Astronomy and Computing 2 (2013) 27–39

Contents lists available at ScienceDirect

Astronomy and Computing

journal homepage: www.elsevier.com/locate/ascom

Full length article

CosmoHammer: Cosmological parameter estimation with the MCMC Hammer☆

Joël Akeret [a,*], Sebastian Seehars [b], Adam Amara [b], Alexandre Refregier [b], André Csillaghy [a]

[a] University of Applied Sciences Northwestern Switzerland, Institute of 4D Technologies, Steinackerstrasse 5, 5210 Windisch, Switzerland
[b] ETH Zurich, Department of Physics, Wolfgang-Pauli-Strasse 27, 8093 Zurich, Switzerland

HIGHLIGHTS

- We analyse MCMC methods for cosmology regarding parallelisability and efficiency.
- We present the Python framework CosmoHammer for parallelised MCMC sampling.
- It enables us to estimate cosmological parameters on high performance clusters.
- To test the efficiency of CosmoHammer we use an elastic cloud computing environment.

ARTICLE INFO

Article history:
Received 6 December 2012
Accepted 25 June 2013

Keywords:
Markov chain Monte Carlo methods
Cloud computing
Cosmological parameter estimation

ABSTRACT

We study the benefits and limits of parallelised Markov chain Monte Carlo (MCMC) sampling in cosmology. MCMC methods are widely used for the estimation of cosmological parameters from a given set of observations and are typically based on the Metropolis–Hastings algorithm. Some of the required calculations can however be computationally intensive, meaning that a single long chain can take several hours or days to calculate. In practice, this can be limiting, since the MCMC process needs to be performed many times to test the impact of possible systematics and to understand the robustness of the measurements being made. To achieve greater speed through parallelisation, MCMC algorithms need to have short autocorrelation times and minimal overheads caused by tuning and burn-in. The resulting scalability is hence influenced by two factors, the MCMC overheads and the parallelisation costs. In order to efficiently distribute the MCMC sampling over thousands of cores on modern cloud computing infrastructure, we developed a Python framework called `CosmoHammer` which embeds `emcee`, an implementation by Foreman-Mackey et al. (2012) of the affine invariant ensemble sampler by Goodman and Weare (2010). We test the performance of `CosmoHammer` for cosmological parameter estimation from cosmic microwave background data. While Metropolis–Hastings is dominated by overheads, `CosmoHammer` is able to accelerate the sampling process from a wall time of 30 h on a dual core notebook to 16 min by scaling out to 2048 cores. Such short wall times for complex datasets open possibilities for extensive model testing and control of systematics.

1. Introduction

Bayesian inference is a standard procedure in cosmology when the measurement results are compared to predictions of a parameter-dependent model. The likelihood function $\mathcal{L}(\theta)$ is defined as the conditional probability of a measurement outcome x given the model parameters are fixed to θ: $\mathcal{L}(\theta) := p(x|\theta)$. Bayesian inference tells us how to update our knowledge from a prior distribution $q(\theta)$ to a new posterior distribution $p^{\text{new}}(\theta)$ which accounts for the recent measurement:

$$p^{\text{new}}(\theta) \propto q(\theta)p(x|\theta).$$

For exploring the likelihood function $\mathcal{L}$ or the posterior distribution p^{new} of the parameters, Markov chain Monte Carlo (MCMC) algorithms are today's method of choice when no functional expressions are available. Starting with the analysis of cosmic microwave background (CMB) data in 2001 (Christensen et al., 2001; Knox et al., 2001), MCMC methods turned into a vital tool in the analysis of astronomical data from all sorts of cosmological probes.

* Corresponding author. Tel.: +41 446338169.
E-mail address: jakeret@phys.ethz.ch (J. Akeret).

2213-1337/$ – see front matter © 2013 The Authors. Published by Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.ascom.2013.06.003

Most of the analyses in the literature rely on a particular instance of an MCMC method, the so-called Metropolis–Hastings algorithm from 1970 (Metropolis et al., 1953; Hastings, 1970). It is based on a random walk in the parameter space of the likelihood, serially proposing new positions that are accepted or rejected according to its likelihood weights. The Fortran program `CosmoMC` by Lewis and Bridle (2002), for example, is a widely used and very successful tool for parameter estimation in the cosmology community, based on the Metropolis–Hastings algorithm and the Boltzmann integrator `CAMB` (Code for Anisotropies in the Microwave Background by Lewis et al., 2000). A large number of scientific projects have used `CosmoMC`, among them prominent observations as the Wilkinson Microwave Anisotropy Probe (WMAP) (Dunkley et al., 2009) or the Sloan Digital Sky Survey (Tegmark et al., 2004).

Recently, Foreman-Mackey et al. (2012) presented `emcee`, a Python implementation of a novel MCMC algorithm by Goodman and Weare (2010) which has several potential advantages over Metropolis–Hastings: the sampling depends on less tuning-parameters and is independent of linear transformations of the parameters. Furthermore, `emcee` is not based on a single iterative random walk but uses an ensemble of walkers which can be moved in parallel.

Creating samples for the estimation of cosmological parameters from CMB measurements, for example, typically takes a few hours or even days on a desktop computer when using the Metropolis–Hastings algorithm. Whenever the sampling process has to be repeated a number of times in order to study the fit of various distinct models to the data or to find out about the systematics of a measurement, such a long run time gets increasingly problematic. We therefore focus on the parallelisability of MCMC methods in order to minimise the wall time of the calculation.

One of the main questions we try to answer is how fast one can, in principle, generate a useful sample of a given distribution when using `emcee` on an extended computing environment such as a cloud service or grid computer. For this purpose, we combined the Boltzmann integrator `CAMB` and the WMAP likelihood code and data (Larson et al., 2011; Komatsu et al., 2011) (both written in Fortran90) with the `emcee` sampler by Foreman-Mackey et al. (2012) in a Python framework – called `CosmoHammer` in the following – allowing us to study the example of parameter inference from CMB data in detail. As parameter estimation with CMB data is well documented in the literature and standard MCMC tools are publicly available, it is also a good reference point for the performance of the algorithm by Goodman and Weare as compared to a Metropolis–Hastings sampler.

`CosmoHammer` has been designed for optimal computational performance on large scale computing environments such as the Amazon elastic compute cloud (EC2).[1] We carry out a careful analysis of `CosmoHammer`'s sampling efficiency, focusing in particular on the implications arising from the simultaneous sampling on multiple computers.

The architecture of our code makes `CosmoHammer` easily extendable to other applications, as it is straight forward to plug in the Python modules containing further likelihood functions or codes for theory predictions.

We proceed as follows: in Section 2, the analysis of cosmological data using MCMC methods is briefly introduced and limitations of the current state-of-the-art are discussed. Section 3 explains how we address these limitations using the algorithm by Goodman and Weare (2010) and its implementation by Foreman-Mackey et al. (2012). Introducing our code in Section 4, we review its components, explain its architecture and outline the parallelisation scheme. We test the code in Section 5 by sampling the WMAP 7 likelihood and comparing the results to MCMC chains from the Metropolis–Hastings sampler `CosmoMC` by Lewis and Bridle (2002). In Section 6 we assess the performance of `CosmoHammer` on different cloud computing configurations. We discuss the results and conclude in Section 7. Finally, we explain the installation of the package and give detailed examples for running the algorithm in the Appendix.

2. Markov chain Monte Carlo for cosmology

In the following, we give a brief introduction to MCMC methods by discussing the Metropolis–Hastings (MH) algorithm as an example. Afterwards, we focus on the difficulties one faces when applying those methods to cosmological data.

2.1. MCMC methods

Assume that we are interested in a probability density function $p(\theta)$ (called target distribution in the following) which is not given explicitly but can be calculated numerically up to a constant factor. If we want to learn about $p(\theta)$ we need to estimate it from a finite number of numerical evaluations. MCMC algorithms generate a sample distributed according to the target distribution in a probabilistic fashion. We illustrate it using the MH algorithm as an example. A classic textbook for further information is MacKay (2003), which is also on the web.[2]

Imagine the sampling process as a random walk in the parameter space of the target distribution. The walk has to be initialised at a point θ_0 and a common method is to start at a random position close to the region where the likelihood is expected to be centred. In order to determine the next position, we need to specify a proposal density P which has to be easy to sample and is usually chosen to be a Gaussian distribution. The probability of randomly proposing the new position θ' when being at θ_t – the tth position in the sample – is given by $P(\theta'; \theta_t)$. After sampling θ' from P, the new position is accepted with probability

$$\min\left(1, \frac{p(\theta')}{p(\theta_t)}\frac{P(\theta_t;\theta')}{P(\theta';\theta_t)}\right).$$

The updated θ_{t+1} is finally given by θ' if the step is accepted or by θ_t if it is rejected. The set $\{\theta_t\}_{t\in\{0,\dots,T\}}$ converges to a sample from the target distribution $p(\theta)$ for large T (Metropolis et al., 1953; Hastings, 1970).

