21CMMC

Release 0.1.0

The 21cmFAST Collaboration

May 21, 2020

CONTENTS

An extensible MCMC framework for 21cmFAST.

This code uses semantic versioning, though this will strictly begin when *v1.0.0* is officially shipped.

- Free software: MIT license

CHAPTER

ONE

FEATURES

- Seamless integration with `emcee`-based MCMC.
- MCMC is easily extensible via the addition of different likelihoods using the same underlying data.

CHAPTER

TWO

DOCUMENTATION

See https://21CMMC.readthedocs.org.

CHAPTER

THREE

ACKNOWLEDGING

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

Bradley Greig and Andrei Mesinger, "21CMMC: an MCMC analysis tool enabling astrophysical parameter studies of the cosmic 21 cm signal", Monthly Notices of the Royal Astronomical Society, Volume 449, Issue 4, p.4246-4263 (2015), https://doi.org/10.1093/mnras/stv571

Bradley Greig and Andrei Mesinger, "Simultaneously constraining the astrophysics of reionization and the epoch of heating with 21CMMC", Monthly Notices of the Royal Astronomical Society, Volume 472, Issue 3, p.2651-2669 (2017), https://doi.org/10.1093/mnras/stx2118

Bradley Greig and Andrei Mesinger, "21CMMC with a 3D light-cone: the impact of the co-evolution approximation on the astrophysics of reionization and cosmic dawn", Monthly Notices of the Royal Astronomical Society, Volume 477, Issue 3, p.3217-3229 (2018), https://doi.org/10.1093/mnras/sty796

Jaehong Park et al., "Inferring the astrophysics of reionization and cosmic dawn from galaxy luminosity functions and the 21-cm signal", Monthly Notices of the Royal Astronomical Society, Volume 484, Issue 1, p.933-949 (2018), https://doi.org/10.1093/mnras/stz032

CHAPTER

FOUR

CONTENTS

4.1 Installation

As may be expected, `21CMMC` depends on `21cmFAST`, and this has some non-python dependencies. Thus, you must ensure that these dependencies (usually system-wide ones) are installed *before* attempting to install `21CMMC`. See https://21cmfast.readthedocs.org/en/latest/installation for details on these dependencies.

Then follow the instructions below, depending on whether you are a user or developer.

4.1.1 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
```

If you are confident that the non-python dependencies are installed, you can simply install `21CMMC` in the usual fashion:

```
pip install 21CMMC
```

Note that if `21cmFAST` is not installed, it will be installed automatically. There are several environment variables which control the compilation of `21cmFAST`, and these can be set during this call. See the above installation docs for details.

4.1.2 For Developers

If you are developing `21CMMC`, 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 -r requirements_dev.txt
pip install -e .
```

And if you would like to also compile documentation:

```
pip install -r docs/requirements.txt
```

4.2 Tutorials and FAQs

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

4.2.1 MCMC Introduction

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

from py21cmmc import analyse
from py21cmmc import mcmc
import py21cmmc as p21mc

%load_ext autoreload
%autoreload 2
```

In this notebook we demonstrate how to do the simplest possible MCMC to fit just two astrophysical parameters to a series of coeval brightness temperature boxes without noise, and then visualise the results.

This tutorial will introduce the basic components of using py21cmmc for doing MCMC to fit parameters. For more advanced topics, see the relevant tutorials, guides or FAQs.

[3]:
```
p21mc.__version__
```

[3]:
```
'1.0.0dev'
```

The Structure of 21MCMC

As a bit of a primer, we discuss some of the implementation structure for how likelihoods are actually evaluated in 21CMMC. Understanding this structure will help to identify any issues that may arise, construct useful likelihoods, utilise the various options that the built-in structures have, and eventually to create your own (see XXX for more details of how to do this).

The structure of an MCMC routine in 21CMMC is based on that offered by the cosmoHammer library, but with heavy modifications. This structure is based on a single master LikelihoodComputationChain (let's just call it the Chain), which houses a number of what we'll call cores and likelihoods. These are instances of Python classes which can be named arbitrarily (and in principle do not need to be subclassed from any particular object), but follow a minimal API, which we'll outline shortly. In short, any Chain requires at *least* one core, and at least one likelihood. Multiples of each are allowed and will work together seamlessly.

Any MCMC should be run using the run_mcmc function in 21CMMC. While the MCMC can be run manually by setting up a Chain with its cores and likelihoods, there are various pitfalls and gotchas associated with this that usually make it easier just to use the in-built function. This function will take a list of cores and likelihoods, along with some specifications of how the MCMC is to proceed, set up a Chain, run the MCMC sampling, and return the Chain object to you.

Thus, almost all of the flexibility of 21CMMC lies in the construction of the various core and likelihood modules.

Core and Likelihood Modules

Let's briefly discuss these important `core` and `likelihood` modules – the philosophy behind them, and what they can and can't do. We'll leave a discussion of how to go about implementing your own for another tutorial.

The basic idea behind `core` and `likelihood` modules is that the `core`s are supposed to do the calculations to fill up what is called a `context` dictionary on every iteration of the MCMC. The `likelihood`s then use the contents of this `context` to evaluate a likelihood (this ends up being the sum of the likelihood returned by each `likelihood` module).

In practice, there is no hard-and-fast limit to the scope of the `core` or `likelihood`: the `core` could evaluate the likelihood itself, store it in the `context`, and the `likelihood` could merely access and return it. Likewise, the `core` could do nothing, and let the `likelihood` perform the entire calculation.

For practical and philosophical reasons, however there are a few properties of each which dictate how *best* to separate the work that each kind of module does:

1. All `core` modules constructively interfere. That is, they are invoked in sequence, and the output of one propagates as potential input to the next. The various quantities that are computed by the `core`s are not over-written (unless specifically directed to), but are rather accumulated.
2. Conversely, all `likelihood` modules are invoked after all of the `core` modules have been invoked. Each likelihood module is expected to have access to all information from the sequency of `core`s, and is expected not to modify that information. The operation of each `likelihood` is thus in principle independent of each of the other `likelihood`s. This implies that the total posterior is the sum of each of the likelihoods, which are considered statistically independent.
3. Due to the first two points, we consider `core`s as *constructive* and `likelihood`s as *reductive*. That is, it is most useful to put calculations that *build* data given a set of parameters in `core`s, and operations that *reduce* that data to some final form (eg. a power spectrum) in the `likelihoods`. This is because a given dataset, produced by the accumulation of `core`s, may yield a number of independent likelihoods, while these `likelihood`s may be equally valid for a range of different data models (eg. inclusion or exclusion of various systematic effects).
4. Point 3 implies that it is cleanest if all operations that explicitly require the model parameters occur in `core`s. The reduction of data should not in general be model-dependent. In practice, the current parameters *are* available in the `likelihood`s, but we consider it cleaner if this can be separated.
5. In general, as both the `core`s and `likelihood`s are used to build a probabilistic *model* which can be used to evaluate the likelihood of given *data*, both of their output should in principle be deterministic with respect to input parameters. Nevertheless, one may consider the process as a forward-model, and a forward-model is able to *produce* mock data. Indeed, 21CMMC adds a framework to the base `cosmoHammer` for producing such mock data, which are inherently stochastic. A useful way to conceptualize the separation of `core` and `likelihood` is to ensure that all stochasticity can in principle be added in the `core`, and that the `likelihood` should have no aspect of randomness in any of its calculations. The `likelihood` should reduce real/mock data in the same way that it reduces model data.

Given these considerations, the `core` modules that are implemented within `21CMMC` perform `21cmFAST` simulations and add these large datasets to the `context`, while the various `likelihoods` will use this "raw" data and compress it down to a likelihood – either by taking a power spectrum, global average or some other reduction.

Some of the features of `core`s as implemented in 21CMMC are the following. Each `core`: * has access to the entire `Chain` in which it is embedded (if it is indeed embedded), which enables sharing of information between `core`s (and ensuring that they are consistent with one another, if applicable). * has access to the names of the parameters which are currently being constrained. * is enabled for equality testing with other `core`s. * has a special method (and runtime parameters) for storing arbitrary data in the `Chain` on a per-iteration basis (see the advanced MCMC FAQ for more info). * has an optional method for converting a data model into a "mock" (i.e. incorporating a random component), which can be used to simulate mock data for consistency testing.

Some of the features of `likelihoods` as implemented in 21CMMC are that each `likelihood`: * also has access to the `Chain` and parameter names, as well as equality testing, like the `cores`. * computes the likelihood in two steps: first reducing model data to a "final form", and then computing the likelihood from this form (eg. reducing a simulation cube to a 1D power spectrum, and then computing a χ^2 likelihood on the power spectrum). This enables two things: (i) production and saving of reduced mock data, which can be used directly for consistency tests, and (ii) the ability to use *either* raw data or reduced data as input to the likelihood. * has methods for loading data and noise from files. * has the ability to check that a list of `required_cores` are loaded in the `Chain`.

Running MCMC

Enough discussion, let's create our `core` and `likelihood`. In this tutorial we use a single `core` – one which evaluates the coeval brightness temperature field at an arbitrary selection of redshifts, and a single `likelihood` – one which reduces the 3D field(s) into a 1D power spectrum/spectra and evaluates the likelihood based on a χ-square fit to data.

[4]:
```
core = p21mc.CoreCoevalModule( # All core modules are prefixed by Core* and end with␣
↪*Module
    redshift = [7,8,9],           # Redshifts of the coeval fields produced.
    user_params = dict(       # Here we pass some user parameters. Also cosmo_params,␣
↪astro_params and flag_options
        HII_DIM = 50,         # are available. These specify only the *base*␣
↪parameters of the data, *not* the
        BOX_LEN = 125.0       # parameters that are fit by MCMC.
    ),
    regenerate=False          # Don't regenerate init_boxes or perturb_fields if they␣
↪are already in cache.
) # For other available options, see the docstring.

# Now the likelihood...
datafiles = ["data/simple_mcmc_data_%s.npz"%z for z in core.redshift]

likelihood = p21mc.Likelihood1DPowerCoeval(  # All likelihood modules are prefixed by␣
↪Likelihood*
    datafile = datafiles,                   # All likelihoods have this, which␣
↪specifies where to write/read data
    noisefile= None,                        # All likelihoods have this, specifying␣
↪where to find noise profiles.
    logk=False,                             # Should the power spectrum bins be log-
↪spaced?
    min_k=0.1,                              # Minimum k to use for likelihood
    max_k=1.0,                              # Maximum ""
    simulate = True,                        # Simulate the data, instead of reading␣
↪it in.
                                            # will be performed.
) # For other available options, see the docstring
```

Now we have all we need to start running the MCMC. The most important part of the call to `run_mcmc` is the specification of `params`, which specifies which are the parameters *to be fit*. This is passed as a dictionary, where the keys are the parameter names, and *must* come from either `cosmo_params` or `astro_params`, and be of `float` type. The values of the dictionary are length-4 lists: `(guess, min, max, width)`. The first specifies where the best guess of the true value lies, and the initial walkers will be chosen around here. The `min/max` arguments provide upper and lower limits on the parameter, outside of which the likelihood will be `-infinity`. The `width` affects the initial distribution of walkers around the best-guess (it does *not* influence any kind of "prior").

Finally, the `model_name` merely affects the file name of the output chain data, along with the `datadir` argument.

[5]:
```
model_name = "SimpleTest"

chain = mcmc.run_mcmc(
    core, likelihood,        # Use lists if multiple cores/likelihoods required. These will be eval'd in order.
    datadir='data',          # Directory for all outputs
    model_name=model_name,   # Filename of main chain output
    params=dict(             # Parameter dict as described above.
        HII_EFF_FACTOR = [30.0, 10.0, 50.0, 3.0],
        ION_Tvir_MIN = [4.7, 4, 6, 0.1],
    ),
    walkersRatio=3,          # The number of walkers will be walkersRatio*nparams
    burninIterations=0,      # Number of iterations to save as burnin. Recommended to leave as zero.
    sampleIterations=150,    # Number of iterations to sample, per walker.
    threadCount=6,           # Number of processes to use in MCMC (best as a factor of walkersRatio)
    continue_sampling=False  # Whether to contine sampling from previous run *up to* sampleIterations.
)
```

