EPIC Documentation

Release 1.0.4

Rafael J. F. Marcondes

May 19, 2018

Contents

1 Welcome to the EPIC user's guide!

Easy Parameter Inference in Cosmology (EPIC) is my implementation in Python of a MCMC code for Bayesian inference of parameters of cosmological models and model comparison via the computation of Bayesian evidences.

1.1 Why EPIC?

I started to develop EPIC as a means of learning how inference can be made with Markov Chain Monte Carlo, rather than trying to decipher other codes or using them as black boxes. The program has fulfilled this purposed and went on to incorporate a few cosmological observables that I have actually employed in some of my publications. Now I release this code in the hope it can be useful for students to learn some of the methods used in Observational Cosmology and even use it for their own work. It still lacks some important features. A Boltzmann solver is not available. It is possible that I will integrate it with CLASS[1] to make it more useful for more advanced research. At this moment it can be used very well for comparisons using background data alone or maybe some perturbative aspects that can be calculated in a simple way (for example the approximation of the growth rate in some models). Stay tuned for more. Meanwhile, enjoy these nice features:

- The code is compatible with both Python 2 and Python 3 (maybe not with Python below 2.7 or 2.6), in any operating system.
- It uses Python's `multiprocessing` library for evolution of chains in parallel. The separate processes can communicate with each other through some `multiprocessing` utilities, which made possible the implementation of the Parallel Tempering algorithm. This method is capable of detecting and accurately sampling posterior distributions that present two or more separated peaks.
- Convergence between independent chains is tested with the multivariate version of the Gelman and Rubin test, a very robust method.
- Also, the plots are beautiful and can be customized to a certain extent directly from the command line, without having to change the code. You can view triangle plots with marginalized distributions of parameters, derived parameters, two-dimensional joint-posterior distributions, autocorrelation plots, cross-correlation plots, sequence plots, convergence diagnosis and more.

Besides, of course it can be altered to your needs A few changes can be made to the code so you can constrain other models and/or use different datasets. In this case consider cloning the repository with git so you can leverage all the version control tools and even contribute to this project.

The main changes will be in the `observables` module. This is where the physical observables are calculated. There is a general function for computing the Hubble rate as function of the redshift (or

[1] Lesgourgues, J. "The Cosmic Linear Anisotropy Solving System (CLASS) I: Overview". arXiv:1104.2932 [astro-ph.IM]; Blas, D., Lesgourgues, J., Tram, T. "The Cosmic Linear Anisotropy Solving System (CLASS). Part II: Approximation schemes". JCAP07(2011)034

the scale factor converted to redshift), with conditionals for the different models. For some cases a numeric integration is done with Runge-Kutta. The simplified JLA binned data only depends on distances, for this case just properly including your model calculation in the function `Eh`, that is, $hE(z) = H(z)/100$ or `H` will be sufficient. BAO calculations also depend mainly on the Hubble rate but you should pay attention to the necessary density parameters too. CMB shift parameters is similar. In any case, this is the place where you should introduce your calculations. You will choose a label for your model, which should be used in you `.ini` file. This label becomes the attribute `model` of your `Cosmology` class object. Insert the free parameters (and their priors) accordingly and their LaTeX representation at the end of the file for the plots.

If you want to use other data, you can look at the ones provided to see the format used. There are a few different ways. Standard Gaussian likelihoods generally use the function `simple`, if the data contain only the measurements and their uncertainties, or the function `matrixform` if there is a covariance matrix. Of course you can create a new one if these are not adequate for the specific needs. Then you just need to tell which function this dataset uses in the dictionary `allprobes` at the end of the file. The `load_data` module contains functions to read the data files. You will have to create one there too. Detailed explanations are given at the end of the sections about the *data* and the *models*.

Try it now!

2 How to install

There are two ways to install this program. You can download and install from PyPi or you can clone it from BitBucket. But first, it is recommended that you make these changes inside a virtual environment.

2.1 Setting up a virtual environment

if you are on Python 3.3 or superior, you can run:

```
$ python3 -m venv vEPIC
```

to create a virtual Python environment inside a folder named `vEPIC`. Activate it with:

```
$ source vEPIC/bin/activate
```

When you finish using the environment and want to leave it you can just use `deactivate`. To activate it again, which you need in a new session, just run the command above, you do not need to create it again. More details about Python3's venv here.

With inferior versions of Python, you can install pyenv and pyenv-virtualenv, which let you create a virtual environment and even choose another versionof Python to install. This is done with:

```
$ pyenv virtualenv 3.6.1 vEPIC # or other version you like.
```

Then activate it running:

```
$ pyenv activate vEPIC # use pyenv deactivate vEPIC to deactivate it.
```

Your virtual environment name will appear in front of your username at the beginning of the line to indicate that the environment is activated.

2.2 Installing with pip

The easiest way to install this program is to get it from PyPi. It is also the most recommended since it can be easily updated when new versions come out. Inside your virtual environment, run:

```
$ pip install epic-code
```

Then, if you are using Linux, create a bind mount from your python path to an external directory for your convenience. For example, in your home, create the directories `cosmology_codes/EPIC` and `cosmology_codes/EPIC-simulations`. Run:

```
$ sudo mount --bind $PYTHONPATH/lib/python3.5/site-packages/EPIC/ cosmology_
↪codes/EPIC
```

This will make your newly created folder reflect the contents of the installation directory and you will be able to run the scripts and access the associated files from there, rather than only importing the modules from inside the python interactive interpreter. If you do not have the privileges to mount the directory then follow the instructions below for downloading and extracting manually to your home directory.

Note: If you are on Mac OS X or macOS, you can achieve the same effect but will need to install osxfuse and bindfs to create the link:

```
$ brew install osxfuse # or brew cask install osxfuse
$ brew install bindfs
$ bindfs $PYTHONPATH/lib/python3.5/site-packages/EPIC/ cosmology_codes/EPIC
```

Next, `cd` into `cosmology_codes/EPIC`, run:

```
$ python define_altdir.py
```

and pass the full path to `cosmology_codes/EPIC-simulations` to define this folder as the location for saving simulations results. You are now good to go. To check and install updates if available, just run:

```
$ pip install --upgrade epic-code
```

On a Windows computer, if you have Windows 10 Pro, I recommend enabling the Windows Subsystem for Linux and installing Bash on Ubuntu on Windows 10, then proceeding with the instructions above for installation on Linux. If not possible, then use the `zip` file as indicated below.

2.3 Downloading and extracting the tarball

If you rather not install it to your system you can just extract the `tar.gz` files from the Python Package Index at https://pypi.python.org/pypi/epic-code. Extract the file with:

```
$ tar -xzf epic-code-[version].tar.gz
```

`cd` into the root folder and install to your (virtual) environment:

```
$ python setup.py install
```

This is necessary to guarantee that all modules will be loaded correctly. It also applies to the source code extracted from the `zip` file or other compression format. You can then run the program from the `EPIC` folder. To update the program you will have to download the new tarball/zip file and execute this process again.

2.4 Cloning the repository

If you plan to contribute to this program you can clone the git repository at https://bitbucket.org/rmarcondes/epic.

3 Introduction to MCMC

Users familiar with the Markov Chain Monte Carlo (MCMC) method may want to skip to the next section. The typical problem the user will want to tackle with this program is the problem of parameter estimation of a given theoretical model confronted with one or more sets of observational data. This is a very common task in Cosmology these days, specially in the light of numerous data from several surveys, with increasing quality. Important discoveries are expected to be made with the data from new generation telescopes in the next decade.

In the following I give a very brief introduction to the MCMC technique and describe how to use program.

3.1 The Bayes Theorem

Bayesian inference is based on the inversion of the data-parameters probability relation, which is the Bayes theorem[1]. This theorem states that the posterior probability $p(\theta \mid D, \mathcal{M})$ of the parameter set θ given the data D and other information from the model $\mathcal{M}$ can be given by

$$p(\theta \mid D, \mathcal{M}) = \frac{\mathcal{L}(D \mid \theta, \mathcal{M})\,\pi(\theta \mid \mathcal{M})}{p(D, \mathcal{M})},$$

where $\mathcal{L}(D \mid \theta, \mathcal{M})$ is the likelihood of the data given the model parameters, $\pi(\theta \mid \mathcal{M})$ is the prior probability, containing any information known *a priori* about the distribution of the parameters, and $p(D, \mathcal{M})$ is the marginal likelihood, also popularly known as the evidence, giving the normalization of the posterior probability. The evidence is not required for the parameter inference but is essential in problems of selection model, when comparing two or more different models to see which of them is favored by the data.

Direct evaluation of $p(\theta \mid D, \mathcal{M})$ is generally a difficult integration in a multiparameter space that we do not know how to perform. Usually we do know how to compute the likelihood $\mathcal{L}(D \mid \theta, \mathcal{M})$ that is assigned to the experiment (most commonly a distribution that is Gaussian on the data or the parameters), thus the use of the Bayes theorem to give the posterior probability. Flat priors are commonly assumed, which makes the computation of the right-hand side of the equation above trivial. Remember that the evidence is a normalization constant not necessary for us to learn about the most likely values of the parameters.

3.2 The Metropolis-Hastings sampler

The MCMC method shifts the problem of calculating the unknown posterior probability distribution in the entire space, which can be extremly expensive for models with large number of parameters, to the problem of sampling from the posterior distribution. This is possible, for example, by growing a Markov chain with new states generated by the Metropolis sampler[2].

The Markov chain has the property that every new state depends on its current state, and only on this current state. Dependence on more previous states or on some statistics involving all states is not allowed. That can be done and can even also be useful for purposes like ours, but then the chain can not be called Markovian.

The standard MCMC consists of generating a random state y according to a proposal probability $Q(\cdot \mid x_t)$ given the current state x_t at time t. Then a random number u is drawn from a uniform distribution between 0 and 1. The new state is accepted if $r \geq u$, where

$$r = \min\left[1, \frac{p(y \mid D, \mathcal{M})Q(x_t \mid y)}{p(x_t \mid D, \mathcal{M})Q(y \mid x_t)}\right].$$

[1] Hobson M. P., Jaffe A. H., Liddle A. R., Mukherjee P. & Parkinson D., "Bayesian methods in cosmology". (Cambridge University Press, 2010).

[2] Gayer C., "Introduction to Markov Chain Monte Carlo". in "Handbook of Markov Chain Monte Carlo" http://www.mcmchandbook.net/

The fraction is the Metropolis-Hastings ratio. When the proposal function is symmetrical, $\frac{Q(x_t|y)}{Q(y|x_t)}$ reduces to 1 and the ratio is just the original Metropolis ratio of the posteriors. If the new state is accepted, we set $x_{t+1} := y$, otherwise we repeat the state in the chain by setting $x_{t+1} := x_t$.

The acceptance rate $\alpha = \frac{\text{number of accepted states}}{\text{total number of states}}$ of a chain should be around 0.234 for optimal efficiency[3]. This can be obtained by tuning the parameters of the function Q. In this implementation, I use a multivariate Gaussian distribution with a diagonal covariance matrix S.

