Practical Bayesian Sampling in Python and Julia

Cian Roche*

Abstract. In this report, Markov chain Monte Carlo (MCMC) algorithms are implemented in both python and Julia and benchmarked in both serial and parallel via a parameter fitting problem. Both the user experience of implementing and using these algorithms **and** the raw computational performance are considered in order to form a recommendation to young scientists interested in MCMC, in particular those with existing experience in Python. The conclusion is that although prototyping and development are faster and more transparent in python, the vast performance increases obtained in Julia even before optimization make it the better choice, in particular for large datasets or high-dimensional parameter spaces. It is found that MCMC algorithms run in Julia on a single core of a consumer laptop outperform almost identical implementations (and the popular package `emcee`) in python run on hundreds of cores on a supercomputing cluster.

Key words. Markov chain Monte Carlo, MCMC, Python, Julia, Bayesian sampling, Affine, Invariant, Metropolis, Hastings, benchmark, parallel, emcee

AMS subject classifications. 60J22

1. Introduction.

The estimation of model parameters via Bayesian sampling techniques such as MCMC or nested sampling is an integral part of modern science, appearing frequently in the fields of astrophysics, biology, linguistics, nuclear physics, cryptography, political science, and many more. As problems become more high-dimensional and intractable, parallel computing grows only more valuable as a resource in these fields. However, most of the researchers making use of these sampling techniques do not have broad backgrounds in computer science, or perhaps have only engaged in formal education in interpreted languages like python and R. To such a population, the barrier to using a compiled language such as C++ or Julia is relatively high, due to less accessible documentation for non-experts, relatively smaller amounts of example code available easily online, and potentially greater complexity in the coding and debugging processes.

This said, the performance gains when using a compiled language are a significant advantage, in particular for the large-scale computations which are good candidates for parallelism. As a result, modern researchers are faced with the decision between a **potentially** smoother development experience, and **potentially** much faster and more efficient code. As the representative languages we choose python (due to its simplicity and popularity) and Julia (due to its speed and relative user-friendliness for a compiled language). The aim of this project is to illuminate, in the context of Bayesian sampling as a means of model fitting, if indeed Julia outpaces python considerably when appropriate use is made of optimization in packages like Numpy, and if the implementation of a highly-parallel Bayesian sampler is in fact more time-consuming in Julia for a non-expert.

*Massachusetts Institute of Technology (roche@mit.edu, https://github.com/CianMRoche/6.338project).

The layout of the report will be as follows: In Sec. 2, we will outline the MCMC algorithms which will be tested, namely the Metropolis-Hastings (MH) and affine-invariant algorithms often used in modern research. We will then define the test problem which will be used for the remainder of the report, and provide details on the parallel computing environment used to test the implementations. In Sec. 3 we will describe the "basic" python implementation of the MH algorithm and the details of the package `emcee` used for the affine-invariant implementation. In Sec. 4 we describe the same implementations in Julia. In Sec. 5 we compare the performance of the algorithms in both languages both in serial and parallel environments. We conclude in Sec. 6 with an overview of the results and a recommendation to young researchers who intend to use MCMC techniques in their work.

2. Background.

2.1. Markov Chain Monte Carlo. Markov chain Monte Carlo (MCMC) methods are well documented, so we provide only a brief overview here. The second half of their name "Monte Carlo" is a reference to the Monte Carlo Casino, Monaco, and refers to the class of computational techniques which fundamentally rely on the generation of pseudo-random numbers. A "Markov chain" is any process consisting of a series of steps, wherein the decision one makes about where to step next depends only on ones current position, and not on the history of steps which took one to that position. Mathematically, a Markov chain of steps $x_i,\ i \in \mathbb{N}$ is described by $x_{n+1} = g(x_n; \{\text{parameters}\})$ for some function g.

The combination of these two ideas, the "Markov chain Monte Carlo" methods, are a class of methods which implement an algorithm that samples from a probability distribution, using *random* walkers (a list of the x_i of the Markov chain) whose steps depend *only on their current position* and some algorithm parameters, and where one is more likely to step into regions of higher probability density of the desired distribution, and correspondingly less likely to step into regions of lower probability density of the desired distribution. Heuristically, after a walker has had sufficient time to "explore" the space of interest, the "amount of time" a walker spends in a region will be proportional to the desired probability distribution. Each step of a walker is not a *pure* sample from the desired distribution (as it depends on the current position of the walker), but the aggregate effect after many steps typically provides a good estimate of the desired distribution.

To fix notation, consider Bayes' theorem, where $P(H|E)$ is the probability of the hypothesis given the evidence ("posterior distribution" or "posterior"), $P(H)$ is the probability that the hypothesis is true without any new evidence (the "prior distribution" or "prior", which represents our prior beliefs), $P(E|H)$ is the probability of seeing the evidence give that the hypothesis is true (the "likelihood"), and $P(E)$ is the probability of observing the evidence ("marginal distribution" or "marginal")

$$P(H|E) = \frac{P(H)P(E|H)}{P(E)} \tag{2.1}$$

In real-world parameter estimation problems, we have some evidence or data (E) and some proposed model (H), and we wish to know what is the probability that the model is correct

given our evidence, that is $P(H|E)$. Unfortunately it is often the case that the marginal $P(E)$ is intractable or unreasonable to calculate, as for continuous parameters (of which there can be many) calculating $P(E)$ involves a large multidimensional integral for which numerical integration is very rarely the best solution. However, $\mathcal{L} := P(H)P(E|H)$ (the "joint probability"[1]) is typically much easier to calculate, as the prior is often some simple function of the model parameters and when the evidence in independent and identically distributed, the likelihood is given by