It is worth noting that the efficiency of the sampling crucially depends on the chosen proposal distribution. If P mainly proposes positions in the relevant parts of parameter space where the target distribution is large, the chain very quickly converges to a sample of $p(\theta)$. Yet, if the proposal distribution tends towards positions where $p(\theta)$ has low probability, most of the steps are rejected and samples will be heavily correlated. Using a proposal which is close to $p(\theta)$ is a convenient choice, as it guarantees that the proposed positions and the target are distributed similarly.

2.2. Application to cosmology

In cosmology the target distribution often arises from Bayesian inference. We consider the example of parameter estimation from CMB data for illustration. An observation measuring the CMB yields the angular power spectrum of the radiation as a result. At the same time, this power spectrum can be predicted from theoretical

[1] http://aws.amazon.com/ec2/.

[2] http://www.inference.phy.cam.ac.uk/itprnn/book.html.

models for the evolution of the universe. The minimal concordance model, ΛCDM, depends on six parameters and it takes a few seconds to calculate it using a numerical Boltzmann integrator like `CAMB` .

The likelihood function $\mathcal{L}$ of the parameters in the model arises from the conditional probability of the measurement result X given the parameters $\theta = (\theta_1, \ldots, \theta_d)$:

$$\mathcal{L}(\theta) := p(X|\theta).$$

If we already have information on the parameters in the form of a prior distribution $q(\theta)$ from the previous observations, $\mathcal{L}$ is used to update it according to Bayes' rule:

$$p^{\text{new}}(\theta) \propto \mathcal{L}(\theta)q(\theta).$$

Both $\mathcal{L}$ and the posterior p^{new} depend on the results of the numerical Boltzmann integrator and are hence not available as analytical functions of θ. As the number of parameters in this problem is usually greater than or equal to six, MCMC methods have to be used for estimating $\mathcal{L}$ or p^{new}. Although MCMC sampling makes the estimation feasible for large dimensions, it still suffers from the fact that calling the likelihood function – i.e. running the Boltzmann integrator for some specified parameters and comparing the results to the data in the CMB example – is costly in terms of time and resources. Consequently, we want our sampler to be as efficient as possible in terms of likelihood function calls per converged sample.

Additionally, MH sampling is by definition a serial process, iterating the calls to the likelihood function several thousand times. Using a single MH chain for inferring the parameters of ΛCDM from CMB data with an existing MCMC software package like CosmoMC, for example, takes at least a few hours on a notebook. Such runtimes are practical in cases where a single parameter estimation run suffices, but quickly become limiting in general applications.

Testing how well various other models different from ΛCDM fit the data implies that the likelihood has to be explored separately for every distinct model. Furthermore, it is often necessary to introduce so-called nuisance parameters which model the process of data generation. Altering the number and the effect of the nuisance parameters is a way to test for the systematics in a measurement and requires the MCMC sampling process to be repeated multiple times. Whenever the MCMC analyses have to be iterated, it is hence desirable to decrease their run time—either by improving on the underlying MCMC process or by running the calculations in parallel on a cluster or cloud computing environment. In the next section, we explain why the Python sampler `emcee` is a good choice for achieving such a speed-up.

3. Affine invariant ensemble sampler

We learned in Section 2 that MCMC sampling can be very time consuming in cosmological applications when the likelihood function is hard to evaluate numerically. Here we study how this can be overcome when using more recent MCMC algorithms such as the one by Goodman and Weare (2010) (called GW in the following) instead of MH. We therefore introduce the algorithm and its implementation by Foreman-Mackey et al. (2012) before discussing its advantages in terms of efficiency and parallelisation.

3.1. The ensemble sampler by Goodman and Weare

Instead of a single position which is updated during the course of the sampling, GW uses an ensemble of *walkers* which are spread on the parameter space of the target distribution. At every iteration, the walkers are randomly assigned to a partner walker chosen from the ensemble and a random point on a line connecting their positions is proposed as the next step.

More formally, let θ_t^i denote the position of walker i after t iterations. When updating this position to θ_{t+1}^i, we pick another walker θ_t^j at random with $j \neq i$, sample a value z from the *fixed* distribution

$$q(z) = \begin{cases} \dfrac{1}{\sqrt{z}} & \text{if } z \in \left[\dfrac{1}{a}, a\right] \\ 0 & \text{otherwise,} \end{cases}$$

with tuning parameter a and propose the position $\theta' = \theta_t^j + z(\theta_t^k - \theta_t^j)$. After evaluating the target distribution p at the proposed position, we accept the step if

$$z^{d-1}\frac{p(\theta')}{p(\theta_t^i)} \geq r,$$

with r being a random number from $[0, 1]$ and d the dimension of p.

GW is affine invariant, i.e. invariant under linear transformations of the target distribution. This implies in particular that the sampler is not sensitive to the scales of the parameters and does not depend on the covariances of the target distribution.

3.2. Emcee

The algorithm by Goodman and Weare (2010) was slightly altered and implemented as the Python module `emcee` by Foreman-Mackey et al. (2012). In this implementation, the algorithm does not update the walkers serially but instead divides them into two subgroups and updates all of the walkers from one subgroup at a time, using the other half of the walkers as their references.

3.3. Advantages

In Section 2 we mentioned that MH needs a proposal distribution P which is governing the efficiency of the algorithm. Thinking of P as a d-dimensional Gaussian distribution with unknown covariance matrix, one finds that the MH has $d(d+1)/2$ tuning parameters. When the scales of variances and covariances of the target distribution are well known before the sampling, the tuning parameters are very helpful: by supplying the MH with an appropriate covariance matrix, the parameter space is explored efficiently and the sample quickly converges. Yet, if the target distribution is unknown and the covariance matrix is only a guess, the efficiency of the algorithm generally decreases. In the case of non-Gaussian target distributions even a tuned Gaussian proposal might not match the target and the efficiency can be low.

On the other hand, GW is affine invariant and thus not sensitive to the scales of the target distribution. Hence, the efficiency of the sampling process does not depend on having a good estimate of the target distribution before the sampling and a tuning of the algorithm is not necessary.

Equally important is the use of an ensemble of walkers instead of a single chain. As the `emcee` implementation updates a large number of positions simultaneously at every iteration of the process, it can be parallelised in order to take advantage of a compute cluster (see Section 4).

The `emcee` sampler based on GW thus has the potential to improve upon the limitations outlined in Section 2. First, we expect the sampling process to be fairly robust to changes in data and model. Second, it is possible to distribute the sampling on the multiple nodes of modern cluster or cloud computing environments. Both advantages can be achieved without tuning of the sampler or parallelisation of the underlying likelihood and theory codes.

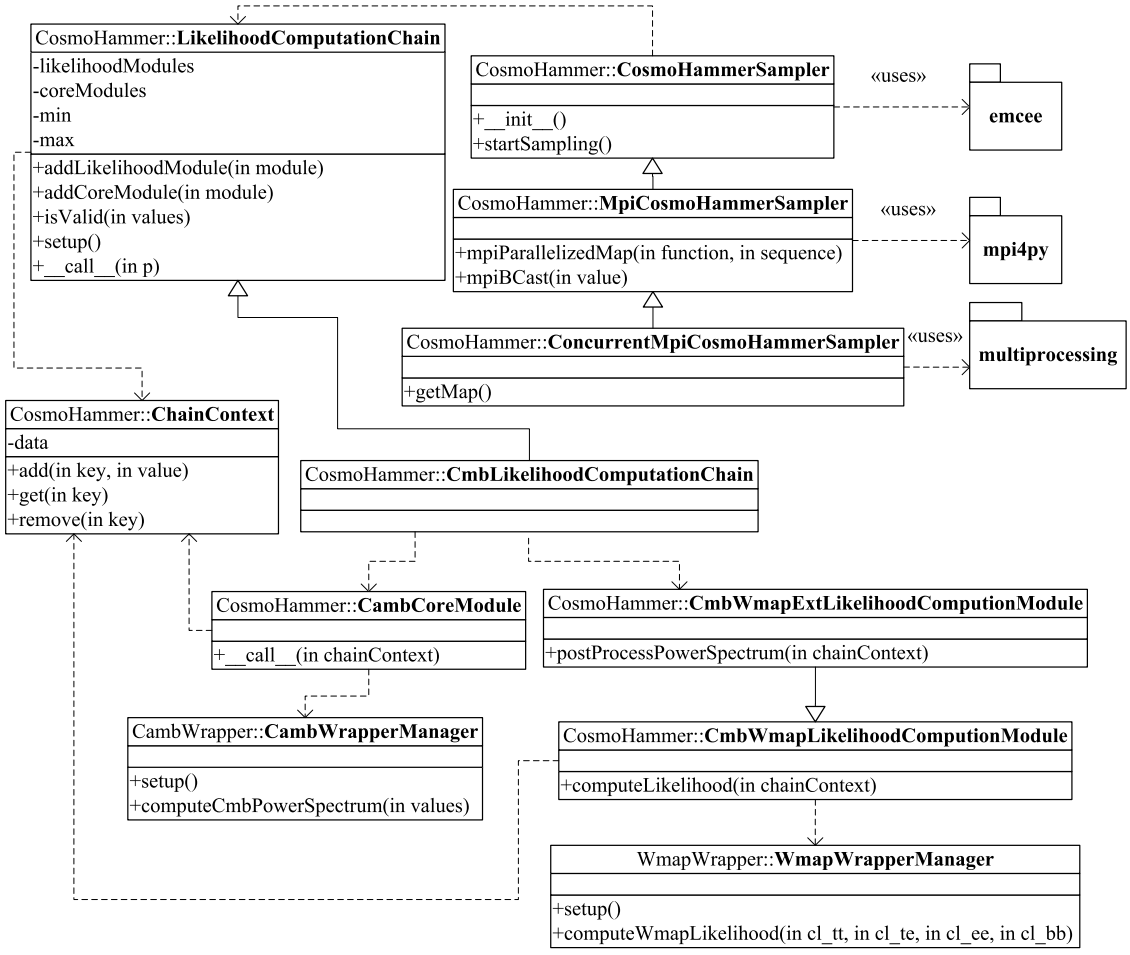

Fig. 1. Class diagram of `CosmoHammer`'s main architecture components.