3.3 The Parallel Tempering algorithm

Standard MCMC is powerful and works in most cases but there are some problems where the user may be better off using some other method. Due to the characteristic behavior of a Markov chain, it is possible (and even likely) that a chain become stuck in a single mode of a multimodal distribution. If two or more peaks are far away from each other, the proposal function tuned for good performance in a peak may have difficulty escaping that peak to explore the other, because the jump may be too short. To overcome this inefficiency, a neat variation of MCMC, called Parallel Tempering[4], favors a better exploration of the entire parameter space in such cases thanks to an arrangement of multiple chains that are run in parallel, each one with a different ''temperature" T. The posterior is calculated as $\mathcal{L}^{\beta}\pi$, with $\beta = 1/T$. The first chain is the one that corresponds to the real life posterior we are interested in; the other chains, at higher temperatures, will have wider distributions, which makes it easier to jump between peaks, thus exploring more properly the parameter space. Periodically, a swap of states between neighboring chains is proposed and accepted or rejected according to a Hastings-like ratio.

4 Using EPIC

You can use standard MCMC and Parallel-Tempering MCMC with this program. Besides, there is one experimental adaptive feature to each method that adjusts the covariance of the proposal multivariate Gaussian probability density for optimal efficiency, aiming at an acceptance rate around 0.234. In the PT-MCMC, the adaptation also adjusts the temperature levels of the chains and reduces the number of necessary chains if possible.

In the following sections I guide the user through the two approaches, with typical examples for demonstration.

[3] Roberts G. O. & Rosenthal J. S., "Optimal scaling for various Metropolis-Hastings algorithms". Statistical Science 16 (2001) 351-367.

[4] Gregory P. C., "Bayesian logical data analysis for the physical sciences: a comparative approach with Mathematica support". (Cambridge University Press, 2005).

4.1 Before starting

The problem is set up through a `.ini` file containing the parameters, priors, model type, data files to be used and tex strings for plotting. The program creates a folder in the working directory with the same name of this `.ini` file, if not already existing. Another folder is created with the date and time for the output of each run of the code, but you can always continue a previous run from where it stopped, just giving the folder name instead of the `.ini` file.

The folder `epic` is the root of the project. The python codes are in the `EPIC` source folder, where the `.ini` files should also be placed. The default working directory is the `EPIC`'s parent directory, i.e., the `epic` repository folder.

Changing the default working directory

By default, the folders with the name of the `.ini` files are created at the repository root level. But the chains can get very long and you might want to have them stored in a different drive. In order to set a new default location for all the new files, run:

```
python define_altdir.py
```

This will ask for the path of the folder where you want to save all the output of the program and keep this information in a file `altdir.txt`. If you want to revert this change you can delete the `altdir.txt` file or run again the command above and leave the answer empty when prompted.

The parameters names

Each cosmological parameter already implemented has a name to be used in the code that must be respected in the `.ini` file. The table below lists some of the parameters, their definitions and their names in the program.

Parameter	Description	Name
h	Reduced Hubble constant	`h`
$\Omega_{b0}h^2$	Physical baryonic density parameter	`Obh2`
$\Omega_{c0}h^2$	Physical cold dark matter density parameter	`Och2`
$\Omega_{r0}h^2$	Physical radiation density parameter	`Orh2`
Ω_{b0}	Baryonic density parameter	`Ob0`
Ω_{c0}	Cold dark matter density parameter	`Oc0`
Ω_{m0}	Matter (baryons plus DM) density parameter	`Om0`
Ω_{d0}	Dark energy density parameter	`Oc0`
w_d	Dark energy equation-of-state parameter	`w` or `wd`

The structure of the `.ini` file

Let us work with an example, with a simple flat ΛCDM model. Suppose we want to constrain its parameters with $H(z)$, supernovae data, CMB shift parameters and BAO data. The model parameters are the reduced Hubble constant h, the present-day values of the physical density parameters of dark matter $\Omega_{c0}h^2$, baryons $\Omega_{b0}h^2$ and radiation $\Omega_{r0}h^2$. We will not consider perturbations, we are only constraining the parameters at the background level. Since we are using supernovae data we must include a nuisance parameter M, which represents a shift in the absolute magnitudes of the supernovae. It is also possible to use the full JLA catalogue with this program. In this case, we would have to include also the nuisance parameters α, β and ΔM from the light-curve fit. The first section of the `LCDM.ini` file defines the priors and at the same time the names of the parameters. They must be inserted in the following form:

```
## PRIORS

h       0.6     0.8     4e-4
Och2    0.08    0.20    2e-4
Obh2    0.01    0.08    5e-5
Orh2    4e-5    8e-5    2e-7
M       -0.3    0.3     2e-3    nuisance

## end of priors
```

Delimiting lines are mandatory: the line defining the beginning of the priors section must contain the word `PRIORS`. The priors are read until the word `end` is found. The first column are the names of the parameters in the code. The second and third columns set the flat priors intervals. The fourth column is optional. Gaussian priors are also supported. In this case, one should add `gaussian` at the end of the line. The second and third values will then be the Gaussian parameters μ and σ instead. It gives the standard deviation for each parameter in the covariance of the proposal function. The values will vary according to the problem but a first guess can be 1/10 or other small fraction of the prior interval. Notice that the nuisance parameter is signaled as such by the word `nuisance` at the end of the line. To remove a parameter, you can just comment out its line with the character `#`. If the parameter removed is necessary for the model predictions then a default value will be assumed.

After the priors come the type of the model, needed to select the proper functions in the calculation of model predictions, and a display name, both with identifiers `MODEL` and `NAME`. They are signaled in the `.ini` file in the same way of the priors, but since they are single-line the final delimiter is not needed:

```
## MODEL
lcdm

## NAME
$\Lambda$CDM
```

The class model could also be `wcdm` for a wCDM model, of which ΛCDM is a special case.

Omitting the dark energy equation of state parameter would automatically set it to -1. Notice that the use of LaTeX input is allowed in the model name. Next is the list of observables to be used, with the filename or directory with the data. The list is read until an occurrence of `end` is found:

```
## DATA files

JLAsimple               jla_likelihood_v4
Planck2015_distances    Planck2015_LCDM
BAO                     BAO-6dF+SDSS+BOSS+Lyalpha+WiggleZ.txt
CosmicChrono            thirtypointsHz.txt

## end of data
```

Finally, tex strings must be given to be used in the plots:

```
## TEX REPRESENTATION

h           h
Och2        \Omega_{c0}h^2
Obh2        \Omega_{b0}h^2
Orh2        \Omega_{r0}h^2
Oc0         \Omega_{c0}
Ob0         \Omega_{b0}
Or0         \Omega_{r0}
Od0         \Omega_{d0}
Om0         \Omega_{m0}
M           \Delta{M}

# end of tex
```

Here, strings for the derived parameters are also needed. The tex strings should not contain any spaces. If you need to separate a tex command from a following variable, use `{}` instead, as is done for the last parameter in this example. Also note that the first column needs to be consistent with the names given in the priors and also in the `parameters_names.txt` dictionary.

To summarize, we end up with the `LCDM.ini` file in the `EPIC` folder looking like this:

```

## PRIORS

h       0.6     0.8     4e-4
Och2    0.08    0.20    2e-4
Obh2    0.01    0.08    5e-5
Orh2    4e-5    8e-5    2e-7
M       -0.3    0.3     2e-3    nuisance

## end of priors
```

(continues on next page)

(continued from previous page)

```
## MODEL
lcdm

## NAME
$\Lambda$CDM

## DATA files

JLAsimple               jla_likelihood_v4
Planck2015_distances    Planck2015_LCDM
BAO                     BAO-6dF+SDSS+BOSS+Lyalpha+WiggleZ.txt
CosmicChrono            thirtypointsHz.txt

## end of data

## TEX REPRESENTATION

h           h
Och2        \Omega_{c0}h^2
Obh2        \Omega_{b0}h^2
Orh2        \Omega_{r0}h^2
Oc0         \Omega_{c0}
Ob0         \Omega_{b0}
Or0         \Omega_{r0}
Od0         \Omega_{d0}
Om0         \Omega_{m0}
M           \Delta{M}

# end of tex
```

4.2 The datasets

We list and detail below the datasets already implemented and give instructions on how to include other data.

Type Ia supernovae

Two types of analyses can be made with the JLA catalogue. Here we are using the binned data consisting of distance modulus estimates at 31 points (defining 30 bins of redshift). This dataset is select via the `JLAsimple` entry in the list of data sources. If you want to use the full dataset (which

makes the analysis much slower since it involves three more nuisance parameters and requires the program to invert a 740 by 740 matrix at every iteration for the calculation of the JLA likelihood), insert `SNeJLA` instead. Also, you need to download the full data from http://supernovae.in2p3.fr/sdss_snls_jla/ReadMe.html or from my copy on Google Drive. Since the data files are too big, EPIC includes only the `jla_likelihood_v4/data` folder. In this case you will need the `jla_likelihood_v4/covmat`. Make sure to properly include it in you installation directory.

Either way, the argument is the same data folder `jla_likelihood_v4`. Note that the binned dataset introduces one nuisance parameter `M`, representing an overall shift in the absolute magnitudes, and the full dataset introduce other four nuisance parameters related to the light-curve parametrization. See Betoule et al. (2014)[1] for more details.

Note: This version of EPIC supports version V4 (June 2014) of the dataset release. Current version is V6 (March 2015). An update to implement use of V6 is being considered for the near future.

CMB distance priors

Constraining models with temperature or polarization anisotropy amplitudes is not currently implemented. However, you can include the CMB distance priors from Planck2015 with `Planck2015_distances`. These data consist of an acoustic scale l_A, a shift parameter R and the physical density of baryons $\Omega_{b0}h^2$. You can choose between the data for ΛCDM, wCDM and ΛCDM + Ω_k models[2], specifying the folders `Planck2015_LCDM`, `Planck2015_wCDM` or `Planck2015_LCDM+Omega_k`, respectively.

BAO data

Measurements of the baryon acoustic scales from the Six Degree Field Galaxy Survey (6dF), the Main Galaxy Sample of Data Release 7 of Sloan Digital Sky Survey (SDSS-MGS), the LOWZ and CMASS galaxy samples of the Baryon Oscillation Spectroscopic Survey (BOSS-LOWZ and BOSS-CMASS), the WiggleZ Dark Energy Survey and the distribution of the Lyman α forest in BOSS (BOSS-Ly) are compiled file `BAO-6dF+SDSS+BOSS+Lyalpha+WiggleZ.txt`. The subsamples `BAO-6dF+SDSS+BOSS+Lyalpha.txt` and `BAO-6dF.txt` are also available, the first excluding the WiggleZ data and the second only with the 6dF data. Since these files simply contain the redshift of the measurement, the value of the characteristic ratio $r_{\rm BAO}(z) \equiv r_s(z_d)/d_V(z)$ between the sound horizon r_s at decoupling time (z_d) and the effective BAO distance d_V and the measurement error, it is really simple to remove or add new measurements of this observable.

[1] Betoule M. et al. "Improved cosmological constraints from a joint analysis of the SDSS-II and SNLS supernova samples". Astronomy & Astrophysics 568, A22 (2014).

[2] Huang Q.-G., Wang K. & Wang S. "Distance priors from Planck 2015 data". Journal of Cosmology and Astroparticle Physics 1512, 022 (2015).