$$P(E = \{v_i\}_{\ i\in\{1,\dots,N\}}|H = \mathbf{p}) = \prod_{i=1}^{N} f(v_i;\mathbf{p}) \tag{2.2}$$

where $f(v;\mathbf{p})$ is the proposed probability density function (PDF) for the variable v, that is, our model with parameters $\mathbf{p} = \{p_j\}_{\ j\in\{1,\dots,n\}}$ and $\{v_i\}_{\ i\in\{1,\dots,N\}}$ are our evidence (or "data points"). here, and in the remainder of this report, N is the number of data points and n is the dimension of our parameter space in which the walkers "walk", corresponding to the number of parameters in our model which we are trying to fit. MCMC algorithms typically decide where the walkers should step (in a probabilistic sense) based on the ratio of $\mathcal{L}$(current set of parameters) to $\mathcal{L}$(set of parameters of the proposed next step). When $\mathcal{L}_{new} > \mathcal{L}_{current}$, it is likely that we take the step, and vice versa. In this way, we avoid calculating the (difficult) marginal by considering *ratios* of joint probabilities, and stepping in a manner that we usually go toward regions of higher joint probability. The specifics of this stepping depend on the algorithm used, which we will discuss next. Note that for numerical stability, is is standard to work with the logarithm of these quantities, for example we actually compare $\log(\mathcal{L})$ for the current and proposed parameters. This will be evident in the implementations of future sections.

2.1.1. Metropolis-Hastings. The Metropolis-Hastings algorithm [4] [2] is among the simplest examples of an MCMC "stepping scheme". It is best illustrated by simply examining a schematic of a single iteration of the walker algorithm (see algorithm 2.1). Say we start at an initial guess for the parameters of our model $\mathbf{p}_0$ (a vector with n elements), and we have chosen a "stepping distribution" $g(\mathbf{p},\dots)$ where the $\dots$ represents some fixed parameters of the stepping distribution which can be tuned for the problem at hand. Consider as an example for g the n-dimensional normal distribution centred at $\mathbf{p}$ with standard deviations $\sigma_1,\dots,\sigma_n$ (i.e. a diagonal covariance matrix), which should be chosen based on the typical scale of the parameters of your model[2]. The algorithm always steps if the proposed new parameters are more likely, and sometimes steps when the proposed parameters are less likely, in a manner proportional to how "bad" (unlikely) the new parameters are.

2.1.2. Affine-Invariant MCMC. The advantage of affine-invariant MCMC [1] is that the "stepping function" can now stretch how far it steps dynamically. This is a significant advantage, as if we consider a "desired" distribution (the one we are sampling from) which is

[1] The joint probability is in fact **not** a valid probability distribution, as it is not normalized, and so it is sometimes referred to as the "unnormalized posterior".

[2] If the first parameter of your model is typically order 1 then σ_1 should be something like 0.2, but if it is typically order 1000 then convergence will be extremely slow with a small σ_1, and so it should be adjusted and tuned accordingly for each parameter.

Algorithm 2.1 Metropolis-Hastings

```
Chains = empty list [ ]
p = p_0
while i < desired number of walker steps do
  Propose new set of parameters p̃ via g(p, ...)
  Current_prob = L(p)
  Proposed_prob = L(p̃)
  if Proposed_prob > Current_prob then
    p = p̃
    Append p to Chains
  else
    Generate uniform random number acceptance_threshold between 0 and 1
    if Proposed_prob / Current_prob > acceptance_threshold then
      p = p̃
      Append p to Chains
    end if
  end if
end while
return Chains
```

skewed along one of its axes such that in some areas it is very steep and in others more gently sloped, then the stepping function of the MH algorithm will be a suboptimal choice, as it has a fixed stepping distance scale. In the case of a multivariate normal distribution as our stepping function, if the standard deviation in this "skewed" dimension is large, we will resolve poorly the details of the steeply sloped region, but if it is small, we will converge extremely slowly to the accurate shape of the more gently sloped region. The affine-invariant algorithm is largely the same as the MH algorithm, but we start with an ensemble of N_w walkers $\{\mathbf{p}^1, \dots, \mathbf{p}^{N_w}\}$, and the proposed stepping of walker k starting at parameters $\mathbf{p}$ is determined by

$$\tilde{\mathbf{p}}^k = \mathbf{p}^j + Z(\mathbf{p} - \mathbf{p}^j) \tag{2.3}$$

where $\mathbf{p}^j$ is one of the other walkers and Z is a distribution which we sample from (analogously to the multivariate normal distribution mentioned before) whose density g satisfies

$$g\left(\frac{1}{z}\right) = zg(z) \tag{2.4}$$

as in this case the stepping algorithm is symmetric, that is the probability to step from $\mathbf{p}$ to $\tilde{\mathbf{p}}$ is equal to the probability to step in the other direction. An example of such a g is

$$g(z) \propto \begin{cases} \frac{1}{\sqrt{z}} & \text{for } z \in \left[\frac{1}{a}, a\right] \\ 0 & \text{otherwise} \end{cases} \tag{2.5}$$

where $a > 1$ is a scale factor, a single number which replaces the many metaparameters of Metropolis-Hastings that can be tuned based on the problem. In this case a random sample

from Z can be obtained as a function of a using pseudorandom numbers via inverse transform sampling as

$$(\text{Sample from } Z)\,(a) = \frac{1}{a}\left((a-1)(U_{[0,1]})+1\right)^2 \tag{2.6}$$

where $U_{[0,1]}$ is a pseudorandom number obtained by sampling the uniform distribution on the interval $[0,1]$. This stepping procedure mans that the space of possible proposals starting at $\mathbf{p}^k$ is the line connecting $\mathbf{p}^k$ and $\mathbf{p}^j$.