4. Implementation of CosmoHammer

We developed a Python framework called `CosmoHammer` for the estimation of cosmological parameters. The software embeds the Python package `emcee` by Foreman-Mackey et al. (2012) and gives the user the possibility to plug in modules for the computation of any desired likelihood. The major goal of the software is to reduce the complexity when one wants to extend or replace the existing computation by modules which fit the user's needs as well as to provide the possibility to easily use large scale computing environments.

4.1. Architecture

We applied the principle of chaining modules for the computation of the likelihood as depicted in Fig. 1. The class diagram shows the underlying architecture and the most important classes. The architecture allows for the development of self-contained and tested modules which can be assembled in different sampling configurations. The internal design separates parameter space exploration, theory prediction and likelihood computation. This makes it easy to extend or replace these modules by new algorithms.

The concept of `CosmoHammer` divides the modules in two logical groups: modules for the computation of the likelihood and core modules. The core modules produce information like the CMB power spectrum or other theory predictions for the likelihood modules. The individual core modules can be combined in an instance of the *LikelihoodComputationChain* module (see Fig. 1). The chain stores the modules and initialises them in the right order. Furthermore, it ensures that the sampled parameters stay within physically motivated bounds during the sampling process.

The modules in the *LikelihoodComputationChain* communicate via the *ChainContext* in which arbitrary data can be stored and retrieved. This minimises the dependencies between the individual modules and ensures that they can be replaced without the need to change or extend `CosmoHammer`.

4.2. Components

`CosmoHammer` comes with a set of modules which compute the CMB power spectrum and the WMAP likelihood by wrapping the theory prediction code `CAMB` by Lewis et al. (2000) and the likelihood code from the WMAP team (Larson et al., 2011; Komatsu et al., 2011). It integrates the two Fortran modules by wrapping them using numpys F2PY.[3] The Fortran to Python interface generator provides the connection between Python and Fortran. In this way we can take full advantage of the well tested and widely used Boltzmann integrator `CAMB` as well as of the WMAP likelihood computation module while combining them with the `emcee` sampling algorithm. Furthermore, by using the wrapped code we benefit from the performance of Fortran in the convenient Python environment.

[3] http://www.scipy.org/F2py.

4.2.1. CAMB and WMAP modules

To use CAMB and the WMAP code we create an instance of the *LikelihoodComputationChain* and add an instance of the *CambCoreModule* and *CmbWmapLikelihoodComputationModule*. The *CambCoreModule* delegates the computation of the power spectrum to the *CambWrapperManager* and the *CmbWmapLikelihoodComputationModule* in turn delegates the likelihood computation to the *WmapWrapperManager*.

Alternatively an instance of the preconfigured *CmbLikelihoodComputationChain* can be created. This chain extends the regular *LikelihoodComputationChain* and ensures the correct setup of the two modules. A detailed example is given in Appendix B.

4.2.2. Samplers

CosmoHammer also provides different samplers. The samplers embed the emcee package and are responsible for logging and storing the obtained results. The *CosmoHammerSampler* is used when running on a single physical computer using one or multiple threads. The functionality, however, is limited to a single computer. The extended *MpiCosmoHammerSampler* provides the required functionality when CosmoHammer should take advantage of a computation cluster with multiple physical nodes like a cloud or grid computer. This sampler uses Message Passing Interface (MPI)[4] for the communication between the nodes in the cluster.

The implementation uses the paradigm of workload partitioning in which the work is split into blocks of nearly equal length. Every node then processes its block and returns the results to the master node. The master node gathers all results and merges them into a single list which then is returned to emcee.

When using a compute cluster the nodes often come with a large number of computational cores. Writing code that fully benefits from such a large number of cores is usually difficult. Therefore, it makes sense to split the workload also on the node since using a smaller number of cores per computation while performing multiple computations in parallel is typically more efficient. In Section 6.2 it can be seen how the execution time decreases when CosmoHammer uses multiple processes on one physical node.

If it is desired to distribute the workload to several nodes in a cluster as well as to spawn multiple processes on a node, the provided *ConcurrentMpiCosmoHammerSampler* can be used. This sampler introduces another level of parallelisation by using Python's built in multiprocessing package.

4.2.3. Initialisation

By default, all samplers initialise their walkers in a ball around a given centre point as suggested in Foreman-Mackey et al. (2012) The starting position θ_0^i of every walker i is then computed as follows:

$$\theta_0^i = P_c + N(0, 1) * P_w,$$

where P_c is the centre point and P_w is the start width. Both P_c and P_w have to be supplied by the user for every parameter. Furthermore, CosmoHammer comes with a built in generator producing a top-hat distribution as follows:

$$\theta_0^i = P_c + \epsilon * P_w,$$

where ϵ is a pseudorandom number ranging from -1 to 1.

If one prefers to launch the sampling process with a different starting strategy, it is possible to pass a custom implementation of a *PositionGenerator* to the *CosmoHammerSampler* instance.

[4] http://mpi4py.scipy.org/.

5. Tests and results

For testing the efficiency of CosmoHammer we compare it to CosmoMC, a widely used MCMC engine for cosmological parameter estimation by Lewis and Bridle (2002). CosmoMC is a Fortran90 code which also uses CAMB and the WMAP likelihood code for estimating cosmological parameters from CMB data, but it employs the MH algorithm for creating its MCMC chains. It furthermore contains an extensive choice of additional datasets and analysis modules.

To optimise the sampling process, Lewis and Bridle (2002) use the following seven default parameters for describing the ΛCDM model: the physical baryon density $\Omega_b h^2$, the physical dark matter density $\Omega_{\mathrm{DM}} h^2$, the ratio of the approximate sound horizon to the angular diameter distance θ, the reionisation optical depth τ_{re}, the scalar spectral index n_s, the primordial superhorizon power in the curvature perturbation on 0.05 Mpc^{-1} scales Δ_R^2, and finally A_{SZ}, a Sunyaev–Zel'dovich template normalisation. Furthermore, some of the parameters are rescaled to end up with similar orders of magnitude.

As emcee is affine invariant, the particular choice of parametrisation is not expected to influence its performance. For simplicity, we decided to use the same ones as CosmoMC up to rescaling, yet replacing θ by Hubble's constant H_0.

5.1. Analysis of MCMC chains

Following the lines of Foreman-Mackey et al. (2012) we adopt the *autocorrelation time* as our primary quality criterion for an MCMC sampler. Consider a probability density function $p(\theta)$, an MCMC sample $\{\theta_t\}$ of this distribution, and a function $f(\theta)$ whose mean $\langle f \rangle = \int f(\theta) p(\theta)\, d\theta$ we wish to estimate. The autocorrelation C_{ff} of $f(\theta)$ evaluated at the sample points $\{\theta_t\}$ with lag T is defined as:

$$C_{ff}(T) := \langle (f(\theta_t) - \langle f \rangle)(f(\theta_{t+T}) - \langle f \rangle) \rangle.$$

Typically, the autocorrelation of an MCMC sample is non-zero and decaying with increasing lag: $C_{ff}(T) \propto \exp(-\frac{T}{\tau_{ff}})$. However, if the points $\{\theta_t\}$ were independent of each other, the autocorrelation function would vanish for all $T \geq 1$.

Let us also define the normalised autocorrelation function

$$\rho_{ff}(T) = \frac{C_{ff}(T)}{C_{ff}(0)},$$

where $C_{ff}(0)$ is the variance of the sample $\{f_t\} := \{f(\theta_t)\}$. There are two relevant timescales connected to $\rho_{ff}(T)$. On the one hand, there is the exponential autocorrelation time τ_{exp} which is defined by

$$\tau_{\mathrm{exp}} = \limsup_{T \to \infty} \frac{T}{-\log |\rho_{ff}(T)|}. \tag{1}$$

On the other hand, the integrated autocorrelation time τ_{int} is given by

$$\tau_{\mathrm{int}} = \frac{1}{2} + \sum_{T=1}^{\infty} \rho_{ff}(T). \tag{2}$$

In general τ_{exp} and τ_{int} are not equivalent, although they are both equal to τ_{ff} for exponentially decaying autocorrelation functions C_{ff}.