```
/home/steven/miniconda3/envs/21CMMC/lib/python3.7/site-packages/powerbox/dft.py:121: UserWarning: You do not have pyFFTW installed. Installing it should give some speed increase.
  warnings.warn("You do not have pyFFTW installed. Installing it should give some speed increase.")
2019-11-15 12:25:59,295 | WARNING | likelihood.py::_write_data() | File data/simple_mcmc_data_7.npz already exists. Moving previous version to data/simple_mcmc_data_7.npz.bk
2019-11-15 12:25:59,297 | WARNING | likelihood.py::_write_data() | File data/simple_mcmc_data_8.npz already exists. Moving previous version to data/simple_mcmc_data_8.npz.bk
2019-11-15 12:25:59,298 | WARNING | likelihood.py::_write_data() | File data/simple_mcmc_data_9.npz already exists. Moving previous version to data/simple_mcmc_data_9.npz.bk
```

Analysis

Accessing chain data

The full chain data, as well as any stored data (as "blobs") is available within the chain as the `samples` attribute. If access to this "chain" object is lost (eg. the MCMC was run via CLI and is finished), an exact replica of the store object can be read in from file. Unified access is provided through the `get_samples` function in the `analyse` module. Thus all these are equivalent:

[6]:
```
samples1 = chain.samples

samples2 = analyse.get_samples(chain)

samples3 = analyse.get_samples("data/%s"%model_name)

# Equivalent:
# samples = analyse.get_samples("data/%s"%model_name)
```

[7]:
```
print(np.all(samples1.accepted == samples2.accepted))
print(np.all(samples1.accepted == samples3.accepted))
```

```
True
True
```

Do note that while the first two methods return exactly the same object, occupying the same memory:

[8]:
```
samples1 is samples2
```

[8]:
```
True
```

this is not true when reading in the samples from file:

[9]:
```
print(samples1 is samples3)
del samples3; samples=samples1 # Remove unnecessary memory, and rename to samples
```

```
False
```

Several methods are defined on the sample object (which has type `HDFStore`), to ease interactions. For example, one can access the various dimensions of the chain:

[10]:
```
niter = samples.size
nwalkers, nparams = samples.shape
```

We can also check what the parameter names of the run were, and their initial "guess" (this is the first value passed to the "parameters" dictionary in `run_mcmc`:

[11]:
```
print(samples.param_names)
print([samples.param_guess[k] for k in samples.param_names])
```

```
('HII_EFF_FACTOR', 'ION_Tvir_MIN')
[array([30.]), array([4.7])]
```

Or one can view how many iterations were accepted for each walker:

[12]:
```
samples.accepted, np.mean(samples.accepted/niter)
```

[12]:
```
(array([106,  98,  95, 105, 109, 109]), 0.6911111111111111)
```

We can also see what extra data we saved along the way by getting the blob names (see below for more details on this):

[13]:
```
samples.blob_names
```

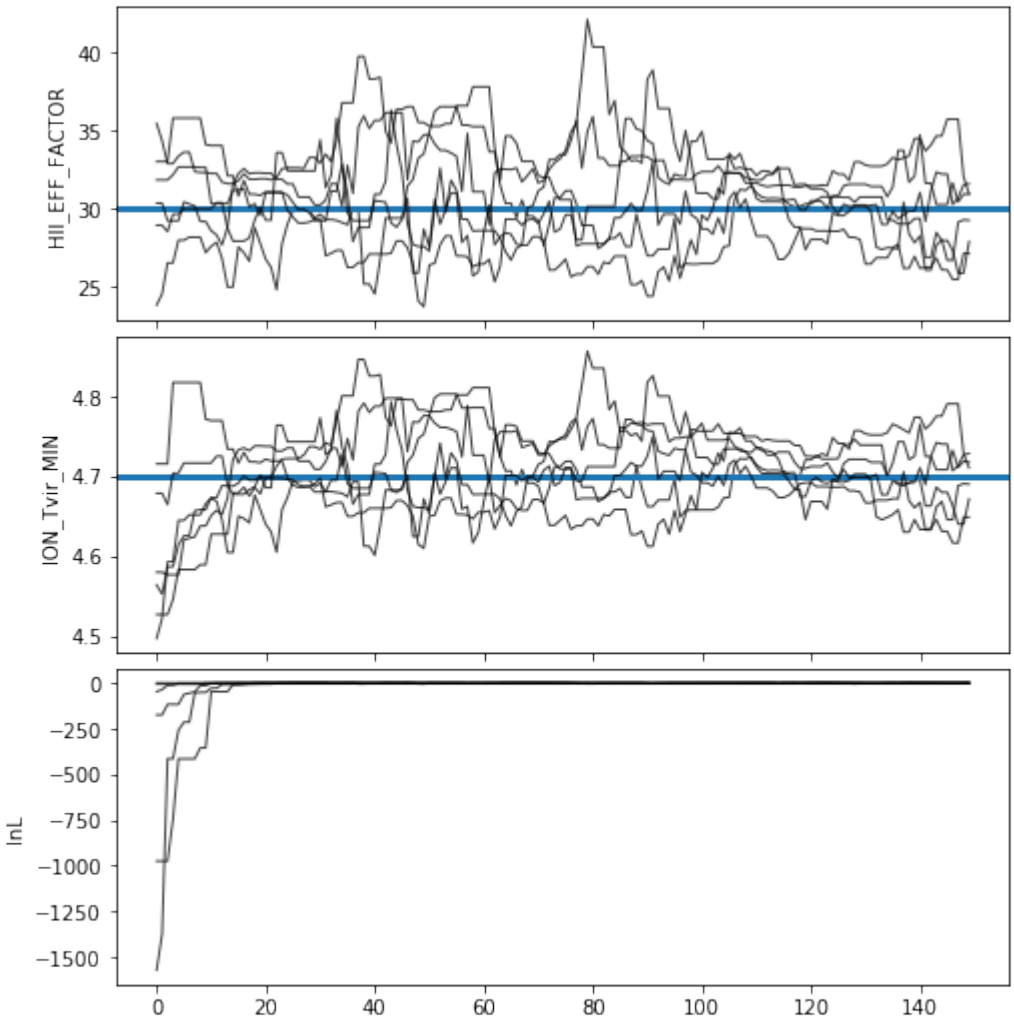
[13]:
```
('k_z7', 'delta_z7', 'k_z8', 'delta_z8', 'k_z9', 'delta_z9')
```

Finally, we can get the actual chain data, using `get_chain` and friends. However, this is best done dynamically, as the `samples` object itself does *not* hold the chain in memory, rather transparently read it from file when accessed.

Trace Plot

Often, for diagnostic purposes, the most useful plot to start with is the trace plot. This enables quick diagnosis of burnin time and walkers that haven't converged. The function in `py21cmmc` by default plots the log probability along with the various parameters that were fit. It also supports setting a starting iteration, and a thinning amount.

[14]:
```
analyse.trace_plot(samples, include_lnl=True, start_iter=0, thin=1, colored=False,␣
↪show_guess=True);
```



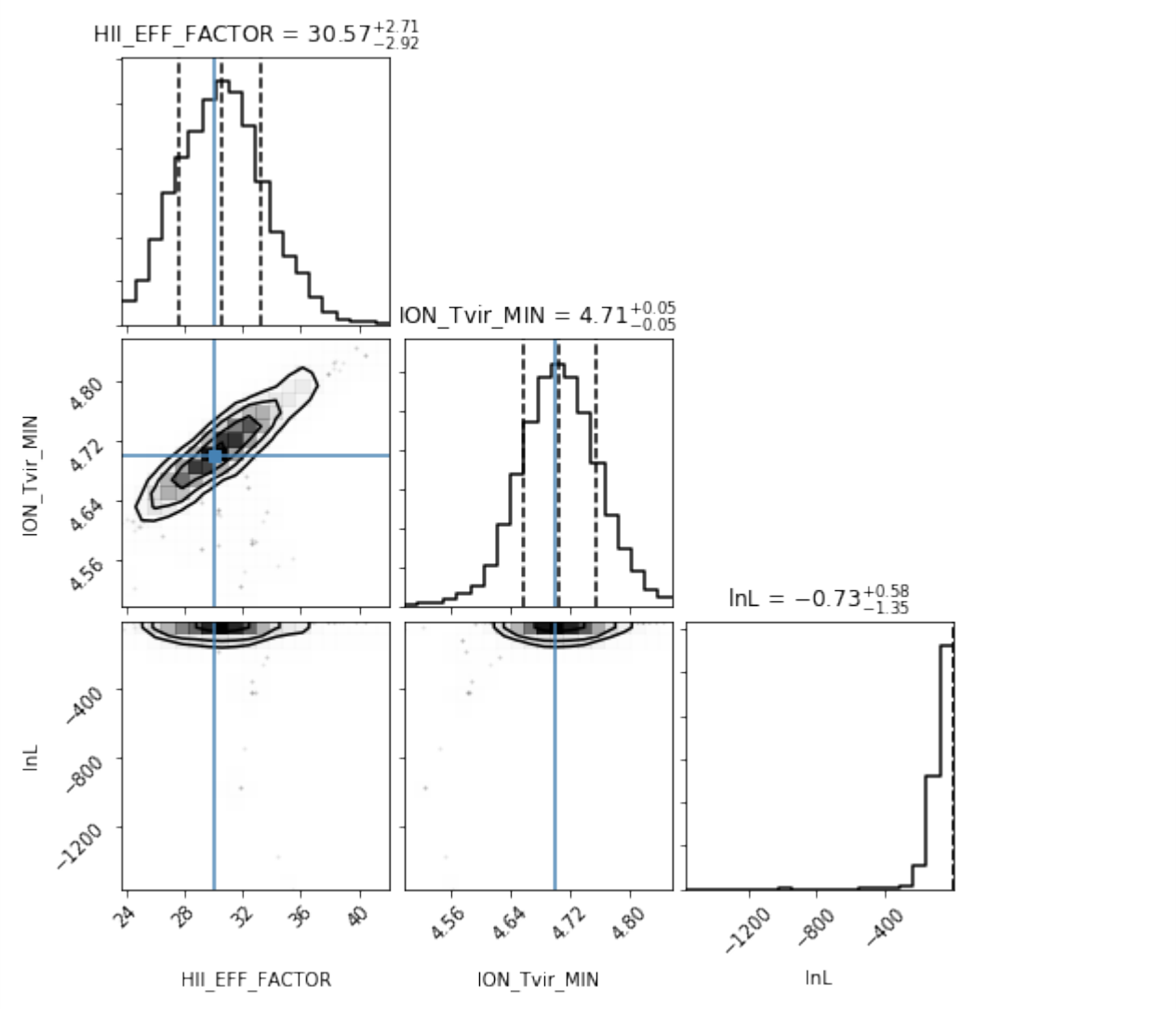
Corner Plot

One of the most useful plots to come out of an MCMC analysis is the "corner" plot, which shows the correlation of each parameter with every other parameter. The function in `py21cmmc` will by default also show the original "guess" for each parameter as a blue line, and also show the log-likelihood as a psuedo-"parameter", though this can be turned off.

[15]:
```
samples.param_guess
```

[15]:
```
array([(30., 4.7)],
      dtype=[('HII_EFF_FACTOR', '<f8'), ('ION_Tvir_MIN', '<f8')])
```

[16]:
```
analyse.corner_plot(samples);
```



Model Comparison Plot

Another plot of interest is a "model comparison" plot – i.e. comparing the range of outputted models to the data itself. These will differ significantly depending on the kind of data produced by the likelihood function, and thus they depend also on the actual data used. We thus do not provide a general function for plotting this. We do however show how one might go about this task in the function below.

First, however, we show how one might interact with the data and saved models/blobs.

To extract all blob data from the samples:

[17]:
```
blobs = samples.get_blobs()
```

For simplicity, let's extract each kind of blob from the blob structured array:

[18]:
```
k = blobs['k_z7']
model_power = blobs['delta_z7'], blobs['delta_z8'], blobs['delta_z9']
```

(continues on next page)

(continued from previous page)