2.2. Test Problem Definition. As our test problem, we will use a model for the high-velocity distribution of stellar speeds in a galaxy [3], and fit the parameters of this model to generated mock data. The is an approporiate model for the testing of these algorithms as the covariance distribution of its parameters is in fact skewed (see Fig. 8) and thus should prduce visibly different results for the MH and affine-invariant implementations. In the "real world", this analysis technique is applied to observations and the outputs of galactic-scale simulations in order to estimate the escape velocity profile[3] (and therefore mass profile) of galaxies. The PDF of the model is given by

$$f(v|v_{esc},k) = \frac{k+1}{{v_{esc}}^{k+1}}(v_{esc}-v)^k \quad , \quad v\in[0,v_{esc}] \tag{2.7}$$

and is 0 otherwise. That is, the parameters of this model are an escape velocity v_{esc} and the slope of a power law in the speed v which we denote k. By fitting measurements of stellar speeds in a galaxy with this model, we can obtain an estimate for the escape velocity at the location of those stars in the galaxy of interest, and also the slope of the distribution in log space k. The logarithm of the likelihood is then

$$\log\left(P(E=\{v_i\}_{i\in\{1,\ldots,N\}}|v_{esc},k)\right) = \sum_{i=1}^{N}\log(f(v_i|v_{esc},k) \tag{2.8}$$

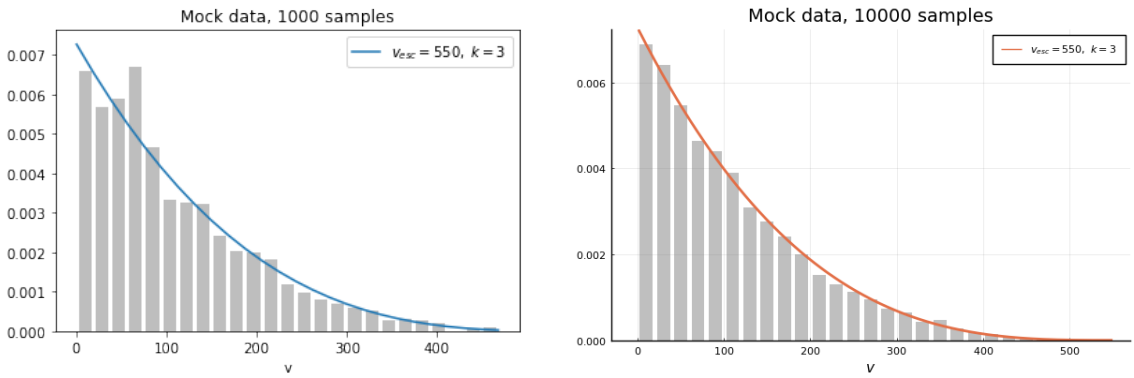
where each term in the sum is of the form

$$\log(f(v_i|v_{esc},k) = \log(k+1) - (k+1)\log(v_{esc}) + k\ \log(v_{esc}-v) \tag{2.9}$$

For our prior beliefs, we choose the least restrictive choice which is reasonable, that is a "hard cutoff" for unreasonable values, and the uniform distribution between those values. We consider the hard cutoffs for v_{esc} as $v_{esc}\in[100,800]$ km s^{-1} and for the slope we choose $k\in[0.2,10]$. How this is implemented in code (and in log-space) is clear from the accompanying notebooks, which all share the same joint probabilities. In the literature, typically stricter priors are places on these parameters, but we choose to be as nonrestrictive as possible.

Now we just require some mock data with which we can estimate our parameters. Real-world and simulation data inherit kinematic substructure which would obscure the results of

[3]The analysis we will describe here would yield only a single value for the escape velocity, but by repeating the analysis for collections of data at varying galactocentric radius, one obtains an approximation for $v_{esc}(r_{gal})$ where r_{gal} is the distance from the center of the galaxy.



(a) 1000 samples, Python implementation

(b) 10000 samples, Julia implementation

Figure 1: Sampling from the target distribution with different number of "data points". Evidently the parameter fitting will be more accurate but more computationally expensive as the number of data points increases. Establishes also the color scheme for the report: that Python outputs will feature blue, and Julia outputs will feature orange/red.

