELD* is True.
- **SECOND_ORDER_LPT_CORRECTIONS** (*bool*) – Use second-order Lagrangian perturbation theory (2LPT). Set this to True if the density field or the halo positions are extrapolated to low redshifts. The current implementation is very naive and adds a factor ~6 to the memory requirements. Reference: Scoccimarro R., 1998, MNRAS, 299, 1097-1118 Appendix D.
- **HII_ROUND_ERR** (*float*) – Rounding error on the ionization fraction. If the mean xHI is greater than `1 - HII_ROUND_ERR`, then finding HII bubbles is skipped, and a homogeneous xHI field of ones is returned. Added in v1.1.0.
- **FIND_BUBBLE_ALGORITHM**(*int, {1,2}*) – Choose which algorithm used to find HII bubbles. Options are: (1) Mesinger & Furlanetto 2007 method of overlapping spheres: paint an ionized sphere with radius R, centered on pixel where R is filter radius. This method, while somewhat more accurate, is slower than (2), especially in mostly ionized universes, so only use for lower resolution boxes (HII_DIM<~400). (2) Center pixel only method (Zahn et al. 2007). This is faster.
- **N_POISSON** (*int*) – If not using the halo field to generate HII regions, we provide the option of including Poisson scatter in the number of sources obtained through the conditional collapse fraction (which only gives the *mean* collapse fraction on a particular scale. If the predicted mean collapse fraction is less than *N_POISSON * M_MIN*, then Poisson scatter is added to mimic discrete halos on the subgrid scale (see Zahn+2010).Use a negative number to turn it off.

Note: If you are interested in snapshots of the same realization at several redshifts,it is recommended to turn off this feature, as halos can stochastically "pop in and out of" existence from one redshift to the next.

- **R_OVERLAP_FACTOR** (*float*) – When using USE_HALO_FIELD, it is used as a factor the halo's radius, R, so that the effective radius is R_eff = R_OVERLAP_FACTOR * R. Halos whose centers are less than R_eff away from another halo are not allowed. R_OVERLAP_FACTOR = 1 is fully disjoint R_OVERLAP_FACTOR = 0 means that centers are allowed to lay on the edges of neighboring halos.

- **DELTA_CRIT_MODE** (*int*) – The delta_crit to be used for determining whether a halo exists in a cell 0: delta_crit is constant (i.e. 1.686) 1: delta_crit is the sheth tormen ellipsoidal collapse correction to delta_crit
- **HALO_FILTER** (*int*) – Filter for the density field used to generate the halo field with EPS 0: real space top hat filter 1: sharp k-space filter 2: gaussian filter
- **OPTIMIZE** (*bool*) – Finding halos can be made more efficient if the filter size is sufficiently large that we can switch to the collapse fraction at a later stage.
- **OPTIMIZE_MIN_MASS** (*float*) – Minimum mass on which the optimization for the halo finder will be used.
- **T_USE_VELOCITIES** (*bool*) – Whether to use velocity corrections in 21-cm fields

  ---

  Note: The approximation used to include peculiar velocity effects works only in the linear regime, so be careful using this (see Mesinger+2010)

- **MAX_DVDR** (*float*) – Maximum velocity gradient along the line of sight in units of the hubble parameter at z. This is only used in computing the 21cm fields.

  Note: Setting this too high can add spurious 21cm power in the early stages, due to the 1-e^-tau ~ tau approximation (see Mesinger's 21cm intro paper and mao+2011). However, this is still a good approximation at the <~10% level.

- **VELOCITY_COMPONENT** (*int*) – Component of the velocity to be used in 21-cm temperature maps (1=x, 2=y, 3=z)
- **DELTA_R_FACTOR** (*float*) – Factor by which to scroll through filter radius for halos
- **DELTA_R_HII_FACTOR** (*float*) – Factor by which to scroll through filter radius for bubbles
- **HII_FILTER** (*int, {0, 1, 2}*) – Filter for the Halo or density field used to generate ionization field: 0. real space top hat filter 1. k-space top hat filter 2. gaussian filter
- **INITIAL_REDSHIFT** (*float*) – Used to perturb field
- **CRIT_DENS_TRANSITION** (*float*) – A transition value for the interpolation tables for calculating the number of ionising photons produced given the input parameters. Log sampling is desired, however the numerical accuracy near the critical density for collapse (i.e. 1.69) broke down. Therefore, below the value for *CRIT_DENS_TRANSITION* log sampling of the density values is used, whereas above this value linear sampling is used.
- **MIN_DENSITY_LOW_LIMIT** (*float*) – Required for using the interpolation tables for the number of ionising photons. This is a lower limit for the density values that is slightly larger than -1. Defined as a density contrast.
- **RecombPhotonCons** (*int*) – Whether or not to use the recombination term when calculating the filling factor for performing the photon non-conservation correction.
- **PhotonConsStart** (*float*) – A starting value for the neutral fraction where the photon non-conservation correction is performed exactly. Any value larger than this the photon non-conservation correction is not performed (i.e. the algorithm is perfectly photon conserving).
- **PhotonConsEnd** (*float*) – An end-point for where the photon non-conservation correction is performed exactly. This is required to remove undesired numerical artifacts in the resultant neutral fraction histories.

- **PhotonConsAsymptoteTo** (*float*) – Beyond *PhotonConsEnd* the photon non-conservation correction is extrapolated to yield smooth reionisation histories. This sets the lowest neutral fraction value that the photon non-conservation correction will be applied to.
- **HEAT_FILTER** (*int*) – Filter used for smoothing the linear density field to obtain the collapsed fraction: 0: real space top hat filter 1: sharp k-space filter 2: gaussian filter
- **CLUMPING_FACTOR** (*float*) – Sub grid scale. If you want to run-down from a very high redshift (>50), you should set this to one.
- **Z_HEAT_MAX** (*float*) – Maximum redshift used in the Tk and x_e evolution equations. Temperature and x_e are assumed to be homogeneous at higher redshifts. Lower values will increase performance.
- **R_XLy_MAX** (*float*) – Maximum radius of influence for computing X-ray and Lya pumping in cMpc. This should be larger than the mean free path of the relevant photons.
- **NUM_FILTER_STEPS_FOR_Ts** (*int*) – Number of spherical annuli used to compute df_coll/dz' in the simulation box. The spherical annuli are evenly spaced in logR, ranging from the cell size to the box size. `spin_temp()` will create this many boxes of size *HII_DIM*, so be wary of memory usage if values are high.
- **ZPRIME_STEP_FACTOR** (*float*) – Logarithmic redshift step-size used in the z' integral. Logarithmic dz. Decreasing (closer to unity) increases total simulation time for lightcones, and for Ts calculations.
- **TK_at_Z_HEAT_MAX** (*float*) – If positive, then overwrite default boundary conditions for the evolution equations with this value. The default is to use the value obtained from RECFAST. See also *XION_at_Z_HEAT_MAX*.
- **XION_at_Z_HEAT_MAX** (*float*) – If positive, then overwrite default boundary conditions for the evolution equations with this value. The default is to use the value obtained from RECFAST. See also *TK_at_Z_HEAT_MAX*.
- **Pop** (*int*) – Stellar Population responsible for early heating (2 or 3)
- **Pop2_ion** (*float*) – Number of ionizing photons per baryon for population 2 stellar species.
- **Pop3_ion** (*float*) – Number of ionizing photons per baryon for population 3 stellar species.
- **NU_X_BAND_MAX** (*float*) – This is the upper limit of the soft X-ray band (0.5 - 2 keV) used for normalising the X-ray SED to observational limits set by the X-ray luminosity. Used for performing the heating rate integrals.
- **NU_X_MAX** (*float*) – An upper limit (must be set beyond *NU_X_BAND_MAX*) for performing the rate integrals. Given the X-ray SED is modelled as a power-law, this removes the potential of divergent behaviour for the heating rates. Chosen purely for numerical convenience though it is motivated by the fact that observed X-ray SEDs apprear to turn-over around 10-100 keV (Lehmer et al. 2013, 2015)
- **NBINS_LF** (*int*) – Number of bins for the luminosity function calculation.
- **P_CUTOFF** (*bool*) – Turn on Warm-Dark-matter power suppression.
- **M_WDM** (*float*) – Mass of WDM particle in keV. Ignored if *P_CUTOFF* is False.
- **g_x** (*float*) – Degrees of freedom of WDM particles; 1.5 for fermions.
- **OMn** (*float*) – Relative density of neutrinos in the universe.
- **OMk** (*float*) – Relative density of curvature.

- **OMr** (*float*) – Relative density of radiation.
- **OMtot** (*float*) – Fractional density of the universe with respect to critical density. Set to unity for a flat universe.
- **Y_He** (*float*) – Helium fraction.
- **wl** (*float*) – Dark energy equation of state parameter (wl = -1 for vacuum)
- **SHETH_b** (*float*) – Sheth-Tormen parameter for ellipsoidal collapse (for HMF).

Note: The best fit b and c ST params for these 3D realisations have a redshift, and a DELTA_R_FACTOR dependence, as shown in Mesinger+. For converged mass functions at z~5-10, set *DELTA_R_FACTOR=1.1* and *SHETH_b=0.15* and *SHETH_c~0.05*.

For most purposes, a larger step size is quite sufficient and provides an excellent match to N-body and smoother mass functions, though the b and c parameters should be changed to make up for some "stepping-over" massive collapsed halos (see Mesinger, Perna, Haiman (2005) and Mesinger et al., in preparation).

For example, at z~7-10, one can set *DELTA_R_FACTOR=1.3* and *SHETH_b=0.15* and *SHETH_c=0.25*, to increase the speed of the halo finder.

- **SHETH_c** (*float*) – Sheth-Tormen parameter for ellipsoidal collapse (for HMF). See notes for *SHETH_b*.
- **Zreion_HeII** (*float*) – Redshift of helium reionization, currently only used for tau_e
- **FILTER** (*int, {0, 1}*) – Filter to use for smoothing. 0. tophat 1. gaussian
- ***external_table_path*** (*str*) – The system path to find external tables for calculation speedups. DO NOT MODIFY.
- **R_BUBBLE_MIN** (*float*) – Minimum radius of bubbles to be searched in cMpc. One can set this to 0, but should be careful with shot noise if running on a fine, non-linear density grid. Default is set to L_FACTOR which is (4PI/3)^(-1/3) = 0.620350491.
- **M_MIN_INTEGRAL** – Minimum mass when performing integral on halo mass function.
- **M_MAX_INTEGRAL** – Maximum mass when performing integral on halo mass function.
- **T_RE** – The peak gas temperatures behind the supersonic ionization fronts during reionization.

Methods

__init__(wrapped, ffi)	Initialize self.
filtered_repr(filter_params)	Get a fully unique representation of the instance that filters out some parameters.
items()	Yield (name, value) pairs for each element of the struct.
keys()	Return a list of names of elements in the struct.
use(**kwargs)	Set given parameters for a certain context.

py21cmfast.inputs.GlobalParams.__init__

GlobalParams.**__init__**(*wrapped*, *ffi*)
Initialize self. See help(type(self)) for accurate signature.

py21cmfast.inputs.GlobalParams.filtered_repr

GlobalParams.**filtered_repr**(*filter_params*)
Get a fully unique representation of the instance that filters out some parameters.

Parameters **filter_params** (*list of str*) – The parameter names which should not appear in the representation.

py21cmfast.inputs.GlobalParams.items

GlobalParams.**items**()
Yield (name, value) pairs for each element of the struct.

py21cmfast.inputs.GlobalParams.keys

GlobalParams.**keys**()
Return a list of names of elements in the struct.

py21cmfast.inputs.GlobalParams.use

GlobalParams.**use**(***kwargs*)
Set given parameters for a certain context.

Note: Keywords are *not* case-sensitive.

Examples

```
>>> from py21cmfast import global_params, run_lightcone
>>> with global_params.use(zprime_step_factor=1.1, Sheth_c=0.06):
>>>     run_lightcone(redshift=7)
```

Attributes

external_table_path	An ffi char pointer to the path to which external tables are kept.

py21cmfast.inputs.GlobalParams.external_table_path

property GlobalParams.**external_table_path**
An ffi char pointer to the path to which external tables are kept.

py21cmfast.inputs.UserParams

class py21cmfast.inputs.**UserParams**(**args*, ***kwargs*)
Structure containing user parameters (with defaults).

To see default values for each parameter, use `UserParams._defaults_`. All parameters passed in the constructor are also saved as instance attributes which should be considered read-only. This is true of all input-parameter classes.

Parameters

- **HII_DIM** (*int, optional*) – Number of cells for the low-res box. Default 200.
- **DIM** (*int,optional*) – Number of cells for the high-res box (sampling ICs) along a principal axis. To avoid sampling issues, DIM should be at least 3 or 4 times HII_DIM, and an integer multiple. By default, it is set to 3*HII_DIM.
- **BOX_LEN** (*float, optional*) – Length of the box, in Mpc. Default 300 Mpc.
- **HMF** (*int or str, optional*) – Determines which halo mass function to be used for the normalisation of the collapsed fraction (default Sheth-Tormen). If string should be one of the following codes: 0: PS (Press-Schechter) 1: ST (Sheth-Tormen) 2: Watson (Watson FOF) 3: Watson-z (Watson FOF-z)
- **USE_RELATIVE_VELOCITIES** (*int, optional*) – Flag to decide whether to use relative velocities. If True, POWER_SPECTRUM is automatically set to 5. Default False.
- **POWER_SPECTRUM** (*int or str, optional*) – Determines which power spectrum to use, default EH (unless *USE_RELATIVE_VELOCITIES* is True). If string, use the following codes: 0: EH 1: BBKS 2: EFSTATHIOU 3: PEEBLES 4: WHITE 5: CLASS (single cosmology)
- **N_THREADS** (*int, optional*) – Sets the number of processors (threads) to be used for performing 21cmFAST. Default 1.
- **PERTURB_ON_HIGH_RES** (*bool, optional*) – Whether to perform the Zel'Dovich or 2LPT perturbation on the low or high resolution grid.
- **NO_RNG** (*bool, optional*) – Ability to turn off random number generation for initial conditions. Can be useful for debugging and adding in new features
- **USE_FFTW_WISDOM** (*bool, optional*) – Whether or not to use stored FFTW_WISDOMs for improving performance of FFTs
- **USE_INTERPOLATION_TABLES** (*bool, optional*) – If True, calculates and evaluates quantites using interpolation tables, which is considerably faster than when performing integrals explicitly.

Methods

__init__(*args, **kwargs)	Initialize self.
clone(**kwargs)	Make a fresh copy of the instance with arbitrary parameters updated.
convert(key, val)	Make any conversions of values before saving to the instance.
refresh_cstruct()	Delete the underlying C object, forcing it to be rebuilt.
update(**kwargs)	Update the parameters of an existing class structure.

py21cmfast.inputs.UserParams.__init__

UserParams.**__init__**(**args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

py21cmfast.inputs.UserParams.clone

UserParams.**clone**(***kwargs*)
Make a fresh copy of the instance with arbitrary parameters updated.

py21cmfast.inputs.UserParams.convert

UserParams.**convert**(*key*, *val*)
Make any conversions of values before saving to the instance.

py21cmfast.inputs.UserParams.refresh_cstruct

UserParams.**refresh_cstruct**()
Delete the underlying C object, forcing it to be rebuilt.

py21cmfast.inputs.UserParams.update

UserParams.**update**(***kwargs*)
Update the parameters of an existing class structure.

This should always be used instead of attempting to *assign* values to instance attributes. It consistently re-generates the underlying C memory space and sets some book-keeping variables.

Parameters **kwargs** – Any argument that may be passed to the class constructor.