$H(z)$ data

These are the cosmic chronometer data. 30 measurements of the Hubble expansion rate $H(z)$ at redshifts between 0 and 2, plus a 2.4% precision local measure of H_0. The values of redshift, H and the uncertainties are given in the file `thirtypointsHz.txt`.

$f\sigma_8$ data

Large-scale structure data from redshift-space distortion and peculiar velocity measurements giving the growth rate times the RMS amplitude of matter perturbations $f\sigma_8(z)$ can also be used. Notice that in this case you need to provide an analytic evaluation of the growth rate (for example, as $f = \Omega_m^\gamma$) in your model since this program does not evolve perturbations numerically at the present moment. See for example the implementation in Marcondes et al. (2016)[3].

Including other data

You can include other data if you want. In the simplest case, you will have measurement values and error bars for a certain quantity, say $f(z)$, at given points. Put them on a text file in three columns separated by TAB or spaces, as in, for example, the `SDSS+BOSS.txt` file:

```
0.15        4.47        0.17
0.32        8.47        0.17
0.57        13.77       0.13
```

Choose a label for this dataset and put it in the `.ini` file in the data files section:

```
my_data     my_data_file.txt
```

Next, you need to tell the code how to read this new type of dataset. In this simple case you add a function like:

```
def my_data(obsble):
    return simplest_data(obsble, 'my_data', r'$f(z)$')
```

to the `load_data` module. The name of this function must be the same as the label that goes in the `.ini` file. This function, or its more complicated equivalent, is the only change needed to be made in the `load_data.py` file. In the `simplest_data` function call, the `obsble` variable carries the dataset file information (the filename or directory), the second argument is a string to be displayed indicating that your data is loaded. The third argument is a raw string, possibly including LaTeX notation, to be displayed in the triangle plots compounding the label that describes the datasets used in this analysis when you plot together results from more than one analysis.

[3] Marcondes R. J. F., Landim R. C. G., Costa A. A., Wang B. & Abdalla E. "Analytic study of the effect of dark energy-dark matter interaction on the growth of structures". Journal of Cosmology and Astroparticle Physics 1612, 009 (2016).

You then need to define the likelihood calculation in the `likelihood` module. At the end of the file, add to the dictionary `allprobes` an entry with the key `'my_data'` (again the same label) and the likelihood function, say `my_data_likelihood`, as its value. For a Gaussian likelihood, this function can be:

```
def my_data_likelihood(datapoints, cosmology):
    return simple(datapoints, observables.my_f, cosmology)
```

Now, as you may have already figured out, the next step is to add the theoretical calculation `my_f` of your observable f to the `observables.py` file. Of course, this will vary with each case. Do not forget to use the `model` attribute of the `Cosmology` class object named `cosmology` to set different calculations according to each cosmological model you might want to use.

4.3 The cosmological models

A few models are already implemented. I give a brief description below, with references for works that discuss some of them in detail and works that analyzed them with this code. There are some models for which we have analytical solutions for the evolutions of the energy densities and can write the Hubble rate in a closed form: this is done via the function $h\,E(z)$ (`Eh` in the module `observables`) from which $H(z) = 100\,h\,E(z)$ is obtained in the function `H` from the same module. Generally, this is implemented using the physical densities $\Omega_{i0}h^2$ and h as parameters. For other models, a numerical integral is performed with fourth-order Runge-Kutta. Since this integration is done backwards in time from $a = 1$, it is easier to deal with the density parameters Ω_{i0} and H_0. The result is a $H(z)$ function that can be interpolated for any redshift within the interval of integration. This is registered in a `Cosmology` class object as the attribute `H_solution`, from which $h\,E(z)$ as well can be obtained when convenient. The code is written so that the integration is done only once given a set of parameters and model. In all cases the dark energy density parameter is obtained, as a derived parameter, from the flatness condition that all the density parameters must add up to 1. At the end of this section, I instruct the user on how to include a new model for use with this program.

The ΛCDM model

The standard models accepts the parameters h, $\Omega_{c0}h^2$, $\Omega_{b0}h^2$, $\Omega_{r0}h^2$, and has Ω_{c0}, Ω_{b0}, Ω_{r0}, Ω_{d0}, as derived parameters. Extending this model to allow curvature is not completely supported yet. The Friedmann equation is

$$\frac{H(z)}{100\ \mathrm{km\ s^{-1}\ Mpc^{-1}}} = hE(z) = \sqrt{(\Omega_{b0}h^2 + \Omega_{c0}h^2)(1+z)^3 + \Omega_{r0}h^2(1+z)^4 + \Omega_d h^2}$$

or

$$H(z) = H_0\sqrt{(\Omega_{b0} + \Omega_{c0})(1+z)^3 + \Omega_{r0}(1+z)^4 + \Omega_d},$$

with $\Omega_d = 1 - \Omega_{b0} - \Omega_{c0} - \Omega_{r0}$ and $H_0 = 100\,h\ \mathrm{km\ s^{-1}\ Mpc^{-1}}$. This model is identified in the code by the label `lcdm`.

The wCDM model

Identified by `wcdm`, this is like the standard model except that the dark energy equation of state can be any constant w_d, thus having the ΛCDM model as a specific case with $w_d = -1$. The Friedmann equation is like the above but with the dark energy contribution multiplied by $(1+z)^{3(1+w_d)}$.

Interacting Dark Energy models

A comprehensive review of models that consider a possible interaction between dark energy and dark matter is given by Wang *et al.*[2]. In interacting models, the individual conservation equations of the two dark fluids are violated, although still preserving the total energy conservation:

$$\dot{\rho}_c + 3H\rho_c = Q$$
$$\dot{\rho}_d + 3H(1+w_d)\rho_d = -Q.$$

The shape of Q is what characterizes each model. Common forms are proportional to ρ_c, to ρ_d or to some combination of both.

Proportional to ρ_d

This model uses $Q = 3H\xi\rho_d$. We were interested in studying the growth rate of matter perturbations in such a model in an analytic way in a previous work[3]. Some assumptions were made and the growth rate was approximated as $f \approx \Omega_m^\gamma$, with γ determined in terms of the coupling constant ξ, a free parameter of the model. Other parameters involved are $\sigma_{8,0}$ and Ω_{d0}, and w_0 and w_1 from

$$w_d(\Omega_d) = w_0 + w_1\Omega_d + O(\Omega_d^2).$$

A recurrence relation is used to approximate the evolution of densities. This implementation, dubbed `model2` in the code, is meant to be used with the `myfsigma8` or `growth_rate_sigma_eight` functions only, to have its predictions compared with the $f\sigma_8(z)$ data.

Interacting dark energy in the dark $SU(2)_R$ model

As per the recent paper by Landim *et al*[4], This model actually proposes an interaction possibly between more than two components of a dark sector from the decay of the heaviest particle (φ^+)

[2] Wang B., Abdalla E., Atrio-Barandela F., Pavón D., "Dark matter and dark energy interactions: theoretial challenges, cosmological implications and observational signatures". Reports on Progress in Physics 79 (2016) 096901.

[3] Marcondes R. J. F., Landim R. C. G., Costa A. A., Wang B. and Abdalla E., "Analytic study of the effect of dark energy-dark matter interaction on the growth of structures". Journal of Cosmology and Astroparticle Physics 1612, 009 (2016).

[4] Landim R. C. G., Marcondes R. J. F., Bernardi F. F. and Abdalla E., "Interacting dark energy in the dark SU(2)R model". arXiv:1711.07282 [astro-ph.CO]

of the dark energy doublet (φ^0, φ^+). The other particles in the sector are a dark matter candidate ψ, a dark matter neutrino ν_d. The right-hand side of the conservation equations of these particles are obtained in terms of a characteristic decay time scale Γ and mass ratios with respect to the heaviest particle. The model is implemented in two flavours: the more general `darkSU2` and the specific case with the neutrino mass set to zero, `darkSU2wnu0`. The solution to the full set of background density evolutions is obtained numerically.

Fast-varying dark energy equation-of-state models

Models of dark energy with fast-varying equation-of-state parameter have been studied in some works[1]. Three such models were implemented as described in Marcondes and Pan (2017)[5]. We used this code in that work. They have all the density parameters present in the ΛCDM model besides the dark energy parameters that we describe in the following.

Model 1

This model `fastvarying1` has the free parameters w_p, w_f, a_t and τ characterizing the equation of state

$$w_d(a) = w_f + \frac{w_p - w_f}{1 + (a/a_t)^{1/\tau}}.$$

w_p and w_f are the asymptotic values of w_d in the past ($a \to 0$) and in the future ($a \to \infty$), respectively; a_t is the scale factor at the transition epoch and τ is the transition width. The Friedmann equation is

$$\frac{H(a)^2}{H_0^2} = \frac{\Omega_{r0}}{a^4} + \frac{\Omega_{m0}}{a^3} + \frac{\Omega_{d0}}{a^{3(1+w_p)}} f_1(a),$$

where

$$f_1(a) = \left(\frac{a^{1/\tau} + a_t^{1/\tau}}{1 + a_t^{1/\tau}} \right)^{3\tau(w_p - w_f)}.$$

[1] Corasaniti P. S. & Copeland E. J., "Constraining the quintessence equation of state with SnIa data and CMB peaks". Physical Review D 65 (2002) 043004; Basset B. A., Kunz M., Silk J., "A late-time transition in the cosmic dark energy?". Monthly Notices of the Royal Astronomical Society 336 (2002) 1217-1222; De Felice A., Nesseris S., Tsujikawa S., "Observational constraints on dark energy with a fast varying equation of state". Journal of Cosmology and Astroparticle Physics 1205, 029 (2012).

[5] Marcondes R. J. F. & Pan S., "Cosmic chronometer constraints on some fast-varying dark energy equations of state". arXiv:1711.06157 [astro-ph.CO].

Model 2

This model `fastvarying2` alters the previous model to allow the dark energy to feature an extremum value of the equation of state:

$$w_d(a) = w_p + (w_0 - w_p)a\frac{1-(a/a_t)^{1/\tau}}{1-(1/a_t)^{1/\tau}},$$

where w_0 is the current value of the equation of state and the other parameters have the interpretation as in the previous model. The Friedmann equation is

$$\frac{H(a)^2}{H_0^2} = \frac{\Omega_{r0}}{a^4} + \frac{\Omega_{m0}}{a^3} + \frac{\Omega_{d0}}{a^{3(1+w_p)}}e^{f_2(a)},$$

with

$$f_2(a) = 3(w_0 - w_p)\frac{1 + (1-a_t^{-1/\tau})\tau + a[\{(a/a_t)^{1/\tau} - 1\}\tau - 1]}{(1+\tau)(1-a_t^{-1/\tau})}.$$

Model 3

Finally, we have a third model `fastvarying3` with the same parameters as in Model 2 but with equation of state

$$w_d(a) = w_p + (w_0 - w_p)a^{1/\tau}\frac{1-(a/a_t)^{1/\tau}}{1-(1/a_t)^{1/\tau}}.$$

It has a Friedmann equation identical to Model 2's except that $f_2(a)$ is replaced by

$$f_3(a) = 3(w_0 - w_p)\tau\frac{2 - a_t^{-1/\tau} + a_t^{1/\tau}[(a/a_t)^{1/\tau} - 2]}{2(1-a_t^{-1/\tau})}.$$

Including a new model

To define a new model in the program, you will need to implement your model calculation of the observables you want to use. Choose a label for the model and use conditionals on the `cosmology.model` attribute. In your `.ini` file, use the same label to describe this new model in the corresponding section and do not forget to give LaTeX representations for your parameters. If there are derived parameters that you want to get, include their recipes in `derived.py` and `derived_bf.py`. The first one handles the Markov chains, while the second calculates the best-fit estimate point. No other changes are required throughout the code.