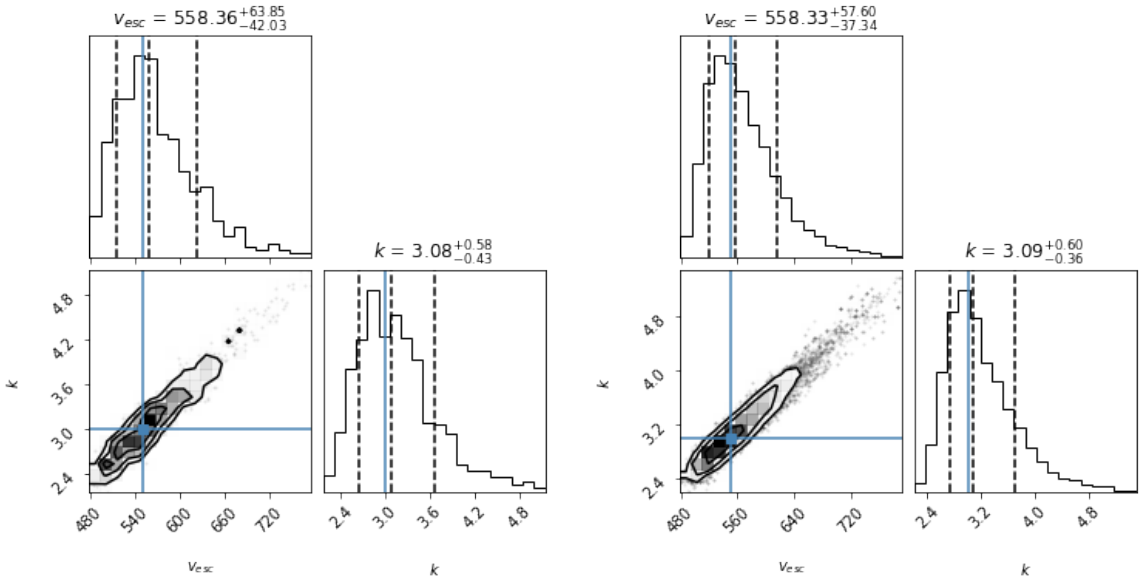
testing these algorithms, so in the interest of simplicity we simply generate data from a test distribution Eq. 2.7 with some particular values of v_{esc} and k. For demonstration purposes, we choose $v_{esc} = 550, k = 3$. We then sample from this distribution via inverse transform sampling to obtain the data in Figure 1. This speed data is what we will use as our evidence E when fitting the model parameters v_{esc} and k.

2.3. Computing Environments. For the majority of testing, we simply use a consumer-grade laptop with an AMD Ryzen 9 5900HS processor (base clock 3.3 GHz). The multi-threaded tests will be performed on 4 threads (by starting Julia with `julia -t 4`). For the python implementation of affine invariant MCMC, we will use 16 nodes of MIT engaging cluster (64 CPUs per node)

3. Python Implementation. For all cases below, we use python 3.9.7 with standard modules such as `numpy`, `matplotlib` and the `corner` module[4] for plotting the results of MCMC runs, as it also depicts the covariances of the achieved parameters. We write a simple implementation of the MH algorithm without particular attention paid to optimization past the usage of modules like `numpy`, as an average researcher in a primarily non-computational field is likely to either use a package such as `emcee` or write their own implementation in a reasonable but sub-optimal way. These tests attempt to reproduce a "typical" use case. The MH implementation can be found in the notebooks accompanying this report under the name "`MH_python.ipynb`".

For the affine-invariant algorithm implementation, we use a standard package used often

[4] https://corner.readthedocs.io/en/latest/



(a) Metropolis-Hastings (b) Affine-invariant

Figure 2: Python: A comparison of the final posterior distributions obtained and their errors when fitting 1000 data points via both MCMC algorithms with a total of 10,000 walker steps each. The affine-invariant method is capable of resolving much more finely the longer tail of the skewed covariance distribution. Blue lines represent true values and dotted lines represent the median and 1σ quantiles (16%,50%,84%).

in research: `emcee`. This package benefits from a relative ease of use due to its user-friendly API and documentation, and even though its is reasonably well-optimized and parallel-ready, it typically performs better by a factor of order unity relative to the "unintelligent" MH implementation mentioned above. The algorithm used is only slightly more computationally expensive, but converges more accurately for problems with skewed distributions and requires the tuning of far fewer meta-parameters as mentioned in Sec. 2.

3.1. Test Problem Results. As a representative benchmark, we choose to generate $N = 1000$ mock data points (speeds v_i) and perform the MH analysis with 10,000 steps of one walker, and the affine-invariant analysis with 2,000 steps of 5 walkers. The resulting corner plots can be seen in Fig. 2. The MH algorithm executed on the order of 125 iterations per second, whereas the `emcee` version executed roughly 250. A more thorough analysis of the performance scaling with N (size of data set) can be seen in Sec. 5.

The bottlenecks of the python implementation are those standard for interpreted languages. Even though numpy arrays are used when possible, MCMC at its core involves very large numbers of iterations through a loop in a way which is not easily vectorized, operations which are notoriously difficult for interpreted languages. The most reasonable route for

optimization then comes from precompiled joint probability functions etc. using a tool like `cython`, however not only does this not entirely fix the problem of the large loops, but at the point of mixing interpreted and precompiled code, it may be more reasonable to attempt a more seamless approach in a language like Julia.

4. Julia Implementation.

For this analysis we use Julia 1.7.1, and as in the case of python, we use a series of standard modules for the implementation such as Plots, Distributions, Statistics, and Random for the core functionality. We again implement the Metropolis-Hastings algorithm manually, and in a manner as identical as possible to that of our python implementation. For the affine-invariant implementation, we utilize an existing but outdated package MCJulia[5] though we make some changes and add to the multi-threading functionality). These functions which implement the affine-invariant stepping algorithm are collected in "`AffineInvariantMCMC.jl`".

4.1. Test Problem Results.

As in the previous section, we execute the algorithm for 10,000 steps total using an initial data set of 1000 samples from the mock distribution. The results for a single run of this analysis can be seen in Fig. 3. We construct our own corner plots, though it is worth noting that there does exist packages analogous to Python's `corner` in Julia. Note that the accuracy of the median of a single run relative to the "true" value (with only 10,000 samples) should not be treated as an absolute measure of accuracy, as with any Monte Carlo method each run carries with it an inherent randomness which is only "washed out" in the infinite step limit. Not only this, but the data generation process is also inherently random. These effects will become less important as we benchmark more effectively in Sec. 5.