```
print(k.shape, model_power[0].shape)
nk = k.shape[-1]
```

```
(150, 6, 22) (150, 6, 22)
```

Here we notice that `k` should be the same on each iteration, so we take just the first:

[19]:
```
print(np.all(k[0] == k[1]))
k = k[0]
```

```
True
```

Finally, we also want to access the *data* to which the models have been compared. Since we have access to the original `likelihood` object, we can easily pull this from it. However, we equivalently could have read it in from file (this file is *not* always present, only if `datafile` is present in the likelihood constructor):

[20]:
```
p_data = np.array([d['delta'] for d in likelihood.data])
k_data = np.array([d['k'] for d in likelihood.data])

# Equivalent
# data = np.genfromtxt("simple_mcmc_data.txt")
# k_data = data[:,0]
# p_data = data[:,1:]
```

Now, let's define a function which will plot our model comparison:

[21]:
```
def model_compare_plot(samples, p_data, k_data, thin=1, start_iter=0):
    chain = samples.get_chain(thin=thin, discard=start_iter, flat=True)
    blobs = samples.get_blobs(thin=thin, discard=start_iter, flat=True)

    k = blobs['k_z7'][0]
    model_power = [blobs[name] for name in samples.blob_names if name.startswith(
↪"delta_")]

    print(k.shape)

    nz = len(model_power)
    nk = k.shape[-1]

    fig, ax = plt.subplots(1, nz, sharex=True, sharey=True, figsize=(6*nz, 4.5),
                           subplot_kw={"xscale":'log', "yscale":'log'}, gridspec_kw={
↪"hspace":0.05, 'wspace':0.05},
                           squeeze=False)

    for i in range(nz):
        this_power = model_power[i]
        this_data = p_data[i]

        label="models"

        for pp in this_power:
            ax[0,i].plot(k, pp, color='k', alpha=0.2, label=label, zorder=1)
            if label:
                label=None

        mean = np.mean(this_power, axis=0)
        std = np.std(this_power, axis=0)
```

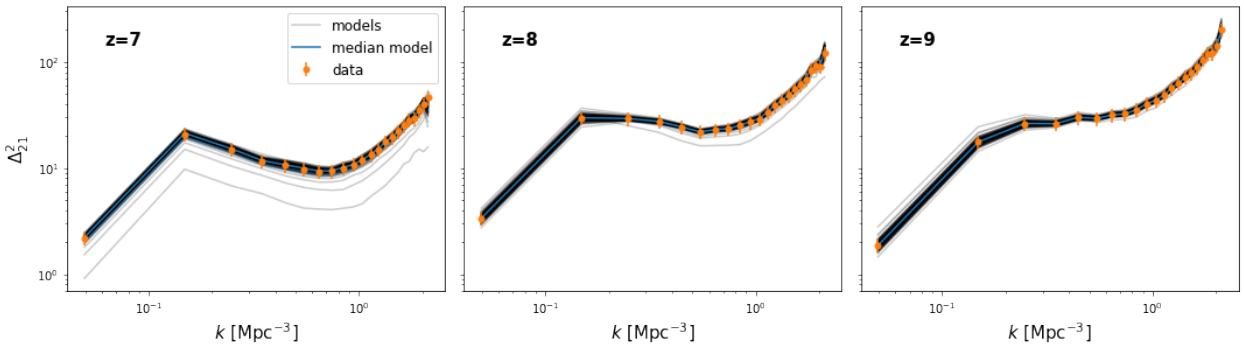
(continues on next page)

(continued from previous page)

```
        md = np.median(this_power, axis=0)

        ax[0,i].fill_between(k, mean - std, mean+std, color="C0", alpha=0.6)
        ax[0,i].plot(k, md, color="C0", label="median model")

        ax[0,i].errorbar(k_data, this_data, yerr = (0.15*this_data), color="C1",
                     label="data", ls="None", markersize=5, marker='o')

        ax[0,i].set_xlabel("$k$ [Mpc$^{-3}$]", fontsize=15)
        ax[0,i].text(0.1, 0.86, "z=%s"%core.redshift[i], transform=ax[0,i].transAxes,␣
↪fontsize=15, fontweight='bold')

    ax[0,0].legend(fontsize=12)
    #plt.ylim((3.1e2, 3.5e3))


    ax[0,0].set_ylabel("$\Delta^2_{21}$", fontsize=15)

#plt.savefig(join(direc, modelname+"_power_spectrum_plot.pdf"))
```

[22]:
```
model_compare_plot(samples, p_data, k_data[0], thin=10)
```

```
(22,)
```



4.2.2 MCMC with LightCones

[3]:
```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np

from py21cmmc import analyse
from py21cmmc import mcmc
import py21cmmc as p21mc

%load_ext autoreload
%autoreload 2

import logging
print("Version of py21cmmc: ", p21mc.__version__)
```

```
Version of py21cmmc:  1.0.0dev
```

In this tutorial we demonstrate how to do MCMC with a lightcone, to fit just two astrophysical parameters without

noise, and then visualise the results. This tutorial follows a very similar pattern to the MCMC intro, and you should follow that one first.

Running MCMC

To perform an MCMC on a lightcone is *very* similar to a Coeval cube. Merely use the `CoreLightConeModule` as the core module, and the `Likelihood1DPowerLightcone` as the likelihood. One extra parameter to the `core` is available – `max_redshift`, which specifies the approximate upper limit on the lightcone's depth. Note that this does **not** necessarily specify the maximum redshift at which the ionization will be computed (this is specified by `z_heat_max`), it merely specifies where to start saving the ionization boxes into a lightcone.

Furthermore, one extra parameter to the likelihood is available – `nchunks` – which allows to break the full lightcone up into independent chunks for which the power spectrum will be computed.

Thus, for example:

[5]:
```
core = p21mc.CoreLightConeModule( # All core modules are prefixed by Core* and end␣
↪with *Module
    redshift = 7.0,              # Lower redshift of the lightcone
    max_redshift = 9.0,          # Approximate maximum redshift of the lightcone␣
↪(will be exceeded).
    user_params = dict(
        HII_DIM = 50,
        BOX_LEN = 125.0
    ),
    z_step_factor=1.04,          # How large the steps between evaluated redshifts␣
↪are (log).
    z_heat_max=18.0,             # Completely ineffective since no spin temp or␣
↪inhomogeneous recombinations.
    regenerate=False
) # For other available options, see the docstring.

# Now the likelihood...
datafiles = ["data/lightcone_mcmc_data_%s.npz"%i for i in range(4)]
likelihood = p21mc.Likelihood1DPowerLightcone(  # All likelihood modules are prefixed␣
↪by Likelihood*
    datafile = datafiles,        # All likelihoods have this, which specifies where␣
↪to write/read data
    logk=False,                 # Should the power spectrum bins be log-spaced?
    min_k=0.1,                  # Minimum k to use for likelihood
    max_k=1.0,                  # Maximum ""
    nchunks = 4,                 # Number of chunks to break the lightcone into
    simulate=True
) # For other available options, see the docstring
```

Actually run the MCMC:

[ ]:
```
model_name = "LightconeTest"

chain = mcmc.run_mcmc(
    core, likelihood,        # Use lists if multiple cores/likelihoods required.␣
↪These will be eval'd in order.
    datadir='data',          # Directory for all outputs
    model_name=model_name,   # Filename of main chain output
    params=dict(             # Parameter dict as described above.
        HII_EFF_FACTOR = [30.0, 10.0, 50.0, 3.0],
        ION_Tvir_MIN = [4.7, 4, 6, 0.1],
```

(continues on next page)

(continued from previous page)

```
    ),
    walkersRatio=3,          # The number of walkers will be walkersRatio*nparams
    burninIterations=0,      # Number of iterations to save as burnin. Recommended to␣
↪leave as zero.
    sampleIterations=100,    # Number of iterations to sample, per walker.
    threadCount=2,           # Number of processes to use in MCMC (best as a factor␣
↪of walkersRatio)
    continue_sampling=False,  # Whether to contine sampling from previous run *up to*␣
↪sampleIterations.
    log_level_stream=logging.DEBUG
)
```

```
/home/steven/miniconda3/envs/21CMMC/lib/python3.7/site-packages/powerbox/dft.py:121:␣
↪UserWarning: You do not have pyFFTW installed. Installing it should give some speed␣
↪increase.
  warnings.warn("You do not have pyFFTW installed. Installing it should give some␣
↪speed increase.")
2019-09-30 18:33:22,616 INFO:Using CosmoHammer 0.6.1
2019-09-30 18:33:22,617 INFO:Using emcee 2.2.1
2019-09-30 18:33:22,630 INFO:all burnin iterations already completed
2019-09-30 18:33:22,633 INFO:Sampler: <class 'py21cmmc.cosmoHammer.CosmoHammerSampler.
↪CosmoHammerSampler'>
configuration:
  Params: [30.   4.7]
  Burnin iterations: 0
  Samples iterations: 100
  Walkers ratio: 3
  Reusing burn in: True
  init pos generator: SampleBallPositionGenerator
  stop criteria: IterationStopCriteriaStrategy
  storage util: <py21cmmc.cosmoHammer.storage.HDFStorageUtil object at 0x7f333044b358>
likelihoodComputationChain:
Core Modules:
  CoreLightConeModule
Likelihood Modules:
  Likelihood1DPowerLightcone