The reason why τ_{exp} and τ_{int} are relevant for the analysis of MCMC samples is connected to the issues of thermalisation at the beginning of an MCMC chain (often called *burn-in*) and the statistical errors one has to account for when evaluating the expectation value of $f(\theta)$ using the sample $\{\theta_t\}$ (see the lecture notes by Sokal, 1989 for more information).

Table 1
Each `emcee` walker and `CosmoMC` chain is initialised at a random position around the mean, within a deviation sampled from a Gaussian distribution with given width.

Parameter	H_0	$\Omega_b h^2$	$\Omega_{\mathrm{DM}} h^2$	$10^9 \Delta_R^2$	n_s	τ_{re}	A_{SZ}	θ
Mean	70	0.0226	0.122	2.1	0.96	0.09	1	1.04
Width	3	0.001	0.01	0.1	0.02	0.03	0.4	0.002
Lower bound	40	0.005	0.01	1.48	0.5	0.01	0	0.5
Upper bound	100	0.1	0.99	5.45	1.5	0.8	2	10

The exponential autocorrelation time tells us how many iterations should be discarded at the beginning of the Markov chain in order to avoid an initialisation bias, since it measures the time we have to wait for two positions in the sample $\{f_t\}$ to be close to uncorrelated. Discarding a few exponential autocorrelation times at the beginning of the sampling typically suffices to suppress the bias.

At the same time, the integrated autocorrelation time determines the standard error of the mean through:

$$\mathrm{Var}(\bar{f}) = \frac{2\tau_{\mathrm{int}}}{N}\mathrm{Var}(f_t), \tag{3}$$

where N is the size, $\bar{f}$ is the mean, and $\mathrm{Var}(f_t)$ is the variance of the sample $\{f_t\}$.

An estimate for $C_{ff}(T)$ is given by

$$C_{ff}(T) \approx \hat{C}_{ff}(T) = \frac{1}{N-T}\sum_{t=1}^{N-T}(f_t - \bar{f})(f_{t+T} - \bar{f}). \tag{4}$$

Estimating the integrated autocorrelation time from (4) is not easy because of issues regarding the statistical noise in the large T limit of $\hat{C}_{ff}(T)$. Yet, the autocorrelation function for our problem is exponentially decaying, meaning that we can also evaluate τ_{exp} instead of τ_{int}. We find that estimating τ_{int} from a fit to $\hat{C}_{ff}(T)$ is the best choice for evaluating the autocorrelation time of this particular problem (see Appendix C for more information). As τ_{exp} and τ_{int} are equivalent for our purposes, we denote both as the autocorrelation time τ for simplicity.

5.2. Multiple walkers

We typically demand that MCMC samples satisfy the following accuracy criterion: the statistical error of the mean $\bar{f}$ we wish to estimate – defined in Eq. (3) – has to be smaller than a given fraction ϵ of the standard deviation of its marginal distribution. In other words

$$\sqrt{\frac{\mathrm{Var}(\bar{f})}{\mathrm{Var}(f)}} = \sqrt{\frac{2\tau}{N}} \leq \epsilon \tag{5}$$

and consequently

$$N \geq \frac{2\tau}{\epsilon^2}, \tag{6}$$

where N is the total size of the sample. For multiple walkers or independent chains, N is given by the number of steps per walker or chain n (called sequential steps in the following) times the number of walkers L and hence

$$n \geq \frac{2\tau}{\epsilon^2 L}. \tag{7}$$

Eq. (5) only holds when the sample is unbiased by the initialisation and this is true in the asymptotic limit of large n. When sampling on a cloud computing infrastructure, however, we need a large number of walkers L for maximum parallelisability and hence expect rather small n according to (7).

Yet, the sample gets close to unbiased when discarding the initial steps of every walker or chain as burn-in. We already mentioned in Section 5.1 that such a burn-in phase is expected to last for a few autocorrelation times. When using only a few walkers, i.e. L close to one, n is large and the burn-in phase is a subdominant part of the overall sampling. As L grows, though, the number of sequential steps n becomes comparable to the burn-in length and the discarded samples turn into a dominant fraction of the overall sample size. We analyse the consequences of this result in Section 5.4.

5.3. Configuration of CosmoMC and CosmoHammer

In the following we want to compare `CosmoHammer` to two different configurations of `CosmoMC`. In each case we sample the likelihood given by the Fortran90 code and the data of the WMAP 7 team.

The first `CosmoMC` instance is an out-of-the-box approach, using the standard configuration shipped with the `CosmoMC` package. It initialises the chain in a small ball around an estimated mean of the likelihood using the numbers from Table 1 and supplies a covariance matrix to specify the Gaussian distribution which is used as the proposal. We will refer to this configuration as *fine-tuned `CosmoMC`*.

The second approach employs an option of `CosmoMC` which splits the sampling into two phases. Starting with an initial guess for the covariance matrix in the tuning phase – a diagonal matrix with estimated variances in our case – the sampler continuously updates the proposal's covariance matrix from the last half of the generated samples. Afterwards, `CosmoMC` uses the generated proposal to create the samples for estimating the likelihood. This approach will be called *self-tuning `CosmoMC`* from now on. When using multiple independent chains, `CosmoMC` allows one to automatically stop the tuning phase once a convergence criterion is fulfilled. We use this option to ensure that the sampling after the tuning phase is comparable to the fine-tuned `CosmoMC` process. For our analyses, we used 10 independent chains for both `CosmoMC` configurations.

For `CosmoHammer` we call `CAMB` in exactly the same fashion as the standard `CosmoMC` configuration does (i.e. we use the same theoretical model for our cosmology) and also initialise the walkers according to Table 1. We furthermore used `emcee` with 350 walkers, a rather arbitrary pick which was convenient to work with.

The bounds listed in Table 1 are needed to ensure that the parameters which are passed to the Boltzmann integrator `CAMB` make sense physically. Whenever the sampler proposes a position which is out of bounds, `CosmoHammer` returns zero probability immediately.

5.4. Efficiency

A good estimate for the autocorrelation function is important for the analysis of the sample. For both `CosmoMC` and `CosmoHammer` the autocorrelations

$$\hat{C}_X(T) = \frac{1}{n-T}\sum_{t=1}^{n-T}(X_t - \bar{X})(X_{t+T} - \bar{X}),$$

Table 2
Estimated autocorrelation time τ for the seven parameters and the three sampler configurations.

Sampler	H_0	$\Omega_b h^2$	$\Omega_{\mathrm{DM}} h^2$	Δ_R^2	n_s	τ_{re}	A_{SZ}
`CosmoHammer`	44.4	44.7	44.4	43.4	43.8	43.8	48.2
Fine-tuned `CosmoMC`	17.3	17.1	15.1	13.8	17.6	14.4	18.4
Self-tuning `CosmoMC`	16.6	14.9	19.1	13.3	15.0	14.8	16.3

with X being one of the seven dimensions of the parameter-space, behave equivalently for all parameters. As the estimation of the autocorrelation times for the different sampler configurations is not straightforward, we give a detailed description of our procedure in Appendix C. The results can be found in Table 2. It is not very surprising that the `CosmoMC` processes are more efficient in terms of calls per independent sample, as they are using a well tuned proposal to sample a target distribution which is itself close to normally distributed. On the other hand, `emcee` needed no tuning while still performing reasonably well in terms of autocorrelation time.

Additionally we need to consider the optimal length of the burn-in period before the actual sampling starts. As was discussed in Section 5.2, this is particularly important if we want to iterate a large number of walkers. Let $\{X_t^i\}$ denote the position of walker $i \in \{1, \ldots, L\}$ at iteration $t \in \{1, \ldots, n\}$ for parameter X. For finding the burn-in length, we observed the mean sequential step $\bar{X}_t$

$$\bar{X}_t = \sum_{i=1}^{L} X_t^i$$

for increasing time t. Once the positions of the walkers are drawn from the target distribution in an unbiased way, we expect $\bar{X}_t$ to vary around the true mean as predicted by the standard error in the mean

$$\sigma(\bar{X}_t) = \frac{\sigma}{\sqrt{L}},$$

where σ is the standard deviation of the marginal target distribution for parameter X. As long as this is not the case, the walkers are still biased by the initialisation.

Using this method we find that a safe choice for the burn-in period is 250 for both the fine-tuned `CosmoMC` and `CosmoHammer`, as can be seen in Fig. 2 for parameter Δ_R^2. This is surprising as both processes have very different autocorrelation times, but the MCMC algorithm of `emcee` turns out to be very efficient in getting rid of its initialisation bias. The self-tuning algorithm needs about 2000 iterations to estimate its proposal distribution, but already starts at an unbiased position afterwards. The reason for such a long tuning phase is that the sampling efficiency is very poor when the sampler is untuned in the beginning.