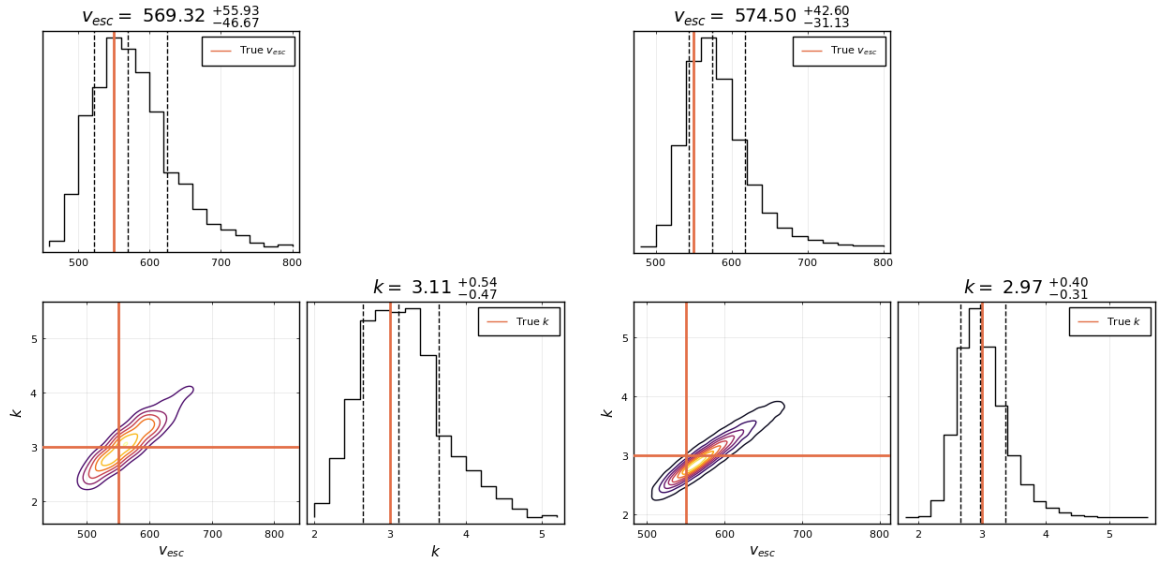
The basic (and relatively unoptimized) MH implementation in Julia performed at 33,000 iterations per second (250 times faster than the simple python MH implementation), and the affine-invariant implementation operated at an average of 50,000 iterations per second (1,000 times faster than `emcee`). The performance gain over python is expanded upon in the next section (Sec. 5). For a very large data set of "perfect" data ($N = 100,000$) we present the MCMC results in the appendix, Fig. 8. Evidently the estimate is much more accurate (see 1σ errors) but in the real world, such a perfect data set is hard to come by to say the least.

5. Comparison.

5.1. Performance of Python vs Julia.

As was evident in the previous section, almost identical Julia implementations of the same algorithms have the potential to outperform python + numpy by a factor of two to 3 orders of magnitude, so we now examine how this difference scales for both algorithms with the size of the data set which is being fit. Though we can easily obtain estimates for the per-second performance of all implementations, data for final results will extend significantly further for the Julia implementations than for those in Python, as even with 1000 data points the python versions take on the order of minutes per run. These results are presented in the appendix.

[5] https://git.physics.byu.edu/Modeling/MCJulia.jl



(a) Metropolis-Hastings (b) Affine-invariant

Figure 3: Julia: A comparison of the final posterior distributions obtained and their errors when fitting 1000 data points via both MCMC algorithms with a total of 10,000 walker steps each. The affine-invariant samples were taken using 5 walkers. Again the affine-invariant algorithm preserves better the skew structure. Orange lines represent true values and dotted lines represent the median and 1σ quantiles (16%,50%,84%).

The number of algorithm steps executed per second for all implementations in the *serial* regime is shown in Fig. 4, in which it is clear that the number of iterations per second scales simply with N as approximately $1/N$. In all test cases (even very small data sets) the Julia implementations outperformed those in Python by a factor of several hundred. To put this in perspective, this is the difference between an analysis running in one minute versus on the order of three hours. This deficit can be "accounted for" by running the Python code on extensive computing resources in parallel, but for reasons of cost, time and energy usage, this seems inadvisable.

We investigate the parallel performance of the Python affine-invariant implementation, as shown in fig. 5, where it is evident that depending on the number of walkers chosen, there is a limit of CPU cores after which adding further parallel resources will yield no benefit. We also examine the multi-threaded performance of Julia, wherein for this application it is unsurprising that the iterations per second (after some almost negligible initial overhead) scale approximately linearly with the number of threads available to Julia. The multi-threaded performance as a function of data size N is shown in Fig. 6. Exact numbers shown are only approximate, and should be interpreted primarily as demonstrating a general trend.