2019-09-30 18:33:22,636 INFO:start sampling after burn in
2019-09-30 18:35:04,787 INFO:Iteration finished:10
2019-09-30 18:36:38,799 INFO:Iteration finished:20
2019-09-30 18:38:18,352 INFO:Iteration finished:30
2019-09-30 18:39:46,722 INFO:Iteration finished:40
```

Analysis

Accessing chain data

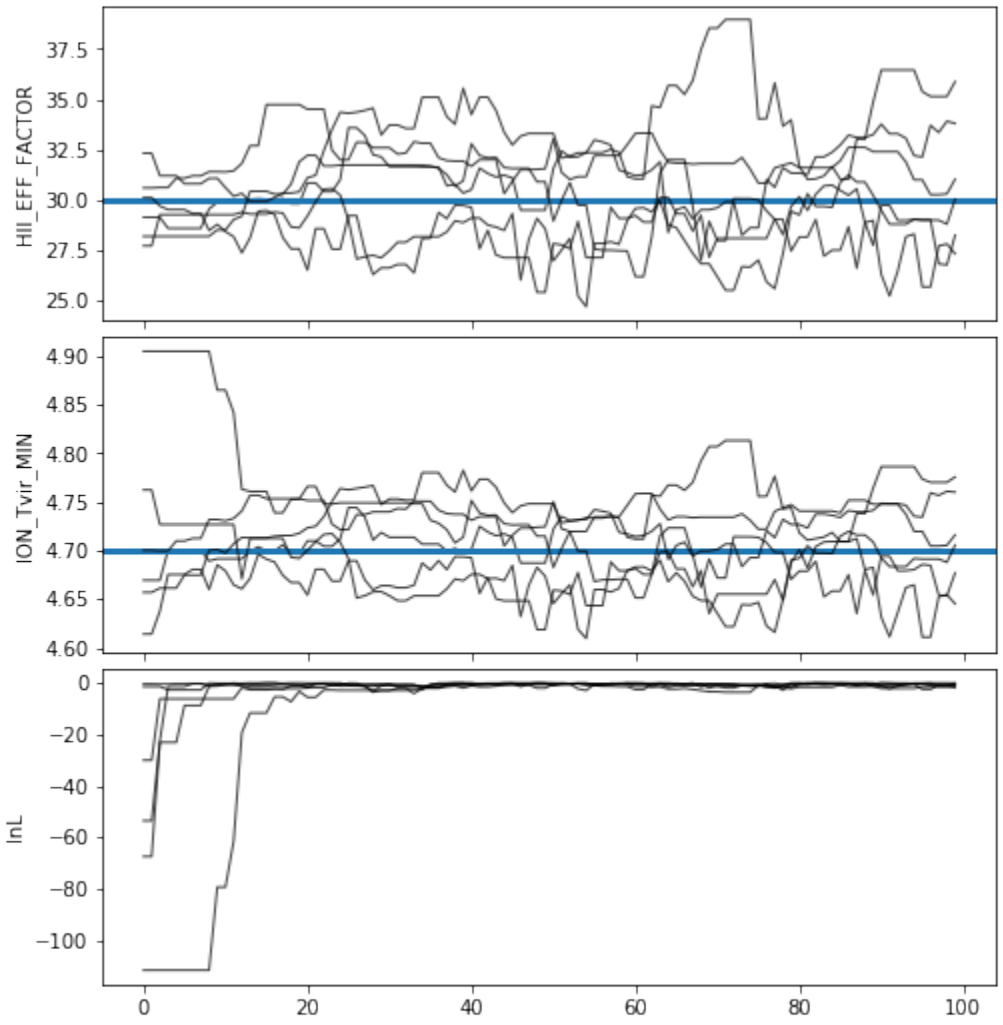
Access the samples object within the chain (see the intro for more details):

[5]:
```
samples = chain.samples
```

Trace Plot

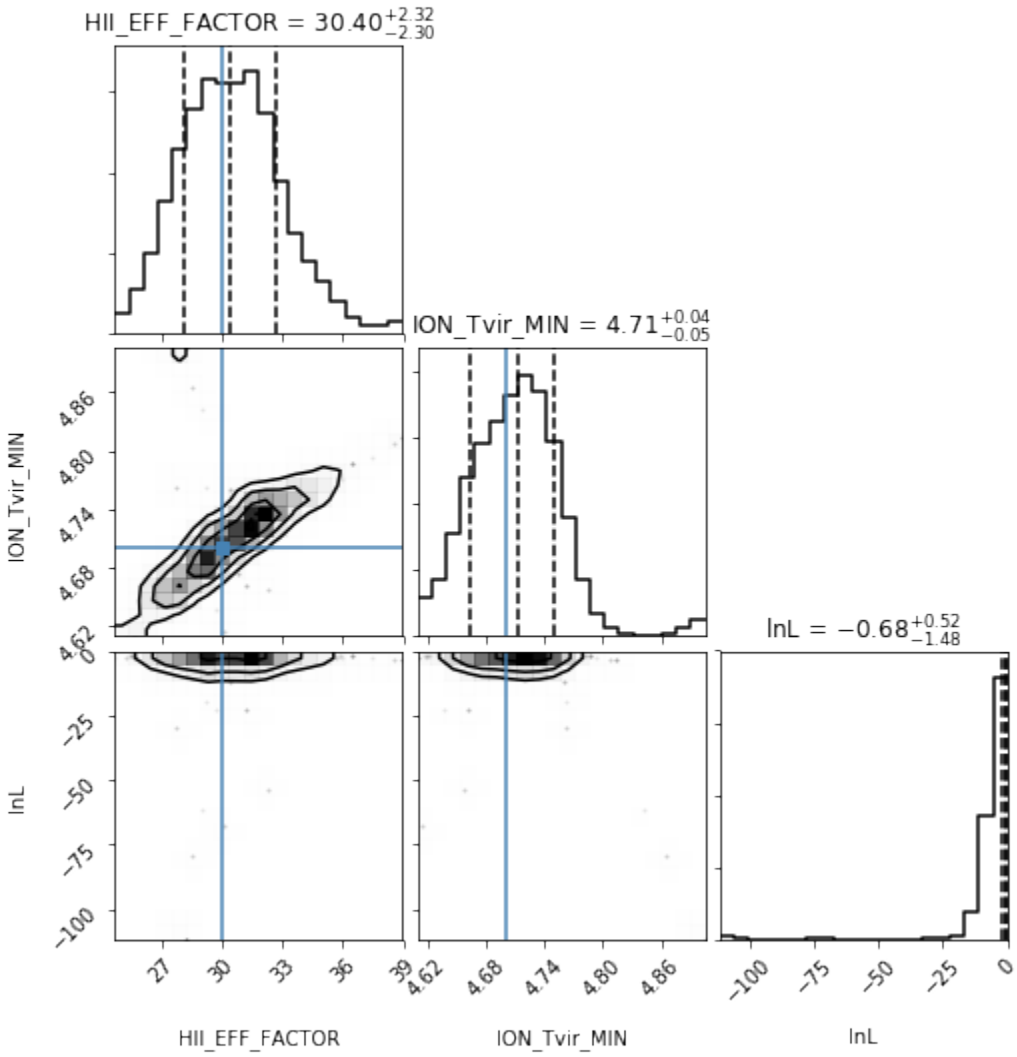
Often, for diagnostic purposes, the most useful plot to start with is the trace plot. This enables quick diagnosis of burnin time and walkers that haven't converged. The function in `py21cmmc` by default plots the log probability along with the various parameters that were fit. It also supports setting a starting iteration, and a thinning amount.

[6]:
```
analyse.trace_plot(samples, include_lnl=True, start_iter=0, thin=1, colored=False,␣
↪show_guess=True);
```



Corner Plot

[7]:
```
analyse.corner_plot(samples);
```



Model Comparison Plot

Extract all blob data from the samples:

[8]:
```
blobs = samples.get_blobs()
```

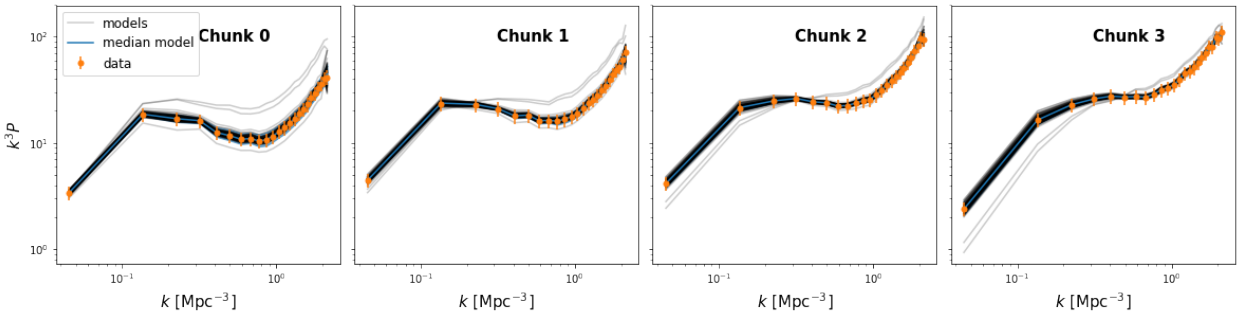
Read in the data:

[9]:
```
delta_data = [d['delta'] for d in likelihood.data]
k_data = [d['k'] for d in likelihood.data]
```

Now, let's define a function which will plot our model comparison:

[10]:
```
def model_compare_plot(samples, k_data, delta_data, thin=1, start_iter=0):
    chain = samples.get_chain(thin=thin, discard=start_iter, flat=True)
    blobs = samples.get_blobs(thin=thin, discard=start_iter, flat=True)

    ks = [blobs[name] for name in samples.blob_names if name.startswith("k")]
    models = [blobs[name] for name in samples.blob_names if name.startswith("delta")]

    fig, ax = plt.subplots(1, len(ks), sharex=True, sharey=True, figsize=(5*len(ks),␣
↪4.5),
                           subplot_kw={"xscale":'log', "yscale":'log'}, gridspec_kw={
↪"hspace":0.05, 'wspace':0.05},
                           squeeze=False)

    for i,(k,model, kd, data) in enumerate(zip(ks,models, k_data, delta_data)):
        label="models"

        for pp in model:
            ax[0,i].plot(k[0], pp, color='k', alpha=0.2, label=label, zorder=1)
            if label:
                label=None

        mean = np.mean(model, axis=0)
        std = np.std(model, axis=0)
        md = np.median(model, axis=0)

        ax[0,i].fill_between(k[0], mean - std, mean+std, color="C0", alpha=0.6)
        ax[0,i].plot(k[0], md, color="C0", label="median model")

        ax[0,i].errorbar(kd, data, yerr = (0.15*data), color="C1",
                     label="data", ls="None", markersize=5, marker='o')

        ax[0,i].set_xlabel("$k$ [Mpc$^{-3}$]", fontsize=15)
        ax[0,i].text(0.5, 0.86, "Chunk %s"%i, transform=ax[0,i].transAxes,␣
↪fontsize=15, fontweight='bold')

    ax[0,0].legend(fontsize=12)
    #plt.ylim((3.1e2, 3.5e3))


    ax[0,0].set_ylabel("$k^3 P$", fontsize=15)

#plt.savefig(join(direc, modelname+"_power_spectrum_plot.pdf"))
```

[11]:
```
model_compare_plot(samples, k_data, delta_data, thin=5)
```



[ ]:

4.2.3 MCMC with Multiple Likelihoods

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

from py21cmmc.mcmc import analyse
from py21cmmc import mcmc

%load_ext autoreload
%autoreload 2
```

In this tutorial we demonstrate two main ideas: how to use the non-power-spectrum likelihoods provided in `21CMMC`, and how to use more than one likelihood at a time. The extra likelihoods that we use were called "priors" in previous versions and publications, and they can help break parameter degeneracies in some cases.

[2]:
```
import py21cmmc
py21cmmc.__version__
```

[2]:
```
'0.1.0'
```

Setting up cores and likelihoods

The various extra likelihoods provided work *best* (but not exclusively) with a Lightcone core, so we will use them in conjunction with a lightcone likelihood. This is because they typically measure the global average of a quantity (either brightness temperature or neutral fraction) as a function of redshift.

For the core, we take the same as that used in the lightcone MCMC intro:

[3]:
```
core = mcmc.CoreLightConeModule( # All core modules are prefixed by Core* and end␣
↪with *Module
    redshift = 5.5,              # Lower redshift of the lightcone
    max_redshift = 8.0,          # Approximate maximum redshift of the lightcone␣
↪(will be exceeded).
    user_params = dict(
        HII_DIM = 50,
        BOX_LEN = 125.0
    ),
    z_step_factor=1.04,          # How large the steps between evaluated redshifts␣
↪are (log).
    regenerate=False,
    nchunks=4
) # For other available options, see the docstring.
```

Now we instantiate several different likelihoods, beginning with the basic 1D power spectrum:

[4]:
```
# Now the likelihood...
datafiles = ["data/lightcone_mcmc_data_%s.npz"%i for i in range(4)]
likelihood_ps = mcmc.Likelihood1DPowerLightcone(  # All likelihood modules are␣
↪prefixed by Likelihood*
    datafile = datafiles,        # All likelihoods have this, which specifies where␣
↪to write/read data
```

(continues on next page)

(continued from previous page)

```
    logk=False,                  # Should the power spectrum bins be log-spaced?
    min_k=0.1,                   # Minimum k to use for likelihood
    max_k=1.0,                   # Maximum ""
    nchunks = 4,                  # Number of chunks to break the lightcone into
    simulate=True
) # For other available options, see the docstring

likelihood_planck = mcmc.LikelihoodPlanck()  # there are no options here, though some␣
↪class variables
                                             # can be set to change the behaviour (eg.
↪ tau_mean, tu_sigma)

likelihood_mcgreer = mcmc.LikelihoodNeutralFraction()  # Again, no required options,␣
↪though the options
                                                       # redshift, xHI_mean, xHI_
↪sigma can be set. By default
                                                       # they evaluate to the McGreer␣
↪values.

likelihood_greig = mcmc.LikelihoodGreig()
```

Running MCMC

[ ]:
```
model_name = "MultiLikelihoodTest"

chain = mcmc.run_mcmc(
    core, [likelihood_ps, likelihood_mcgreer, likelihood_greig],#, likelihood_planck],
    datadir='data',           # Directory for all outputs
    model_name=model_name,    # Filename of main chain output
    params=dict(              # Parameter dict as described above.
        HII_EFF_FACTOR = [30.0, 10.0, 50.0, 3.0],
        ION_Tvir_MIN = [4.7, 4, 6, 0.1],
    ),
    walkersRatio=4,          # The number of walkers will be walkersRatio*nparams
    burninIterations=0,       # Number of iterations to save as burnin. Recommended to␣
↪leave as zero.
    sampleIterations=100,     # Number of iterations to sample, per walker.
    threadCount=4,            # Number of processes to use in MCMC (best as a factor␣
↪of walkersRatio)
    continue_sampling=False  # Whether to contine sampling from previous run *up to*␣
↪sampleIterations.
)
```

```
2019-03-27 11:13:34,257 | WARNING | likelihood.py::_write_data() | File data/
↪lightcone_mcmc_data_0.npz already exists. Moving previous version to data/lightcone_
↪mcmc_data_0.npz.bk
2019-03-27 11:13:34,260 | WARNING | likelihood.py::_write_data() | File data/
↪lightcone_mcmc_data_1.npz already exists. Moving previous version to data/lightcone_
↪mcmc_data_1.npz.bk
2019-03-27 11:13:34,263 | WARNING | likelihood.py::_write_data() | File data/
↪lightcone_mcmc_data_2.npz already exists. Moving previous version to data/lightcone_
↪mcmc_data_2.npz.bk
2019-03-27 11:13:34,265 | WARNING | likelihood.py::_write_data() | File data/
↪lightcone_mcmc_data_3.npz already exists. Moving previous version to data/lightcone_
↪mcmc_data_3.npz.bk
```