We can now consider the statistical error ϵ from Eq. (7) as a function of burn-in or tuning length b, sequential steps n, number of walkers L, and autocorrelation time τ:

$$\epsilon = \sqrt{\frac{2\tau}{(n-b)L}} \quad \text{for } n > b. \tag{8}$$

This is visualised in Fig. 3 for all sampler configurations, using τ from Table 2 and the corresponding burn-in and tuning values.

We find that the most efficient way to create a sample which satisfies the accuracy criterion (5) is to use a fine-tuned `CosmoMC`. This is not surprising, as it has a good estimate of the target distribution before it even starts the sampling process, resulting in a short autocorrelation time and burn-in phase. Yet, a well tuned MH is typically not available when analysing new data. In this case, the self-tuning `CosmoMC` configuration or similar concepts with lengthy tuning phases have to be used for configuring the sampler.

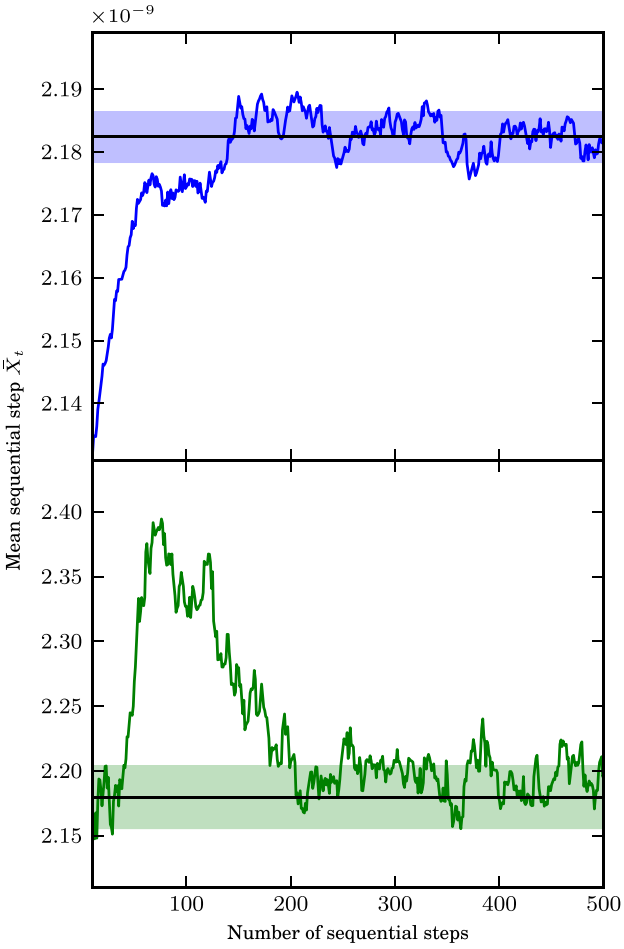

Fig. 2. Mean sequential steps $\bar{X}_t$ of `CosmoHammer` (top) and fine-tuned `CosmoMC` (bottom) for parameter Δ_R^2. The shaded regions depict the error in the mean $\sigma(\bar{X}_t)$ for every mean sequential step. Self-tuning `CosmoMC` runs need about 2000 initial steps for tuning, but start at an unbiased position afterwards.

It is this tuning phase which turns out to be the bottle-neck for the parallelisation of a MH sampler. We can see in Fig. 3 that for as few as 10 walkers, `CosmoHammer` reaches the 10% error regime $\epsilon \leq 0.1$ before the self-tuning `CosmoMC` run even finalises the tuning at $n = 2000$. When estimating the parameter X from the target distribution, the result usually reads

$$\hat{X} = \bar{X} \pm \sigma(X),$$

with estimate $\hat{X}$, sample mean $\bar{X}$, and standard deviation $\sigma(X)$. Consequently, the statistical error does not affect $\hat{X}$ up to the second digit of $\sigma(X)$ when ϵ is smaller than 0.1 and is hence sufficiently small for most applications in cosmological parameter estimation.

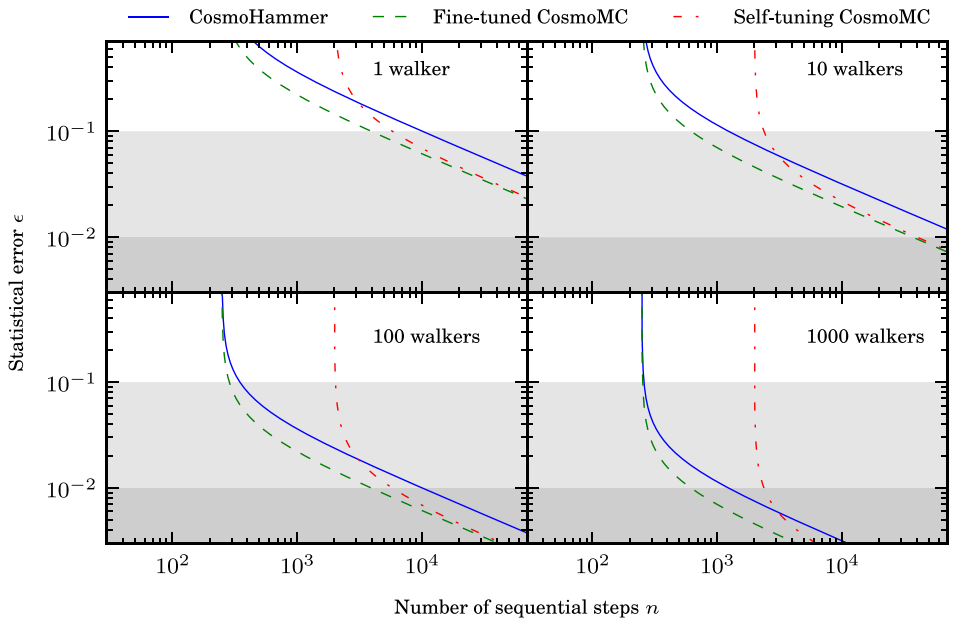

Fig. 3. Statistical error ϵ, defined in Eq. (8), as a function of sequential steps n and for different numbers of walkers. For simplicity, we assume in this plot that burn-in length and autocorrelation time are independent of the number of walkers or chains. The shaded regions show where the respective configurations reach statistical error $\epsilon = 10\%$ and $\epsilon = 1\%$.

Table 3
Mean, standard deviation and the expected statistical error of the parameters sampled by `CosmoHammer` and `CosmoMC` .

Parameter	`CosmoHammer`	`CosmoMC`	Statistical error
H_0	$70.4^{+2.8}_{-2.5}$	$70.2^{+2.4}_{-2.1}$	± 0.1
$100\Omega_{b0}h^2$	$2.247^{+0.065}_{-0.051}$	$2.243^{+0.057}_{-0.057}$	± 0.002
$\Omega_{dm0}h^2$	$0.1107^{+0.0063}_{-0.0049}$	$0.1116^{+0.0048}_{-0.0055}$	± 0.0002
$10^9\,\Delta^2_R$	$2.174^{+0.078}_{-0.069}$	$2.168^{+0.077}_{-0.068}$	± 0.003
n_s	$0.967^{+0.014}_{-0.014}$	$0.966^{+0.014}_{-0.012}$	± 0.0005
$100\tau_{\mathrm{re}}$	$8.7^{+1.5}_{-1.3}$	$8.6^{+1.5}_{-1.3}$	± 0.05
A_{SZ}	$0.92^{+0.71}_{-0.64}$	$1.01^{+0.65}_{-0.72}$	± 0.02

5.5. Parameter estimation

We know from the previous discussion that the errors we expect for our estimates of the mean are of the order $\sigma\sqrt{2\tau/N}$, where σ is the standard deviation of the target distribution for the respective parameter. The total sample size after the burn-in was chosen to be $N = 250 \times 350 = 87{,}500$ for `CosmoHammer`, predicting an accuracy in the mean estimate of about 3.4% relative to the standard deviation. The same precision will be reached when using about $N = 3500 \times 10 = 35{,}000$ samples from `CosmoMC`.

As we chose N such that the error is smaller than 3.4% of the standard deviation, the parameter estimates of the different sampler configurations are supposed to vary on this scale, too. We can see from Table 3 that this is indeed the case. Finally, Fig. 4 shows the projections of the 7-dimensional likelihood into one and two dimensional marginal distributions.

We conclude that the samples of `CosmoMC` and `CosmoHammer` behave just as expected from our analysis in Section 5.1. In particular, this means that the quality of the sampling is well understood once the autocorrelation of the MCMC process is known. Tests based on multiple independent instances of the employed sampler configurations support these conclusions.

6. Performance measures and metrics

The performed computations required a large amount of computational power. We therefore decided to explore the possible benefits of cloud computing by means of `CosmoHammer`. One of the major advantages of this computing strategy is that the configuration of the cloud can be easily tailored to the problem at hand. In the cloud more computational power can be added within minutes by renting extra compute instances on demand, resulting in an optimised execution time. The following section describes the environment and the configuration used to perform the benchmarks.