Attributes

DIM	Number of cells for the high-res box (sampling ICs) along a principal axis.
HII_tot_num_pixels	Total number of pixels in the low-res box.
HMF	The HMF to use (an int, mapping to a given form).
POWER_SPECTRUM	The power spectrum generator to use, as an integer.
defining_dict	Pure python dictionary representation of this class, as it would appear in C.
fieldnames	List names of fields of the underlying C struct.
fields	List of fields of the underlying C struct (a list of tuples of "name, type").
hmf_model	String representation of the HMF model used.
pointer_fields	List of names of fields which have pointer type in the C struct.
power_spectrum_model	String representation of the power spectrum model used.
primitive_fields	List of names of fields which have primitive type in the C struct.
pystruct	A pure-python dictionary representation of the corresponding C structure.
self	Dictionary which if passed to its own constructor will yield an identical copy.
tot_fft_num_pixels	Total number of pixels in the high-res box.

py21cmfast.inputs.UserParams.DIM

property UserParams.**DIM**
Number of cells for the high-res box (sampling ICs) along a principal axis.

py21cmfast.inputs.UserParams.HII_tot_num_pixels

property UserParams.**HII_tot_num_pixels**
Total number of pixels in the low-res box.

py21cmfast.inputs.UserParams.HMF

property UserParams.**HMF**
The HMF to use (an int, mapping to a given form).

See hmf_model for a string representation.

py21cmfast.inputs.UserParams.POWER_SPECTRUM

property UserParams.**POWER_SPECTRUM**
The power spectrum generator to use, as an integer.

See *power_spectrum_model()* for a string representation.

py21cmfast.inputs.UserParams.defining_dict

property UserParams.**defining_dict**
Pure python dictionary representation of this class, as it would appear in C.

Note: This is not the same as *pystruct*, as it omits all variables that don't need to be passed to the constructor, but appear in the C struct (some can be calculated dynamically based on the inputs). It is also not the same as *self*, as it includes the 'converted' values for each variable, which are those actually passed to the C code.

---

py21cmfast.inputs.UserParams.fieldnames

property UserParams.**fieldnames**
List names of fields of the underlying C struct.

py21cmfast.inputs.UserParams.fields

property UserParams.**fields**
List of fields of the underlying C struct (a list of tuples of "name, type").

py21cmfast.inputs.UserParams.hmf_model

property UserParams.**hmf_model**
String representation of the HMF model used.

py21cmfast.inputs.UserParams.pointer_fields

property UserParams.**pointer_fields**
List of names of fields which have pointer type in the C struct.

py21cmfast.inputs.UserParams.power_spectrum_model

property UserParams.**power_spectrum_model**
String representation of the power spectrum model used.

py21cmfast.inputs.UserParams.primitive_fields

property UserParams.**primitive_fields**
List of names of fields which have primitive type in the C struct.

py21cmfast.inputs.UserParams.pystruct

property UserParams.**pystruct**
A pure-python dictionary representation of the corresponding C structure.

py21cmfast.inputs.UserParams.self

property UserParams.**self**
Dictionary which if passed to its own constructor will yield an identical copy.

Note: This differs from *pystruct* and *defining_dict* in that it uses the hidden variable value, if it exists, instead of the exposed one. This prevents from, for example, passing a value which is 10**10**val (and recurring!).

---

py21cmfast.inputs.UserParams.tot_fft_num_pixels

property UserParams.**tot_fft_num_pixels**
Total number of pixels in the high-res box.

py21cmfast.outputs

Output class objects.

The classes provided by this module exist to simplify access to large datasets created within C. Fundamentally, ownership of the data belongs to these classes, and the C functions merely accesses this and fills it. The various boxes and lightcones associated with each output are available as instance attributes. Along with the output data, each output object contains the various input parameter objects necessary to define it.

Warning: These should not be instantiated or filled by the user, but always handled as output objects from the various functions contained here. Only the data within the objects should be accessed.

Classes

BrightnessTemp([astro_params, flag_options, ...])	A class containing the brightness temperature box.
Coeval(redshift, initial_conditions, ...[, ...])	A full coeval box with all associated data.
HaloField([astro_params, flag_options, ...])	A class containing all fields related to halos.
InitialConditions(*[, user_params, cosmo_params])	A class containing all initial conditions boxes.
IonizedBox([astro_params, flag_options, ...])	A class containing all ionized boxes.
LightCone(redshift, user_params, ...[, ...])	A full Lightcone with all associated evolved data.

continues on next page

Table 13 – continued from previous page

PerturbHaloField([astro_params, ...])	A class containing all fields related to halos.
PerturbedField(*[, user_params, cosmo_params])	A class containing all perturbed field boxes.
TsBox([astro_params, flag_options, first_box])	A class containing all spin temperature boxes.

py21cmfast.outputs.BrightnessTemp

class py21cmfast.outputs.**BrightnessTemp**(*astro_params=None*, *flag_options=None*, *first_box=False*, ***kwargs*)

A class containing the brightness temperature box.

Methods

__init__([astro_params, flag_options, first_box])	Initialize self.
compute(direc, *args[, write])	Compute the actual function that fills this struct.
exists([direc])	Return a bool indicating whether a box matching the parameters of this instance is in cache.
find_existing([direc])	Try to find existing boxes which match the parameters of this instance.
from_file(fname[, direc, load_data])	Create an instance from a file on disk.
read([direc, fname])	Try find and read existing boxes from cache, which match the parameters of this instance.
refresh_cstruct()	Delete the underlying C object, forcing it to be rebuilt.
save([fname, direc])	Save the box to disk.
write([direc, fname, write_inputs, mode])	Write the struct in standard HDF5 format.

py21cmfast.outputs.BrightnessTemp.__init__

BrightnessTemp.**__init__**(*astro_params=None*, *flag_options=None*, *first_box=False*, ***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

py21cmfast.outputs.BrightnessTemp.compute

BrightnessTemp.**compute**(*direc*, **args*, *write=True*)

Compute the actual function that fills this struct.

py21cmfast.outputs.BrightnessTemp.exists

BrightnessTemp.**exists**(*direc=None*)
Return a bool indicating whether a box matching the parameters of this instance is in cache.

Parameters **direc** (*str, optional*) – The directory in which to search for the boxes. By default, this is the centrally-managed directory, given by the `config.yml` in `~/.21cmfast/`.

py21cmfast.outputs.BrightnessTemp.find_existing

BrightnessTemp.**find_existing**(*direc=None*)
Try to find existing boxes which match the parameters of this instance.

Parameters **direc** (*str, optional*) – The directory in which to search for the boxes. By default, this is the centrally-managed directory, given by the `config.yml` in `~/.21cmfast/`.

Returns *str* – The filename of an existing set of boxes, or None.

py21cmfast.outputs.BrightnessTemp.from_file

classmethod BrightnessTemp.**from_file**(*fname*, *direc=None*, *load_data=True*)
Create an instance from a file on disk.

Parameters

- **fname** (*str, optional*) – Path to the file on disk. May be relative or absolute.
- **direc** (*str, optional*) – The directory from which fname is relative to (if it is relative). By default, will be the cache directory in config.
- **load_data** (*bool, optional*) – Whether to read in the data when creating the instance. If False, a bare instance is created with input parameters – the instance can read data with the *read()* method.

py21cmfast.outputs.BrightnessTemp.read

BrightnessTemp.**read**(*direc=None*, *fname=None*)
Try find and read existing boxes from cache, which match the parameters of this instance.

Parameters **direc** (*str, optional*) – The directory in which to search for the boxes. By default, this is the centrally-managed directory, given by the `config.yml` in `~/.21cmfast/`.

py21cmfast.outputs.BrightnessTemp.refresh_cstruct

BrightnessTemp.**refresh_cstruct**()
Delete the underlying C object, forcing it to be rebuilt.

py21cmfast.outputs.BrightnessTemp.save

`BrightnessTemp.`**`save`**(*fname=None*, *direc='.'*)

Save the box to disk.

In detail, this just calls write, but changes the default directory to the local directory. This is more user-friendly, while *write()* is for automatic use under-the-hood.

Parameters

- **fname** (*str, optional*) – The filename to write. Can be an absolute or relative path. If relative, by default it is relative to the current directory (otherwise relative to `direc`). By default, the filename is auto-generated as unique to the set of parameters that go into producing the data.
- **direc** (*str, optional*) – The directory into which to write the data. By default the current directory. Ignored if `fname` is an absolute path.

py21cmfast.outputs.BrightnessTemp.write

`BrightnessTemp.`**`write`**(*direc=None*, *fname=None*, *write_inputs=True*, *mode='w'*)

Write the struct in standard HDF5 format.

Parameters

- **direc** (*str, optional*) – The directory in which to write the boxes. By default, this is the centrally-managed directory, given by the `config.yml` in `~/.21cmfast/`.
- **fname** (*str, optional*) – The filename to write to. By default creates a unique filename from the hash.
- **write_inputs** (*bool, optional*) – Whether to write the inputs to the file. Can be useful to set to False if the input file already exists and has parts already written.

Attributes

arrays_initialized	Whether all necessary arrays are initialized.
fieldnames	List names of fields of the underlying C struct.
fields	List of fields of the underlying C struct (a list of tuples of "name, type").
filename	The base filename of this object.
global_Tb	
pointer_fields	List of names of fields which have pointer type in the C struct.
primitive_fields	List of names of fields which have primitive type in the C struct.
random_seed	The random seed for this particular instance.

py21cmfast.outputs.BrightnessTemp.arrays_initialized

property BrightnessTemp.**arrays_initialized**
Whether all necessary arrays are initialized.

Note: This must be true before passing to a C function.

py21cmfast.outputs.BrightnessTemp.fieldnames

property BrightnessTemp.**fieldnames**
List names of fields of the underlying C struct.

py21cmfast.outputs.BrightnessTemp.fields

property BrightnessTemp.**fields**
List of fields of the underlying C struct (a list of tuples of "name, type").

py21cmfast.outputs.BrightnessTemp.filename

property BrightnessTemp.**filename**
The base filename of this object.

py21cmfast.outputs.BrightnessTemp.global_Tb

BrightnessTemp.**global_Tb = <MagicMock name='mock()' id='140323935381968'>**

py21cmfast.outputs.BrightnessTemp.pointer_fields

property BrightnessTemp.**pointer_fields**
List of names of fields which have pointer type in the C struct.

py21cmfast.outputs.BrightnessTemp.primitive_fields

property BrightnessTemp.**primitive_fields**
List of names of fields which have primitive type in the C struct.

py21cmfast.outputs.BrightnessTemp.random_seed

property BrightnessTemp.**random_seed**
The random seed for this particular instance.

py21cmfast.outputs.Coeval

class py21cmfast.outputs.**Coeval**(*redshift: float, initial_conditions: py21cmfast.outputs.InitialConditions, perturbed_field: py21cmfast.outputs.PerturbedField, ionized_box: py21cmfast.outputs.IonizedBox, brightness_temp: py21cmfast.outputs.BrightnessTemp, ts_box: [<class 'py21cmfast.outputs.TsBox'>, None] = None, photon_nonconservation_data=None, _globals=None*)
A full coeval box with all associated data.

Methods

__init__(redshift, initial_conditions, ...)	Initialize self.
get_unique_filename()	Generate a unique hash filename for this instance.
read(fname[, direc])	Read a lightcone file from disk, creating a LightCone object.
save([fname, direc])	Save to disk.

py21cmfast.outputs.Coeval.__init__

Coeval.**__init__**(*redshift: float, initial_conditions: py21cmfast.outputs.InitialConditions, perturbed_field: py21cmfast.outputs.PerturbedField, ionized_box: py21cmfast.outputs.IonizedBox, brightness_temp: py21cmfast.outputs.BrightnessTemp, ts_box: [<class 'py21cmfast.outputs.TsBox'>, None] = None, photon_nonconservation_data=None, _globals=None*)
Initialize self. See help(type(self)) for accurate signature.

py21cmfast.outputs.Coeval.get_unique_filename

Coeval.**get_unique_filename**()
Generate a unique hash filename for this instance.

py21cmfast.outputs.Coeval.read

classmethod Coeval.**read**(*fname*, *direc='.'*)
Read a lightcone file from disk, creating a LightCone object.

Parameters

- **fname** (*str*) – The filename path. Can be absolute or relative.
- **direc** (*str*) – If fname, is relative, the directory in which to find the file. By default, both the current directory and default cache and the will be searched, in that order.

Returns *LightCone* – A *LightCone* instance created from the file's data.

py21cmfast.outputs.Coeval.save

Coeval.**save**(*fname=None*, *direc='.'*)
Save to disk.

This function has defaults that make it easy to save a unique box to the current directory.

Parameters

- **fname** (*str, optional*) – The filename to write, default a unique name produced by the inputs.
- **direc** (*str, optional*) – The directory into which to write the file. Default is the current directory.

Returns *str* – The filename to which the box was written.

Attributes

astro_params	Astro params shared by all datasets.
cosmo_params	Cosmo params shared by all datasets.
flag_options	Flag Options shared by all datasets.
random_seed	Random seed shared by all datasets.
user_params	User params shared by all datasets.

py21cmfast.outputs.Coeval.astro_params

property Coeval.**astro_params**
Astro params shared by all datasets.

py21cmfast.outputs.Coeval.cosmo_params

property Coeval.**cosmo_params**
Cosmo params shared by all datasets.

py21cmfast.outputs.Coeval.flag_options

property Coeval.**flag_options**
Flag Options shared by all datasets.

py21cmfast.outputs.Coeval.random_seed

property Coeval.**random_seed**
Random seed shared by all datasets.

py21cmfast.outputs.Coeval.user_params

property Coeval.**user_params**
User params shared by all datasets.

py21cmfast.outputs.HaloField

class py21cmfast.outputs.**HaloField**(*astro_params=None*, *flag_options=None*, *first_box=False*, ***kwargs*)
A class containing all fields related to halos.

Methods

__init__([astro_params, flag_options, first_box])	Initialize self.
compute(direc, *args[, write])	Compute the actual function that fills this struct.
exists([direc])	Return a bool indicating whether a box matching the parameters of this instance is in cache.
find_existing([direc])	Try to find existing boxes which match the parameters of this instance.
from_file(fname[, direc, load_data])	Create an instance from a file on disk.
read([direc, fname])	Try find and read existing boxes from cache, which match the parameters of this instance.
refresh_cstruct()	Delete the underlying C object, forcing it to be rebuilt.
save([fname, direc])	Save the box to disk.
write([direc, fname, write_inputs, mode])	Write the struct in standard HDF5 format.

py21cmfast.outputs.HaloField.__init__

HaloField.**__init__**(*astro_params=None*, *flag_options=None*, *first_box=False*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

py21cmfast.outputs.HaloField.compute

HaloField.**compute**(*direc*, **args*, *write=True*)
Compute the actual function that fills this struct.

py21cmfast.outputs.HaloField.exists

HaloField.**exists**(*direc=None*)
Return a bool indicating whether a box matching the parameters of this instance is in cache.

Parameters **direc** (*str, optional*) – The directory in which to search for the boxes. By default, this is the centrally-managed directory, given by the `config.yml` in `~/.21cmfast/`.

py21cmfast.outputs.HaloField.find_existing

HaloField.**find_existing**(*direc=None*)
Try to find existing boxes which match the parameters of this instance.

Parameters **direc** (*str, optional*) – The directory in which to search for the boxes. By default, this is the centrally-managed directory, given by the `config.yml` in `~/.21cmfast/`.

Returns *str* – The filename of an existing set of boxes, or None.

py21cmfast.outputs.HaloField.from_file

classmethod HaloField.**from_file**(*fname*, *direc=None*, *load_data=True*)
Create an instance from a file on disk.

Parameters

- **fname** (*str, optional*) – Path to the file on disk. May be relative or absolute.
- **direc** (*str, optional*) – The directory from which fname is relative to (if it is relative). By default, will be the cache directory in config.
- **load_data** (*bool, optional*) – Whether to read in the data when creating the instance. If False, a bare instance is created with input parameters – the instance can read data with the *read()* method.

py21cmfast.outputs.HaloField.read

HaloField.**read**(*direc=None*, *fname=None*)
Try find and read existing boxes from cache, which match the parameters of this instance.