Analysis

Accessing chain data

Access the samples object within the chain (see the intro for more details):

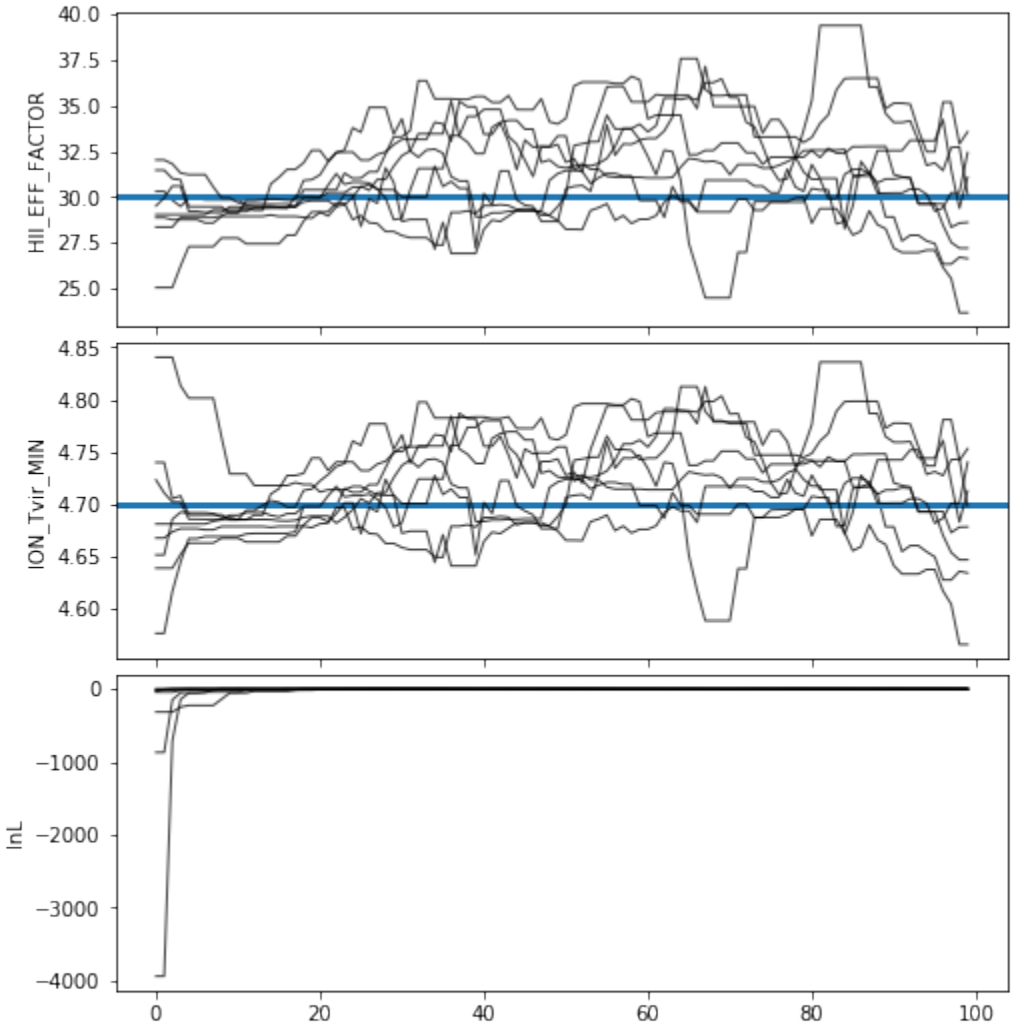
[6]:
```
samples = chain.samples
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-6-b4c91829d4bf> in <module>
----> 1 samples = chain.samples

NameError: name 'chain' is not defined
```

Trace Plot

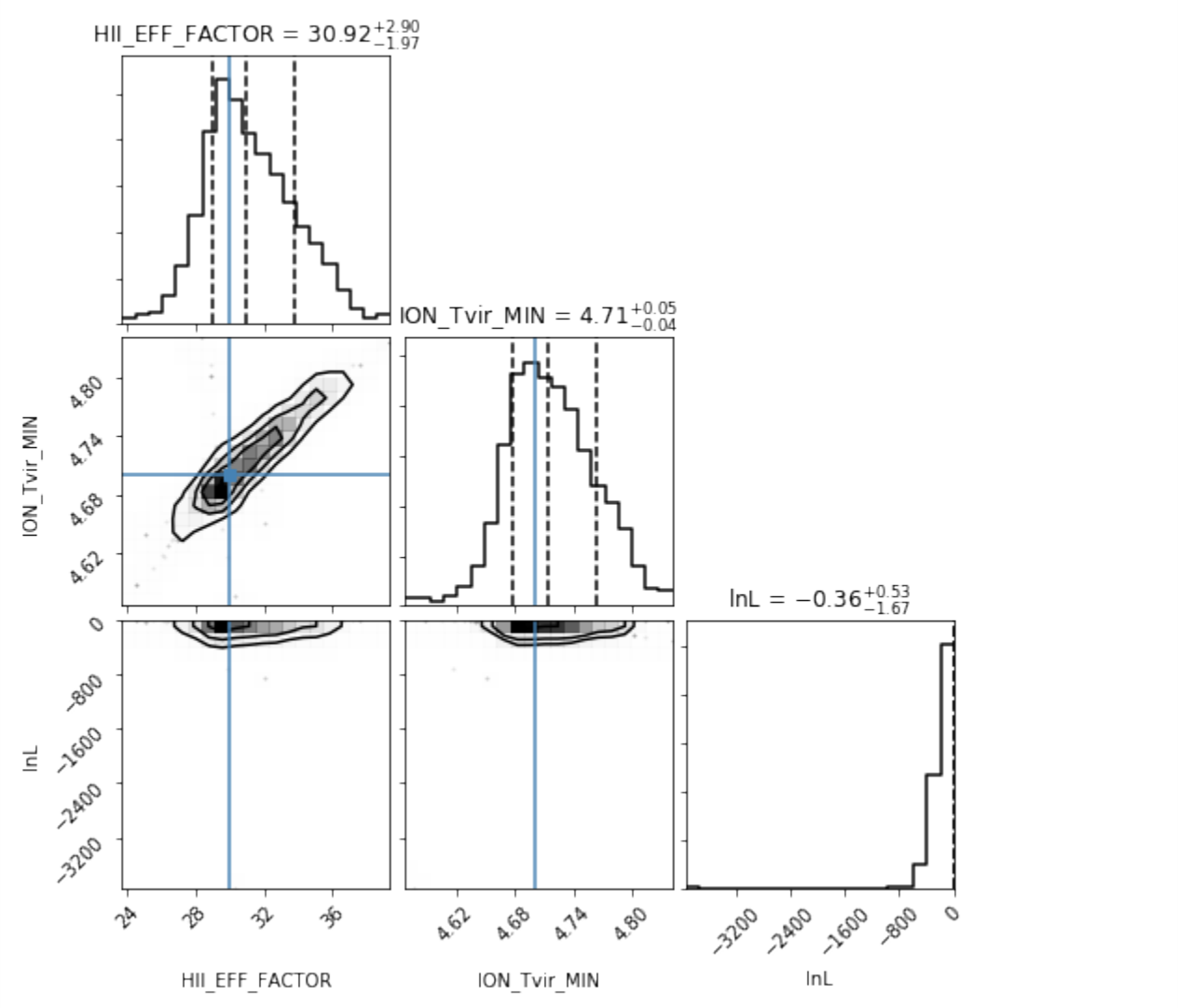
Often, for diagnostic purposes, the most useful plot to start with is the trace plot. This enables quick diagnosis of burnin time and walkers that haven't converged. The function in `py21cmmc` by default plots the log probability along with the various parameters that were fit. It also supports setting a starting iteration, and a thinning amount.

[7]:
```
analyse.trace_plot(samples, include_lnl=True, start_iter=0, thin=1, colored=False,␣
↪show_guess=True);
```



Corner Plot

[8]:
```
analyse.corner_plot(samples);
```



Using the Luminosity Function Likelihood

Another `Likelihood` which can be important in breaking degeneracies is the `LikelihoodLuminosityFunction`. Let's create the structure for this:

[ ]:
```
core_lf = mcmc.CoreLuminosityFunction(
    redshifts = [4, 5, 6],     # Generate luminosity functions at z=4,5,6
    user_params = core.user_params,
)

likelihood_lf = mcmc.LikelihoodLuminosityFunction(
        simulate=True
)
```

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

4.2.4 MCMC FAQ

This FAQ will cover several "advanced" topics that might come up in the context of doing MCMC with `py21cmmc`. Throughout we will use the following imports:

[2]:
```
from py21cmmc import mcmc, __version__
import logging

%matplotlib inline
import matplotlib.pyplot as plt

print("Using version ", __version__, " of 21CMMC.")
```

```
0.1.0
```

How to Continue Running a Chain After it has Stopped?

`21CMMC` has an inbuilt custom chain storage mechanism which uses `h5py`. The storage class itself is found here:

[2]:
```
from py21cmmc.mcmc.cosmoHammer.storage import HDFStorage
```

This class should not be required to be instantiated directly, however. In any case, this class has in-built mechanisms for detecting which iteration the chain is currently at, and saves the complete random state of the chain at each iteration. Thus, it can be started from wherever it stops (due to an exception, or walltime limits on a shared system). In practice, doing so is as easy as calling `run_mcmc` with the `continue_sampling` set to `True` (which it is by default). One just needs to ensure that the data directory and model name are consistent so that the previous chain can be located:

[3]:
```
core = mcmc.CoreCoevalModule(
    redshift = 7.0,
    user_params = {"HII_DIM":40, "BOX_LEN":80.0}
)

likelihood = mcmc.Likelihood1DPowerCoeval(simulate=True) # require simulate=True to␣
↪explicitly allow simulation
```

To see what is happening directly from the notebook, we're going to set the error level of the stream logger to INFO, so that the entire log file is also written to stdout:

[4]:
```
mcmc.run_mcmc(
    core, likelihood,
    params = {"HII_EFF_FACTOR":[30,20,40,3]},
    datadir='.', model_name='a_model',
    sampleIterations = 10,
    burninIterations = 0,
    walkersRatio = 2,
    log_level_stream = logging.INFO,
    continue_sampling=False
);
```

```
Initializing init and perturb boxes for the entire chain... done.
```

```
2018-08-07 10:40:35,639 INFO:Using CosmoHammer 0.6.1
2018-08-07 10:40:35,639 INFO:Using emcee 2.2.1
2018-08-07 10:40:35,644 INFO:all burnin iterations already completed
2018-08-07 10:40:35,645 INFO:Sampler: <class 'py21cmmc.mcmc.cosmoHammer.
↪CosmoHammerSampler.CosmoHammerSampler'>
configuration:
```

(continues on next page)

(continued from previous page)

```
  Params: [30]
  Burnin iterations: 0
  Samples iterations: 10
  Walkers ratio: 2
  Reusing burn in: True
  init pos generator: SampleBallPositionGenerator
  stop criteria: IterationStopCriteriaStrategy
  storage util: <py21cmmc.mcmc.cosmoHammer.storage.HDFStorageUtil object at␣
↪0x7f0955d2bc18>
likelihoodComputationChain:
Core Modules:
  CoreCoevalModule
Likelihood Modules:
  Likelihood1DPowerCoeval

2018-08-07 10:40:35,646 INFO:start sampling after burn in
2018-08-07 10:40:41,991 INFO:Iteration finished:10
2018-08-07 10:40:41,991 INFO:sampling done! Took: 6.3448s
2018-08-07 10:40:41,992 INFO:Mean acceptance fraction:0.65
```