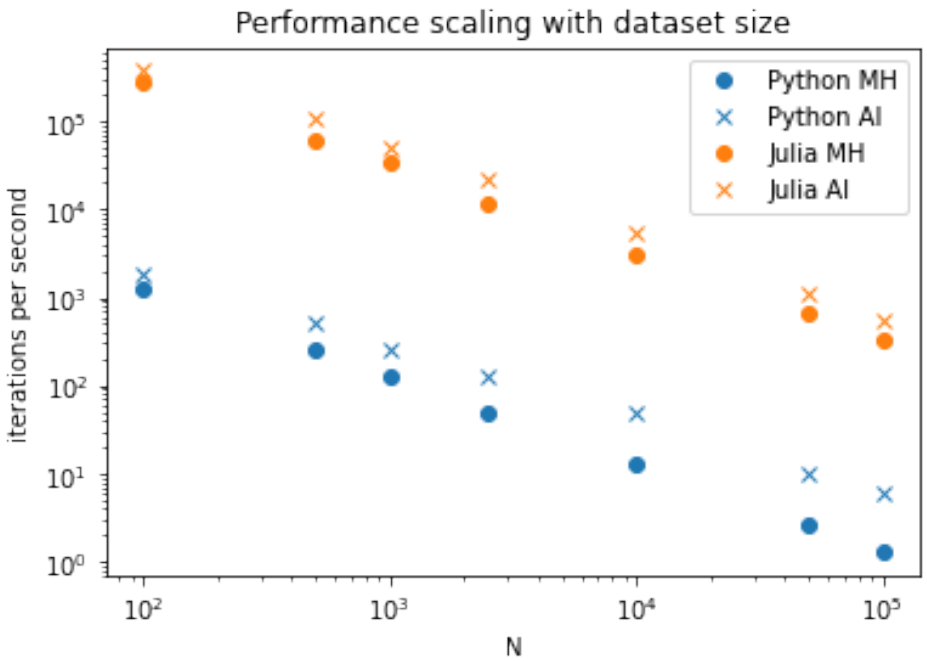


Figure 4: Performance of Metropolis-Hastings (MH) and affine-invariant (AI) MCMC implementations in both Python and Julia, and their scaling with size of dataset used in fitting.

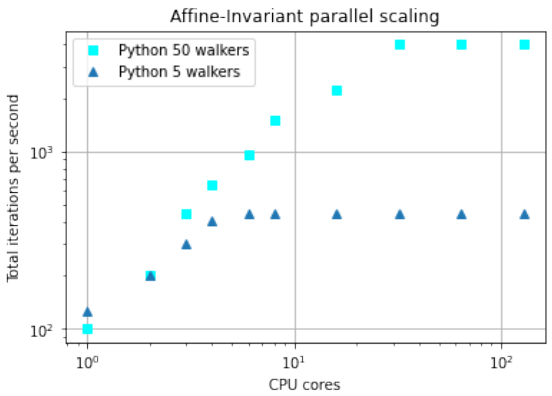


Figure 5: Scaling of affine-invariant MCMC implementation in Python with the number of CPU cores used to run the algorithm. Tests performed with 1000 data points, 2,000 iterations per walker and variable number of walkers.

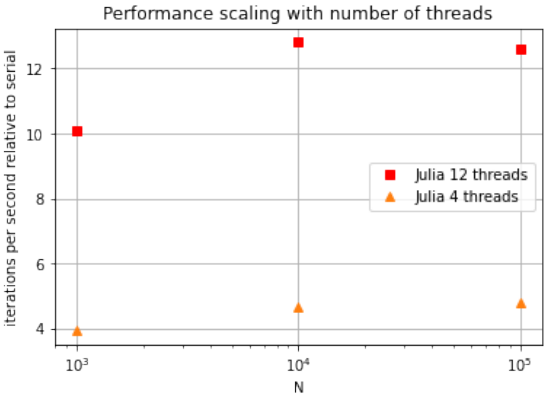


Figure 6: Julia: Performance scaling of affine-invariant MCMC implementation with number of threads available to Julia, shown as iterations per second relative to the serial implementation. Tests performed with 5 walkers.

5.2. Ease of Implementation. The process of development and debugging in Julia is undoubtedly more cumbersome than in Python (but simpler than in other compiled languages), primarily due to 1) longer initial compile times and load times which hinder the prototyping stage, and 2) less readily available example code online, with often harder to interpret documentation for the non-expert. One can spend an hour searching for how to set the lower limit for the contours on a contour plot, and find no elegant solution in the end (to name but one example). In contrast, Python development typically involves regular and **fast** internet searches which yield exactly the solution the user was looking for, partly owing to better search engine optimization (SEO) and partly due to a long history of activity on forums such as stackoverflow.

This said, the readily available source code for Julia means that, in theory, one could accomplish any task they wished with a given set of functions or module, but this may involve a much greater time investment than the "run into problem, google it, get back to work" ethos of doing relatively non-complex tasks in python. For some, this means that a workflow in which data is interfaced and plotted etc. through Python, but computations are done in Julia is desirable, however since Julia can in principle do all the same things python can (and usually faster), it may be optimal to invest time in becoming familiar with the Julia analysis and plotting ecosystems.

6. Conclusions and Recommendations. Evidently, Julia vastly outpaces Python in terms of raw computational performance, to a degree of hundreds or even thousands of times for identical algorithms and almost identical implementations. However, the development process can be slow relative to Python depending on the knowledge and experience of the user. In fact for this report, the Metropolis-Hastings algorithm was first implemented in Python, confirmed to function as expected, and then copied and edited to the Julia format. This exercise was surprisingly easy, as the general syntax is very similar and changes were only needed in specialty functions from modules, and superficial additions like `end` statements. For researchers with significant Python experience, this is not an unreasonable way to become familiar with Julia.