Parameters **direc** (*str, optional*) – The directory in which to search for the boxes. By default, this is the centrally-managed directory, given by the `config.yml` in `~/.21cmfast/`.

py21cmfast.outputs.HaloField.refresh_cstruct

HaloField.**refresh_cstruct**()
Delete the underlying C object, forcing it to be rebuilt.

py21cmfast.outputs.HaloField.save

HaloField.**save**(*fname=None*, *direc='.'*)
Save the box to disk.

In detail, this just calls write, but changes the default directory to the local directory. This is more user-friendly, while *write()* is for automatic use under-the-hood.

Parameters

- **fname** (*str, optional*) – The filename to write. Can be an absolute or relative path. If relative, by default it is relative to the current directory (otherwise relative to `direc`). By default, the filename is auto-generated as unique to the set of parameters that go into producing the data.
- **direc** (*str, optional*) – The directory into which to write the data. By default the current directory. Ignored if `fname` is an absolute path.

py21cmfast.outputs.HaloField.write

HaloField.**write**(*direc=None*, *fname=None*, *write_inputs=True*, *mode='w'*)
Write the struct in standard HDF5 format.

Parameters

- **direc** (*str, optional*) – The directory in which to write the boxes. By default, this is the centrally-managed directory, given by the `config.yml` in `~/.21cmfast/`.
- **fname** (*str, optional*) – The filename to write to. By default creates a unique filename from the hash.
- **write_inputs** (*bool, optional*) – Whether to write the inputs to the file. Can be useful to set to False if the input file already exists and has parts already written.

Attributes

arrays_initialized	Whether all necessary arrays are initialized.
fieldnames	List names of fields of the underlying C struct.
fields	List of fields of the underlying C struct (a list of tuples of "name, type").
filename	The base filename of this object.
pointer_fields	List of names of fields which have pointer type in the C struct.
primitive_fields	List of names of fields which have primitive type in the C struct.
random_seed	The random seed for this particular instance.

py21cmfast.outputs.HaloField.arrays_initialized

property HaloField.**arrays_initialized**
Whether all necessary arrays are initialized.

Note: This must be true before passing to a C function.

py21cmfast.outputs.HaloField.fieldnames

property HaloField.**fieldnames**
List names of fields of the underlying C struct.

py21cmfast.outputs.HaloField.fields

property HaloField.**fields**
List of fields of the underlying C struct (a list of tuples of "name, type").

py21cmfast.outputs.HaloField.filename

property HaloField.**filename**
The base filename of this object.

py21cmfast.outputs.HaloField.pointer_fields

property HaloField.**pointer_fields**
List of names of fields which have pointer type in the C struct.

py21cmfast.outputs.HaloField.primitive_fields

property HaloField.**primitive_fields**
List of names of fields which have primitive type in the C struct.

py21cmfast.outputs.HaloField.random_seed

property HaloField.**random_seed**
The random seed for this particular instance.

py21cmfast.outputs.InitialConditions

class py21cmfast.outputs.**InitialConditions**(*, *user_params=None*, *cosmo_params=None*, ***kwargs*)
A class containing all initial conditions boxes.

Methods

__init__(*[, user_params, cosmo_params])	Initialize self.
compute(direc, *args[, write])	Compute the actual function that fills this struct.
exists([direc])	Return a bool indicating whether a box matching the parameters of this instance is in cache.
find_existing([direc])	Try to find existing boxes which match the parameters of this instance.
from_file(fname[, direc, load_data])	Create an instance from a file on disk.
read([direc, fname])	Try find and read existing boxes from cache, which match the parameters of this instance.
refresh_cstruct()	Delete the underlying C object, forcing it to be rebuilt.
save([fname, direc])	Save the box to disk.
write([direc, fname, write_inputs, mode])	Write the struct in standard HDF5 format.

py21cmfast.outputs.InitialConditions.__init__

InitialConditions.**__init__**(*, *user_params=None*, *cosmo_params=None*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

py21cmfast.outputs.InitialConditions.compute

InitialConditions.**compute**(*direc*, **args*, *write=True*)
Compute the actual function that fills this struct.

py21cmfast.outputs.InitialConditions.exists

InitialConditions.**exists**(*direc=None*)
Return a bool indicating whether a box matching the parameters of this instance is in cache.

Parameters **direc** (*str, optional*) – The directory in which to search for the boxes. By default, this is the centrally-managed directory, given by the `config.yml` in `~/.21cmfast/`.

py21cmfast.outputs.InitialConditions.find_existing

InitialConditions.**find_existing**(*direc=None*)
Try to find existing boxes which match the parameters of this instance.

Parameters **direc** (*str, optional*) – The directory in which to search for the boxes. By default, this is the centrally-managed directory, given by the `config.yml` in `~/.21cmfast/`.

Returns *str* – The filename of an existing set of boxes, or None.

py21cmfast.outputs.InitialConditions.from_file

classmethod InitialConditions.**from_file**(*fname*, *direc=None*, *load_data=True*)
Create an instance from a file on disk.

Parameters

- **fname** (*str, optional*) – Path to the file on disk. May be relative or absolute.
- **direc** (*str, optional*) – The directory from which fname is relative to (if it is relative). By default, will be the cache directory in config.
- **load_data** (*bool, optional*) – Whether to read in the data when creating the instance. If False, a bare instance is created with input parameters – the instance can read data with the *read()* method.

py21cmfast.outputs.InitialConditions.read

InitialConditions.**read**(*direc=None*, *fname=None*)
Try find and read existing boxes from cache, which match the parameters of this instance.

Parameters **direc** (*str, optional*) – The directory in which to search for the boxes. By default, this is the centrally-managed directory, given by the `config.yml` in `~/.21cmfast/`.

py21cmfast.outputs.InitialConditions.refresh_cstruct

InitialConditions.**refresh_cstruct**()
Delete the underlying C object, forcing it to be rebuilt.

py21cmfast.outputs.InitialConditions.save

InitialConditions.**save**(*fname=None*, *direc='.'*)
Save the box to disk.

In detail, this just calls write, but changes the default directory to the local directory. This is more user-friendly, while *write()* is for automatic use under-the-hood.

Parameters

- **fname** (*str, optional*) – The filename to write. Can be an absolute or relative path. If relative, by default it is relative to the current directory (otherwise relative to direc). By default, the filename is auto-generated as unique to the set of parameters that go into producing the data.
- **direc** (*str, optional*) – The directory into which to write the data. By default the current directory. Ignored if fname is an absolute path.

py21cmfast.outputs.InitialConditions.write

InitialConditions.**write**(*direc=None*, *fname=None*, *write_inputs=True*, *mode='w'*)
Write the struct in standard HDF5 format.

Parameters

- **direc** (*str, optional*) – The directory in which to write the boxes. By default, this is the centrally-managed directory, given by the config.yml in ~/.21cmfast/.
- **fname** (*str, optional*) – The filename to write to. By default creates a unique filename from the hash.
- **write_inputs** (*bool, optional*) – Whether to write the inputs to the file. Can be useful to set to False if the input file already exists and has parts already written.

Attributes

arrays_initialized	Whether all necessary arrays are initialized.
fieldnames	List names of fields of the underlying C struct.
fields	List of fields of the underlying C struct (a list of tuples of "name, type").
filename	The base filename of this object.
pointer_fields	List of names of fields which have pointer type in the C struct.
primitive_fields	List of names of fields which have primitive type in the C struct.
random_seed	The random seed for this particular instance.

py21cmfast.outputs.InitialConditions.arrays_initialized

property InitialConditions.**arrays_initialized**
Whether all necessary arrays are initialized.

Note: This must be true before passing to a C function.

py21cmfast.outputs.InitialConditions.fieldnames

property InitialConditions.**fieldnames**
List names of fields of the underlying C struct.

py21cmfast.outputs.InitialConditions.fields

property InitialConditions.**fields**
List of fields of the underlying C struct (a list of tuples of "name, type").

py21cmfast.outputs.InitialConditions.filename

property InitialConditions.**filename**
The base filename of this object.

py21cmfast.outputs.InitialConditions.pointer_fields

property InitialConditions.**pointer_fields**
List of names of fields which have pointer type in the C struct.

py21cmfast.outputs.InitialConditions.primitive_fields

property InitialConditions.**primitive_fields**
List of names of fields which have primitive type in the C struct.

py21cmfast.outputs.InitialConditions.random_seed

property InitialConditions.**random_seed**
The random seed for this particular instance.

py21cmfast.outputs.IonizedBox

class py21cmfast.outputs.**IonizedBox**(*astro_params=None*, *flag_options=None*, *first_box=False*, ***kwargs*)

A class containing all ionized boxes.

Methods

__init__([astro_params, flag_options, first_box])	Initialize self.
compute(direc, *args[, write])	Compute the actual function that fills this struct.
exists([direc])	Return a bool indicating whether a box matching the parameters of this instance is in cache.
find_existing([direc])	Try to find existing boxes which match the parameters of this instance.
from_file(fname[, direc, load_data])	Create an instance from a file on disk.
read([direc, fname])	Try find and read existing boxes from cache, which match the parameters of this instance.
refresh_cstruct()	Delete the underlying C object, forcing it to be rebuilt.
save([fname, direc])	Save the box to disk.
write([direc, fname, write_inputs, mode])	Write the struct in standard HDF5 format.

py21cmfast.outputs.IonizedBox.__init__

IonizedBox.**__init__**(*astro_params=None*, *flag_options=None*, *first_box=False*, ***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

py21cmfast.outputs.IonizedBox.compute

IonizedBox.**compute**(*direc*, **args*, *write=True*)

Compute the actual function that fills this struct.

py21cmfast.outputs.IonizedBox.exists

IonizedBox.**exists**(*direc=None*)
Return a bool indicating whether a box matching the parameters of this instance is in cache.

Parameters **direc** (*str, optional*) – The directory in which to search for the boxes. By default, this is the centrally-managed directory, given by the `config.yml` in `~/.21cmfast/`.

py21cmfast.outputs.IonizedBox.find_existing

IonizedBox.**find_existing**(*direc=None*)
Try to find existing boxes which match the parameters of this instance.

Parameters **direc** (*str, optional*) – The directory in which to search for the boxes. By default, this is the centrally-managed directory, given by the `config.yml` in `~/.21cmfast/`.

Returns *str* – The filename of an existing set of boxes, or None.

py21cmfast.outputs.IonizedBox.from_file

classmethod IonizedBox.**from_file**(*fname*, *direc=None*, *load_data=True*)
Create an instance from a file on disk.

Parameters

- **fname** (*str, optional*) – Path to the file on disk. May be relative or absolute.
- **direc** (*str, optional*) – The directory from which fname is relative to (if it is relative). By default, will be the cache directory in config.
- **load_data** (*bool, optional*) – Whether to read in the data when creating the instance. If False, a bare instance is created with input parameters – the instance can read data with the *read()* method.

py21cmfast.outputs.IonizedBox.read

IonizedBox.**read**(*direc=None*, *fname=None*)
Try find and read existing boxes from cache, which match the parameters of this instance.

Parameters **direc** (*str, optional*) – The directory in which to search for the boxes. By default, this is the centrally-managed directory, given by the `config.yml` in `~/.21cmfast/`.

py21cmfast.outputs.IonizedBox.refresh_cstruct

IonizedBox.**refresh_cstruct**()
Delete the underlying C object, forcing it to be rebuilt.

py21cmfast.outputs.IonizedBox.save

IonizedBox.**save**(*fname=None*, *direc='.'*)

Save the box to disk.

In detail, this just calls write, but changes the default directory to the local directory. This is more user-friendly, while *write()* is for automatic use under-the-hood.

Parameters

- **fname** (*str, optional*) – The filename to write. Can be an absolute or relative path. If relative, by default it is relative to the current directory (otherwise relative to `direc`). By default, the filename is auto-generated as unique to the set of parameters that go into producing the data.
- **direc** (*str, optional*) – The directory into which to write the data. By default the current directory. Ignored if `fname` is an absolute path.

py21cmfast.outputs.IonizedBox.write

IonizedBox.**write**(*direc=None*, *fname=None*, *write_inputs=True*, *mode='w'*)

Write the struct in standard HDF5 format.

Parameters

- **direc** (*str, optional*) – The directory in which to write the boxes. By default, this is the centrally-managed directory, given by the `config.yml` in `~/.21cmfast/`.
- **fname** (*str, optional*) – The filename to write to. By default creates a unique filename from the hash.
- **write_inputs** (*bool, optional*) – Whether to write the inputs to the file. Can be useful to set to False if the input file already exists and has parts already written.

Attributes

arrays_initialized	Whether all necessary arrays are initialized.
fieldnames	List names of fields of the underlying C struct.
fields	List of fields of the underlying C struct (a list of tuples of "name, type").
filename	The base filename of this object.
global_xH	
pointer_fields	List of names of fields which have pointer type in the C struct.
primitive_fields	List of names of fields which have primitive type in the C struct.
random_seed	The random seed for this particular instance.

py21cmfast.outputs.IonizedBox.arrays_initialized

property IonizedBox.**arrays_initialized**
Whether all necessary arrays are initialized.

Note: This must be true before passing to a C function.

py21cmfast.outputs.IonizedBox.fieldnames

property IonizedBox.**fieldnames**
List names of fields of the underlying C struct.

py21cmfast.outputs.IonizedBox.fields

property IonizedBox.**fields**
List of fields of the underlying C struct (a list of tuples of "name, type").

py21cmfast.outputs.IonizedBox.filename

property IonizedBox.**filename**
The base filename of this object.

py21cmfast.outputs.IonizedBox.global_xH

IonizedBox.**global_xH = <MagicMock name='mock()' id='140323935381968'>**

py21cmfast.outputs.IonizedBox.pointer_fields

property IonizedBox.**pointer_fields**
List of names of fields which have pointer type in the C struct.

py21cmfast.outputs.IonizedBox.primitive_fields

property IonizedBox.**primitive_fields**
List of names of fields which have primitive type in the C struct.

py21cmfast.outputs.IonizedBox.random_seed

property IonizedBox.**random_seed**
The random seed for this particular instance.

py21cmfast.outputs.LightCone

class py21cmfast.outputs.**LightCone**(*redshift*, *user_params*, *cosmo_params*, *astro_params*, *flag_options*, *random_seed*, *lightcones*, *node_redshifts=None*, *global_quantities=None*, *photon_nonconservation_data=None*, *_globals=None*)
A full Lightcone with all associated evolved data.

Methods

__init__(redshift, user_params, ..., [, ...])	Initialize self.
get_unique_filename()	Generate a unique hash filename for this instance.
read(fname[, direc])	Read a lightcone file from disk, creating a LightCone object.
save([fname, direc])	Save to disk.

py21cmfast.outputs.LightCone.__init__

LightCone.**__init__**(*redshift*, *user_params*, *cosmo_params*, *astro_params*, *flag_options*, *random_seed*, *lightcones*, *node_redshifts=None*, *global_quantities=None*, *photon_nonconservation_data=None*, *_globals=None*)
Initialize self. See help(type(self)) for accurate signature.

py21cmfast.outputs.LightCone.get_unique_filename

LightCone.**get_unique_filename**()
Generate a unique hash filename for this instance.

py21cmfast.outputs.LightCone.read

classmethod LightCone.**read**(*fname*, *direc='.'*)
Read a lightcone file from disk, creating a LightCone object.

Parameters

- **fname** (*str*) – The filename path. Can be absolute or relative.
- **direc** (*str*) – If fname, is relative, the directory in which to find the file. By default, both the current directory and default cache and the will be searched, in that order.

Returns *LightCone* – A *LightCone* instance created from the file's data.

py21cmfast.outputs.LightCone.save

LightCone.**save**(*fname=None*, *direc='.'*)
Save to disk.

This function has defaults that make it easy to save a unique box to the current directory.

Parameters

- **fname** (*str, optional*) – The filename to write, default a unique name produced by the inputs.
- **direc** (*str, optional*) – The directory into which to write the file. Default is the current directory.