Now let's continue running the chain until 20 iterations:

[5]:
```
mcmc.run_mcmc(
    core, likelihood,
    params = {"HII_EFF_FACTOR":[30,20,40,3]},
    datadir='.', model_name='a_model',
    sampleIterations = 20,
    walkersRatio = 2,
    burninIterations = 0,
    continue_sampling=True,
    log_level_stream=logging.INFO
);
```

```
Initializing init and perturb boxes for the entire chain... done.
```

```
2018-08-07 10:40:44,156 INFO:Using CosmoHammer 0.6.1
2018-08-07 10:40:44,157 INFO:Using emcee 2.2.1
2018-08-07 10:40:44,158 INFO:all burnin iterations already completed
2018-08-07 10:40:44,160 INFO:Sampler: <class 'py21cmmc.mcmc.cosmoHammer.
↪CosmoHammerSampler.CosmoHammerSampler'>
configuration:
  Params: [30]
  Burnin iterations: 0
  Samples iterations: 20
  Walkers ratio: 2
  Reusing burn in: True
  init pos generator: SampleBallPositionGenerator
  stop criteria: IterationStopCriteriaStrategy
  storage util: <py21cmmc.mcmc.cosmoHammer.storage.HDFStorageUtil object at␣
↪0x7f09394e6fd0>
likelihoodComputationChain:
Core Modules:
  CoreCoevalModule
Likelihood Modules:
  Likelihood1DPowerCoeval

2018-08-07 10:40:44,161 INFO:reusing previous samples: 10 iterations
2018-08-07 10:40:44,166 INFO:start sampling after burn in
```

(continues on next page)

(continued from previous page)

```
2018-08-07 10:40:49,616 INFO:Iteration finished:20
2018-08-07 10:40:49,616 INFO:sampling done! Took: 5.4498s
2018-08-07 10:40:49,617 INFO:Mean acceptance fraction:0.6
```

Note that continuing the sampling will only complete up to the number of iterations given, rather than *doing* that number of iterations on that particular call.

How to store arbitrary data on every iteration of the chain?

One may wish to store extra derived quantities along the way while the MCMC is performed. For example, to create an interesting visual of how a lightcone changes with varying MCMC parameters, one may wish to store a slice of the lightcone itself within the MCMC storage file. This can be done in both the `core` and `likelihood` steps of the computation.

Storage in the Core

The simplest way to store arbitrary data in the `core` module is to use the `store` keyword. This is a dictionary, where each key specifies the name of the resulting data entry in the samples object, and the value is a callable which receives the `context`, and returns a value from it.

This means that the context can be inspected and arbitrarily summarised before storage. In particular, this allows for taking slices of arrays and saving them. One thing to note is that the context is dictionary-like, but is not a dictionary. The elements of the context are only available by using the `get` method, rather than directly subscripting the object like a normal dictionary.

An example:

[6]:
```
storage = {
    "brightness_temp": lambda ctx: ctx.get('brightness_temp')[0].brightness_temp[0,:,:
↪]
}
```

[7]:
```
core = mcmc.CoreCoevalModule(
    redshift = 7.0,
    store = storage
)
```

[8]:
```
chain = mcmc.run_mcmc(
    core, likelihood,
    params = {"HII_EFF_FACTOR":[30,20,40,3]},
    datadir='.', model_name='storage_model',
    sampleIterations = 5,
    burninIterations = 0,
    walkersRatio = 2,
    continue_sampling=False
);
```

```
Initializing init and perturb boxes for the entire chain... done.
```

```
/home/steven/Documents/Projects/powerbox/powerbox/tools.py:106: UserWarning: One or␣
↪more radial bins had no cells within it.
  warnings.warn("One or more radial bins had no cells within it.")
```

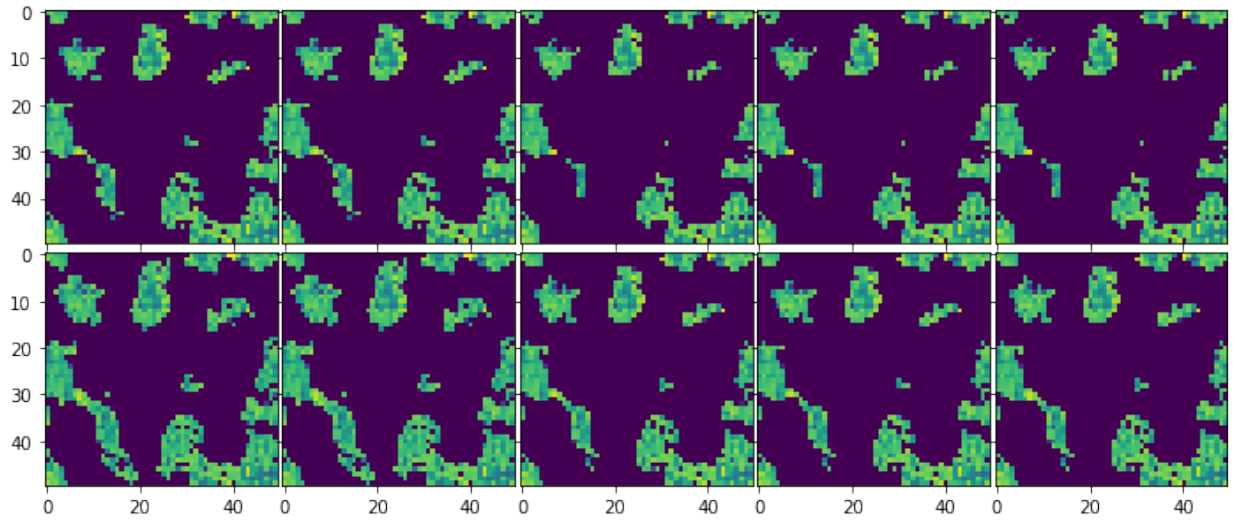
Now we can access the stored blobs within the samples object:

[9]:
```
blobs = chain.samples.get_blobs()
print(blobs.dtype)
```

```
[('brightness_temp', '<f4', (50, 50)), ('power', '<f8', (1, 14)), ('k', '<f8', (14,))]
```

We can also analyse or plot the blobs:

[10]:
```
fig, ax = plt.subplots(chain.samples.shape[0], chain.samples.iteration, figsize=(12,
↪5), sharex=True, sharey=True,
                       gridspec_kw={"hspace":0.02, 'wspace':0.02})

bt_slices = blobs['brightness_temp']

for i in range(chain.samples.iteration):
    for j in range(chain.samples.shape[0]): # n walkers
        ax[j,i].imshow(bt_slices[i,j])
```



Storage in the Likelihood

To store arbitrary data in the `Likelihood` components, a dedicated `store()` method exists. To modify what is stored on each iteration, this method should be overwritten. The `store` method takes two arguments: `model` and `storage`. The first is exactly the output of the `simulate` method of the `Likelihood`. It is usually either a dictionary of quantities or a list of such dictionaries. The latter is the storage container which will be internally accessed to write out the quantities. This just needs to be updated with any quantity of interest. For example, let's imagine the built-in `Likelihood1DPowerLightcone` had no defined storage method, and we wanted to store *everything* in its model (which currently consists of `k` and `delta`). We would do the following:

[ ]:
```
class MyLCModule(core.CoreLightConeModule):
    def store(self, model, storage):
        for i, m in enumerate(model):
            storage.update({k + "_%s" %i : v for k, v in m.items()})
```

(this is, of course, the actual definition of the method in the source code). The `blobs` in the stored chain will now include the entries `k` and `delta` for each iteration of the chain.

How to add thermal noise to observations/mocks?

By default, both the coeval and lightcone likelihoods provided in 21CMMC do not include any thermal noise or sample variance. The only noise used by default is the modelling uncertainty. One way to add noise is to subclass an existing likelihood and over-write the `computeLikelihood` method, which can then be implemented with artbirary noise characteristics. We urge this point because 21CMMC is written to be *extensible* in this manner.

However, there is an easier way to add thermal-like noise to the two basic likelihoods provided in 21CMMC. Both contain a `noisefile` argument. This should be an existing file, in a numpy `.npz` format, with *at least* the arrays "ks" and "errs" defined within it. **OR**, it can be a list of such files, one for each coeval box or redshift chunk. This format is precisely what is written by `21cmSense`, and so that can be natively used to compute the noise characteristics. However, we stress that any code that produces noise characteristics can be used, and the output simply reformatted into this simple format.

If no such file is provided, the likelihood will assume that no special noise characteristics are being used. There is one caveat to this. If `simulate=True`, and a `define_noise` method has been defined on the class, then this will be used to produce the noise properties on-the-fly. It must produce a list of dictionaries, each of which contains the arrays "ks" and "errs", as above, unless the derived class also over-writes the `_write_noise` method and the `computeLikelihood` method which use this data structure. If a noisefile *is* provided but does not exist, and the noise has been simulated in this way, it will instead be *written* to the provided file.

Finally, if for some reason it is not possible to re-format your noise data, you can subclass the likelihood and overwrite the `_read_noise` method. This method should iterate through a list of files, and return a list of dicts containing the arrays "ks" and "errs".

As an example of using a noise file in the default manner:

[6]:
```
core = mcmc.CoreCoevalModule(
    redshift = 7.0,
    user_params = {"HII_DIM":40, "BOX_LEN":80.0}
)

likelihood = mcmc.Likelihood1DPowerCoeval(
    noisefile = "hera127.drift_mod_0.135.npz", # File produced with 21cmSense
    simulate=True
)
```

[8]:
```
mcmc.run_mcmc(
    core, likelihood,
    params = {"HII_EFF_FACTOR":[30,20,40,3]},
    datadir='.', model_name='a_noise_model',
    sampleIterations = 10,
    burninIterations = 0,
    walkersRatio = 2,
    log_level_stream = logging.INFO,
    continue_sampling=False
);
```

```
Initializing init and perturb boxes for the entire chain...
```

```
2018-09-03 12:12:42,279 INFO:Using CosmoHammer 0.6.1
2018-09-03 12:12:42,279 INFO:Using emcee 2.2.1
2018-09-03 12:12:42,283 INFO:all burnin iterations already completed
2018-09-03 12:12:42,285 INFO:Sampler: <class 'py21cmmc.mcmc.cosmoHammer.
↪CosmoHammerSampler.CosmoHammerSampler'>
configuration:
  Params: [30]
  Burnin iterations: 0
```

(continues on next page)

(continued from previous page)

```
  Samples iterations: 10
  Walkers ratio: 2
  Reusing burn in: True
  init pos generator: SampleBallPositionGenerator
  stop criteria: IterationStopCriteriaStrategy
  storage util: <py21cmmc.mcmc.cosmoHammer.storage.HDFStorageUtil object at␣
↪0x7fe137d044a8>
likelihoodComputationChain:
Core Modules:
  CoreCoevalModule
Likelihood Modules:
  Likelihood1DPowerCoeval

2018-09-03 12:12:42,286 INFO:start sampling after burn in
```

```
 done.
['burnin', 'sample_0']
```

```
2018-09-03 12:12:46,359 INFO:Iteration finished:10
2018-09-03 12:12:46,359 INFO:sampling done! Took: 4.0724s
2018-09-03 12:12:46,360 INFO:Mean acceptance fraction:0.4
```

How does 21CMMC deal with parameters outside their range?

The default behaviour of `cosmoHammer` is to immediately return `-np.inf` if the sample parameters on a given iteration/walker are outside their bounds (bounds are specified by the `params` argument to `run_mcmc`). A log-likelihood of `-inf` means that the likelihood is zero, and the sampler will not accept that position. This is completely self-consistent and is the recommended procedure in `emcee` (on which `cosmoHammer` is based).

However, in `21CMMC`, this can lead to significant inefficiencies due to load balancing. If a process returns instantly, but the MCMC step blocks until all walkers are finished, then that particular processor/thread is essentially idle, and the overall efficiency goes down.