4.4 Running MCMC

This is the vanilla Monte Carlo Markov Chain with the Metropolis algorithm, as introduced in the previous section *The Metropolis-Hastings sampler*.

We will now proceed to run MCMC for the ΛCDM model. All the following commands must be issued from your terminal at the `EPIC` directory.

Initializing the chains

Standard MCMC is the default option for sampling. The chains are initialized with:

```
python initialize_chains.py LCDM.ini
```

We already know the `.ini` file, a configuration file containing all the information for that specific simulation. It defines the parameters names and priors, their TeX strings, the type of cosmological model, which will determine how to calculate distances and the observables and sets a label for the model.

The code will ask the number of chains to be used. It must be at least `2`. You can also run the command above with `chains=8` as an argument to suppress this prompt. Each chain will create an independent process with Python's `multiprocessing`, so you should not choose a number higher than the number of CPUs `multiprocessing.cpu_count()` of your machine.

This command will create a folder with the files that will store the states of the chains, named `reducedchain1.txt`, `reducedchain2.txt`, etc, besides other relevant files. The sampling mode (`MCMC` or `PT`) is prepended to the name of the folder, which is always the date-time of creation (in UTC), unless the option `cfullname=MyFirstRun` (c standing for custom) is used, then `MCMC-MyFirstRun` will be the folder name. A custom label can also be prepended with `cname=my_label` creating, for example, the folder `my_label-MCMC-171110-173000`. It will be stored within another folder with the same name of the `.ini` file, thus in this case `LCDM/my_label-MCMC-170704-201125`. The folder name will be displayed in the first line of the output.

Alternatively, this first command can be skipped by running the main file `epic.py` directly, as I explain next. The main file can distinguish if the first argument is a `.ini` file or a directory. In the first case, it will call `initialize_chains.py` with the same arguments. In the second case, if the path given corresponds to a `MCMC` or `PT` (signaled by the `mode.txt` file) directory just created with `initialize_chains.py` or any previously initiated simulation, it will continue to evolve its chains from where they stopped.

Sampling the posterior distribution

The magic starts with:

```
python epic.py <FULL-PATH-TO>/LCDM/MCMC-MyFirstRun/
```

or simply:

```
python epic.py LCDM.ini
```

if you have not initialized the chains yet with the previous command or want to start a new run. Two parameters of the code will be prompted:

```
Please specify the number of steps in each passage in the MCMC loop: 10000
Please specify the tolerance for convergence: 1e-2
```

but these can also be informed directly when executing `epic.py` in the form of arguments `steps=3000` and `tol=1e-2`. This number of steps means that the chains will be written to the disk (the new states are appended to the chains files) after each `steps` states in all chains. A large number prevents frequent writing operations, which could otherwise affect overall performance unnecessarily. The relevant information will be displaying, in our example case looking similar to the following:

```
Loading INI file...
    Reading simplified JLA dataset...
        jla_likelihood_v4
    Reading distance priors from Planck 2015...
        Planck2015_LCDM
    Reading BAO (6dF+SDSS+BOSS+Lyalpha+WiggleZ) data...
        BAO-6dF+SDSS+BOSS+Lyalpha+WiggleZ.txt
    Reading cosmic chorometer H(z) data...
        thirtypointsHz.txt
File <FULL-PATH-TO>/LCDM/MCMC-MyFirstRun/
```

and the MCMC will start.

In the MCMC mode, the code will periodically check for convergence according to the Gelman-Rubin method (by default it is done every two hours but can be specified differently as `12h` or `45min` in the arguments, for example. Ideally this does not need to be a small time interval, but the option of specifying this time in minutes or even in seconds (`30sec`) is implemented and available for testing purposes. The MCMC run stops if convergence is achieved with a tolerance smaller than `tol`.

The following is the output of our example after the MCMC has started. The 10000 steps take a bit less than twelve minutes in my workstation running 8 chains in parallel. The number of chains will not make much impact on this unless we use too many steps by iteration and are close to the limit of memory. After approximately two hours, convergence is checked. Since it is smaller than our required `tol`, the code continues with new iterations for more two hours before checking convergence again and so on. When convergence smaller than `tol` is achieved the code makes the relevant plots and quits.

```
Initiating MCMC...
i 1, 10000 steps, 8 ch; 11m48s, Mon Oct 16 19:09:09 2017. Next: ~1h48m.
i 2, 20000 steps, 8 ch; 11m50s, Mon Oct 16 19:21:00 2017. Next: ~1h36m.
i 3, 30000 steps, 8 ch; 11m51s, Mon Oct 16 19:32:51 2017. Next: ~1h24m.
i 4, 40000 steps, 8 ch; 11m50s, Mon Oct 16 19:44:42 2017. Next: ~1h12m.
i 5, 50000 steps, 8 ch; 11m49s, Mon Oct 16 19:56:31 2017. Next: ~1h0m.
i 6, 60000 steps, 8 ch; 11m51s, Mon Oct 16 20:08:23 2017. Next: ~48m58s.
i 7, 70000 steps, 8 ch; 11m50s, Mon Oct 16 20:20:14 2017. Next: ~37m7s.
i 8, 80000 steps, 8 ch; 11m50s, Mon Oct 16 20:32:04 2017. Next: ~25m16s.
i 9, 90000 steps, 8 ch; 11m52s, Mon Oct 16 20:43:57 2017. Next: ~13m24s.
i 10, 100000 steps, 8 ch; 11m53s, Mon Oct 16 20:55:50 2017. Checking now...
Loading chains...                                          [##########]  8/8
n = 100000
Saving information for histograms...                       [##########]  100%
Monitoring convergence...                                  [##########]  100%
Total time for data analysis: 14.5 seconds.
After 1h58m and 100000 steps, with 8 chains,
Convergence 3.73e-01 achieved. Current time: Mon Oct 16 20:56:04 2017.
i 11, 110000 steps, 8 ch; 11m49s, Mon Oct 16 21:07:54 2017. Next: ~1h48m.
...
```

At any time after the first iteration, the user can inspect the acceptance ratio of the chains. The information after updated at every iteration and can be found in the file `LCDM/MCMC-MyFirstRun/llc.txt`.

Adaptation (beta)

In development. Come back later to check!

Analyzing the chains

The code reads the chains and compile the distributions for a nice plot with the command:

```
python analyze.py <FULL-OR-RELATIVE-PATH-TO>/LCDM/MCMC-MyFirstRun/
```

Histograms are generated for the marginalized distributions using 20 bins or any other number given with `combobins=40`, for example, if you think you have sufficient data. At any time one can run this command with `calculate_R` to check the state of convergence. By default, it will calculate $\hat{R}^p - 1$ for twenty different sizes considering the size of the chains to provide an idea of the evolution of the convergence.

Convergence is assessed based on the Gelman-Rubin criteria. I will not enter into details of the

method here, but I refer the reader to the original papers[1][2] for more information. The variation of the original method for multivariate distribution is implemented. When using `MCMC`, all the chains are checked for convergence and the final resulting distribution which is analyzed and plotted is the concatenation of all the chains, since they are essentially all the same once they have converged.

Additional options

If for some reason you want to view the results for an intermediate point of the simulation, you can tell the script to `stop_at=18000`, everything will be analyzed until that point.

Depending on the model being analyzed, one may be interested in seeing the derived parameters. Once these are defined appropriately in the files `derived.py` and `derived_bf.py`, include them in the analysis with the option `use_derived`.

If you want to check the random walk of the chains you can plot the sequences with `plot_sequences`. This will make a grid plot containing all the chains and all parameters. Keep in mind that this can take some time and generate a big output file if the chains are very long. You can contour this problem by thinning the distributions by some factor `thin=10`. This also applies for the calculation of the correlation of the parameters in each chain, enabled with the `ACF` option.

We generally represent distributions by their histograms but sometimes we may prefer to exhibit smooth curves. Although it is possible to choose a higher number of histogram bins, this may not be sufficient and may required much more data. Much better (although possibly slow when there are too many states) are the kernel density estimates (KDE) tuned for Gaussian-like distributions[3]. To obtain smoothed shapes use `smooth_hist=kde`. This will compute KDE curves for the marginalized parameter distributions and also the two-parameter joint posterior probabilities. Try it for the 1D distributions only if it is taking too long, using `kde1` instead. The option `kde2` for the 2D distributions only is also available for completeness.

Making the triangle plots

The `analyze.py` routine will also produce the plots automatically (unless supressed with `dontdraw`), but you can always redraw everything when you want, maybe you would like to tweak some colors? Loading the chains again is not necessarily since the analysis already saves the information for the plots anyway. So, let's say that we already saved this information for the chains after 20000 steps. Now we must pass the path to the folder containing the results for those specific size of chains and number of bins:

[1] Gelman A & Rubin D. B. "Inference from Iterative Simulation Using Multiple Sequences". Statistical Science 7 (1992) 457-472.

[2] Brooks S. P. & Gelman A. "General Methods for Monitoring Convergence of Iterative Simulations". Journal of Computational and Graphical Statistics 7 (1998) 434.

[3] Kristan M., Leonardis A. and Skocaj D. "Multivariate online kernel density estimation with Gaussian kernels". Pattern Recognit 44 (2011) 2630-2642.

```
python drawhistograms.py <FULL-OR-RELATIVE-PATH-TO>/LCDM/MCMC-MyFirstRun/n20000/
↪results20/
```

You can choose to view the best-fit point with `bf=+` or any other marker. To change the color, use `singlecolor=m` (for magenta) or any other Python color name. The default is `C0` (a shade of blue, from the default Matplotlib palette).

The 1σ and 2σ confidence levels are shown and are written to the file `hist_table.tex` inside the `results20` folder. This is a LaTeX-ready file that can be compiled to make a nice table in a PDF file or included in your paper as you want. To view and save information for more levels, use the argument `levels=1,2,3`, for example.

If you have used the option `smooth_hist` with either `kde`, `kde1` or `kde2` option you need to specify it again in order to make the corresponding plots, otherwise the histograms will be drawn.

Additional options

You can further tweak your plots and tables. `usetex` will make the program render the plot using LaTeX, fitting nicely to the rest of your paper. You can even choose the Times New Roman font if you prefer it over the default Computer Modern, using `font=Times`.

If you have nuisance parameters, you can opt to not show them with `nonuisance`.

`fmt=3` can be used to set the number of figures to report the results (default is `5`). You can plot Gaussian fits together with the histograms (the Gaussian curve and the two first sigma levels), using `show_gaussian_fit`, but probably only for the sake of comparison since this is not usually done in publications.