Returns *str* – The filename to which the box was written.

Attributes

cell_size	Cell size [Mpc] of the lightcone voxels.
global_xHI	Global neutral fraction function.
lightcone_coords	Co-ordinates [Mpc] of each cell along the redshift axis.
lightcone_dimensions	Lightcone size over each dimension – tuple of (x,y,z) in Mpc.
lightcone_distances	Comoving distance to each cell along the redshift axis, from z=0.
lightcone_redshifts	Redshift of each cell along the redshift axis.
n_slices	Number of redshift slices in the lightcone.
shape	Shape of the lightcone as a 3-tuple.

py21cmfast.outputs.LightCone.cell_size

property LightCone.**cell_size**
Cell size [Mpc] of the lightcone voxels.

py21cmfast.outputs.LightCone.global_xHI

property LightCone.**global_xHI**
Global neutral fraction function.

py21cmfast.outputs.LightCone.lightcone_coords

property LightCone.**lightcone_coords**
Co-ordinates [Mpc] of each cell along the redshift axis.

py21cmfast.outputs.LightCone.lightcone_dimensions

property LightCone.**lightcone_dimensions**
Lightcone size over each dimension – tuple of (x,y,z) in Mpc.

py21cmfast.outputs.LightCone.lightcone_distances

property LightCone.**lightcone_distances**
Comoving distance to each cell along the redshift axis, from z=0.

py21cmfast.outputs.LightCone.lightcone_redshifts

property LightCone.**lightcone_redshifts**
Redshift of each cell along the redshift axis.

py21cmfast.outputs.LightCone.n_slices

property LightCone.**n_slices**
Number of redshift slices in the lightcone.

py21cmfast.outputs.LightCone.shape

property LightCone.**shape**
Shape of the lightcone as a 3-tuple.

py21cmfast.outputs.PerturbHaloField

class py21cmfast.outputs.**PerturbHaloField**(*astro_params=None*, *flag_options=None*, *first_box=False*, ***kwargs*)
A class containing all fields related to halos.

Methods

__init__([astro_params, flag_options, first_box])	Initialize self.
compute(direc, *args[, write])	Compute the actual function that fills this struct.
exists([direc])	Return a bool indicating whether a box matching the parameters of this instance is in cache.
find_existing([direc])	Try to find existing boxes which match the parameters of this instance.
from_file(fname[, direc, load_data])	Create an instance from a file on disk.
read([direc, fname])	Try find and read existing boxes from cache, which match the parameters of this instance.
refresh_cstruct()	Delete the underlying C object, forcing it to be rebuilt.
save([fname, direc])	Save the box to disk.
write([direc, fname, write_inputs, mode])	Write the struct in standard HDF5 format.

py21cmfast.outputs.PerturbHaloField.__init__

PerturbHaloField.**__init__**(*astro_params=None*, *flag_options=None*, *first_box=False*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

py21cmfast.outputs.PerturbHaloField.compute

PerturbHaloField.**compute**(*direc*, **args*, *write=True*)
Compute the actual function that fills this struct.

py21cmfast.outputs.PerturbHaloField.exists

PerturbHaloField.**exists**(*direc=None*)
Return a bool indicating whether a box matching the parameters of this instance is in cache.

Parameters **direc** (*str, optional*) – The directory in which to search for the boxes. By default, this is the centrally-managed directory, given by the `config.yml` in `~/.21cmfast/`.

py21cmfast.outputs.PerturbHaloField.find_existing

PerturbHaloField.**find_existing**(*direc=None*)
Try to find existing boxes which match the parameters of this instance.

Parameters **direc** (*str, optional*) – The directory in which to search for the boxes. By default, this is the centrally-managed directory, given by the `config.yml` in `~/.21cmfast/`.

Returns *str* – The filename of an existing set of boxes, or None.

py21cmfast.outputs.PerturbHaloField.from_file

classmethod PerturbHaloField.**from_file**(*fname*, *direc=None*, *load_data=True*)
Create an instance from a file on disk.

Parameters

- **fname** (*str, optional*) – Path to the file on disk. May be relative or absolute.
- **direc** (*str, optional*) – The directory from which fname is relative to (if it is relative). By default, will be the cache directory in config.
- **load_data** (*bool, optional*) – Whether to read in the data when creating the instance. If False, a bare instance is created with input parameters – the instance can read data with the *read()* method.

py21cmfast.outputs.PerturbHaloField.read

PerturbHaloField.**read**(*direc=None*, *fname=None*)

Try find and read existing boxes from cache, which match the parameters of this instance.

Parameters **direc** (*str, optional*) – The directory in which to search for the boxes. By default, this is the centrally-managed directory, given by the `config.yml` in `~/.21cmfast/`.

py21cmfast.outputs.PerturbHaloField.refresh_cstruct

PerturbHaloField.**refresh_cstruct**()

Delete the underlying C object, forcing it to be rebuilt.

py21cmfast.outputs.PerturbHaloField.save

PerturbHaloField.**save**(*fname=None*, *direc='.'*)

Save the box to disk.

In detail, this just calls write, but changes the default directory to the local directory. This is more user-friendly, while *write()* is for automatic use under-the-hood.

Parameters

- **fname** (*str, optional*) – The filename to write. Can be an absolute or relative path. If relative, by default it is relative to the current directory (otherwise relative to `direc`). By default, the filename is auto-generated as unique to the set of parameters that go into producing the data.
- **direc** (*str, optional*) – The directory into which to write the data. By default the current directory. Ignored if `fname` is an absolute path.

py21cmfast.outputs.PerturbHaloField.write

PerturbHaloField.**write**(*direc=None*, *fname=None*, *write_inputs=True*, *mode='w'*)

Write the struct in standard HDF5 format.

Parameters

- **direc** (*str, optional*) – The directory in which to write the boxes. By default, this is the centrally-managed directory, given by the `config.yml` in `~/.21cmfast/`.
- **fname** (*str, optional*) – The filename to write to. By default creates a unique filename from the hash.
- **write_inputs** (*bool, optional*) – Whether to write the inputs to the file. Can be useful to set to False if the input file already exists and has parts already written.

Attributes

arrays_initialized	Whether all necessary arrays are initialized.
fieldnames	List names of fields of the underlying C struct.
fields	List of fields of the underlying C struct (a list of tuples of "name, type").
filename	The base filename of this object.
pointer_fields	List of names of fields which have pointer type in the C struct.
primitive_fields	List of names of fields which have primitive type in the C struct.
random_seed	The random seed for this particular instance.

py21cmfast.outputs.PerturbHaloField.arrays_initialized

property PerturbHaloField.**arrays_initialized**
Whether all necessary arrays are initialized.

Note: This must be true before passing to a C function.

py21cmfast.outputs.PerturbHaloField.fieldnames

property PerturbHaloField.**fieldnames**
List names of fields of the underlying C struct.

py21cmfast.outputs.PerturbHaloField.fields

property PerturbHaloField.**fields**
List of fields of the underlying C struct (a list of tuples of "name, type").

py21cmfast.outputs.PerturbHaloField.filename

property PerturbHaloField.**filename**
The base filename of this object.

py21cmfast.outputs.PerturbHaloField.pointer_fields

property PerturbHaloField.**pointer_fields**
List of names of fields which have pointer type in the C struct.

py21cmfast.outputs.PerturbHaloField.primitive_fields

property PerturbHaloField.**primitive_fields**
List of names of fields which have primitive type in the C struct.

py21cmfast.outputs.PerturbHaloField.random_seed

property PerturbHaloField.**random_seed**
The random seed for this particular instance.

py21cmfast.outputs.PerturbedField

class py21cmfast.outputs.**PerturbedField**(*, *user_params=None*, *cosmo_params=None*, ***kwargs*)
A class containing all perturbed field boxes.

Methods

__init__(*[, user_params, cosmo_params])	Initialize self.
compute(direc, *args[, write])	Compute the actual function that fills this struct.
exists([direc])	Return a bool indicating whether a box matching the parameters of this instance is in cache.
find_existing([direc])	Try to find existing boxes which match the parameters of this instance.
from_file(fname[, direc, load_data])	Create an instance from a file on disk.
read([direc, fname])	Try find and read existing boxes from cache, which match the parameters of this instance.
refresh_cstruct()	Delete the underlying C object, forcing it to be rebuilt.
save([fname, direc])	Save the box to disk.
write([direc, fname, write_inputs, mode])	Write the struct in standard HDF5 format.

py21cmfast.outputs.PerturbedField.__init__

PerturbedField.**__init__**(*, *user_params=None*, *cosmo_params=None*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

py21cmfast.outputs.PerturbedField.compute

PerturbedField.**compute**(*direc*, **args*, *write=True*)
Compute the actual function that fills this struct.

py21cmfast.outputs.PerturbedField.exists

PerturbedField.**exists**(*direc=None*)
Return a bool indicating whether a box matching the parameters of this instance is in cache.

Parameters **direc** (*str, optional*) – The directory in which to search for the boxes. By default, this is the centrally-managed directory, given by the `config.yml` in `~/.21cmfast/`.

py21cmfast.outputs.PerturbedField.find_existing

PerturbedField.**find_existing**(*direc=None*)
Try to find existing boxes which match the parameters of this instance.

Parameters **direc** (*str, optional*) – The directory in which to search for the boxes. By default, this is the centrally-managed directory, given by the `config.yml` in `~/.21cmfast/`.

Returns *str* – The filename of an existing set of boxes, or None.

py21cmfast.outputs.PerturbedField.from_file

classmethod PerturbedField.**from_file**(*fname*, *direc=None*, *load_data=True*)
Create an instance from a file on disk.

Parameters

- **fname** (*str, optional*) – Path to the file on disk. May be relative or absolute.
- **direc** (*str, optional*) – The directory from which fname is relative to (if it is relative). By default, will be the cache directory in config.
- **load_data** (*bool, optional*) – Whether to read in the data when creating the instance. If False, a bare instance is created with input parameters – the instance can read data with the *read()* method.

py21cmfast.outputs.PerturbedField.read

PerturbedField.**read**(*direc=None*, *fname=None*)
Try find and read existing boxes from cache, which match the parameters of this instance.

Parameters **direc** (*str, optional*) – The directory in which to search for the boxes. By default, this is the centrally-managed directory, given by the `config.yml` in `~/.21cmfast/`.

py21cmfast.outputs.PerturbedField.refresh_cstruct

`PerturbedField.`**`refresh_cstruct`**`()`
Delete the underlying C object, forcing it to be rebuilt.

py21cmfast.outputs.PerturbedField.save

`PerturbedField.`**`save`**(*fname=None*, *direc='.'*)
Save the box to disk.

In detail, this just calls write, but changes the default directory to the local directory. This is more user-friendly, while *write()* is for automatic use under-the-hood.

Parameters

- **fname** (*str, optional*) – The filename to write. Can be an absolute or relative path. If relative, by default it is relative to the current directory (otherwise relative to `direc`). By default, the filename is auto-generated as unique to the set of parameters that go into producing the data.
- **direc** (*str, optional*) – The directory into which to write the data. By default the current directory. Ignored if `fname` is an absolute path.

py21cmfast.outputs.PerturbedField.write

`PerturbedField.`**`write`**(*direc=None*, *fname=None*, *write_inputs=True*, *mode='w'*)
Write the struct in standard HDF5 format.

Parameters

- **direc** (*str, optional*) – The directory in which to write the boxes. By default, this is the centrally-managed directory, given by the `config.yml` in `~/.21cmfast/`.
- **fname** (*str, optional*) – The filename to write to. By default creates a unique filename from the hash.
- **write_inputs** (*bool, optional*) – Whether to write the inputs to the file. Can be useful to set to False if the input file already exists and has parts already written.

Attributes

arrays_initialized	Whether all necessary arrays are initialized.
fieldnames	List names of fields of the underlying C struct.
fields	List of fields of the underlying C struct (a list of tuples of "name, type").
filename	The base filename of this object.
pointer_fields	List of names of fields which have pointer type in the C struct.
primitive_fields	List of names of fields which have primitive type in the C struct.
random_seed	The random seed for this particular instance.

py21cmfast.outputs.PerturbedField.arrays_initialized

property PerturbedField.**arrays_initialized**
Whether all necessary arrays are initialized.

Note: This must be true before passing to a C function.

py21cmfast.outputs.PerturbedField.fieldnames

property PerturbedField.**fieldnames**
List names of fields of the underlying C struct.

py21cmfast.outputs.PerturbedField.fields

property PerturbedField.**fields**
List of fields of the underlying C struct (a list of tuples of "name, type").

py21cmfast.outputs.PerturbedField.filename

property PerturbedField.**filename**
The base filename of this object.

py21cmfast.outputs.PerturbedField.pointer_fields

property PerturbedField.**pointer_fields**
List of names of fields which have pointer type in the C struct.

py21cmfast.outputs.PerturbedField.primitive_fields

property PerturbedField.**primitive_fields**
List of names of fields which have primitive type in the C struct.

py21cmfast.outputs.PerturbedField.random_seed

property PerturbedField.**random_seed**
The random seed for this particular instance.

py21cmfast.outputs.TsBox

class py21cmfast.outputs.**TsBox**(*astro_params=None*, *flag_options=None*, *first_box=False*, ***kwargs*)

A class containing all spin temperature boxes.

Methods

__init__([astro_params, flag_options, first_box])	Initialize self.
compute(direc, *args[, write])	Compute the actual function that fills this struct.
exists([direc])	Return a bool indicating whether a box matching the parameters of this instance is in cache.
find_existing([direc])	Try to find existing boxes which match the parameters of this instance.
from_file(fname[, direc, load_data])	Create an instance from a file on disk.
read([direc, fname])	Try find and read existing boxes from cache, which match the parameters of this instance.
refresh_cstruct()	Delete the underlying C object, forcing it to be rebuilt.
save([fname, direc])	Save the box to disk.
write([direc, fname, write_inputs, mode])	Write the struct in standard HDF5 format.

py21cmfast.outputs.TsBox.__init__

TsBox.**__init__**(*astro_params=None*, *flag_options=None*, *first_box=False*, ***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

py21cmfast.outputs.TsBox.compute

TsBox.**compute**(*direc*, **args*, *write=True*)

Compute the actual function that fills this struct.

py21cmfast.outputs.TsBox.exists

TsBox.**exists**(*direc=None*)
Return a bool indicating whether a box matching the parameters of this instance is in cache.

Parameters **direc** (*str, optional*) – The directory in which to search for the boxes. By default, this is the centrally-managed directory, given by the `config.yml` in `~/.21cmfast/`.

py21cmfast.outputs.TsBox.find_existing

TsBox.**find_existing**(*direc=None*)
Try to find existing boxes which match the parameters of this instance.

Parameters **direc** (*str, optional*) – The directory in which to search for the boxes. By default, this is the centrally-managed directory, given by the `config.yml` in `~/.21cmfast/`.

Returns *str* – The filename of an existing set of boxes, or None.

py21cmfast.outputs.TsBox.from_file

classmethod TsBox.**from_file**(*fname*, *direc=None*, *load_data=True*)
Create an instance from a file on disk.

Parameters

- **fname** (*str, optional*) – Path to the file on disk. May be relative or absolute.
- **direc** (*str, optional*) – The directory from which fname is relative to (if it is relative). By default, will be the cache directory in config.
- **load_data** (*bool, optional*) – Whether to read in the data when creating the instance. If False, a bare instance is created with input parameters – the instance can read data with the *read()* method.

py21cmfast.outputs.TsBox.read

TsBox.**read**(*direc=None*, *fname=None*)
Try find and read existing boxes from cache, which match the parameters of this instance.

Parameters **direc** (*str, optional*) – The directory in which to search for the boxes. By default, this is the centrally-managed directory, given by the `config.yml` in `~/.21cmfast/`.

py21cmfast.outputs.TsBox.refresh_cstruct

TsBox.**refresh_cstruct**()
Delete the underlying C object, forcing it to be rebuilt.

py21cmfast.outputs.TsBox.save

TsBox.**save**(*fname=None*, *direc='.'*)

Save the box to disk.