For modern applications, the dimension of the problem n is typically moderately or even significantly larger than 2, and so the test case presented here represents in some ways a "best case scenario". Not only this, but for the tests considered here only 10,000 steps of the walkers were used, but in applications it can be that the autocorrelation scales are high enough that many more steps are required for convergence (depending on your chosen meta-parameters). Evidently, if the Python code is taking on the order of minutes for this simple problem, actual applications can (and for the majority in fact **do**) take on the order of hours per run. If one then wants to perform the analysis for all points in an m-dimensional[6] discrete parameter space (that is, m^2 runs of the analysis) it is easy to see how one must sacrifice either financially (by buying lots of computers), temporally (by waiting a very long time), or in the

[6]These parameters are not to be confused with the parameters of the model which we wish to fit. Rather they could be, for example in our test problem, collections of stars at different galactocentric radius, or datasets with different cuts applied to them.

number of steps they can run, reducing the accuracy of their final result as the analysis is not properly converged. If one chooses one of the first two choices, they are faced with the vast amount of energy required to perform their analysis, as computing clusters are notoriously power-hungry[7]. Considering one's energy footprint and the footprint of their work has become increasingly important, and so this is now more than ever a significant advantage of more efficient code. Simply running Julia code on a handful of threads on a consumer laptop can match or even outperform Python code run on hundreds of CPUs on an expensive and energy-hungry computing cluster.

As a result, the recommendation of this report is the following: Python is valuable to learn if one does not expect to do much development of code whatsoever, and expects only to do very small-scale calculations for which convenience and existence of example code are the primary concerns. For those people who do intend to do more complex calculations or larger-scale analyses, and are already familiar with Python, we recommend prototyping code in Python and then translating to Julia, and making easily available any code which may be valuable to others online. Then, upon becoming familiar with Julia, one can begin to phase out the Python stage of development. For those people who do intend to do serious development or large-scale analyses, but are not particularly familiar with either language, it is most likely optimal to become well-versed in Julia, and have a reasonable understanding of Python for times when one needs to take inspiration from existing Python packages (as was done in this report for the `corner` Python module). Again, these individuals should make example codes available easily online with appropriate keywords in order to accelerate the growth of the Julia community (good for everybody).

REFERENCES

[1] J. Goodman and J. Weare, *Ensemble samplers with affine invariance*, Communications in Applied Mathematics and Computational Science, 5 (2010), pp. 65 – 80, https://doi.org/10.2140/camcos.2010.5.65, https://doi.org/10.2140/camcos.2010.5.65.

[2] W. K. Hastings, *Monte Carlo sampling methods using Markov chains and their applications*, Biometrika, 57 (1970), pp. 97–109, https://doi.org/10.1093/biomet/57.1.97, https://doi.org/10.1093/biomet/57.1.97, https://arxiv.org/abs/https://academic.oup.com/biomet/article-pdf/57/1/97/23940249/57-1-97.pdf.

[3] P. J. T. Leonard and S. Tremaine, *The Local Galactic Escape Speed*, , 353 (1990), p. 486, https://doi.org/10.1086/168638.

[4] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller, *Equation of state calculations by fast computing machines*, The Journal of Chemical Physics, 21 (1953), pp. 1087–1092, https://doi.org/10.1063/1.1699114, https://doi.org/10.1063/1.1699114, https://arxiv.org/abs/https://doi.org/10.1063/1.1699114.

Appendix.

[7] Though the cluster on which the `emcee` parallel analysis was run is powered primarily by a hydroelectric station.