All these options can be given to the `analyze.py` command, they will be passed to `drawhistograms.py` accordingly.

Below we see the triangle plot of the histograms, with the default settings, in comparison with a perfected version using the smoothed distributions, the Python color `C9`, the LaTeX renderer, including the best-fit point and the 3σ confidence level. By default, the prior intervals are used as axes limits, with ticks at predefined positions, but both images below override this setting reading custom intervals and ticks given in the files `LCDM.range` and `LCDM.ticks`.

With `use_derived`, we can also check out the marginalized distributions of the derived parameters in the `der_pars` directory

Combining two or more simulations in one plot

You just need to run `drawhistograms.py` with two or more paths in the arguments. To illustrate this, I run a second MCMC simulation for the same model but this time without including the distance priors data. It is then interesting to plot both realizations together so we can seethe effect that including that dataset has on the results:

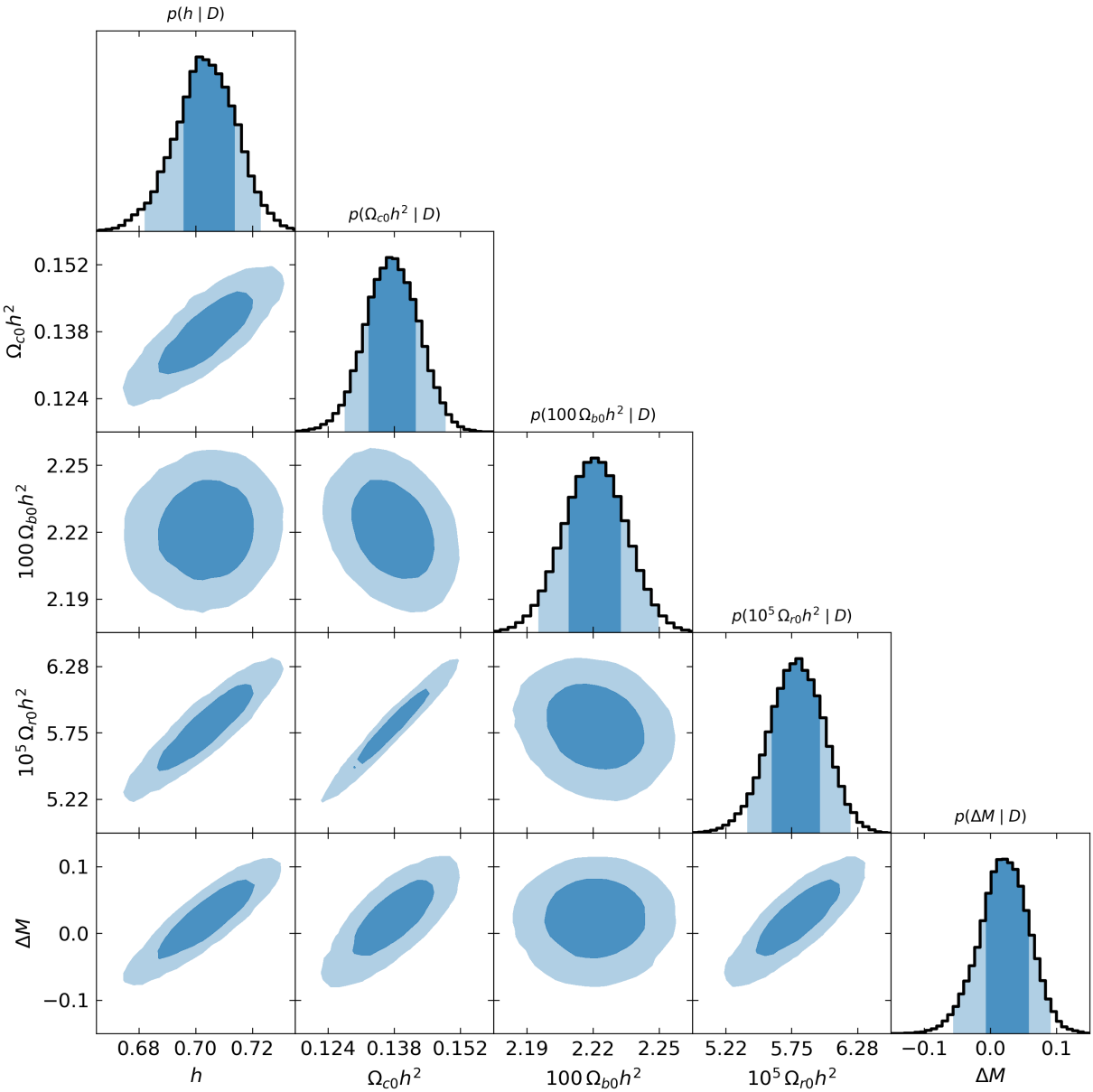

Fig. 1: Triangle plot with default configurations for histograms.

```
python drawhistograms.py \
<FULL-OR-RELATIVE-PATH-TO>/LCDM-no-Planck/MCMC-MySecondRun/n32500/results20/ \
<FULL-OR-RELATIVE-PATH-TO>/LCDM/MCMC-MyFirstRun/n1300000/results20/ \
colors=C3,k smooth_hist=kde units=Orh2:5,Obh2:2 \
range=h:0.66_0.75,Obh2:0.01_0.056,Och2:0.08_0.16,M:-0.12_0.18 \
ticks=Och2:0.09_0.12_0.15,Obh2:1.8e-2_3.3e-2_4.8e-2,Orh2:4.7e-5_6e-5_7.3e-5 \
usetex font=Times gname=noplanck
```

Notice how crucial the CMB data is to resolve the degeneracy between the baryon and radiation

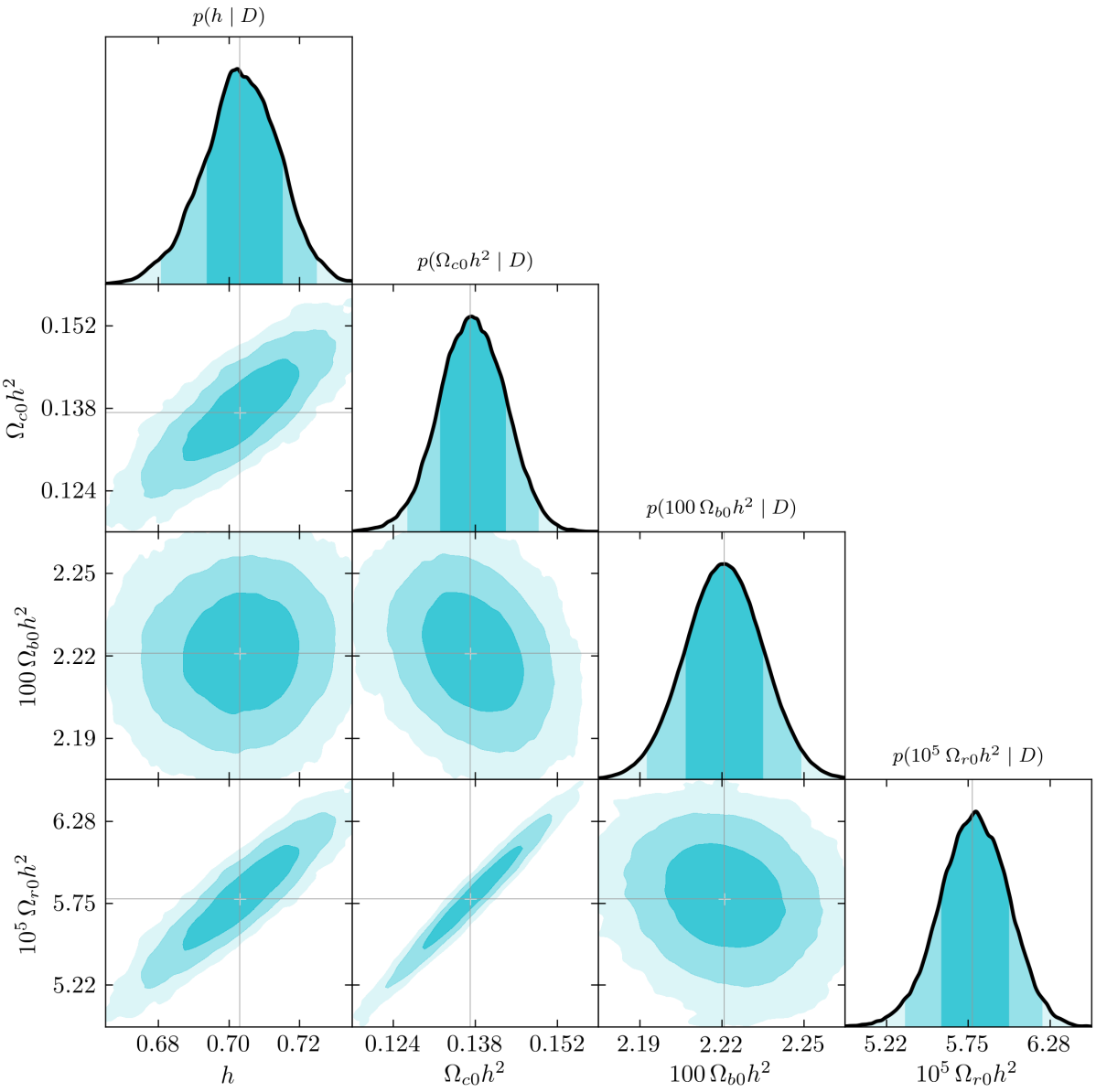

Fig. 2: Customized plot with smoothed distributions.

densities when both are free parameters. The constraints are improved considerably.

You can combine as many results as you like. When this is done, any custom ranges or tick positions defined in `.range` and `.ticks` files will be ignored (as well as the factor in the `.units` file), since different simulations might give considerably different results that could go outside of the ranges of one of them. In this case, after checking the results and determining the ranges that you want to plot, pass this information in the command line as in the example above. The `gname` option specifies the prefix for the name of the pdf file that will be generated. If you omit this option you will be prompted to type it. All other settings are optional.

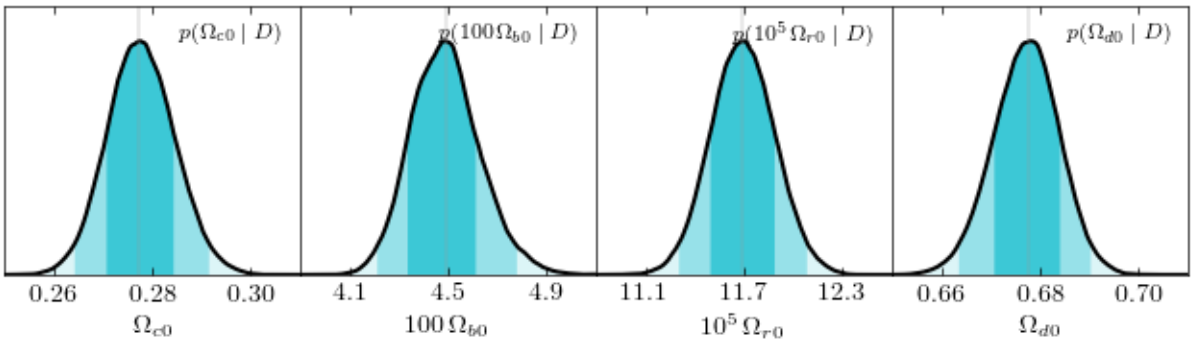

Fig. 3: Distributions of derived paramaters.