In detail, this just calls write, but changes the default directory to the local directory. This is more user-friendly, while *write()* is for automatic use under-the-hood.

Parameters

- **fname** (*str, optional*) – The filename to write. Can be an absolute or relative path. If relative, by default it is relative to the current directory (otherwise relative to `direc`). By default, the filename is auto-generated as unique to the set of parameters that go into producing the data.
- **direc** (*str, optional*) – The directory into which to write the data. By default the current directory. Ignored if `fname` is an absolute path.

py21cmfast.outputs.TsBox.write

TsBox.**write**(*direc=None*, *fname=None*, *write_inputs=True*, *mode='w'*)

Write the struct in standard HDF5 format.

Parameters

- **direc** (*str, optional*) – The directory in which to write the boxes. By default, this is the centrally-managed directory, given by the `config.yml` in `~/.21cmfast/`.
- **fname** (*str, optional*) – The filename to write to. By default creates a unique filename from the hash.
- **write_inputs** (*bool, optional*) – Whether to write the inputs to the file. Can be useful to set to False if the input file already exists and has parts already written.

Attributes

arrays_initialized	Whether all necessary arrays are initialized.
fieldnames	List names of fields of the underlying C struct.
fields	List of fields of the underlying C struct (a list of tuples of "name, type").
filename	The base filename of this object.
global_Tk	
global_Ts	
global_x_e	
pointer_fields	List of names of fields which have pointer type in the C struct.
primitive_fields	List of names of fields which have primitive type in the C struct.
random_seed	The random seed for this particular instance.

py21cmfast.outputs.TsBox.arrays_initialized

property TsBox.**arrays_initialized**
Whether all necessary arrays are initialized.

Note: This must be true before passing to a C function.

py21cmfast.outputs.TsBox.fieldnames

property TsBox.**fieldnames**
List names of fields of the underlying C struct.

py21cmfast.outputs.TsBox.fields

property TsBox.**fields**
List of fields of the underlying C struct (a list of tuples of "name, type").

py21cmfast.outputs.TsBox.filename

property TsBox.**filename**
The base filename of this object.

py21cmfast.outputs.TsBox.global_Tk

TsBox.**global_Tk = <MagicMock name='mock()' id='140323935381968'>**

py21cmfast.outputs.TsBox.global_Ts

TsBox.**global_Ts = <MagicMock name='mock()' id='140323935381968'>**

py21cmfast.outputs.TsBox.global_x_e

TsBox.**global_x_e = <MagicMock name='mock()' id='140323935381968'>**

py21cmfast.outputs.TsBox.pointer_fields

property TsBox.**pointer_fields**
List of names of fields which have pointer type in the C struct.

py21cmfast.outputs.TsBox.primitive_fields

property TsBox.**primitive_fields**
List of names of fields which have primitive type in the C struct.

py21cmfast.outputs.TsBox.random_seed

property TsBox.**random_seed**
The random seed for this particular instance.

py21cmfast.wrapper

The main wrapper for the underlying 21cmFAST C-code.

The module provides both low- and high-level wrappers, using the very low-level machinery in `_utils`, and the convenient input and output structures from *inputs* and *outputs*.

This module provides a number of:

- Low-level functions which simplify calling the background C functions which populate these output objects given the input classes.
- High-level functions which provide the most efficient and simplest way to generate the most commonly desired outputs.

Low-level functions

The low-level functions provided here ease the production of the aforementioned output objects. Functions exist for each low-level C routine, which have been decoupled as far as possible. So, functions exist to create *initial_conditions()*, *perturb_field()*, *ionize_box* and so on. Creating a brightness temperature box (often the desired final output) would generally require calling each of these in turn, as each depends on the result of a previous function. Nevertheless, each function has the capability of generating the required previous outputs on-the-fly, so one can instantly call *ionize_box()* and get a self-consistent result. Doing so, while convenient, is sometimes not *efficient*, especially when using inhomogeneous recombinations or the spin temperature field, which intrinsically require consistent evolution of the ionization field through redshift. In these cases, for best efficiency it is recommended to either use a customised manual approach to calling these low-level functions, or to call a higher-level function which optimizes this process.

Finally, note that `py21cmfast` attempts to optimize the production of the large amount of data via on-disk caching. By default, if a previous set of data has been computed using the current input parameters, it will be read-in from a caching repository and returned directly. This behaviour can be tuned in any of the low-level (or high-level) functions by setting the *write*, *direc*, *regenerate* and *match_seed* parameters (see docs for *initial_conditions()* for details). The function `query_cache()` can be used to search the cache, and return empty datasets corresponding to each (and these can then be filled with the data merely by calling `.read()` on any data set). Conversely, a specific data set can be read and returned as a proper output object by calling the *readbox()* function.

High-level functions

As previously mentioned, calling the low-level functions in some cases is non-optimal, especially when full evolution of the field is required, and thus iteration through a series of redshift. In addition, while `InitialConditions` and `PerturbedField` are necessary intermediate data, it is *usually* the resulting brightness temperature which is of most interest, and it is easier to not have to worry about the intermediate steps explicitly. For these typical use-cases, two high-level functions are available: *run_coeval()* and *run_lightcone()*, whose purpose should be self-explanatory. These will optimally run all necessary intermediate steps (using cached results by default if possible) and return all datasets of interest.

Examples

A typical example of using this module would be the following.

```
>>> import py21cmfast as p21
```

Get coeval cubes at redshift 7,8 and 9, without spin temperature or inhomogeneous recombinations:

```
>>> coeval = p21.run_coeval(
>>>     redshift=[7,8,9],
>>>     cosmo_params=p21.CosmoParams(hlittle=0.7),
>>>     user_params=p21.UserParams(HII_DIM=100)
>>> )
```

Get coeval cubes at the same redshift, with both spin temperature and inhomogeneous recombinations, pulled from the natural evolution of the fields:

```
>>> all_boxes = p21.run_coeval(
>>>                 redshift=[7,8,9],
>>>                 user_params=p21.UserParams(HII_DIM=100),
>>>                 flag_options=p21.FlagOptions(INHOMO_RECO=True),
>>>                 do_spin_temp=True
>>>             )
```

Get a self-consistent lightcone defined between z1 and z2 (*z_step_factor* changes the logarithmic steps between redshift that are actually evaluated, which are then interpolated onto the lightcone cells):

```
>>> lightcone = p21.run_lightcone(redshift=z2, max_redshift=z2, z_step_factor=1.03)
```

Functions

brightness_temperature(*, ionized_box, ...)	Compute a coeval brightness temperature box.
calibrate_photon_cons(user_params, ...)	Set up the photon non-conservation correction.
compute_luminosity_function(*, redshifts[, ...])	Compute a the luminosity function over a given number of bins and redshifts.
compute_tau(*, redshifts, global_xHI[, ...])	Compute the optical depth to reionization under the given model.
configure_redshift(redshift, *structs)	Check and obtain a redshift from given default and structs.
construct_fftw_wisdoms(*[, user_params, ...])	Construct all necessary FFTW wisdoms.
determine_halo_list(*, redshift[, ...])	Find a halo list, given a redshift.
get_all_fieldnames([arrays_only, ...])	Return all possible fieldnames in output structs.
initial_conditions(*[, user_params, ...])	Compute initial conditions.
ionize_box(*[, astro_params, flag_options, ...])	Compute an ionized box at a given redshift.
perturb_field(*, redshift[, init_boxes, ...])	Compute a perturbed field at a given redshift.
perturb_halo_list(*, redshift[, init_boxes, ...])	Given a halo list, perturb the halos for a given redshift.
run_coeval(*[, redshift, user_params, ...])	Evaluate a coeval ionized box at a given redshift, or multiple redshifts.
run_lightcone(*[, redshift, max_redshift, ...])	Evaluate a full lightcone ending at a given redshift.
spin_temperature(*[, astro_params, ...])	Compute spin temperature boxes at a given redshift.

py21cmfast.wrapper.brightness_temperature

py21cmfast.wrapper.**brightness_temperature**(*, *ionized_box*, *perturbed_field*, *spin_temp=None*, *write=None*, *regenerate=None*, *direc=None*, ***global_kwargs*)

Compute a coeval brightness temperature box.

Parameters

- **ionized_box** (`IonizedBox`) – A pre-computed ionized box.
- **perturbed_field** (`PerturbedField`) – A pre-computed perturbed field at the same redshift as *ionized_box*.
- **spin_temp** (`TsBox`, optional) – A pre-computed spin temperature, at the same redshift as the other boxes.
- ****global_kwargs** – Any attributes for *GlobalParams*. This will *temporarily* set global attributes for the duration of the function. Note that arguments will be treated as case-insensitive.

Returns `BrightnessTemp` instance.

py21cmfast.wrapper.calibrate_photon_cons

py21cmfast.wrapper.**calibrate_photon_cons**(*user_params*, *cosmo_params*, *astro_params*, *flag_options*, *init_box*, *regenerate*, *write*, *direc*, ***global_kwargs*)

Set up the photon non-conservation correction.

Scrolls through in redshift, turning off all flag_options to construct a 21cmFAST calibration reionisation history to be matched to the analytic expression from solving the filling factor ODE.

Parameters

- **user_params** (*~UserParams*, optional) – Defines the overall options and parameters of the run.
- **astro_params** (`AstroParams`, optional) – Defines the astrophysical parameters of the run.
- **cosmo_params** (`CosmoParams`, optional) – Defines the cosmological parameters used to compute initial conditions.
- **flag_options** (`FlagOptions`, optional) – Options concerning how the reionization process is run, eg. if spin temperature fluctuations are required.
- **init_box** (`InitialConditions`, optional) – If given, the user and cosmo params will be set from this object, and it will not be re-calculated.
- ****global_kwargs** – Any attributes for *GlobalParams*. This will *temporarily* set global attributes for the duration of the function. Note that arguments will be treated as case-insensitive.

Other Parameters **regenerate, write** – See docs of *initial_conditions()* for more information.

py21cmfast.wrapper.compute_luminosity_function

py21cmfast.wrapper.**compute_luminosity_function**(*, *redshifts*, *user_params=None*, *cosmo_params=None*, *astro_params=None*, *flag_options=None*, *nbins=100*, *mturnovers=None*, *mturnovers_mini=None*, *component=0*)

Compute a the luminosity function over a given number of bins and redshifts.

Parameters

- **redshifts** (*array-like*) – The redshifts at which to compute the luminosity function.
- **user_params** (`UserParams`, optional) – Defines the overall options and parameters of the run.
- **cosmo_params** (`CosmoParams`, optional) – Defines the cosmological parameters used to compute initial conditions.
- **astro_params** (`AstroParams`, optional) – The astrophysical parameters defining the course of reionization.
- **flag_options** (`FlagOptions`, optional) – Some options passed to the reionization routine.
- **nbins** (*int, optional*) – The number of luminosity bins to produce for the luminosity function.
- **mturnovers** (*array-like, optional*) – The turnover mass at each redshift for massive halos (ACGs). Only required when USE_MINI_HALOS is True.
- **mturnovers_mini** (*array-like, optional*) – The turnover mass at each redshift for minihalos (MCGs). Only required when USE_MINI_HALOS is True.
- **component** (*int, optional*) – The component of the LF to be calculated. 0, 1 an 2 are for the total, ACG and MCG LFs respectively, requiring inputs of both mturnovers and mturnovers_MINI (0), only mturnovers (1) or mturnovers_MINI (2).

Returns

- **Muvfunc** (*np.ndarray*) – Magnitude array (i.e. brightness). Shape [nredshifts, nbins]
- **Mhfunc** (*np.ndarray*) – Halo mass array. Shape [nredshifts, nbins]
- **lfunc** (*np.ndarray*) – Number density of haloes corresponding to each bin defined by *Muvfunc*. Shape [nredshifts, nbins].

py21cmfast.wrapper.compute_tau

py21cmfast.wrapper.**compute_tau**(*, *redshifts*, *global_xHI*, *user_params=None*, *cosmo_params=None*)

Compute the optical depth to reionization under the given model.

Parameters

- **redshifts** (*array-like*) – Redshifts defining an evolution of the neutral fraction.
- **global_xHI** (*array-like*) – The mean neutral fraction at *redshifts*.
- **user_params** (`UserParams`) – Parameters defining the simulation run.
- **cosmo_params** (`CosmoParams`) – Cosmological parameters.

Returns **tau** (*float*) – The optional depth to reionization

Raises `ValueError` : – If *redshifts* and *global_xHI* have inconsistent length.

py21cmfast.wrapper.configure_redshift

py21cmfast.wrapper.**configure_redshift**(*redshift*, **structs*)

Check and obtain a redshift from given default and structs.

Parameters

- **redshift** (*float*) – The default redshift to use
- **structs** (list of `OutputStruct`) – A number of output datasets from which to find the redshift.

Raises `ValueError` : – If both *redshift* and *all* structs have a value of *None*, **or** if any of them are different from each other (and not *None*).

py21cmfast.wrapper.construct_fftw_wisdoms

py21cmfast.wrapper.**construct_fftw_wisdoms**(*, *user_params=None*, *cosmo_params=None*)

Construct all necessary FFTW wisdoms.

Parameters **user_params** (`UserParams`) – Parameters defining the simulation run.

py21cmfast.wrapper.determine_halo_list

py21cmfast.wrapper.**determine_halo_list**(*, *redshift*, *init_boxes=None*, *user_params=None*, *cosmo_params=None*, *astro_params=None*, *flag_options=None*, *random_seed=None*, *regenerate=None*, *write=None*, *direc=None*, ***global_kwargs*)

Find a halo list, given a redshift.

Parameters

- **redshift** (*float*) – The redshift at which to determine the halo list.
- **init_boxes** (`InitialConditions`, optional) – If given, these initial conditions boxes will be used, otherwise initial conditions will be generated. If given, the user and cosmo params will be set from this object.
- **user_params** (`UserParams`, optional) – Defines the overall options and parameters of the run.
- **cosmo_params** (`CosmoParams`, optional) – Defines the cosmological parameters used to compute initial conditions.
- **astro_params** (`AstroParams` instance, optional) – The astrophysical parameters defining the course of reionization.
- ****global_kwargs** – Any attributes for *GlobalParams*. This will *temporarily* set global attributes for the duration of the function. Note that arguments will be treated as case-insensitive.

Returns `HaloField`

Other Parameters **regenerate, write, direc, random_seed** – See docs of *initial_conditions()* for more information.

Examples

Fill this in once finalised

py21cmfast.wrapper.get_all_fieldnames

py21cmfast.wrapper.**get_all_fieldnames**(*arrays_only=True*, *lightcone_only=False*, *as_dict=False*)

Return all possible fieldnames in output structs.

Parameters

- **arrays_only** (*bool, optional*) – Whether to only return fields that are arrays.
- **lightcone_only** (*bool, optional*) – Whether to only return fields from classes that evolve with redshift.
- **as_dict** (*bool, optional*) – Whether to return results as a dictionary of `quantity: class_name`. Otherwise returns a set of quantities.

py21cmfast.wrapper.initial_conditions

py21cmfast.wrapper.**initial_conditions**(*, *user_params=None*, *cosmo_params=None*, *random_seed=None*, *regenerate=None*, *write=None*, *direc=None*, ***global_kwargs*)

Compute initial conditions.

Parameters

- **user_params** (`UserParams` instance, optional) – Defines the overall options and parameters of the run.
- **cosmo_params** (`CosmoParams` instance, optional) – Defines the cosmological parameters used to compute initial conditions.
- **regenerate** (*bool, optional*) – Whether to force regeneration of data, even if matching cached data is found. This is applied recursively to any potential sub-calculations. It is ignored in the case of dependent data only if that data is explicitly passed to the function.
- **write** (*bool, optional*) – Whether to write results to file (i.e. cache). This is recursively applied to any potential sub-calculations.
- **direc** (*str, optional*) – The directory in which to search for the boxes and write them. By default, this is the directory given by `boxdir` in the configuration file, `~/.21cmfast/config.yml`. This is recursively applied to any potential sub-calculations.
- ****global_kwargs** – Any attributes for *GlobalParams*. This will *temporarily* set global attributes for the duration of the function. Note that arguments will be treated as case-insensitive.