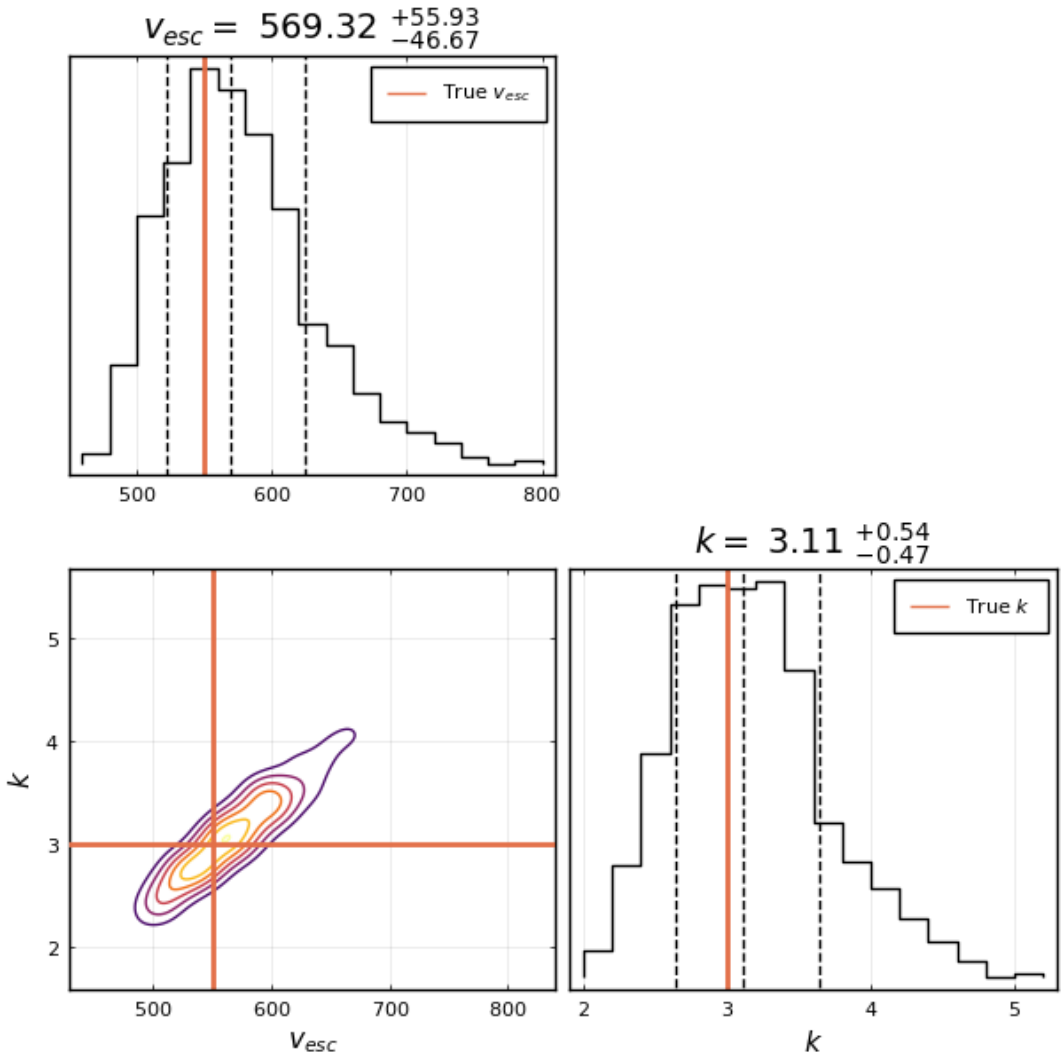


Figure 7: Julia: 10,000 steps of the Metropolis-Hastings algorithm with $\mathbf{N = 100,000}$ data points for parameter fitting. This computation took 30 seconds, but in python would require vast computing resources to complete on a reasonable timescale. Note that the errors are no more converged that for $N = 1000$, which represents the limitations of this algorithm for skewed distributions.

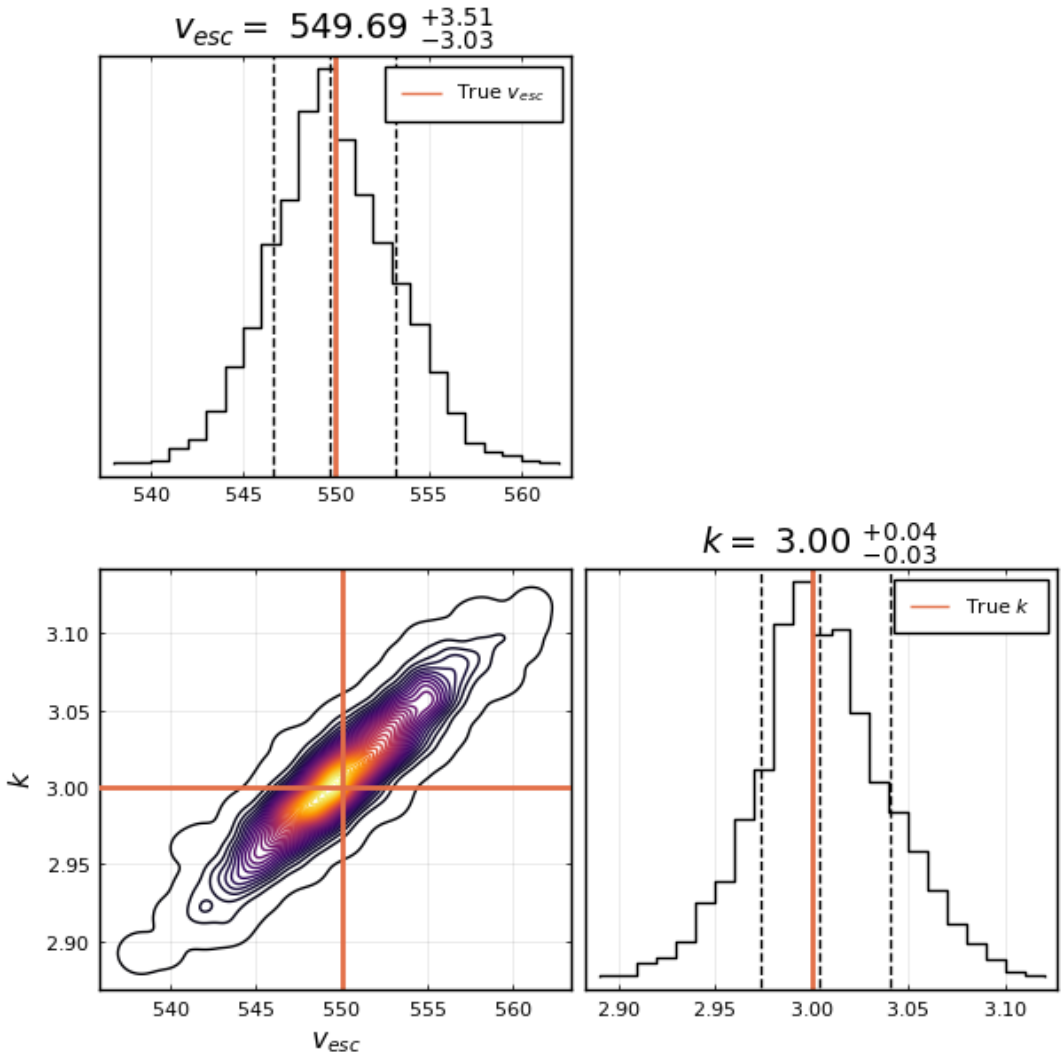


Figure 8: Julia: 10,000 steps (distributed across 5 walkers) of the affine-invariant algorithm with $\mathbf{N = 100,000}$ data points for parameter fitting. This computation took 18 seconds, but in python would require vast computing resources to complete on a reasonable timescale. More contour levels are required to display the sharply peaked distribution relative to the Metropolis-Hastings case.