To circumvent this, in `21CMMC` we over-load the `emcee.EnsembleSampler` class, so that new new samples are vetted *before* passing them to the likelihood function. If a new sample position is outside the range, it is independently re-sampled until it is within range. There is a failsafe `max_attempts` parameter which will shut down this loop of resampling if no suitable parameters can be found (by default it is set to 100). This means that all processes will be running close to 100% of the time, increasing the efficiency of the sampling.

What parameter ranges should I use?

My run crashed with BrokenProcessPool exception, what is that?

The `BrokenProcessPool` exception is a special exception that catches anything that goes *really* wrong in the underlying C-code – like a segfault. When this happens, it's basically impossible for Python to know what really went wrong, since these errors are really errors in the compiled code itself and do not conform to standard exceptions. However, this exception gives Python a way to report to you *something* about the error.

We've overloaded the `emcee` sampler to know about this error and to report about which parameters were being evaluated at the time that something broke. It cannot tell which parameters were actually at fault, but if you test each one of them in turn, you may be able to identify the problem.

Using this information in tandem with DEBUG mode can be particularly helpful.

My run crashed and I don't know why, can I debug?

There are several ways to debug the runtime behaviour of 21CMMC. While for developers an explicit DEBUG mode is posible, as a user it is much more likely that you will just want to inspect the logging output in order to identify a potential problem. We highly recommend sending any errors through to the development team via github (https://github.com/BradGreig/Hybrid21cm/issues/) and including this logging ouput. So, how to get the logging output?

There are actually three different logs running on any given MCMC run:

1. The cosmoHammer log. This gets written to the `<filePrefix>.log` file, but can also be written to stderr/stdout. It contains information about the computation chain, and keeps a running report of the progress of the sampler. To set which level of logging it reports at, you can specify either/both of `logLevel` and `log_level_stream` in `run_mcmc`. The first changes the level of the file, and the latter the output to screen. It can be set to a string identifier such "DEBUG", or an integer, where the *lower* the integer the more will be printed. See https://docs.python.org/3/library/logging.html#logging-levels for details. A good idea is to set to "DEBUG" when debugging, clearly.
2. The 21CMMC log. This is the log of the internal wrapper functions for 21cmFAST, and also the Core's and Likelihoods. By default, it has only one handler (it writes to screen). Much more information will be printed out if this is set to "DEBUG". Note that these logs can be manipulated using the standard python logging module. However, as a convenience, to just change the level (from its default of WARNING) in the MCMC, you can pass `log_level_21CMMC` to `run_mcmc`.
3. The C-code logs. These logs are printed to stderr. By default, the level here is WARNING. However, due to extensive debugging developer-side, there is a lot of information that is logged if the level is DEBUG (or even SUPER_DEBUG or ULTRA_DEBUG!). To change these levels, you need to re-install the package, using the environment variable `LOG_LEVEL=DEBUG`.

Our advice is to set all of these levels to DEBUG (and maybe SUPER_DEBUG) for the C-code, if your code is crashing there) when something goes awry.

Can I get a history of all parameters trialed in the MCMC?

While the sampler object returned by `run_mcmc` will contain a full chain of parameter samples from your posterior, it is sometimes helpful to keep a running history of *all* parameters that were tried, whether they were accepted or not. At this time, we don't offer a simple function for doing this, but it can be reasonably easily achieved by inspecting the logs. If you set `log_level_21CMMC` to `INFO` or less, then the logs will contain entries of `New Positions: <params>` for each set of parameters trialed. If you have redirected your log to a file, you can get a list of trialed parameters using something like the following script:

```
[ ]: import json
params = []

with open("logfile") as f:
    for line in f:
        if "New Positions:" in line:
            s = line.split(":")[-1]
            params += json.loads(s)
```

Can I interactively explore my Likelihood after running MCMC?

Sometimes, especially when first building a new likelihood, you may run an MCMC chain, and realise that it doesn't quite look like what you expect. The easiest way to explore what may have gone wrong would be to re-create the exact same `LikelihoodComputationChain` and interactively produce the likelihood calculations for a few parameters yourself.

Fortunately, that exact behaviour is provided by the output file ".LCC.yml". Using the `yaml` module built in to `21CMMC` (which has a few extra representers/constructors than the default one from `pyyaml`) you can literally read this file in and it will be the entire `LikelihoodComputationChain` object, which you can explicitly call with new parameters, or otherwise investigate.

One word of caution for those writing their own Cores/Likelihoods: the YAML loader/dumper is *slow* and is not intended for dealing with large datasets. For this reason, cores and likelihoods work best when their attributes are small in size (eg. the CoreCoevalModule does not save any of simulated data – it merely saves the parameters so that it can be instantly read in from cache at any time).

To partially mitigate this, the YAML file that is written when calling `run_mcmc` contains the *pre-setup* class, which should fully define the chain, but often does not have much of the data loaded in. THis choice however means that to use the chain interactively after it is read in, you are required to manually call `setup()`. Alternatively, one can use the convenience function in `analyse.py` to read in the file.

Note that this `.LCC.yml` file is internally used to ensure that when you try to "continue" running a chain that all of the parameters exactly line up so that nothing inconsistent happens. It is saved *before* running the MCMC, so none of the actual chain parameters are saved into it.

YAML files are meant to be (mostly) human-readable, so you can also just look at it to determine what your setup was long after running it.

An example of how to read in the file interactively:

[ ]:
```
from py21cmmc import yaml

with open("MyBigModel.LCC.yml") as f:
    lcc = yaml.load(f)

# Call setup to set up the chain properly (this adds data/noise etc.)
lcc.setup()

lnl, blobs = lcc(<parameters>)

core_module = lcc.getCoreModules()[0]
core_module.do_something()
```

Alternatively, one could do the following:

[ ]:
```
from py21cmmc.analyse import load_primitive_chain

lcc = load_primitive_chain("MyBigModel")

# No set-up required.

lnl, blobs = lcc(<parameters>)
```

4.2.5 Allowed parameter ranges

The version of 21cmFAST that comes with `21CMMC` uses numerous interpolation tables. As such, several parameter combinations/choices may have restricted ranges. This, in addition to limits set by observations restricts the ranges for several of the astrophysical parameters that are available within `21CMMC`.

Cosmology

At the present only flat cosmologies are allowed as some in-built cosmological functions do not allow for non-flat cosmologies. This will likely be remedied in the near future. Thus, only the dark matter energy density (Ω_m) is available to be varied as $\Omega_\Lambda = 1 - \Omega_m$ is enforced. In all cases, sensible ranges are restricted by Planck and other cosmological probes. For example, see Kern et al., 2017 for varying cosmology with `21CMMC`.

- Ω_m - should be allowed to vary across (0, 1]
- Ω_b - should be allowed to take any non-zero value
- h - should be allowed to take any non-zero value
- σ_8 - should be allowed to take any non-zero value. Extreme values may cause issues in combination with the astrophysical parameters governing halo mass (either M_{min} or $T_{min,vir}$)
- n_s - should be allowed to take any non-zero value

Ionisation astrophysical parameters

Allowed astrophysical parameter ranges are determined by which parameter set is being used. Dictated by the choice of `USE_MASS_DEPENDENT_ZETA` in the `FlagOptions` struct.

If `USE_MASS_DEPENDENT_ZETA = False` then the user is selecting the older astrophysical parameterisation for the ionising sources. These include any of: (i) ζ - the ionising efficiency (ii) R_{mfp} - minimum photon horizon for ionising photons or (iii) $T_{min,vir}$ - minimum halo mass for the ionising sources, $T_{min,vir}$.

- ζ - Largest range used thus far was [5,200] in Greig & Mesinger, 2015. In principle it can be less than 200 or even extended beyond 200. Going below 5 is also plausible, but starts to cause numerical issues.
- R_{mfp} - Only if `INHOMO_RECO=False`. Typically [5,20] is adopted. Again, going below 5 Mpc cause numerical problems, but any upper limit is allowed. However, beyond 20 Mpc or so, R_{mfp} has very little impact on the 21cm power spectrum.
- $T_{min,vir}$ - Typically [4,6] is selected, corresponding to 10^4 - 10^6 K corresponding to atomically cooled haloes. It becomes numerically unstable to go beyond 10^6 K owing to interpolation tables with respect to halo mass. It is possible to consider $< 10^4$ K, however, in doing so it will generate a discontinuity. Internally, $T_{min,vir}$ is converted to M_{min} using Equation 26 of Barkana and Loeb, 2001 whereby the mean molecular weight, μ differs according to the IGM state ($\mu = 1.22$ for $< 10^4$ K and $\mu = 0.59$ for $> 10^4$ K)

If `USE_MASS_DEPENDENT_ZETA = True` then the user is selecting the newest astrophysical parameterisation allowing for mass dependent ionisation efficiency as well as constructing luminosity functions (e.g. Park et al., 2019). This allows an expanded set of astrophysical parameter including: (i) $f_{*,10}$ - star formation efficiency normalised at $10^{10} M_\odot$ (ii) α_* - power-law scaling of star formation efficiency with halo mass (iii) $f_{esc,10}$ - escape fraction normalised at $10^{10} M_\odot$ (iv) α_{esc} - power-law scaling of escape fraction with halo mass (v) M_{turn} - Turn-over scale for the minimum halo mass (vi) t_{star} - Star-formation time scale and (vii) R_{mfp} - minimum photon horizon for ionising photons.

- $f_{*,10}$ - Typically [-3., 0.] corresponding to a range of $10^{-3} - 1$. In principle $f_{*,10}$ can exceed unity, as it is the normalisation at $10^{10} M_\odot$ and would depend on the power-law index, α_* as to whether or not the star-formation efficiency exceeds unity. However, probably no need to consider this scenario.

- α_* - Typically [-0.5,1]. Could be modified for stronger scalings with halo mass.
- $f_{esc,10}$ - Typically [-3,0.] corresponding to a range of $10^{-3} - 1$. In principle $f_{esc,10}$ can exceed unity, as it is the normalisation at $10^{10} M_\odot$ and would depend on the power-law index, α_{esc} as to whether or not the escape fraction exceeds unity. However, probably no need to consider this scenario.
- α_{esc} - Typically [-1.0,0.5]. Could be modified for stronger scalings with halo mass.
- M_{turn} - Typically [8,10] corresponding to a range of 10^8 - $10^{10} M_\odot$. In principle it could be extended, though less physical. To have $M_{turn} > 10^{10} M_\odot$ would begin to be inconsistent with existing observed luminosity functions. Could go lower than $M_{turn} < 10^8 M_\odot$ though it could begin to clash with internal limits in the code for interpolation tables (which are set to $M_{min} = 10^6 M_\odot$ and $M_{min} = M_{turn}/50$.
- t_{star} - Typically (0,1). This is represented as a fraction of the Hubble time. Thus, cannot go beyond this range
- R_{mfp} - same as above

Heating astrophysical parameters

For the epoch of heating, there are three additional parameters that can be set. These, can only be used if `USE_TS_FLUCT=True` which performs the heating. These include: (i) $L_{X<2keV}/SFR$ - the soft-band X-ray luminosity of the heating sources (ii) E_0 - the minimum threshold energy for X-rays escaping into the IGM from their host galaxies and (iii) α_X - the power-law spectral index of the X-ray spectral energy distribution function.

- $L_{X<2keV}/SFR$ - Typically [38, 42], corresponding to a range of $10^{38} - 10^{42} erg\, s^{-1} M_\odot^{-1} yr$. This range could easily be extended depending on the assumed sources. This range corresponds to high mass X-ray binaries.
- E_0 - [100, 1500]. Range is in eV corresponding to 0.1 - 1.5 keV. Luminosity is determined in the soft-band (i.e. < 2 keV), thus wouldn't want to expand this upper limit too much. Observations limit the lower range to ~0.1 keV.
- α_X - Typically [-2.,2] but depends on the population of sources being considered (i.e. what is producing the X-ray's). Note, the X-ray SED is defined as $\propto \nu^{-\alpha_X}$

4.2.6 Writing Cores and Likelihoods

[4]:
```
import py21cmmc as p21c
from py21cmmc import mcmc
```

[2]:
```
p21c.__version__
```

[2]:
```
'0.1.0'
```

4.2.7 How do I write a Core Module?

One of main aims of `21CMMC` is to be *extensible*, so that the basic capabilities provided can be easily applied to new situations. For example, one use-case which may require a custom `Core` module (the concept of `Core` and `Likelihood` modules is introduced in the MCMC Introduction tutorial) is where the "observation" which is being analyzed is not a theoretical simulation box, but a set of interferometric visibilities at predefined baselines.