Returns `InitialConditions`

py21cmfast.wrapper.ionize_box

py21cmfast.wrapper.**ionize_box**(*, *astro_params=None*, *flag_options=None*, *redshift=None*, *perturbed_field=None*, *previous_perturbed_field=None*, *previous_ionize_box=None*, *spin_temp=None*, *pt_halos=None*, *init_boxes=None*, *cosmo_params=None*, *user_params=None*, *regenerate=None*, *write=None*, *direc=None*, *random_seed=None*, *cleanup=True*, ***global_kwargs*)

Compute an ionized box at a given redshift.

This function has various options for how the evolution of the ionization is computed (if at all). See the Notes below for details.

Parameters

- **astro_params** (`AstroParams` instance, optional) – The astrophysical parameters defining the course of reionization.
- **flag_options** (`FlagOptions` instance, optional) – Some options passed to the reionization routine.
- **redshift** (*float, optional*) – The redshift at which to compute the ionized box. If *perturbed_field* is given, its inherent redshift will take precedence over this argument. If not, this argument is mandatory.
- **perturbed_field** (`PerturbField`, optional) – If given, this field will be used, otherwise it will be generated. To be generated, either *init_boxes* and *redshift* must be given, or *user_params*, *cosmo_params* and *redshift*.
- **previous_perturbed_field** (`PerturbField`, optional) – An perturbed field at higher redshift. This is only used if mini_halo is included.
- **init_boxes** (`InitialConditions` , optional) – If given, and *perturbed_field not* given, these initial conditions boxes will be used to generate the perturbed field, otherwise initial conditions will be generated on the fly. If given, the user and cosmo params will be set from this object.
- **previous_ionize_box** (`IonizedBox` or None) – An ionized box at higher redshift. This is only used if *INHOMO_RECO* and/or *do_spin_temp* are true. If either of these are true, and this is not given, then it will be assumed that this is the "first box", i.e. that it can be populated accurately without knowing source statistics.
- **spin_temp** (`TsBox` or None, optional) – A spin-temperature box, only required if *do_spin_temp* is True. If None, will try to read in a spin temp box at the current redshift, and failing that will try to automatically create one, using the previous ionized box redshift as the previous spin temperature redshift.
- **pt_halos** (`PerturbHaloField` or None, optional) – If passed, this contains all the dark matter haloes obtained if using the USE_HALO_FIELD. This is a list of halo masses and coords for the dark matter haloes. If not passed, it will try and automatically create them using the available initial conditions.
- **user_params** (`UserParams`, optional) – Defines the overall options and parameters of the run.
- **cosmo_params** (`CosmoParams`, optional) – Defines the cosmological parameters used to compute initial conditions.
- **cleanup** (*bool, optional*) – A flag to specify whether the C routine cleans up its memory before returning. Typically, if *spin_temperature* is called directly, you will want this to be true, as if the next box to be calculate has different shape, errors will occur if memory is not

cleaned. However, it can be useful to set it to False if scrolling through parameters for the same box shape.

- ****global_kwargs** – Any attributes for *GlobalParams*. This will *temporarily* set global attributes for the duration of the function. Note that arguments will be treated as case-insensitive.

Returns IonizedBox – An object containing the ionized box data.

Other Parameters **regenerate, write, direc, random_seed** – See docs of *initial_conditions()* for more information.

Notes

Typically, the ionization field at any redshift is dependent on the evolution of xHI up until that redshift, which necessitates providing a previous ionization field to define the current one. This function provides several options for doing so. First, if neither the spin temperature field, nor inhomogeneous recombinations (specified in flag options) are used, no evolution needs to be done. Otherwise, either (in order of precedence)

1. a specific previous :class`~IonizedBox` object is provided, which will be used directly,
2. a previous redshift is provided, for which a cached field on disk will be sought,
3. a step factor is provided which recursively steps through redshift, calculating previous fields up until Z_HEAT_MAX, and returning just the final field at the current redshift, or
4. the function is instructed to treat the current field as being an initial "high-redshift" field such that specific sources need not be found and evolved.

Note: If a previous specific redshift is given, but no cached field is found at that redshift, the previous ionization field will be evaluated based on *z_step_factor*.

Examples

By default, no spin temperature is used, and neither are inhomogeneous recombinations, so that no evolution is required, thus the following will compute a coeval ionization box:

```
>>> xHI = ionize_box(redshift=7.0)
```

However, if either of those options are true, then a full evolution will be required:

```
>>> xHI = ionize_box(redshift=7.0, flag_options=FlagOptions(INHOMO_RECO=True,USE_
↪TS_FLUCT=True))
```

This will by default evolve the field from a redshift of *at least Z_HEAT_MAX* (a global parameter), in logarithmic steps of *ZPRIME_STEP_FACTOR*. To change these:

```
>>> xHI = ionize_box(redshift=7.0, zprime_step_factor=1.2, z_heat_max=15.0,
>>>                  flag_options={"USE_TS_FLUCT":True})
```

Alternatively, one can pass an exact previous redshift, which will be sought in the disk cache, or evaluated:

```
>>> ts_box = ionize_box(redshift=7.0, previous_ionize_box=8.0, flag_options={
>>>                     "USE_TS_FLUCT":True})
```

Beware that doing this, if the previous box is not found on disk, will continue to evaluate prior boxes based on *ZPRIME_STEP_FACTOR*. Alternatively, one can pass a previous `IonizedBox`:

```
>>> xHI_0 = ionize_box(redshift=8.0, flag_options={"USE_TS_FLUCT":True})
>>> xHI = ionize_box(redshift=7.0, previous_ionize_box=xHI_0)
```

Again, the first line here will implicitly use `ZPRIME_STEP_FACTOR` to evolve the field from `Z_HEAT_MAX`. Note that in the second line, all of the input parameters are taken directly from *xHI_0* so that they are consistent, and we need not specify the `flag_options`.

As the function recursively evaluates previous redshift, the previous spin temperature fields will also be consistently recursively evaluated. Only the final ionized box will actually be returned and kept in memory, however intervening results will by default be cached on disk. One can also pass an explicit spin temperature object:

```
>>> ts = spin_temperature(redshift=7.0)
>>> xHI = ionize_box(redshift=7.0, spin_temp=ts)
```

If automatic recursion is used, then it is done in such a way that no large boxes are kept around in memory for longer than they need to be (only two at a time are required).

py21cmfast.wrapper.perturb_field

py21cmfast.wrapper.**perturb_field**(*, *redshift*, *init_boxes=None*, *user_params=None*, *cosmo_params=None*, *random_seed=None*, *regenerate=None*, *write=None*, *direc=None*, ***global_kwargs*)

Compute a perturbed field at a given redshift.

Parameters

- **redshift** (*float*) – The redshift at which to compute the perturbed field.
- **init_boxes** (`InitialConditions`, optional) – If given, these initial conditions boxes will be used, otherwise initial conditions will be generated. If given, the user and cosmo params will be set from this object.
- **user_params** (`UserParams`, optional) – Defines the overall options and parameters of the run.
- **cosmo_params** (`CosmoParams`, optional) – Defines the cosmological parameters used to compute initial conditions.
- ****global_kwargs** – Any attributes for *GlobalParams*. This will *temporarily* set global attributes for the duration of the function. Note that arguments will be treated as case-insensitive.

Returns `PerturbedField`

Other Parameters **regenerate, write, direc, random_seed** – See docs of *initial_conditions()* for more information.

Examples

The simplest method is just to give a redshift:

```
>>> field = perturb_field(7.0)
>>> print(field.density)
```

Doing so will internally call the *initial_conditions()* function. If initial conditions have already been calculated, this can be avoided by passing them:

```
>>> init_boxes = initial_conditions()
>>> field7 = perturb_field(7.0, init_boxes)
>>> field8 = perturb_field(8.0, init_boxes)
```

The user and cosmo parameter structures are by default inferred from the `init_boxes`, so that the following is consistent:

```
>>> init_boxes = initial_conditions(user_params= UserParams(HII_DIM=1000))
>>> field7 = perturb_field(7.0, init_boxes)
```

If `init_boxes` is not passed, then these parameters can be directly passed:

```
>>> field7 = perturb_field(7.0, user_params=UserParams(HII_DIM=1000))
```

py21cmfast.wrapper.perturb_halo_list

py21cmfast.wrapper.**perturb_halo_list**(*, *redshift*, *init_boxes=None*, *halo_field=None*, *user_params=None*, *cosmo_params=None*, *astro_params=None*, *flag_options=None*, *random_seed=None*, *regenerate=None*, *write=None*, *direc=None*, ***global_kwargs*)

Given a halo list, perturb the halos for a given redshift.

Parameters

- **redshift** (*float*) – The redshift at which to determine the halo list.
- **init_boxes** (`InitialConditions`, optional) – If given, these initial conditions boxes will be used, otherwise initial conditions will be generated. If given, the user and cosmo params will be set from this object.
- **user_params** (`UserParams`, optional) – Defines the overall options and parameters of the run.
- **cosmo_params** (`CosmoParams`, optional) – Defines the cosmological parameters used to compute initial conditions.
- **astro_params** (`AstroParams` instance, optional) – The astrophysical parameters defining the course of reionization.
- ****global_kwargs** – Any attributes for *GlobalParams*. This will *temporarily* set global attributes for the duration of the function. Note that arguments will be treated as case-insensitive.

Returns `PerturbHaloField`

Other Parameters **regenerate, write, direc, random_seed** – See docs of *initial_conditions()* for more information.

Examples

Fill this in once finalised

py21cmfast.wrapper.run_coeval

py21cmfast.wrapper.**run_coeval**(*, *redshift=None*, *user_params=None*, *cosmo_params=None*, *astro_params=None*, *flag_options=None*, *regenerate=None*, *write=None*, *direc=None*, *init_box=None*, *perturb=None*, *use_interp_perturb_field=False*, *pt_halos=None*, *random_seed=None*, *cleanup=True*, ***global_kwargs*)

Evaluate a coeval ionized box at a given redshift, or multiple redshifts.

This is generally the easiest and most efficient way to generate a set of coeval cubes at a given set of redshift. It self-consistently deals with situations in which the field needs to be evolved, and does this with the highest memory-efficiency, only returning the desired redshift. All other calculations are by default stored in the on-disk cache so they can be re-used at a later time.

Note: User-supplied redshift are *not* used as previous redshift in any scrolling, so that pristine log-sampling can be maintained.

Parameters

- **redshift** (*array_like*) – A single redshift, or multiple redshift, at which to return results. The minimum of these will define the log-scrolling behaviour (if necessary).
- **user_params** (`UserParams`, optional) – Defines the overall options and parameters of the run.
- **cosmo_params** (`CosmoParams` , optional) – Defines the cosmological parameters used to compute initial conditions.
- **astro_params** (`AstroParams` , optional) – The astrophysical parameters defining the course of reionization.
- **flag_options** (`FlagOptions` , optional) – Some options passed to the reionization routine.
- **init_box** (`InitialConditions`, optional) – If given, the user and cosmo params will be set from this object, and it will not be re-calculated.
- **perturb** (list of `PerturbedField`, optional) – If given, must be compatible with init_box. It will merely negate the necessity of re-calculating the perturb fields.
- **use_interp_perturb_field** (*bool, optional*) – Whether to use a single perturb field, at the lowest redshift of the lightcone, to determine all spin temperature fields. If so, this field is interpolated in the underlying C-code to the correct redshift. This is less accurate (and no more efficient), but provides compatibility with older versions of 21cmFAST.
- **pt_halos** (*bool, optional*) – If given, must be compatible with init_box. It will merely negate the necessity of re-calculating the perturbed halo lists.
- **cleanup** (*bool, optional*) – A flag to specify whether the C routine cleans up its memory before returning. Typically, if *spin_temperature* is called directly, you will want this to be true, as if the next box to be calculate has different shape, errors will occur if memory is not cleaned. Note that internally, this is set to False until the last iteration.

- ****global_kwargs** – Any attributes for *GlobalParams*. This will *temporarily* set global attributes for the duration of the function. Note that arguments will be treated as case-insensitive.

Returns **coevals** (*Coeval*) – The full data for the Coeval class, with init boxes, perturbed fields, ionized boxes, brightness temperature, and potential data from the conservation of photons. If a single redshift was specified, it will return such a class. If multiple redshifts were passed, it will return a list of such classes.

Other Parameters **regenerate, write, direc, random_seed** – See docs of *initial_conditions()* for more information.

py21cmfast.wrapper.run_lightcone

py21cmfast.wrapper.**run_lightcone**(*, *redshift=None*, *max_redshift=None*, *user_params=None*, *cosmo_params=None*, *astro_params=None*, *flag_options=None*, *regenerate=None*, *write=None*, *lightcone_quantities='brightness_temp'*, *global_quantities='brightness_temp', 'xH_box'*, *direc=None*, *init_box=None*, *perturb=None*, *random_seed=None*, *use_interp_perturb_field=False*, *cleanup=True*, ***global_kwargs*)

Evaluate a full lightcone ending at a given redshift.

This is generally the easiest and most efficient way to generate a lightcone, though it can be done manually by using the lower-level functions which are called by this function.

Parameters

- **redshift** (*float*) – The minimum redshift of the lightcone.
- **max_redshift** (*float, optional*) – The maximum redshift at which to keep lightcone information. By default, this is equal to *z_heat_max*. Note that this is not *exact*, but will be typically slightly exceeded.
- **user_params** (*~UserParams*, optional) – Defines the overall options and parameters of the run.
- **astro_params** (`AstroParams`, optional) – Defines the astrophysical parameters of the run.
- **cosmo_params** (`CosmoParams`, optional) – Defines the cosmological parameters used to compute initial conditions.
- **flag_options** (`FlagOptions`, optional) – Options concerning how the reionization process is run, eg. if spin temperature fluctuations are required.
- **lightcone_quantities** (*tuple of str, optional*) – The quantities to form into a lightcone. By default, just the brightness temperature. Note that these quantities must exist in one of the output structures:
  - `InitialConditions`
  - `PerturbField`
  - `TsBox`
  - `IonizedBox`
  - `BrightnessTemp`

  To get a full list of possible quantities, run *get_all_fieldnames()*.

- **global_quantities** (*tuple of str, optional*) – The quantities to save as globally-averaged redshift-dependent functions. These may be any of the quantities that can be used in `lightcone_quantities`. The mean is taken over the full 3D cube at each redshift, rather than a 2D slice.
- **init_box** (`InitialConditions`, optional) – If given, the user and cosmo params will be set from this object, and it will not be re-calculated.
- **perturb** (list of `PerturbedField`, optional) – If given, must be compatible with init_box. It will merely negate the necessity of re-calculating the perturb fields. It will also be used to set the redshift if given.
- **use_interp_perturb_field** (*bool, optional*) – Whether to use a single perturb field, at the lowest redshift of the lightcone, to determine all spin temperature fields. If so, this field is interpolated in the underlying C-code to the correct redshift. This is less accurate (and no more efficient), but provides compatibility with older versions of 21cmFAST.
- **cleanup** (*bool, optional*) – A flag to specify whether the C routine cleans up its memory before returning. Typically, if *spin_temperature* is called directly, you will want this to be true, as if the next box to be calculate has different shape, errors will occur if memory is not cleaned. Note that internally, this is set to False until the last iteration.
- ****global_kwargs** – Any attributes for *GlobalParams*. This will *temporarily* set global attributes for the duration of the function. Note that arguments will be treated as case-insensitive.

Returns **lightcone** (`LightCone`) – The lightcone object.