Table 4
Amazon EC2 instance types used for the benchmarks in Fig. 5.

	Master node	Worker node
Name	Large instance	Cluster compute eight extra large
API Name	m1.large	cc2.8xlarge
Memory	1.7 GB	60.5 GB
Instance storage	160 GB	3370 GB
Processing Power	2 Cores	32 Cores

6.1. Environment

As cloud service provider we decided to use Amazon EC2 in combination with the Starcluster Toolkit (Software Tools for Academics and Researchers).[5] Table 4 shows the configuration of the instance types we used for the benchmarks. The high performance computing cluster consisted of one master node and several worker nodes. At the moment of the benchmarks one cc2.8xlarge Instance ships with 2× Intel Xeon E5-2670, eight-core architecture with Hyper-Threading, resulting in 32 cores per node. We used a m1.large instance as master node mainly to benefit from the high I/O performance in order to reduce the loading time of the WMAP data.

To compile the native Fortran modules, we used Intel's ifort compiler and mkl libraries, Python 2.7, and numpy 1.6.2.

6.2. Benchmarks

The results depicted in Fig. 5 have been realised with one to 64 worker nodes (32–2048 cores) and different combinations of

[5] http://star.mit.edu/cluster/.

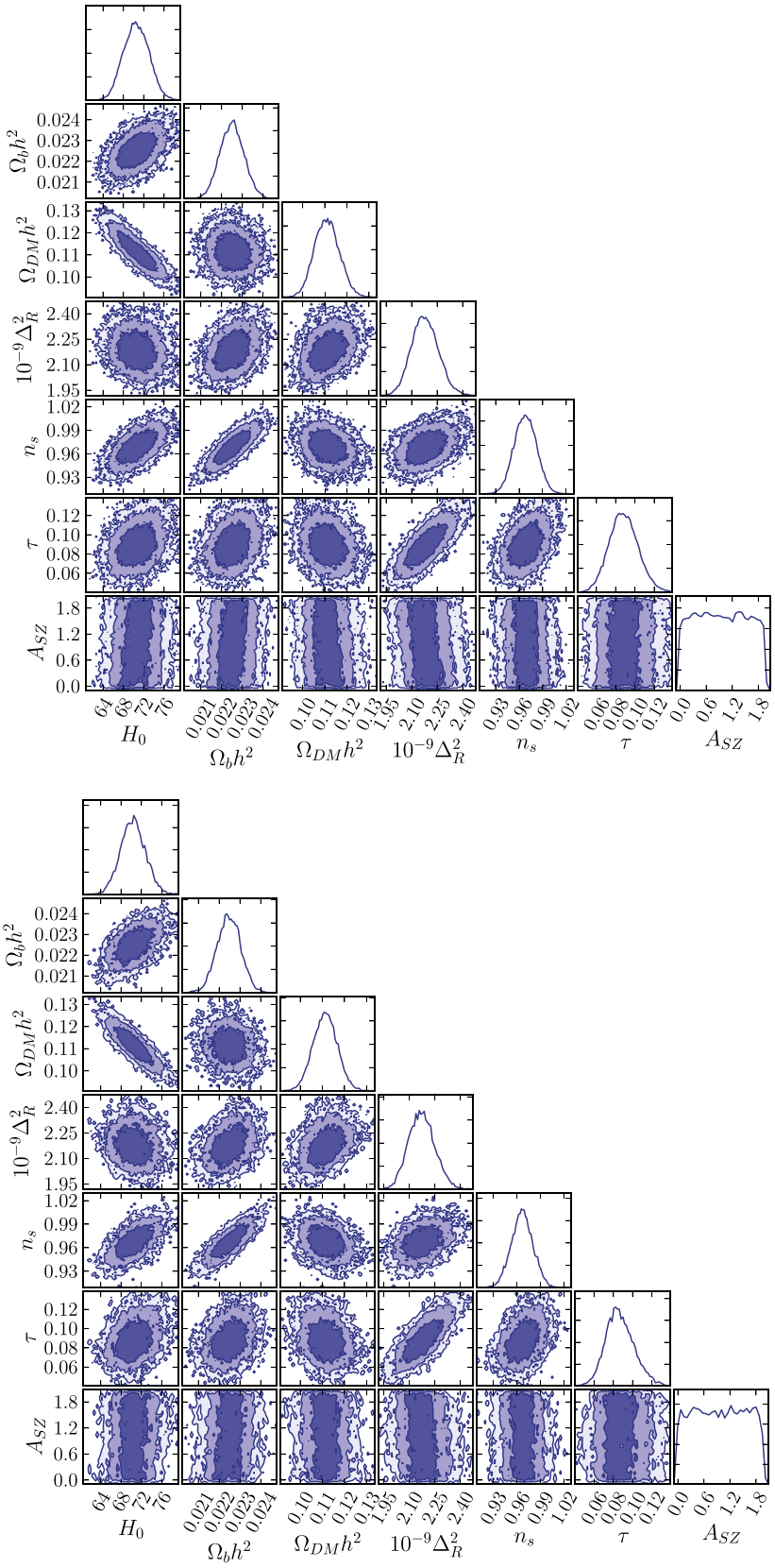

Fig. 4. One and two dimensional marginal distributions of the WMAP 7 likelihood as sampled by `CosmoHammer` (top) and `CosmoMC` (bottom) for all parameter combinations.

processes and threads per node. The processes define the number of computations executed in parallel and the threads represent the number of cores used for one computation. In the case of 4 processes and 8 threads, for instance, there were four Python processes working in parallel on every node, each of them spawning eight threads.

For all test runs, `CosmoHammer` was configured to use 350 walkers and to run 500 sampling iterations, resulting in 175,000

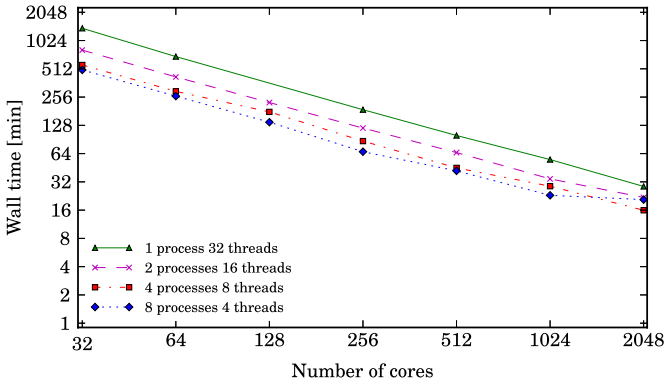

Fig. 5. Run time behaviour of `CosmoHammer` with changing number of cores using different parallelisation schemes.

samples per run. This sample size was also used for the parameter estimation in Section 5.5. We used a *LikelihoodComputationChain* with the *CambCoreModule* and the *CmbWmapExtLikelihoodComputationModule* for the computation of the likelihood.

Fig. 5 shows that the wall time T as a function of number of cores N behaves as a power law: $T \propto N^{-\alpha}$, i.e. it is linear on a logarithmic scale with a mean α of 0.89. The best result was achieved using 64 nodes with 32 cores, four processes and eight threads. Using this configuration, the computation took about 16 min.

6.3. Discussion

The implemented algorithm for the parallelisation is efficient yet easy to understand. Note, however, that this way of parallelisation is only beneficial when the executed computations are time and resource consuming. Distributing the workload in a compute cluster always implies the transfer of information over the network which is typically slower than transferring information between local processes by an order of magnitude. Therefore, the advantage of additional computing resources and the disadvantage of the network overhead have to be weighted.

Since we use `CosmoHammer` for the estimation of cosmological parameters which implies the execution of computationally intensive theory prediction modules like `CAMB`, the overhead of the network latency plays a secondary role. We expect that `CosmoHammer` will be extended by additional theory and likelihood modules so that the network overhead can be neglected and the number of computational nodes and therefore CPUs can be increased. Nevertheless, there is a given maximum: the number of nodes cannot exceed the number of walkers.

`CosmoHammer` cannot only take advantage of multiple nodes in a cluster but can be parallelised also on every node. As illustrated in Fig. 5, there is a maximum for the number of processes on the node, too. The test run using eight processes and four OMP threads reaches a plateau when using 64 nodes. This is caused by the workload partitioning where the 350 walkers are equally distributed on 64 nodes. Consequently, more processes are available than walkers to process, which in turn causes the processes and their spawned threads to idle instead of being used for the computation of the power spectrum.

Inferring parameters from WMAP data takes 30 h on a dual core notebook with a single MH process. Using one chain, 37,000 samples need to be created to obtain results in the specified error regime. When parallelising an MCMC process the scalability is affected by two overheads: First, the number of samples has to be increased to $N = 2 \times 250 \times 350 = 175{,}000$ due to the growth of burn-in overheads and a larger autocorrelation time as described in Section 5.5. This consequently results in additional computational costs. Second, the distribution of the workload in the compute cluster causes a loss of efficiency. In the benchmarks, a power law behaviour with an exponent of $\alpha = 0.89$ was measured, resulting in a scaling overhead of 11% when the number of cores is doubled. Accounting for both effects, the wall time decreases to 16 min on 2048 cores.