A legend will be included in the top right corner using the labels defined in the `labels.txt` files: the lines of these files are combined with a plus sign but you can use whatever label you prefer by replacing the contents of this file, for example, giving a name in a single line. The legend title uses the model name of the first simulation in the arguments. This is intended for showing, at the same time, results from different datasets with the same model. If this is not your case and you would like to use a different title, change accordingly the name entry in the `.ini` file of the corresponding simulation.

Visualizing chain sequences and convergence

If you include the argument `plot_sequences` you will get a plot like this

When monitoring convergence, the values of $\hat{R}^p - 1$ at twenty different lengths for the multivariate analysis and separate one-dimensional analyses for each parameter are plotted in the files `monitor_convergence_<N>.pdf` and `monitor_each_parameter_<N>.pdf`, where `N` is the total utilized length of the chains. The absolute values of the between-chains and within-chains variances, $\hat{V}$ and W are also shown. For our ΛCDM example, we got

Finally, the correlation of the chains can also be inspected if we use the option `ACF`. This includes all the cross- and auto-correlations.

4.5 Running PT-MCMC

Alternatively, we can use the *Parallel Tempering algorithm*. This is specially useful when the posterior distribution is likely to present more than one peak and these peaks are well separated. In such cases, the standard MCMC sampler is not efficient and may even fail to detect peaks away from where the chain begun. With Parallel Tempering, the ground-temperature chain that samples the true posterior distribution is more likely to jump between separate peaks, more efficiently visiting the entire parameter space. This is possible thanks to a mechanism providing state exchange between chains following a temperature ladder, in which the posterior is progressively flattened so

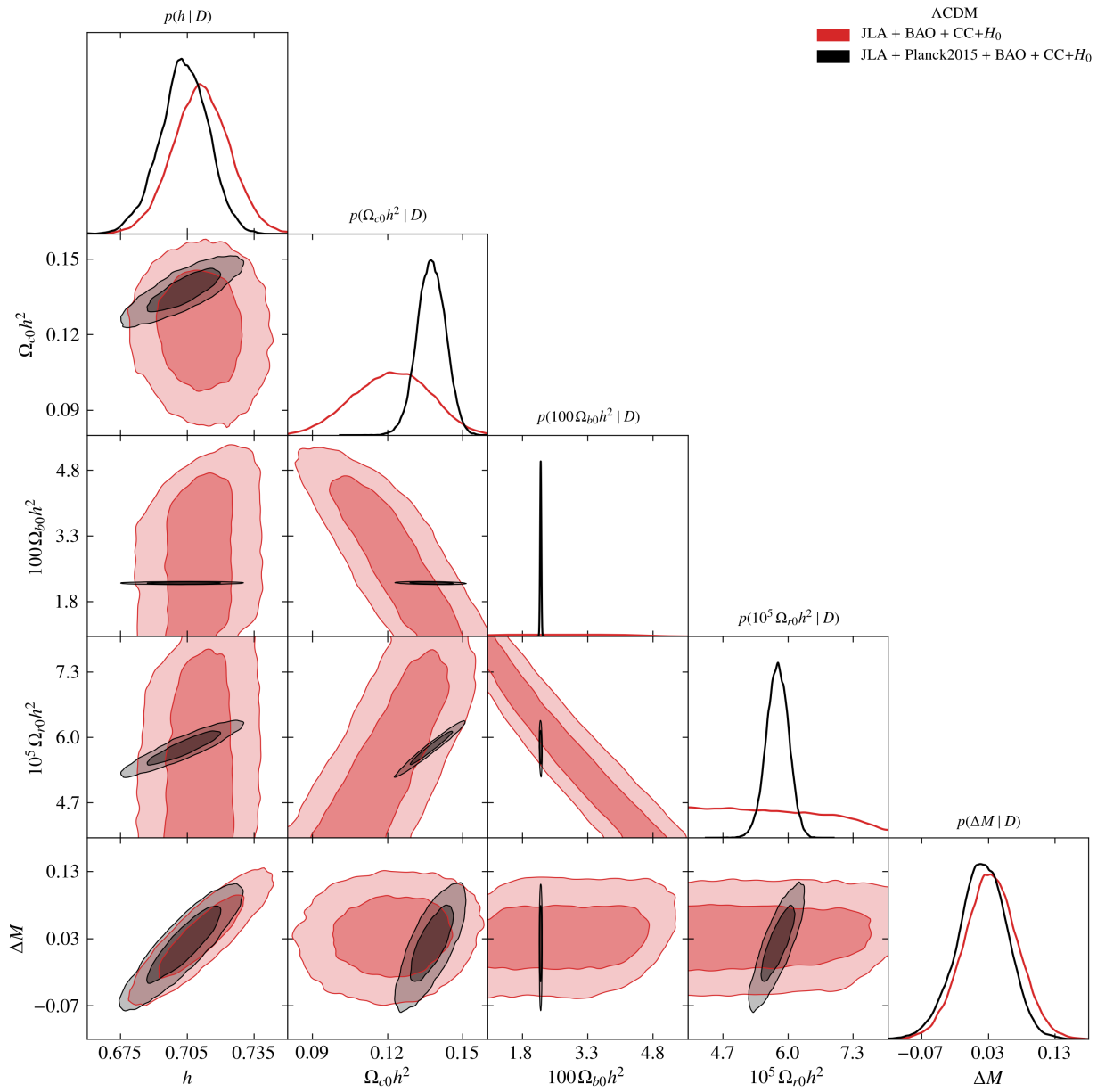

Fig. 4: Results for two different analyses.

as to reduce the differences of probability density amplitude that otherwise would keep the chains locally stuck.

In order to illustrate the potential of this method, we will run a PT-MCMC on an artificial distribution. A test model is created with a likelihood given as a combination of various univariate Gaussian distributions. We will see in the following how we are able to recover the exact marginalized posterior distributions. The function reserved for this is the `disttest` in the `likelihood.py` file. In our example, this model has three free parameters, a, b, c, all with flat priors between 0 and 1. The likelihood consists of the Gaussian functions $N_a(0.5, 0.01)$, $N_b(0.2, 0.03)$ and two Gaussian

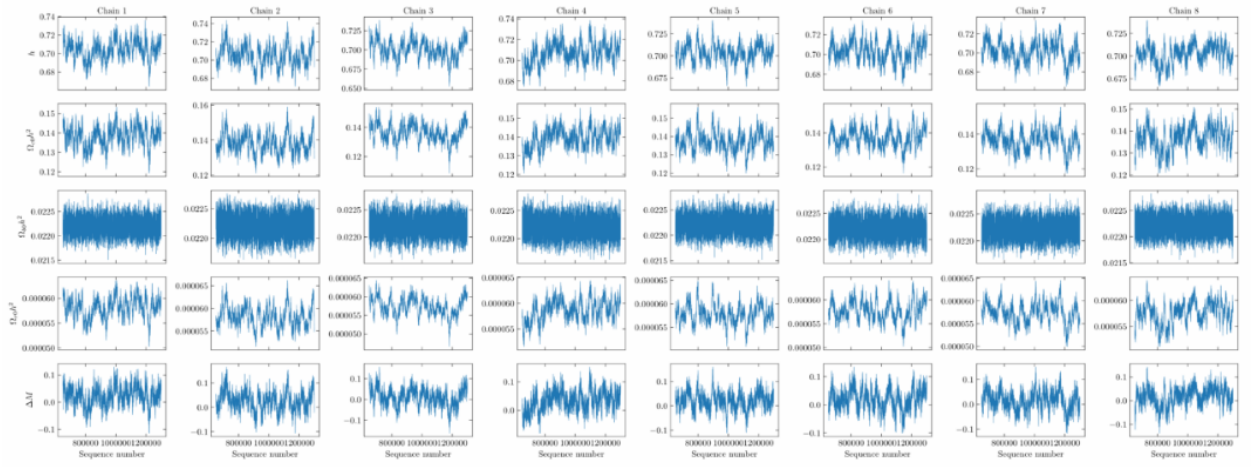

Fig. 5: Chain sequences along each parameter axis, for all chains.

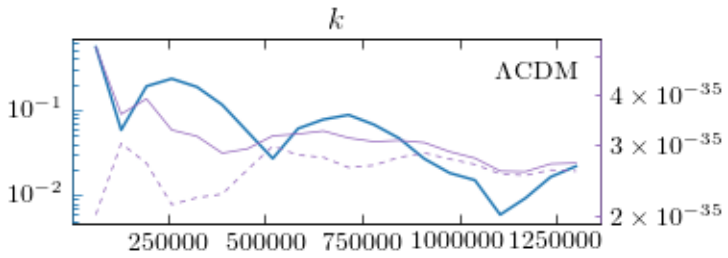

Fig. 6: Multivariate convergence analysis.

functions for the third parameter, $N_c(0.3, 0.02)$ and $N_c(0.75, 0.02)$, giving two distant peaks several sigmas apart, with respective weights $0.7, 0.3$.

Initializing the chains

This works pretty much in the same way as the previous method. Just run:

```
python initialize_chains.py disttest.ini chains=4 mode=PT
```

`mode=PT` is mandatory to specify that you want to use PT and not the default sampler, standard MCMC. You will be prompted about after how many steps the algorithm should propose a state swap:

```
Please specify after how many steps should propose a swap: 30
```

or you can bypass the information with `nswap=30`. At every step a random number u between 0 and 1 is generated and a swap will be proposed if $u < 1/n_s$. The default temperature ladder is between $\beta = 1$ (ground chain) and 2^{10}, evenly distributed over the log scale over the number

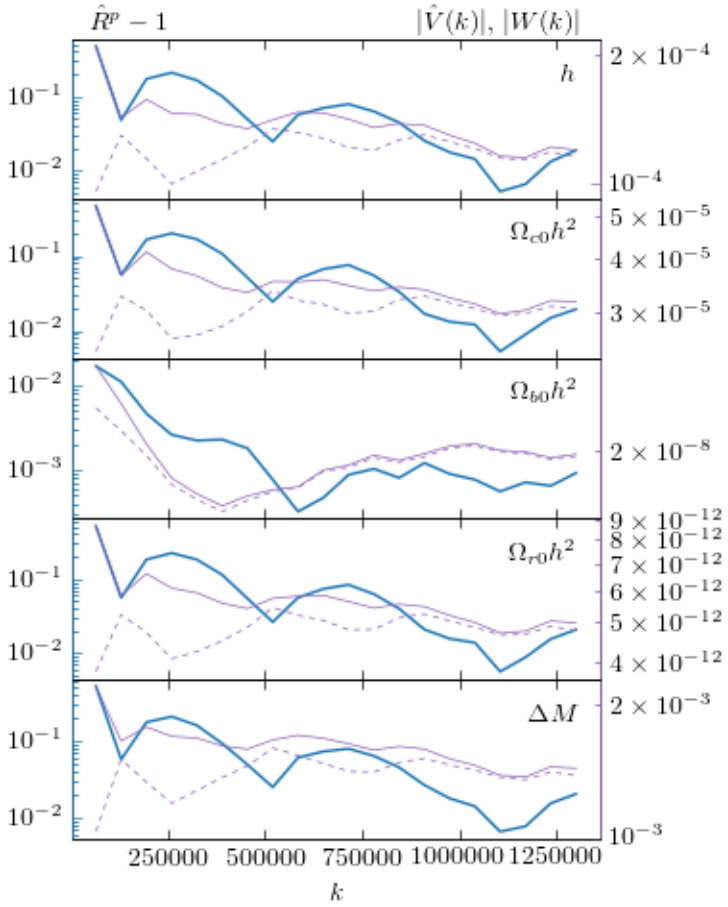

Fig. 7: Individual parameter convergence monitoring.

of chains given. This can be changed with `betascale=linear`, for example (it will actually be linear for whatever given value different than `log`) and `betamax=6` (instead of `10`), for example.