Other Parameters **regenerate, write, direc, random_seed** – See docs of *initial_conditions()* for more information.

py21cmfast.wrapper.spin_temperature

py21cmfast.wrapper.**spin_temperature**(*, *astro_params=None*, *flag_options=None*, *redshift=None*, *perturbed_field=None*, *previous_spin_temp=None*, *init_boxes=None*, *cosmo_params=None*, *user_params=None*, *regenerate=None*, *write=None*, *direc=None*, *random_seed=None*, *cleanup=True*, ***global_kwargs*)

Compute spin temperature boxes at a given redshift.

See the notes below for how the spin temperature field is evolved through redshift.

Parameters

- **astro_params** (`AstroParams`, optional) – The astrophysical parameters defining the course of reionization.
- **flag_options** (`FlagOptions`, optional) – Some options passed to the reionization routine.
- **redshift** (*float, optional*) – The redshift at which to compute the ionized box. If not given, the redshift from *perturbed_field* will be used. Either *redshift*, *perturbed_field*, or *previous_spin_temp* must be given. See notes on *perturbed_field* for how it affects the given redshift if both are given.
- **perturbed_field** (`PerturbField`, optional) – If given, this field will be used, otherwise it will be generated. To be generated, either *init_boxes* and *redshift* must be given, or *user_params*, *cosmo_params* and *redshift*. By default, this will be generated at the same

redshift as the spin temperature box. The redshift of perturb field is allowed to be different than *redshift*. If so, it will be interpolated to the correct redshift, which can provide a speedup compared to actually computing it at the desired redshift.

- **previous_spin_temp** (`TsBox` or None) – The previous spin temperature box.
- **init_boxes** (`InitialConditions`, optional) – If given, and *perturbed_field not* given, these initial conditions boxes will be used to generate the perturbed field, otherwise initial conditions will be generated on the fly. If given, the user and cosmo params will be set from this object.
- **user_params** (`UserParams`, optional) – Defines the overall options and parameters of the run.
- **cosmo_params** (`CosmoParams`, optional) – Defines the cosmological parameters used to compute initial conditions.
- **cleanup** (*bool, optional*) – A flag to specify whether the C routine cleans up its memory before returning. Typically, if *spin_temperature* is called directly, you will want this to be true, as if the next box to be calculate has different shape, errors will occur if memory is not cleaned. However, it can be useful to set it to False if scrolling through parameters for the same box shape.
- ****global_kwargs** – Any attributes for *GlobalParams*. This will *temporarily* set global attributes for the duration of the function. Note that arguments will be treated as case-insensitive.

Returns `TsBox` – An object containing the spin temperature box data.

Other Parameters **regenerate, write, direc, random_seed** – See docs of *initial_conditions()* for more information.

Notes

Typically, the spin temperature field at any redshift is dependent on the evolution of spin temperature up until that redshift, which necessitates providing a previous spin temperature field to define the current one. This function provides several options for doing so. Either (in order of precedence):

1. a specific previous spin temperature object is provided, which will be used directly,
2. a previous redshift is provided, for which a cached field on disk will be sought,
3. a step factor is provided which recursively steps through redshift, calculating previous fields up until Z_HEAT_MAX, and returning just the final field at the current redshift, or
4. the function is instructed to treat the current field as being an initial "high-redshift" field such that specific sources need not be found and evolved.

Note: If a previous specific redshift is given, but no cached field is found at that redshift, the previous spin temperature field will be evaluated based on `z_step_factor`.

Examples

To calculate and return a fully evolved spin temperature field at a given redshift (with default input parameters), simply use:

```
>>> ts_box = spin_temperature(redshift=7.0)
```

This will by default evolve the field from a redshift of *at least Z_HEAT_MAX* (a global parameter), in logarithmic steps of *z_step_factor*. Thus to change these:

```
>>> ts_box = spin_temperature(redshift=7.0, zprime_step_factor=1.2, z_heat_max=15.
↪0)
```

Alternatively, one can pass an exact previous redshift, which will be sought in the disk cache, or evaluated:

```
>>> ts_box = spin_temperature(redshift=7.0, previous_spin_temp=8.0)
```

Beware that doing this, if the previous box is not found on disk, will continue to evaluate prior boxes based on the `z_step_factor`. Alternatively, one can pass a previous spin temperature box:

```
>>> ts_box1 = spin_temperature(redshift=8.0)
>>> ts_box = spin_temperature(redshift=7.0, previous_spin_temp=ts_box1)
```

Again, the first line here will implicitly use `z_step_factor` to evolve the field from around `Z_HEAT_MAX`. Note that in the second line, all of the input parameters are taken directly from *ts_box1* so that they are consistent. Finally, one can force the function to evaluate the current redshift as if it was beyond `Z_HEAT_MAX` so that it depends only on itself:

```
>>> ts_box = spin_temperature(redshift=7.0, zprime_step_factor=None)
```

This is usually a bad idea, and will give a warning, but it is possible.

py21cmfast.plotting

Simple plotting functions for 21cmFAST objects.

Functions

coeval_sliceplot(struct[, kind, cbar_label])	Show a slice of a given coeval box.
lightcone_sliceplot(lightcone[, kind, ...])	Create a 2D plot of a slice through a lightcone.
plot_global_history(lightcone[, kind, ...])	Plot the global history of a given quantity from a lightcone.

py21cmfast.plotting.coeval_sliceplot

py21cmfast.plotting.**coeval_sliceplot**(*struct: [<class 'py21cmfast.outputs._OutputStruct'>, <class 'py21cmfast.outputs.Coeval'>], kind: [<class 'str'>, None] = None, cbar_label: [<class 'str'>, None] = None, **kwargs*)

Show a slice of a given coeval box.

Parameters

- **struct** (`_OutputStruct` or `Coeval` instance) – The output of a function such as *ionize_box* (a class containing several quantities), or *run_coeval*.
- **kind** (*str*) – The quantity within the structure to be shown.
- **cbar_label** (*str, optional*) – A label for the colorbar. Some values of *kind* will have automatically chosen labels, but these can be turned off by setting `cbar_label=''`.

Returns *fig, ax* – figure and axis objects from matplotlib

Other Parameters

- **All other parameters are passed directly to :func:`_imshow_slice`. These include `slice_axis`**
- **and `slice_index`,**
- **which choose the actual slice to plot, optional `fig` and `ax` keywords which enable**
- **over-plotting previous figures,**
- **and the `imshow_kw` argument, which allows arbitrary styling of the plot.**

py21cmfast.plotting.lightcone_sliceplot

py21cmfast.plotting.**lightcone_sliceplot**(*lightcone: py21cmfast.outputs.LightCone, kind: str = 'brightness_temp', lightcone2: py21cmfast.outputs.LightCone = None, vertical: bool = False, xlabel: Optional[str] = None, ylabel: Optional[str] = None, cbar_label: Optional[str] = None, zticks: str = 'redshift', fig: Optional[matplotlib.figure.Figure] = None, ax: Optional[matplotlib.axes._axes.Axes] = None, **kwargs*)

Create a 2D plot of a slice through a lightcone.

Parameters

- **lightcone** (`Lightcone`) – The lightcone object to plot
- **kind** (*str, optional*) – The attribute of the lightcone to plot. Must be an array.
- **lightcone2** (*str, optional*) – If provided, plot the _difference_ of the selected attribute between the two lightcones.
- **vertical** (*bool, optional*) – Whether to plot the redshift in the vertical direction.
- **cbar_label** (*str, optional*) – A label for the colorbar. Some quantities have automatically chosen labels, but these can be removed by setting *cbar_label=''*.

- **zticks** (*str, optional*) – Defines the co-ordinates of the ticks along the redshift axis. Can be "redshift" (default), "frequency", "distance" (which starts at zero for the lowest redshift) or the name of any function in an astropy cosmology that is purely a function of redshift.
- **kwargs** – Passed through to `imshow()`.

Returns

- *fig* – The matplotlib Figure object
- *ax* – The matplotlib Axis object onto which the plot was drawn.

py21cmfast.plotting.plot_global_history

py21cmfast.plotting.**plot_global_history**(*lightcone: [<class 'py21cmfast.outputs.LightCone'>], kind: [<class 'str'>, None] = None, ylabel: [<class 'str'>, None] = None, ylog: [<class 'bool'>] = False, ax: [<class 'matplotlib.axes._axes.Axes'>, None] = None*)

Plot the global history of a given quantity from a lightcone.

Parameters

- **lightcone** (`LightCone` instance) – The lightcone containing the quantity to plot.
- **kind** (*str, optional*) – The quantity to plot. Must be in the *global_quantities* dict in the lightcone. By default, will choose the first entry in the dict.
- **ylabel** (*str, optional*) – A y-label for the plot. If None, will use `kind`.
- **ax** (*Axes, optional*) – The matplotlib Axes object on which to plot. Otherwise, created.

py21cmfast.cache_tools

A set of tools for reading/writing/querying the in-built cache.

Functions

clear_cache(**kwargs)	Delete datasets in the cache.
list_datasets(*[, direc, kind, hsh, seed, ...])	Yield all datasets which match a given set of filters.
query_cache(*[, direc, kind, hsh, seed, ...])	Get or print datasets in the cache.
readbox(*[, direc, fname, hsh, kind, seed, ...])	Read in a data set and return an appropriate object for it.

py21cmfast.cache_tools.clear_cache

py21cmfast.cache_tools.**clear_cache**(***kwargs*)

Delete datasets in the cache.

Walks through the cache, with given filters, and deletes all un-initialised dataset objects, optionally printing their representation to screen.

Parameters **kwargs** – All options passed through to *query_cache()*.

py21cmfast.cache_tools.list_datasets

py21cmfast.cache_tools.**list_datasets**(*, *direc=None*, *kind=None*, *hsh=None*, *seed=None*, *redshift=None*)

Yield all datasets which match a given set of filters.

Can be used to determine parameters of all cached datasets, in conjunction with *readbox()*.

Parameters

- **direc** (*str, optional*) – The directory in which to search for the boxes. By default, this is the centrally-managed directory, given by the `config.yml` in `.21cmfast`.
- **kind** (*str, optional*) – Filter by this kind (one of {"InitialConditions", "PerturbedField", "IonizedBox", "TsBox", "BrightnessTemp"}
- **hsh** (*str, optional*) – Filter by this hsh.
- **seed** (*str, optional*) – Filter by this seed.

Yields

- **fname** (*str*) – The filename of the dataset (without directory).
- **parts** (*tuple of strings*) – The (kind, hsh, seed) of the data set.

py21cmfast.cache_tools.query_cache

py21cmfast.cache_tools.**query_cache**(*, *direc=None*, *kind=None*, *hsh=None*, *seed=None*, *redshift=None*, *show=True*)

Get or print datasets in the cache.

Walks through the cache, with given filters, and return all un-initialised dataset objects, optionally printing their representation to screen. Useful for querying which kinds of datasets are available within the cache, and choosing one to read and use.

Parameters

- **direc** (*str, optional*) – The directory in which to search for the boxes. By default, this is the centrally-managed directory, given by the `config.yml` in `~/.21cmfast`.
- **kind** (*str, optional*) – Filter by this kind. Must be one of "InitialConditions", "PerturbedField", "IonizedBox", "TsBox" or "BrightnessTemp".
- **hsh** (*str, optional*) – Filter by this hsh.
- **seed** (*str, optional*) – Filter by this seed.
- **show** (*bool, optional*) – Whether to print out a repr of each object that exists.

Yields *obj* – Output objects, un-initialized.

py21cmfast.cache_tools.readbox

py21cmfast.cache_tools.**readbox**(*, *direc=None*, *fname=None*, *hsh=None*, *kind=None*, *seed=None*, *redshift=None*, *load_data=True*)

Read in a data set and return an appropriate object for it.

Parameters

- **direc** (*str, optional*) – The directory in which to search for the boxes. By default, this is the centrally-managed directory, given by the `config.yml` in `~/.21cmfast/`.
- **fname** (*str, optional*) – The filename (without directory) of the data set. If given, this will be preferentially used, and must exist.
- **hsh** (*str, optional*) – The md5 hsh of the object desired to be read. Required if *fname* not given.
- **kind** (*str, optional*) – The kind of dataset, eg. "InitialConditions". Will be the name of a class defined in `wrapper`. Required if *fname* not given.
- **seed** (*str or int, optional*) – The random seed of the data set to be read. If not given, and filename not given, then a box will be read if it matches the kind and hsh, with an arbitrary seed.
- **load_data** (*bool, optional*) – Whether to read in the data in the data set. Otherwise, only its defining parameters are read.

Returns *dataset* – An output object, whose type depends on the kind of data set being read.

Raises

- **IOError :** – If no files exist of the given kind and hsh.
- **ValueError :** – If either `fname` is not supplied, or both `kind` and `hsh` are not supplied.

4.5 Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

4.5.1 Bug reports/Feature Requests/Feedback/Questions

It is incredibly helpful to us when users report bugs, unexpected behaviour, or request features. You can do the following:

- Report a bug
- Request a Feature
- Ask a Question

When doing any of these, please try to be as succinct, but detailed, as possible, and use a "Minimum Working Example" whenever applicable.

4.5.2 Documentation improvements

21cmFAST could always use more documentation, whether as part of the official 21cmFAST docs, in docstrings, or even on the web in blog posts, articles, and such. If you do the latter, take the time to let us know about it!

4.5.3 High-Level Steps for Development

This is an abbreviated guide to getting started with development of 21cmFAST, focusing on the discrete high-level steps to take. See our notes for developers for more details about how to get around the 21cmFAST codebase and other technical details.

There are two avenues for you to develop 21cmFAST. If you plan on making significant changes, and working with 21cmFAST for a long period of time, please consider becoming a member of the 21cmFAST GitHub organisation (by emailing any of the owners or admins). You may develop as a member or as a non-member.

The difference between members and non-members only applies to the first step of the development process.

Note that it is highly recommended to work in an isolated python environment with all requirements installed from requirements_dev.txt. This will also ensure that pre-commit hooks will run that enforce the black coding style. If you do not install these requirements, you must manually run black before committing your changes, otherwise your changes will likely fail continuous integration.