7. Conclusion

In cosmological applications, accelerating MCMC methods is crucial whenever the sampling has to be repeated numerous times and evaluating the target distribution is computationally expensive. Examining the measurement systematics of an astrophysical observation in an iterative analysis, for example, requires sampling an extensive amount of target distributions, each depending on time intensive simulations. Whenever it is not possible to massively parallelise the likelihood code, starting a large number of short chains enables us to evolve them in parallel not only on a couple of local CPUs, but globally on a large scale computing infrastructure.

We found that two characteristics are crucial for the performance of parallelised MCMC sampling: it has to be efficient in terms of autocorrelation time and – equally important – it has to sample efficiently without needing much overhead for tuning and equilibration. It is this overhead which puts a fundamental lower bound on the run time of the sampling procedure and can limit the application of MCMC methods when a large number of target distributions has to be explored. The `emcee` sampler by Foreman-Mackey et al. (2012) turned out to be a good choice regarding those preconditions: it not only requires no tuning of the algorithm in general, but also avoids an initialisation bias very quickly while performing fairly well in terms of autocorrelation time when applied to the likelihood of the WMAP 7 observations.

In order to exploit the advantages of `emcee` on large cloud computing configurations or similar infrastructure, we introduced `CosmoHammer`, a Python framework for parallelised MCMC sampling. It enabled us to explore arising computer science technologies like elastic cloud computing for scientific applications. The elastic nature of Amazon EC2, for example, ensures that scientists and engineers do not have to wait in long queues to access shared clusters as the computational power can be increased within minutes by renting additional compute instances. We reduced the time for estimating cosmological parameters using the WMAP 7 likelihood from 30 h on a desktop computer running a MH sampler to about 16 min on a cluster with 2048 cores on Amazon EC2. Furthermore, `CosmoHammer` scales linearly with increasing number of cores, highlighting the efficiency of the parallelisation concept.

The implementation is not limited to inferring the parameters of ΛCDM using `CAMB` and WMAP, but the application programming interface of `CosmoHammer` allows for the extension of its application to further cosmological probes and models using newly developed and self-contained Python modules.

In the Appendix, we describe the installation of `CosmoHammer` and give a guide for the correct setup when using `CAMB` and the WMAP likelihood.

Acknowledgements

We want to thank Joel Bergé, Lukas Gamper and Joe Zuntz for helpful discussions, as well as Laurenz Gamper for his help with licensing and homepage.

Appendix A. Download and installation

The tarballs containing the most recent and stable version of CosmoHammer and the wrapper modules can be found at http://www.astro.ethz.ch/refregier/research/Software/cosmohammer.

CosmoHammer relies on the following Python packages:

- emcee—affine invariant MCMC sampler
- numpy—Numerical Python
- mpi4py—Python wrapper for mpi (optional, only used if CosmoHammer is supposed to be distributed on multiple nodes).

When additionally using the wrapped WMAP likelihood module, the WMAP data[6] needs to be accessible on the filesystem and CFITSIO[7] has to be installed. We tested CosmoHammer with Python 2.6, Python 2.7, numpy 1.6.2, emcee 1.1.2, and mpi4py 1.3, but it is likely to work with earlier versions of these Python packages. For the compilation of the Fortran wrappers, a Fortran compiler and mkl libraries are required. The current distribution has only been tested with Intel's ifort compiler and mkl libraries, though.

To install the components the tarballs have to be extracted and the following commands have to be executed in the root directory of each module:

```
% python setup.py build
% sudo python setup.py install
```

Every module comes with a Readme containing detailed information.

Appendix B. Examples

We show how to use CosmoHammer to estimate cosmological parameters with CAMB and WMAP likelihood. The listed source code is also available in the distributed tarball.

The import statements for both CosmoHammer and numpy have been omitted. We first define the initialisation by specifying the estimated start centre, minimal and maximal value, and the start width using, for example, the values defined in Table 1.

```
#parameter start centre, min, max, start width
params=np.array([[70, 40, 100, 3],
    [0.0226, 0.005, 0.1, 0.001],
    [0.122, 0.01, 0.99, 0.01],
    [2.1e-9, 1.48e-9, 5.45e-9, 1e-10],
    [0.96, 0.5, 1.5, 0.02],
    [0.09, 0.01, 0.8, 0.03]])
```

After instantiating the chain and passing the min and max parameter boundaries as follows, the chain will check if the proposed walker positions are within the boundaries before calling the modules.

```
chain=LikelihoodComputationChain(
    min=params[:,1],
    max=params[:,2])
```

We create an instance of the *CambCoreModule* and add it to the previously instantiated chain. At this point it is possible to add other modules, e.g. further theory prediction modules.

```
camb=CambCoreModule()
chain.addCoreModule(camb)
```

Next, we have to instantiate the WMAP likelihood computation module and then add the module to the chain. Here, further implementations of likelihood modules can be added.

```
wmapLikelihood=CmbWmapLikelihoodComputationModule
    ()
chain.addLikelihoodModule(wmapLikelihood)
```

Alternatively, we could create an instance of a *CmbLikelihoodComputationChain* which automatically adds the previous modules.

```
chain=CmbLikelihoodComputationChain(
    min=params[:,1],
    max=params[:,2])
```

By calling the *setup*() function the CAMB and WMAP modules are initialised.

```
chain.setup()
```

Finally, we create a *CosmoHammerSampler* and pass the arguments. A *walkersRatio* of 50 will launch $50 \times 6 = 300$ walkers, where six is the number of parameters sampled. By calling the *startSampling*() function, CosmoHammer will first sample 250 iterations for burn-in and then run for another 250 iterations while writing the results to a file.

```
sampler=CosmoHammerSampler(
  params=params,
  likelihoodComputationChain=chain,
  filePrefix="example",
  walkersRatio=50,
  burninIterations=250,
  sampleIterations=250)

sampler.startSampling()
```

Further examples can be found in the distributed tarball.

Appendix C. Estimation of autocorrelation

As mentioned in Section 5.1, we want to estimate the correlation function $C_{ff}(T)$ from a finite sample $\{\theta_t\}$. For parameter estimation, we are particularly interested in the case when $f(\theta_t) = \theta_t$ is the identity, as we want to estimate the mean of the marginals of our multidimensional posterior distribution (and maybe higher moments). For a single MCMC chain, the estimator for $C_{ff}(T)$ was introduced in Eq. (4) and for f being the identity, it reads

$$\hat{C}(T) = \frac{1}{n-T}\sum_{t=1}^{n-T}(X_t - \bar{X})(X_{t+T} - \bar{X}),$$

where X is one of the seven parameters of the likelihood and the estimator of the identity's autocorrelation function is denoted by $\hat{C}(T)$.

We now consider the case of an ensemble sampler with sample $\{X_t^i\}_{t\in 1,\dots,n}^{i\in 1,\dots,L}$ where i is the walker and t the iteration. Goodman and Weare (2010) propose the following procedure for estimating $C(T)$ of their ensemble sampler: let $F_t = \frac{1}{L}\sum_{i=1}^{L} X_t^i$ denote the ensemble average per iteration. The estimate $\hat{C}_F(T)$ of the correlation function of the process is then given by:

$$\hat{C}_F(T) = \frac{1}{n-T}\sum_{t=1}^{n-T}(F_t - \bar{F})(F_{t+T} - \bar{F}). \tag{C.1}$$

In our analyses we found that large n are needed for a stable estimator $\hat{C}_F(T)$ using this procedure. However, altering the

[6] http://lambda.gsfc.nasa.gov/product/map/current/likelihood_get.cfm.
[7] http://heasarc.gsfc.nasa.gov/docs/software/fitsio/fitsio.html.