Since our philosophy is that a `Core` module should "construct" a model such that it approximates the pre-reduced "observations", the conversion of the 21cmFAST cube into a set of visibilities should be defined in a new `Core`.

In principle, the `Core` module does not need to be subclassed from any particular class, so long as it implements a minimal API. However, we recommend *always* subclassing from `py21cmmc.mcmc.CoreBase`, which sets default

methods for several of these API components (some of which should almost never have to be changed). For a vanilla `Core` module, the following methods/attributes are highly recommended to be over-ridden/defined:

- `__init__`: this is the only place that user-input can be obtained. This method should almost certainly just save input parameters to the instance with the same name, modulo setting default parameters and simple logic.
- `build_model_data(ctx)`: this is the heart of the `Core`. It is what produces the actual model quantities, which are saved back to the `ctx` object. The current MCMC parameters are available via `ctx.getParams()`, and the model quantities should be saved back to the `ctx` via `ctx.add("key", value)`. These model quantities should be *deterministic*, as they convey a model, not a mock. They may however include quantities of interest for determining probability *distributions* on the model.

The following methods have defaults, but may be required to be over-ridden for particular applications:

- `setup`: this method, if defined, is run once only (if embedded in a `Chain`, the `Chain` checks if it has been previously run and disallows running again) before the beginning of an MCMC. It receives no parameters. In this sense, it is *almost* equivalent to tacking the operations onto the end of `__init__`. The difference is that when `setup` is run, it is generally assumed that the `core` has been embedded into a `Chain`, so that access to other cores (and, if loaded earlier in the core sequence, their respective instance attributes) is available.
- `convert_model_to_mock(ctx)`: this method takes the model output from `build_model_data`and computes a mock simulation from it (i.e. it adds the requisite randomness). If not over-ridden, it is a no-op, which implies that data is considered to be deterministic (as far as this core goes). The method is not invoked in a standard MCMC run, but can be used to generate mock data for consistency tests.
- `__eq__(self, other)`: this method should determine if this `core` instance is *identical* to another `core` instance. It is used for checking whether the an MCMC chain can be continued from file (i.e. if it is consistent with what has already been run). It is defined in `CoreBase` by checking each of the input parameters to `__init__`, as saved in each instance either as its own name, or its name prefaced by "_". This list of instance attributes can be supplemented using the `extra_defininingattributes` class attribute, and can be filtered with the `ignore_attributes` class attribute. It is probably rare that the `__eq__` method should be required to be overwritten; usually these class attributes should be all that is required.

We can see the various methods/attributes made available by default:

[6]:
```
cls = mcmc.CoreBase()
help(cls)
```

```
Help on CoreBase in module py21cmmc.mcmc.core object:

class CoreBase(ModuleBase)
 |  CoreBase(store=None)
 |
 |  Method resolution order:
 |      CoreBase
 |      ModuleBase
 |      builtins.object
 |
 |  Methods defined here:
 |
 |  __call__(self, ctx)
 |      Call the class. By default, it will just build model data, with no␣
↪stochasticity.
 |
 |  __init__(self, store=None)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  build_model_data(self, ctx)
 |      Passed a standard context object, should construct model data and place it in␣
↪the context.
```

(continues on next page)

(continued from previous page)

```
 |
 |      The data generated by this method should ideally be *deterministic*, so that␣
↪input parameters (which are
 |      inherently contained in the `ctx` object) map uniquely to output data. The␣
↪addition of stochasticity in order
 |      to produce mock data is done in the :meth:`~convert_model_to_mock` method.␣
↪All data necessary to full evaulate
 |      probabilities of mock data from the model data should be determined in this␣
↪method (including model
 |      uncertainties, if applicable).
 |
 |      Parameters
 |      ----------
 |      ctx : dict-like
 |          The context, from which parameters are accessed.
 |
 |      Returns
 |      -------
 |      dct : dict
 |          A dictionary of data which was simulated.
 |
 |  convert_model_to_mock(self, ctx)
 |      Given a context object containing data from :meth:`~build_model_data`,␣
↪generate random mock data, which should
 |      represent an exact forward-model of the process under investigation.
 |
 |      Parameters
 |      ----------
 |      ctx : dict-like
 |          The context, from which parameters and other simulated model data can be␣
↪accessed.
 |
 |  prepare_storage(self, ctx, storage)
 |      Add variables to special dict which cosmoHammer will automatically store with␣
↪the chain.
 |
 |  simulate_mock(self, ctx)
 |      Generate all mock data and add it to the context.
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from ModuleBase:
 |
 |  __eq__(self, other)
 |      Return self==value.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from ModuleBase:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
 |
 |  chain
 |      A reference to the LikelihoodComputationChain of which this Core is a part.
 |
```

(continues on next page)

(continued from previous page)

```
|  parameter_names
|
|  ----------------------------------------------------------------------
|  Data and other attributes inherited from ModuleBase:
|
|  __hash__ = None
|
|  extra_defining_attributes = []
|
|  ignore_attributes = []
```

A Visibility Core Example

Back to our proposed example of creating a core which evaluates visibilities. There are two ways of doing this. The first is to subclass an original `Core` class from `21CMMC` (in this case, the `CoreLightConeModule` will be required).

A minimal example would then be:

[ ]:
```python
class MyVisibilityCore(core.CoreLightConeModule):
    def __init__(self, frequencies, baselines,
                 *args, **kwargs # always include *args, **kwargs and call super()
                ):
        # Call super to initialise standard lightcone module
        super().__init__(*args, **kwargs)

        # Add other user-dependent quantities
        self.frequencies = frequencies
        self.baselines = baselines

    def build_model_data(self, ctx):
        # Call the LightConeModule model builder, and add its model to the context.
        super().build_model_data(ctx)

        # Convert the lightcone into visibilities
        vis = convert_lightcone_to_vis(ctx.get("lightcone"))

        # Importantly, add useful quantities to the context, so they are available
        # to the likelihood
        ctx.add("visibilities", vis)

        # Also could clean up any quantities that we don't care about anymore.
        # But it might just be better to leave it there.
        ctx.remove("lightcone")
```

Another method hinges on the fact that multiple cores can be loaded for a given MCMC chain. In this method, the defined `Core` is entirely separate from the `LightConeModule`, but both must be explicitly loaded:

[ ]:
```python
class MyVisibilityCore(mcmc.CoreBase):
    required_cores = [mcmc.CoreLightConeModule]

    def __init__(self, frequencies, baselines, *args, **kwargs):
        super().__init__(*args, **kwargs)
```

(continues on next page)

(continued from previous page)

```
        self.frequencies = frequencies
        self.baselines = baselines

    def build_model_data(self, ctx):
        lightcone = ctx.get("lightcone") # Assumes that the LightConeModule is loaded␣
↪before this one.

        # Convert the lightcone into visibilities
        vis = convert_lightcone_to_vis(lightcone)

        # Importantly, add useful quantities to the context, so they are available
        # to the likelihood
        ctx.add("visibilities", vis)
```

Which of these is preferred is difficult to determine. The first option is simpler, in that it uses well-worn methods for overwriting components of the original. It can also be generally assumed that if the visibilities are to be used, the original simulation will not be required. It also means that when it comes to actually running the MCMC, only the `MyVisibilityCore` will need to be loaded. On the other hand, this method requires passing parameters to `MyVisibilityCore` that actually are just passed through to `CoreLightConeModule`, which is perhaps a bit unclean.

TLDR; either will work.

We could supplement either of the above with an additional `convert_model_to_mock` method, which might add additional complex Gaussian noise to each visibility.

4.2.8 How do I write a Likelihood Module?

If you have read the "Write My Own Core" section just above, then you know how important *extensibility* is to `21CMMC`. This is especially true when it comes to the `Likelihood` modules (a brief intro to the idea behind cores and likelihoods is given in the MCMC intro).

Likelihoods *should* inherit from `py21cmmc.mcmc.BaseLikelihood`, or if you want a simple way to support reading in data and noise from files, `py21cmmc.mcmc.BaseLikelihoodFile`.

Along with the standard `__init__` and `setup` methods which have very similar functions to their `Core` counterparts (see previous section), it is recommended to overwrite the following methods:

- `reduce_data(ctx)`: this takes the data produced from all cores, and reduces it to "final form". This "final form" is essentially defined as the maximally reduced form that can be applied to either data or model *separately*, before calculating the likelihood from their combination. For instance, it may be to perform an FFT to obtain the power spectrum. It returns the reduced data as a dictionary of quantities.
- `computeLikelihood(model)`: this function takes the model dictionary output by `reduce_data` and computes a likelihood from it, using the `self.data` attribute of the instance. The reason for separating these two methods is simple: it allows applying the `reduce_data` method on both data and model, as well as simulating mock datasets.
- `store(model, storage)`: a method which is called on every MCMC iteration, and passed the model output from `reduce_data`. It should save this data to a storage dictionary, and it will subsequently be saved to the storage chain file.

If using the `BaseLikelihoodFile` class, an extra set of methods are available to be over-ridden which define how data and measurement noise should be read in from file. See the docstring or the FAQ on adding noise for details on these methods.

The simplest example of a likelihood would be something like the following:

```
[ ]: class MySimpleLikelihood(mcmc.LikelihoodBaseFile):
         def reduce_data(self, ctx):
             k, power_spectrum = convert_to_power(ctx.get("lightcone"))
             return dict(power_spectrum=power_spectrum, k=k)

         def computeLikelihood(self, model):
             return np.sum((self.data["power_spectrum"] - model['power_spectrum'])**2 / (2␣
     ↪* self.noise(['variance'])))
```

Since we are using the `LikelihoodBaseFile`, the data is automatically read in from a user-input file. The default read method assumes that the file is a numpy format (`.npz`), and contains exactly the same quantities that are returned by `reduce_data` (which allows very easily for simulating mock data and outputting to file, and subsequently reading it back in).

The `self.data` attribute is defined during `setup`, and filled with the contents of the input data file, and likewise the `self.noise` attribute (it comes from a separate noisefile).

4.3 API Reference

4.3.1 py21cmmc

`py21cmmc.mcmc`
`py21cmmc.core`
`py21cmmc.likelihood`
`py21cmmc.analyse`
`py21cmmc.cosmoHammer`

4.4 Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

4.4.1 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.

4.4.2 Documentation improvements

21CMMC could always use more documentation, whether as part of the official 21CMMC 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!

4.4.3 High-Level Steps for Development

This is an abbreviated guide to getting started with development of 21CMMC, focusing on the discrete high-level steps to take. See our notes for developers for more details about how to get around the 21CMMC codebase and other technical details.

There are two avenues for you to develop 21CMMC. If you plan on making significant changes, and working with 21CMMC 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/21CMMC.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/21CMMC.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 ~/21cmmc
source ~/21cmmc/bin/activate
```

   or:

```
conda create -n 21cmmc python=3
conda activate 21cmmc
```

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 Authors

`21CMMC` was originally written by Brad Greig. The updated interface that uses the Python-wrapped `21cmFAST` was primarily developed by Steven Murray. It is maintained by the full `21cmFAST` collaboration.

Here follows a list of individuals who have contributed in any way to `21CMMC`:

- Brad Greig
- Steven Murray
- Andrei Mesinger
- Catherine Watkinson

- Yuxiang Qin
- Jaehong Park
- Tom Binnie

4.6 Changelog

4.6.1 v1.0.0dev

- More fleshed-out interface to cosmoHammer, with base classes abstracting some common patterns.
- New likelihoods and cores that are able to work on any data from the `21cmFAST` pipeline.
- Better logging
- Better exception handling
- pip-installable
- Documentation
- Pipenv support
- Full code formatting applied

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search