Sampling the posterior distribution

Next, we run as we would a standard MCMC set of chains:

```
python epic.py <FULL-PATH-TO>/disttest/PT-171022-185848/ steps=3000 limit=90000
```

Unlike the standard method, convergence will not be checked automatically (remember only the ground temperature chain corresponds to the true posterior distribution). We thus need to specify a `limit` so it stops after some large number of states. Because the higher-temperature distributions are wider, in this code I try to guarantee an efficient sampling for all chains by dividing the variances in the proposal function in each chain by their respective value of β.

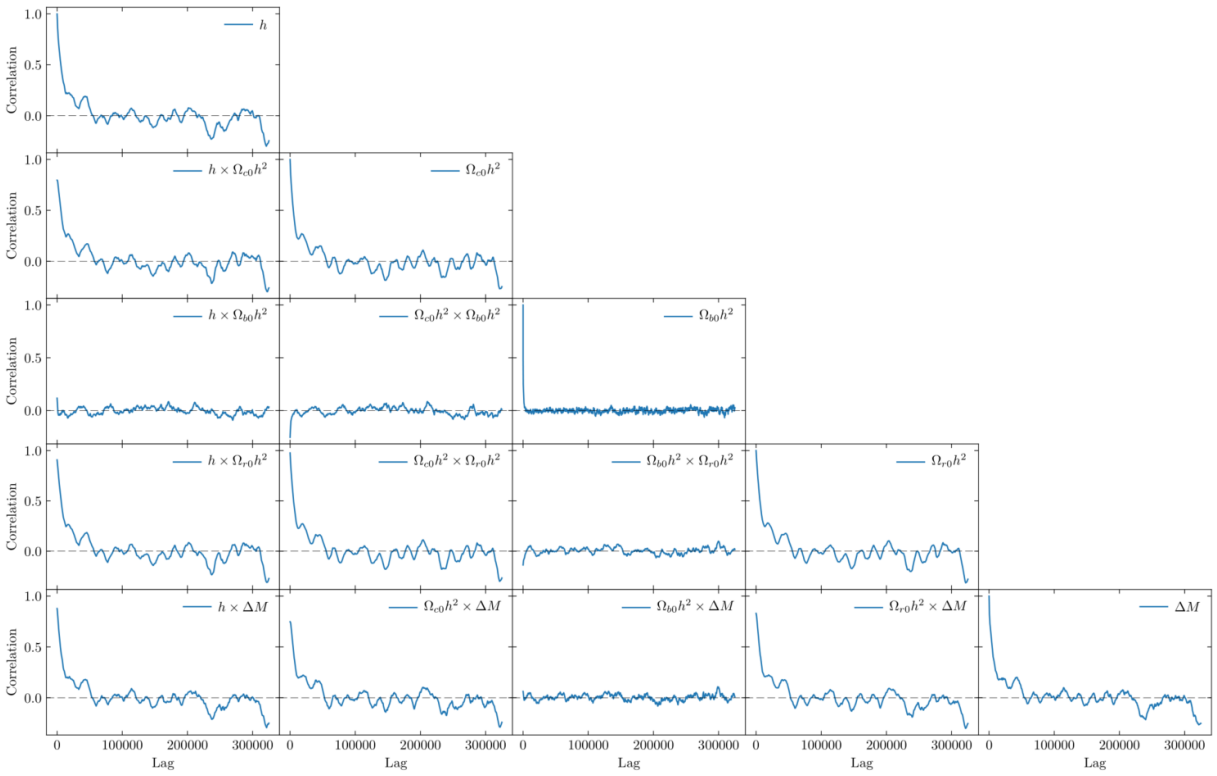

Fig. 8: Correlations in a chain.

Analyzing the chains

In the PT method, convergence is not assessed automatically, since each chain has a different temperature, thus not sampling from the same distribution. We could think of comparing the different chains after taking the frequencies of each bin to the power $1/\beta$, but I believe this could lead to inaccuracies due to statistical noise amplification in the higher-temperature chains, so I have not tested nor implemented this yet. In order to determine if you can already stop evolving your chains, I recommend running a second similar simulation so you can compare the ground temperature chains of the two (or more, but two is enough). To do this, create and run a second simulation with:

```
python epic.py disttest.ini chains=4 mode=PT steps=9000 limit=90000 nswap=30
```

and then:

```
python analyze.py <FULL-OR-RELATIVE-PATH-TO-FIRST-SIMULATION> calculate_R=<FULL-
↪OR-RELATIVE-PATH-TO-SECOND-SIMULATION>
```

You can do this at any time while the simulations are still running, even if they have different lengths. The only requirement is that the one in the `calculate_R` argument is the longer one. The minimum of the lengths of the two ground temperature chains will be used, the rest is disregarded. The directory of the main argument (the shorter chain) will contain the convergence results.

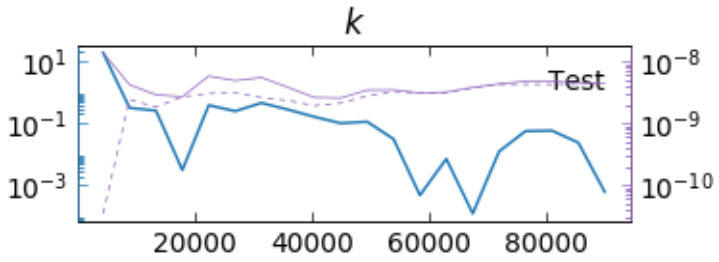

Fig. 9: Convergence between two chains from two Parallel Tempering simulations.

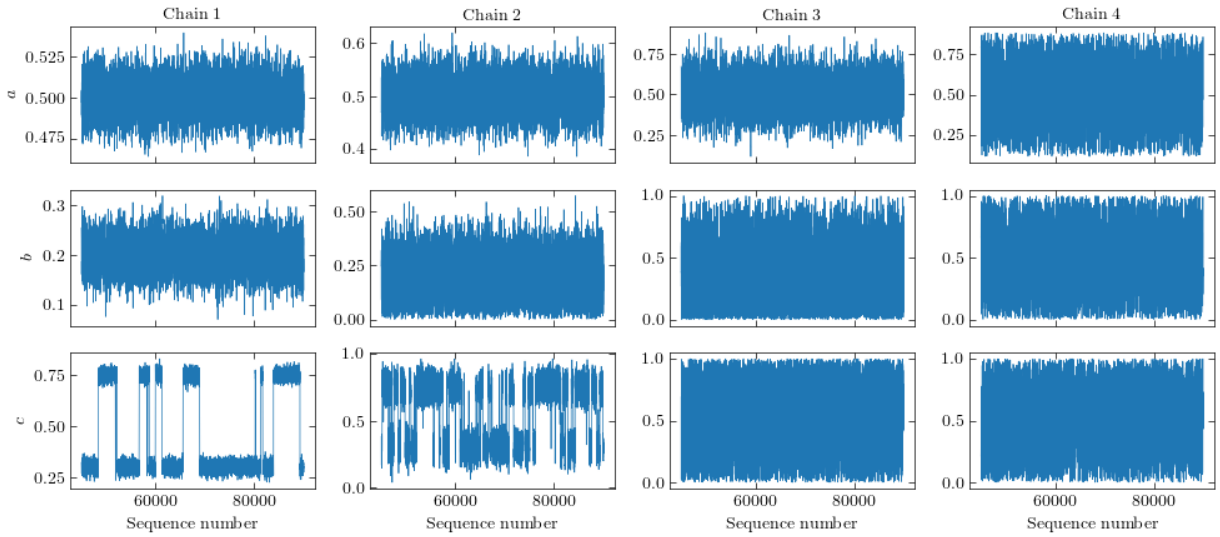

Fig. 10: Sequences of a Parallel Tempering MCMC simulation

During the simulation, the code will print the acceptance rates of the chains every 10 iterations. The acceptance rate of the ground temperature chain and the swap rate are printed every iteration. After `limit` steps, the code outputs the Akaike and Bayesian information criteria, `AIC` and `BIC`, and generates the usual plots.

Analyzing the resulting posterior and even checking convergence is fast since only the ground temperature chains need to be loaded. However, since this is Parallel Temperature, we can use the information of the higher-temperature chains to compute the Bayesian evidence, we just need to specify that we want so with the option `evidence`. If we ask to `plot_sequences` or the correlations (with `ACF`), since all chains will have to be loaded the evidence will also be computed anyway even without the argument `evidence`. It is interesting to inspect the sequences of the different temperatures. We can clearly see the jumps between the peaks occurring in the lower-temperature chains and the good mixing of the high-temperature chains, where the peaks are smoothed out.

The evidence is calculated from the area under the following curve

Of course, the more chains we use the more precise this calculation will be. So let us now see how accurate is our simulation. In the following triangle plot, we show the exact curves of the Gaussian

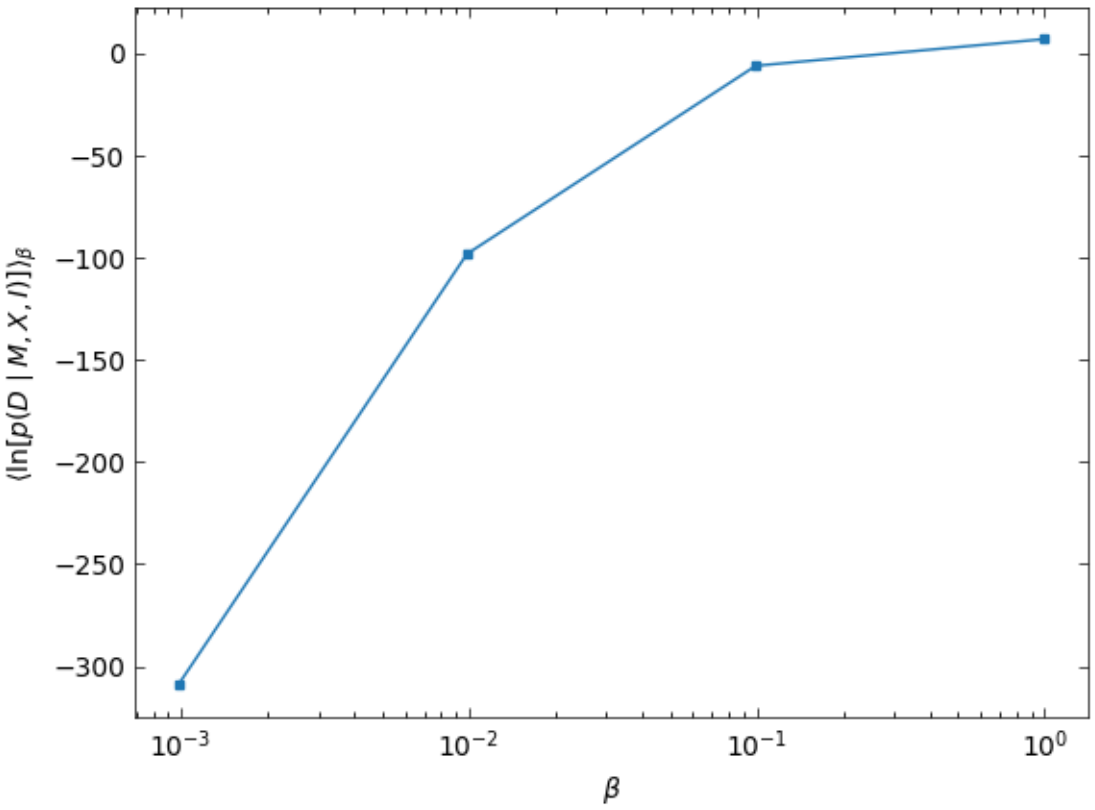

Fig. 11: Log-evidence calculated by thermo-integration of the PT chains.

distributions superposing the histograms.