As a *member*:

1. Clone the repo:

```
git clone git@github.com:21cmFAST/21cmFAST.git
```

As a *non-member*:

1. First fork 21cmFAST (look for the "Fork" button), then clone the fork locally:

```
git clone git@github.com:your_name_here/21cmFAST.git
```

The following steps are the same for both *members* and *non-members*:

2. Install a fresh new isolated environment. This can be either a basic virtualenv or a conda env (suggested). So either:

```
virtualenv ~/21cmfast
source ~/21cmfast/bin/activate
```

   or:

```
conda create -n 21cmfast python=3
conda activate 21cmfast
```

3. Install the *development* requirements for the project. If using the basic *virtualenv*:

```
pip install -r requirements_dev.txt
```

   or if using *conda* (suggested):

```
conda env update -f environment.yml
```

4. Install pre-commit hooks:

```
pre-commit install
```

5. Create a branch for local development:

```
git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally. **Note: as a member, you _must_ do step 5. If you make changes on master, you will _not_ be able to push them**.

6. When you're done making changes, run all the checks, doc builder and spell checker with tox one command:

```
tox
```

7. Commit your changes and push your branch to GitHub:

```
git add .
git commit -m "Your detailed description of your changes."
git push origin name-of-your-bugfix-or-feature
```

   Note that if the commit step fails due to a pre-commit hook, *most likely* the act of running the hook itself has already fixed the error. Try doing the *add* and *commit* again (up, up, enter). If it's still complaining, manually fix the errors and do the same again.

8. Submit a pull request through the GitHub website.

Pull Request Guidelines

If you need some code review or feedback while you're developing the code just make the pull request. You can mark the PR as a draft until you are happy for it to be merged.

Tips

To run a subset of tests:

```
tox -e envname -- py.test -k test_myfeature
```

To run all the test environments in *parallel* (you need to `pip install detox`):

```
detox
```

4.5.4 Developer Documentation

If you are new to developing `21cmFAST`, please read the *Contributing* section *first*, which outlines the general concepts for contributing to development, and provides a step-by-step walkthrough for getting setup. This page lists some more detailed notes which may be helpful through the development process.

Compiling for debugging

When developing, it is usually a good idea to compile the underlying C code in `DEBUG` mode. This may allow extra print statements in the C, but also will allow running the C under `valgrind` or `gdb`. To do this:

```
$ DEBUG=True pip install -e .
```

See *Installation* for more installation options.

Developing the C Code

In this section we outline how one might go about modifying/extending the C code and ensuring that the extension is compatible with the wrapper provided here. It is critical that you run all tests after modifying _anything_ (and see the section below about running with valgrind). When changing C code, before testing, ensure that the new C code is compiled into your environment by running:

```
$ rm -rf build
$ pip install .
```

Note that using a developer install (*-e*) is not recommended as it stores compiled objects in the working directory which don't get updated as you change code, and can cause problems later.

There are two main purposes you may want to write some C code:

1. An external plugin/extension which uses the output data from 21cmFAST.
2. Modifying the internal C code of 21cmFAST.

21cmFAST currently provides no support for external plugins/extensions. It is entirely possible to write your own C code to do whatever you want with the output data, but we don't provide any wrapping structure for you to do this, you will need to write your own. Internally, 21cmFAST uses the *cffi* library to aid the wrapping of the C code into Python. You don't need to do the same, though we recommend it. If your desired "extension" is something that needs to operate in-between steps of 21cmFAST, we also provide no support for this, but it is possible, so long as the next step in the chain maintains its API. You would be required to re-write the low-level wrapping function _preceding_ your inserted step as well. For instance, if you had written a self-contained piece of code that modified the initial conditions box, adding some physical effect which is not already covered, then you would need to write a low-level wrapper _and_ re-write the `initial_conditions` function to modify the box before returning it. We provide no easy "plugin" system for doing this currently. If your external code is meant to be inserted _within_ a basic step of 21cmFAST, this is currently not possible. You will instead have to modify the source code itself.

Modifying the C-code of 21cmFAST should be relatively simple. If your changes are entirely internal to a given function, then nothing extra needs to be done. A little more work has to be done if the modifications add/remove input parameters or the output structure. If any of the input structures are modified (i.e. an extra parameter added to it), then the corresponding class in `py21cmfast.wrapper` must be modified, usually simply to add the new parameter to the `_defaults_` dict with a default value. For instance, if a new variable `some_param` was added to the `user_params` struct in the `ComputeInitialConditions` C function, then the `UserParams` class in the wrapper would be modified, adding `some_param=<default_value>` to its *_default_* dict. If the default value of the parameter is dependent on another parameter, its default value in this dict can be set to `None`, and you can give it a dynamic definition as a Python `@property`. For example, the `DIM` parameter of `UserParams` is defined as:

```
@property
def DIM(self):
    if self._some_param is None:
        return self._DIM or 4 * self.HII_DIM
```

Note the underscore in `_DIM` here: by default, if a dynamic property is defined for a given parameter, the `_default_` value is saved with a prefixed underscore. Here we return either the explicitly set `DIM`, or 4 by the `HII_DIM`. In

addition, if the new parameter is not settable – if it is completely determined by other parameters – then don't put it in `_defaults_` at all, and just give it a dynamic definition.

If you modify an output struct, which usually house a number of array quantities (often float pointers, but not necessarily), then you'll again need to modify the corresponding class in the wrapper. In particular, you'll need to add an entry for that particular array in the `_init_arrays` method for the class. The entry consists of initialising that array (usually to zeros, but not necessarily), and setting its proper dtype. All arrays should be single-pointers, even for multi-dimensional data. The latter can be handled by initalising the array as a 1D numpy array, but then setting its shape attribute (after creation) to the appropriate n-dimensional shape (see the `_init_arrays` method for the `InitialConditions` class for examples of this).

Modifying the `global_params` struct should be relatively straightforward, and no changes in the Python are necessary. However, you may want to consider adding the new parameter to relevant `_filter_params` lists for the output struct wrapping classes in the wrapper. These lists control which global parameters affect which output structs, and merely provide for more accurate caching mechanisms.

C Function Standards

The C-level functions are split into two groups – low-level "private" functions, and higher-level "public" or "API" functions. All API-level functions are callable from python (but may also be called from other C functions). All API-level functions are currently prototyped in *21cmFAST.h*.

To enable consistency of error-checking in Python (and a reasonable standard for any kind of code), we enforce that any API-level function must return an integer status. Any "return" objects must be modified in-place (i.e. passed as pointers). This enables Python to control the memory access of these variables, and also to receive proper error statuses (see below for how we do exception handling). We also adhere to the convention that "output" variables should be passed to the function as its last argument(s). In the case that _only_ the last argument is meant to be "output", there exists a simple wrapper *_call_c_simple* in *wrapper.py* that will neatly handle the calling of the function in an intuitive pythonic way.

Running with Valgrind

If any changes to the C code are made, it is ideal to run tests under valgrind, and check for memory leaks. To do this, install `valgrind` (we have tested v3.14+), which is probably available via your package manager. We provide a suppression file for `valgrind` in the `devel/` directory of the main repository.

It is ideal if you install a development-version of python especially for running these tests. To do this, download the version of python you want and then configure/install with:

```
$ ./configure --prefix=<your-home>/<directory> --without-pymalloc --with-pydebug --
↪with-valgrind
$ make; make install
```

Construct a `virtualenv` on top of this installation, and create your environment, and install all requirements.

If you do not wish to run with a modified version of python, you may continue with your usual version, but may get some extra cruft in the output. If running with Python version > 3.6, consider running with environment variable `PYTHONMALLOC=malloc` (see https://stackoverflow.com/questions/20112989/how-to-use-valgrind-with-python).

The general pattern for using valgrind with python is:

```
$ valgrind --tool=memcheck --track-origins=yes --leak-check=full --suppressions=devel/
↪valgrind-suppress-all-but-c.supp <python script>
```

One useful command is to run valgrind over the test suite (from the top-level repo directory):

```
$ valgrind --tool=memcheck --track-origins=yes --leak-check=full --suppressions=devel/
↪valgrind-suppress-all-but-c.supp pytest
```

While we will attempt to keep the suppression file updated to the best of our knowledge so that only relevant leaks and errors are reported, you will likely have to do a bit of digging to find the relevant parts.

Valgrind will likely run very slowly, and sometimes you will know already which exact tests are those which may have problems, or are relevant to your particular changes. To run these:

```
$ valgrind --tool=memcheck --track-origins=yes --leak-check=full --suppressions=devel/
↪valgrind-suppress-all-but-c.supp pytest -v tests/<test_file>::<test_func>
```

Producing Integration Test Data

There are bunch of so-called "integration tests", which rely on previously-produced data. To produce this data, run *python tests/produce_integration_test_data.py*.

Furthermore, this data should only be produced with good reason – the idea is to keep it static while the code changes, to have something steady to compare to. If a particular PR fixes a bug which affects a certain tests' data, then that data should be re-run, in the context of the PR, so it can be explained.

Logging in C

The C code has a header file `logging.h`. The C code should *never* contain bare print-statements – everything should be formally logged, so that the different levels can be printed to screen correctly. The levels are defined in `logging.h`, and include levels such as `INFO`, `WARNING` and `DEBUG`. Each level has a corresponding macro that starts with `LOG_`. Thus to log run-time information to stdout, you would use `LOG_INFO("message");`. Note that the message does not require a final newline character.

Exception handling in C

There are various places that things can go wrong in the C code, and they need to be handled gracefully so that Python knows what to do with it (rather than just quitting!). We use the simple `cexcept.h` header file from http://www.nicemice.net/cexcept/ to enable a simple form of exception handling. That file itself should **not be edited**. There is another header – `exceptions.h` – that defines how we use exceptions throughout `21cmFAST`. Any time an error arises that can be understood, the developer should add a `Throw <ErrorKind>;` line. The `ErrorKind` can be any of the kinds defined in `exceptions.h` (eg. `GSLError` or `ValueError`). These are just integers.

Any C function that has a header in `21cmFAST.h` – i.e. any function that is callable directly from Python – *must* be globally wrapped in a `Try {} Catch(error_code) {}` block. See `GenerateICs.c` for an example. Most of the code should be in the `Try` block. Anything that does a `Throw` at any level of the call stack within that `Try` will trigger a jump to the `Catch`. The `error_code` is the integer that was thrown. Typically, one will perhaps want to do some cleanup here, and then finally *return* the error code.

Python knows about the exit codes it can expect to receive, and will raise Python exceptions accordingly. From the python side, two main kinds of exceptions could be raised, depending on the error code returned from C. The lesser exception is called a `ParameterError`, and is supposed to indicate an error that happened merely because the parameters that were input to the calculation were just too extreme to handle. In the case of something like an automatic Monte Carlo algorithm that's iterating over random parameters, one would *usually* want to just keep going at this point, because perhaps it just wandered too far in parameter space. The other kind of error is a `FatalCError`, and this is where things went truly wrong, and probably will do for any combination of parameters.

If you add a kind of Exception in the C code (to `exceptions.h`), then be sure to add a handler for it in the `_process_exitcode` function in `wrapper.py`.

4.6 Authors

- Brad Greig - github.com/BradGreig
- Andrei Mesinger - github.com/andreimesinger
- Steven Murray - github.com/steven-murray

4.6.1 Contributors

4.7 Changelog

4.7.1 v3.0.2

4.7.2 Fixed

- Added prototype functions to enable compilation for some standard compilers on MacOS.

4.7.3 v3.0.1

Modifications to the internal code structure of 21cmFAST

Added

- Refactor FFTW wisdom creation to be a python callable function

4.7.4 v3.0.0

Complete overhaul of 21cmFAST, including a robust python-wrapper and interface, caching mechanisms, and public repository with continuous integration. Changes and equations for minihalo features in this version are found in https://arxiv.org/abs/2003.04442

All functionality of the original 21cmFAST v2 C-code has been implemented in this version, including `USE_HALO_FIELD` and performing full integration instead of using the interpolation tables (which are faster).

Added

- Updated the radiation source model: (i) all radiation fields including X-rays, UV ionizing, Lyman Werner and Lyman alpha are considered from two seperated population namely atomic-cooling (ACGs) and minihalo-hosted molecular-cooling galaxies (MCGs); (ii) the turn-over masses of ACGs and MCGs are estimated with cooling efficiency and feedback from reionization and lyman werner suppression (Qin et al. 2020). This can be switched on using new `flag_options` `USE_MINI_HALOS`.

- Updated kinetic temperature of the IGM with fully ionized cells following equation 6 of McQuinn (2015) and partially ionized cells having the volume-weightied temperature between the ionized (volume: 1-xHI; temperature T_RE) and neutral components (volume: xHI; temperature: temperature of HI). This is stored in IonizedBox as temp_kinetic_all_gas. Note that Tk in TsBox remains to be the kinetic temperature of HI.
- Tests: many unit tests, and also some regression tests.
- CLI: run 21cmFAST boxes from the command line, query the cache database, and produce plots for standard comparison runs.
- Documentation: Jupyter notebook demos and tutorials, FAQs, installation instructions.
- Plotting routines: a number of general plotting routines designed to plot coeval and lightcone slices.
- New power spectrum option (`POWER_SPECTRUM=5`) that uses a CLASS-based transfer function. WARNING: If POWER_SPECTRUM==5 the cosmo parameters cannot be altered, they are set to the Planck2018 best-fit values for now (until CLASS is added): (omegab=0.02237, omegac= 0.120, hubble=0.6736 (the rest are irrelevant for the transfer functions, but in case: A_s=2.100e-9, n_s=0.9649, z_reio = 11.357)
- New `user_params` option `USE_RELATIVE_VELOCITIES`, which produces initial relative velocity cubes (option implemented, but not the actual computation yet).
- Configuration management.
- global params now has a context manager for changing parameters temporarily.
- Vastly improved error handling: exceptions can be caught in C code and propagated to Python to inform the user of what's going wrong.
- Ability to write high-level data (`Coeval` and `Lightcone` objects) directly to file in a simple portable format.

Changed

- `POWER_SPECTRUM` option moved from `global_params` to `user_params`.
- Default cosmology updated to Planck18.

4.7.5 v2.0.0

All changes and equations for this version are found in https://arxiv.org/abs/1809.08995.

Changed

- Updated the ionizing source model: (i) the star formation rates and ionizing escape fraction are scaled with the masses of dark matter halos and (ii) the abundance of active star forming galaxies is exponentially suppressed below the turn-over halo mass, M_{turn}, according to a duty cycle of exp(M_{turn}/M_{h}), where M_{h} is a halo mass.
- Removed the mean free path parameter, R_{mfp}. Instead, directly computes inhomogeneous, sub-grid recombinations in the intergalactic medium following the approach of Sobacchi & Mesinger (2014)

4.7.6 v1.2.0

Added

- Support for a halo mass dependent ionizing efficiency: zeta = zeta_0 (M/Mmin)^alpha, where zeta_0 corresponds to HII_EFF_FACTOR, Mmin –> ION_M_MIN, alpha –> EFF_FACTOR_PL_INDEX in ANAL_PARAMS.H

4.7.7 v1.12.0

Added

- Code 'redshift_interpolate_boxes.c' to interpolate between comoving cubes, creating comoving light cone boxes.
- Enabled openMP threading for SMP machines. You can specify the number of threads (for best performace, do not exceed the number of processors) in INIT_PARAMS.H. You do not need to have an SMP machine to run the code. NOTE: YOU SHOULD RE-INSTALL FFTW to use openMP (see INSTALL file)
- Included a threaded driver file 'drive_zscroll_reion_param.c' set-up to perform astrophysical parameter studies of reionization
- Included explicit support for WDM cosmologies; see COSMOLOGY.H. The prescription is similar to that discussed in Barkana+2001; Mesinger+2005, madifying the (i) transfer function (according to the Bode+2001 formula; and (ii) including the effective pressure term of WDM using a Jeans mass analogy. (ii) is approximated with a sharp cuttoff in the EPS barrier, using 60* M_J found in Barkana+2001 (the 60 is an adjustment factor found by fitting to the WDM collapsed fraction).
- A Gaussian filtering step of the PT fields to perturb_field.c, in addition to the implicit boxcar smoothing. This avoids having"empty" density cells, i.e. delta=-1, with some small loss in resolution. Although for most uses delta=-1 is ok, some Lya forest statistics do not like it.
- Added treatment of the risidual electron fraction from X-ray heating when computing the ionization field. Relatedly, modified Ts.c to output all intermediate evolution boxes, Tk and x_e.
- Added a missing factor of Omega_b in Ts.c corresponding to eq. 18 in MFC11. Users who used a previous version should note that their results just effecively correspond to a higher effective X-ray efficiency, scaled by 1/Omega_baryon.
- Normalization optimization to Ts.c, increasing performace on arge resolution boxes

Fixed

- GSL interpolation error in kappa_elec_pH for GSL versions > 1.15
- Typo in macro definition, which impacted the Lya background calculation in v1.11 (not applicable to earlier releases)
- Outdated filename sytax when calling gen_size_distr in drive_xHIscroll
- Redshift scrolling so that drive_logZscroll_Ts.c and Ts.c are in sync.

Changed

- Output format to avoid FFT padding for all boxes
- Filename conventions to be more explicit.
- Small changes to organization and structure

4.7.8 v1.1.0

Added

- Wrapper functions mod_fwrite() and mod_fread() in Cosmo_c_progs/misc.c, which should fix problems with the library fwrite() and fread() for large files (>4GB) on certain operating systems.
- Included print_power_spectrum_ICs.c program which reads in high resolution initial conditions and prints out an ASCII file with the associated power spectrum.
- Parameter in Ts.c for the maximum allowed kinetic temperature, which increases stability of the code when the redshift step size and the X-ray efficiencies are large.

Fixed

- Oversight adding support for a Gaussian filter for the lower resolution field.

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

p

INDEX

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

W

X