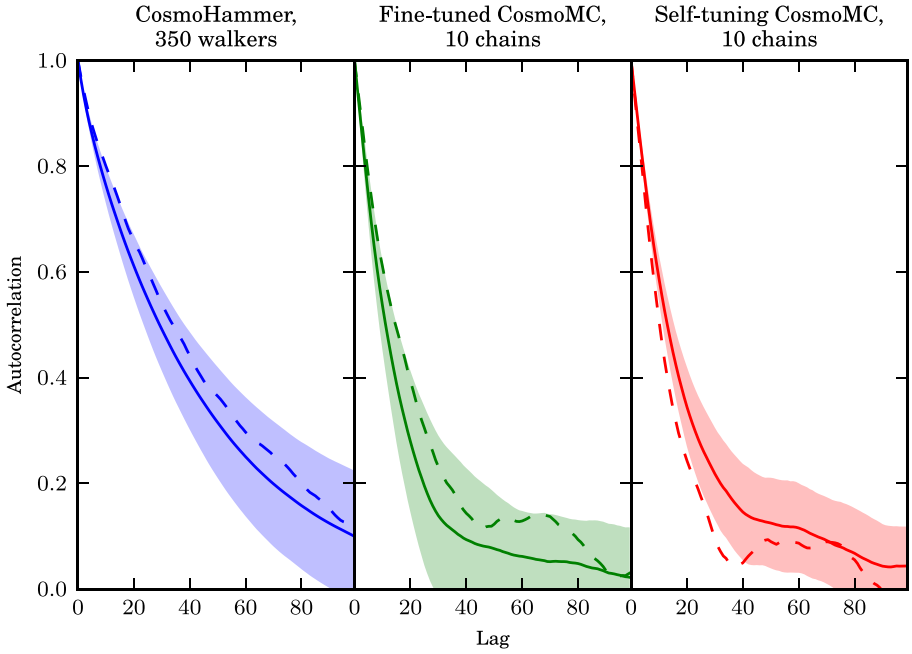

Fig. C.6. Autocorrelations of parameter $\Omega_{\rm DM}h^2$ sampled by `CosmoHammer`, fine-tuned, and self-tuning `CosmoMC` with 3000 sequential steps, estimated by $\tilde{C}(T)$ (solid line) and $\hat{C}_F(T)$ (dashed line). The shaded regions depict the standard deviation of $\hat{C}(T)$ between the individual walkers or chains.

calculations slightly did improve the stability while yielding results consistent with (C.1). Instead of calculating the autocorrelation of the average steps, we calculated $\hat{C}(T)$ per walker and averaged the results afterwards:

$$\tilde{C}(T) = \frac{1}{L}\sum_{i=1}^{L}\frac{1}{n-T}\sum_{t=1}^{n-T}(X_t^i - \bar{X}^i)(X_{t+T}^i - \bar{X}^i).$$

The difference between the two approaches is visualised in Fig. C.6 for parameter $\Omega_{\rm DM}h^2$. It shows that 3000 steps are not sufficient for estimating the autocorrelation time reliably from a single chain, as the deviations between the different chains are still too big. Evaluating the autocorrelation time using the dashed line given by Eq. (C.1) is hence not expected to yield reliable results. Averaging over a large number of walkers suppresses the fluctuations and improves the stability of the estimated autocorrelation time.

Even if the estimate for $C(T)$ is stable, it is non-trivial to deduce $\tau_{\rm exp}$ and $\tau_{\rm int}$ defined in Eqs. (1) and (2). We used the algorithm proposed by Jonathan Goodman[8] and implemented in Python by Dan Foreman-Mackey[9] to estimate the integrated autocorrelation time. On the other hand, we evaluated the exponential autocorrelation time from a simple least-squares fit to $\tilde{C}(T)$ up to the maximal T for which $\tilde{C}(T) < e^{-1}$ holds.

As can be seen from Fig. C.6, the autocorrelation function resembles an exponential decay. We hence expect the estimated values of $\tau_{\rm exp}$ and $\tau_{\rm int}$ to coincide. Indeed, both $\tau_{\rm exp}$ and $\tau_{\rm int}$ converge to the same value, but the exponential autocorrelation time estimate is quite stable already at about 3000 iterations per walker (using 350 walkers), while the integrated autocorrelation has not yet converged. We hence decided to use $\tau_{\rm exp}$ for estimating the relevant time-scales for our sample analysis.

Yet, it turns out that after analysing the autocorrelation time from 3000 iterations per walker and 350 walkers, we found in Section 5.5 that only 250 sequential steps are required for an error in the mean of 3.4% of the standard deviation. Hence, more samples are needed to properly evaluate the error bars than for parameter estimation. Nevertheless, the estimate for $\tau_{\rm exp}$ is typically of the right order even for small numbers of sequential steps. Rounding the result generously should hence suffice to get a rough upper bound on the error bars.

For our comparison of the samplers, we decided to consider the runs with a large number of sequential steps in order to get stable estimates for $\tau_{\rm exp}$. We used 10 independent chains with 3000 iterations each for the `CosmoMC` runs and 3000 iterations of 350 walkers for the `CosmoHammer`. The results are stated in Table 2.

[8] http://www.math.nyu.edu/faculty/goodman/software/acor/.
[9] https://github.com/dfm/acor.

References

Christensen, N., Meyer, R., Knox, L., Luey, B., 2001. Bayesian methods for cosmological parameter estimation from cosmic microwave background measurements. Classical Quantum Gravity 18 (14), 2677.
Dunkley, J., Komatsu, E., Nolta, M.R., Spergel, D.N., Larson, D., Hinshaw, G., Page, L., Bennett, C.L., Gold, B., Jarosik, N., Weiland, J.L., Halpern, M., Hill, R.S., Kogut, A., Limon, M., Meyer, S.S., Tucker, G.S., Wollack, E., Wright, E.L., 2009. Five-year wilkinson microwave anisotropy probe observations: likelihoods and parameters from the WMAP data. Astrophys. J. Suppl. Ser. 180 (2), 306.
Foreman-Mackey, D., Hogg, D.W., Lang, D., Goodman, J., 2012. emcee: the MCMC hammer.
Goodman, J., Weare, J., 2010. Ensemble samplers with affine invariance. Commun. Appl. Math. Comput. Sci. 5 (1), 65.
Hastings, W.K., 1970. Monte Carlo sampling methods using Markov chains and their applications. Biometrika 57 (1), 97.
Knox, L., Christensen, N., Skordis, C., 2001. The age of the universe and the cosmological constant determined from cosmic microwave background anisotropy measurements. Astrophys. J. 563 (2), L95.
Komatsu, E., Smith, K.M., Dunkley, J., Bennett, C.L., Gold, B., Hinshaw, G., Jarosik, N., Larson, D., Nolta, M.R., Page, L., Spergel, D.N., Halpern, M., Hill, R.S., Kogut, A., Limon, M., Meyer, S.S., Odegard, N., Tucker, G.S., Weiland, J.L., Wollack, E., Wright, E.L., 2011. Seven-year wilkinson microwave anisotropy probe (WMAP) observations: cosmological interpretation. Astrophys. J. Suppl. Ser. 192 (2), 18.
Larson, D., Dunkley, J., Hinshaw, G., Komatsu, E., Nolta, M.R., Bennett, C.L., Gold, B., Halpern, M., Hill, R.S., Jarosik, N., Kogut, A., Limon, M., Meyer, S.S., Odegard, N., Page, L., Smith, K.M., Spergel, D.N., Tucker, G.S., Weiland, J.L., Wollack, E., Wright, E.L., 2011. Seven-year wilkinson microwave anisotropy probe (WMAP) observations: power spectra and WMAP-derived parameters. Astrophys. J. Suppl. Ser. 192 (2), 16.
Lewis, A., Bridle, S., 2002. Cosmological parameters from CMB and other data: a Monte Carlo approach. Phys. Rev. D 66 (10), 103511.

Lewis, A., Challinor, A., Lasenby, A., 2000. Efficient computation of cosmic microwave background anisotropies in closed Friedmann–Robertson–Walker models. Astrophys. J. 538 (2), 473.

MacKay, D., 2003. Information Theory, Inference, and Learning Algorithms. Cambridge University Press, New York, NY, USA.

Metropolis, N., Rosenbluth, A.W., Rosenbluth, M.N., Teller, A.H., Teller, E., 1953. Equation of state calculations by fast computing machines. J. Chem. Phys. 21 (6), 1087.

Sokal, A.D., 1989. Monte Carlo Methods in Statistical Mechanics: Foundations and New Algorithms. In: Cours de Troisième Cycle de la Physique en Suisse Romande.

Tegmark, M., Strauss, M.A., Blanton, M.R., Abazajian, K., Dodelson, S., Sandvik, H., Wang, X., Weinberg, D.H., Zehavi, I., Bahcall, N.A., Hoyle, F., Schlegel, D., Scoccimarro, R., Vogeley, M.S., Berlind, A., Budavari, T., Connolly, A., Eisenstein, D.J., Finkbeiner, D., Frieman, J.A., Gunn, J.E., Hui, L., Jain, B., Johnston, D., Kent, S., Lin, H., Nakajima, R., Nichol, R.C., Ostriker, J.P., Pope, A., Scranton, R., Seljak, U.c.v., Sheth, R.K., Stebbins, A., Szalay, A.S., Szapudi, I., Xu, Y., Annis, J., Brinkmann, J., Burles, S., Castander, F.J., Csabai, I., Loveday, J., Doi, M., Fukugita, M., Gillespie, B., Hennessy, G., Hogg, D.W., Ivezić, i.c.v., Knapp, G.R., Lamb, D.Q., Lee, B.C., Lupton, R.H., McKay, T.A., Kunszt, P., Munn, J.A., O'Connell, L., Peoples, J., Pier, J.R., Richmond, M., Rockosi, C., Schneider, D.P., Stoughton, C., Tucker, D.L., Vanden Berk, D.E., Yanny, B., York, D.G., 2004. Cosmological parameters from SDSS and WMAP. Phys. Rev. D 69 (10), 103501.