Notice how the resulting histograms approximate very well the true functions, including the separated peaks given by the sum of two Gaussians. In standard MCMC, we would very probably obtain only one of the peaks. Which one depends on where the chain starts.

Let us improve this a little. These histograms with 20 bins are ok for the first two the parameters, but because of the wider range of data along the c axis, it is too little. Using a higher number of bins helps but most bins would be wasted since the division spans uniformly over the entire range of data and there are practically no points inside a large interval between the two peaks. The standard kernel density estimates with the bandwidth tuned for Gaussian-like distributions does not work well with multi-peaked distributions because the standard deviation will be rather large. We circumvented this limitation in this program by allowing the calculation of KDE on separate subsets of data. For this case, instead of running `make_kde([array,])` we can do `make_kde([array[array < 0.5], array[array > 0.5]])`, since the peak separation is very clear and there will probably be no points above $c = 0.5$ belonging to the first peak and no points below this value belonging to the second peak (when the peaks are closer together this may be much more complicated to accomplish). This will make KDE for the two subsets separately and

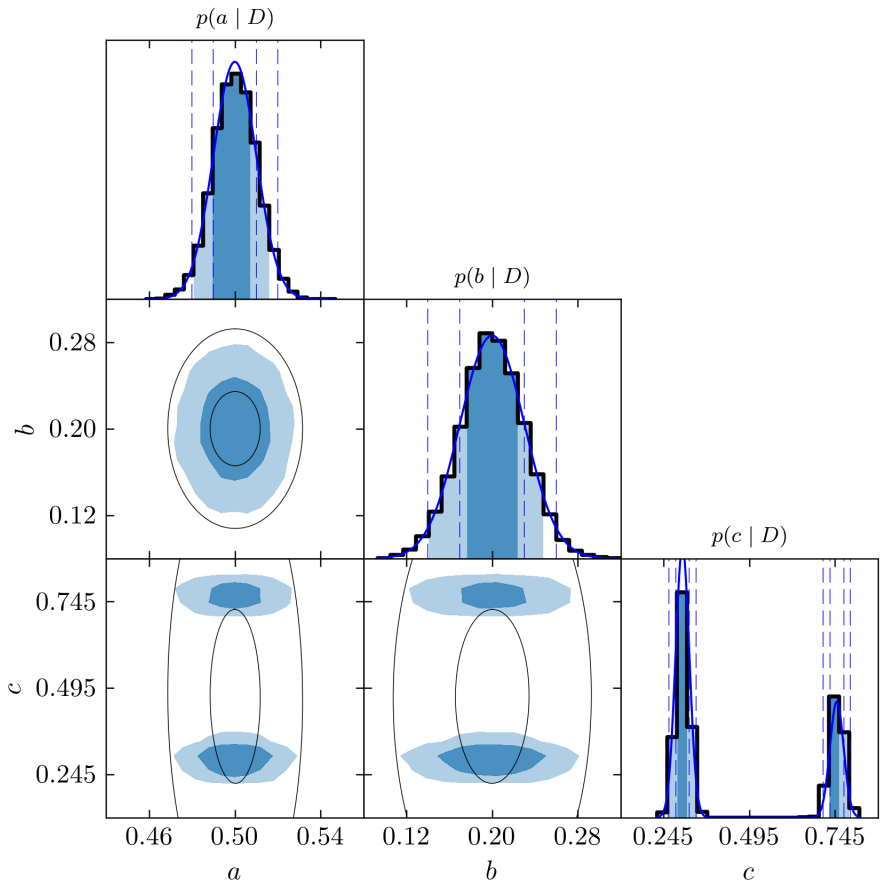

Fig. 12: Triangle plot for the PT simulation with the true marginalized probabilities in blue.

then join them together again, resulting in an accurate representation of the data. The `make_kde` function already receives a list of arrays to make this possible when we have separate arrays of data. In this case, we just need to tell the program where to split the single array in two. This is done by saving a file `disttest.true_fit` containing the weight, μ and σ parameters of the separated true distributions (only Gaussian is supported):

```
1       0.5     0.01
1       0.2     0.03
0.7     0.3     0.02    0.3     0.75    0.02
```

Notice how the KDE curves match very well the true functions, in blue lines:

The confidence levels are extracted from this marginalized distributions. The precision in the determination of the intervals is then limited by the resolution of the histograms (the number of bins). I recommend using KDE whenever is possible. Using more bins is also an option but requires having very long chains so this number can be raised considerably. KDE provides very good precision without needing too much data points (as long as there is enough for the chains to have converged). It is clear from these images how the KDE curves match much better 1σ and

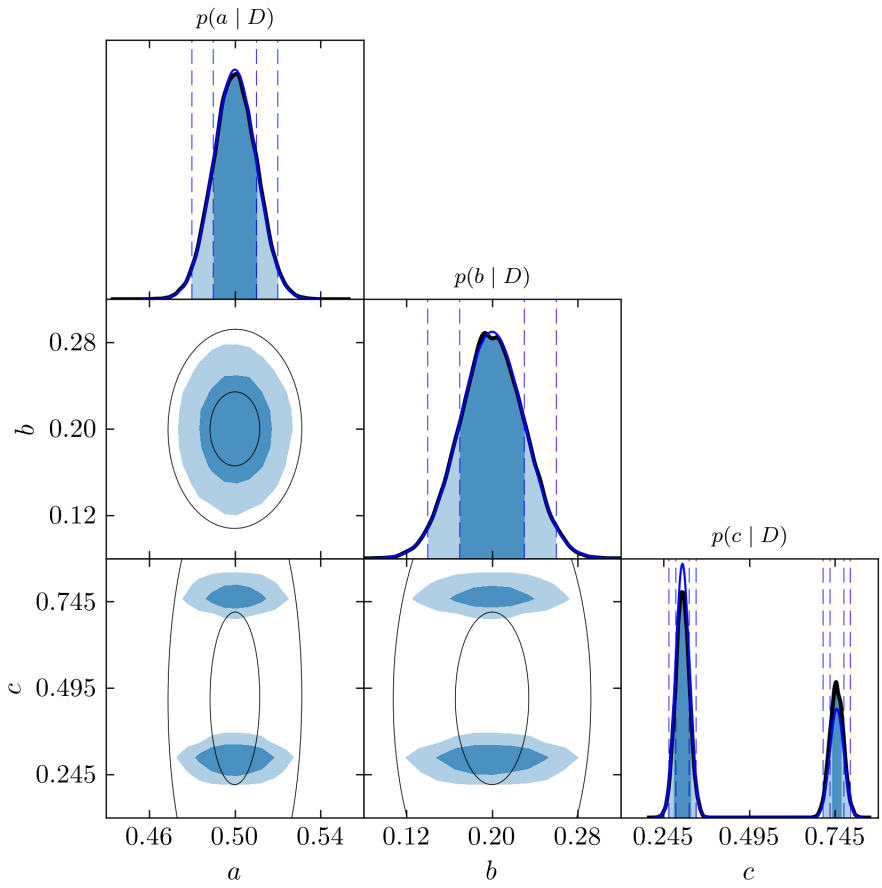

Fig. 13: Triangle plot for the PT simulation with KDE smoothed distributions and true density probabilities.

2σ confidence intervals. The shaded regions are not expected to match the dashed lines in the marginalized distribution of the third parameter (unless the two Gaussians had equal weights and widths), however, as they are calculated globally rather than locally.

At this moment, more than two separated peaks are not supported for this splitting by the `analyze.py` routine, although it can be done directly with any number of separated arrays in a list in the `make_kde` function.

Finally, let us see the results from this simulation. The image below is a screenshot of the `kde_table.pdf` file obtained compiling the `kde_table.tex` file saved in the `kde_estimates` directory:

Since the parameter c has two peaks of high density probability, the resulting confidence regions are composed of two intervals. The union symbol $\cup$ is used to represent this configuration. The result for the parameter b may look different from the true Gaussian function because of an artifact in the peak, showing doubled tips. The reported result considers the highest of the tips as the central value, the error bars being calculated from that value. For cases like this, it is interesting to simply report the Gaussian fits instead. They can be inserted automatically in the tables if we save

TABLE I. Results

Parameter	Prior	Best-fit	1σ C.L.	2σ C.L.
a	$[0.00, 1.00]$	0.49992	0.50002 ± 0.01010	0.50002 ± 0.02019
b	$[0.00, 1.00]$	0.19982	0.19999 ± 0.02985	0.19999 ± 0.05969
c	$[0.00, 1.00]$	0.29938	$0.29934^{+0.02597}_{-0.02404} \cup 0.74794^{+0.01605}_{-0.01149}$	$0.29934^{+0.04351}_{-0.04194} \cup 0.74794^{+0.03806}_{-0.03541}$

Fig. 14: Summary of the results of our analysis with the test distribution.

a file named `normal_kde.txt` in the simulation directory informing the name of the parameters (one per line) that look Gaussian. Then the mean and standard deviation will be printed in the table when `drawhistograms.py` is run again. This has been done for the result shown above.

Adaptation (beta)

An adaptive routine is implemented following the ideas in Łącki & Miasojedow 2016[1], but this needs more tests and is still in beta. Come back later to check.

5 Acknowledgments

I want to thank Gabriel Marcondes and Luiz Irber for their help. This work has made use of the computing facilities of the Laboratory of Astroinformatics (IAG/USP, NAT/Unicsul), whose purchase was made possible by the Brazilian agency FAPESP (grant 2009/54006-4) and the INCT-A.

[1] Łącki M. K. & Miasojedow B. "State-dependent swap strategies and automatic reduction of number of temperatures in adaptive parallel tempering algorithm". Statistics and Computing 26, 951–964 (2016).