Published in final edited form as:

Comput Chem Eng. 2018 February 2; 110: 1–12. doi:10.1016/j.compchemeng.2017.11.011.

APT-MCMC, a C++/Python implementation of Markov Chain Monte Carlo for parameter identification

Li Ang Zhang^a, Alisa Urbano^f, Gilles Clermont^{a,b,c}, David Swigon^d, Ipsita Banerjee^{a,c,e}, Robert S. Parker^{a,b,c,e,*}

^aDepartment of Chemical and Petroleum Engineering, Swanson School of Engineering, University of Pittsburgh, PA, USA

^bClinical Research, Investigation, and Systems Modeling of Acute Illness Laboratory (CRISMA), Department of Critical Care Medicine, University of Pittsburgh, PA, USA

^cDepartment of Bioengineering, Swanson School of Engineering, University of Pittsburgh, PA, USA

^dDepartment of Mathematics, University of Pittsburgh, PA, USA

^eMcGowan Institute for Regenerative Medicine, University of Pittsburgh and UPMC, PA, USA

^fDepartment of Biostatistics, School of Public Health, The University of North Carolina at Chapel Hill, NC, USA

Abstract

The inverse problem associated with fitting parameters of an ordinary differential equation (ODE) system to data is nonlinear and multimodal, which is of great challenge to gradient-based optimizers. Markov Chain Monte Carlo (MCMC) techniques provide an alternative approach to solving these problems and can escape local minima by design. APT-MCMC was created to allow users to setup ODE simulations in Python and run as compiled C++ code. It combines affine-invariant ensemble of samplers and parallel tempering MCMC techniques to improve the simulation efficiency. Simulations use Bayesian inference to provide probability distributions of parameters, which enable analysis of multiple minima and parameter correlation.

Benchmark tests result in a 20×–60× speedup but 14% increase in memory usage against *emcee*, a similar MCMC package in Python. Several MCMC hyperparameters were analyzed: number of temperatures, ensemble size, step size, and swap attempt frequency. Heuristic tuning guidelines are provided for setting these hyperparameters.

Keywords

MCMC; Simulation; Bayesian inference

*Corresponding author at: Department of Chemical and Petroleum Engineering, Swanson School of Engineering, 940 Benedum Hall, University of Pittsburgh, PA, 15261, USA. rparker@pitt.edu (R.S. Parker).

Appendix A. Supplementary data

Supplementary data associated with this article can be found, in the online version, at <https://doi.org/10.1016/j.compchemeng.2017.11.011>.

1. Introduction

The inverse problems that mathematical modelers tackle are often very challenging due to parameter dimensionality, parameter correlation, and multiple optima. A topical and popular set of inverse problems is fitting parameters to a system of ordinary differential equations (ODEs). Most deterministic optimization algorithms can be broken down into two types: line search and trust region (Nocedal and Wright, 1999). During nonlinear fitting, these algorithms often get trapped within local minima because they are highly dependent on the provided starting point. Additionally, many advanced solvers calculate or estimate the Jacobian or Hessian of the objective function, a process which may not converge for complex ODE parameter-fitting problems. This results in heuristic implementations that employ multi-start optimization to characterize the optimal parameter space, and parameter estimates that are likely only locally optimal and for which confidence intervals cannot necessarily be easily constructed.

Markov Chain Monte Carlo (MCMC) is a stochastic sampling technique typically used to gain information about a probability distribution that lacks a closed form. It has been described as a “bad method” for parameter estimation to be used when all alternatives are worse (Sokal, 1997). MCMC has been used in a variety of fields, such as cryptography, statistical mechanics, and astrophysics (Diaconis, 2009; Foreman-Mackey et al., 2013). Long runtimes, simulation stochasticity, and the lack of a robust convergence and stationarity criteria contribute to the stigma against MCMC. However, MCMC is immediately relevant to ill-posed ODE inverse problems. Unlike deterministic algorithms, MCMC is not dependent on the starting point and does not require the computation of Jacobians or Hessians. Furthermore, an MCMC simulation provides parameter probability distributions, which can help elucidate parameter correlations and identify multiple optima (see Fig. 1).

MCMC is formulated as a stationary Markov Chain with a transition probability derived from the parameter probability distribution. Bayesian formalism is used to reformulate a parameter fitting problem as a search for probability distribution. That distribution can be sampled using MCMC. Eq. (1) describes Bayes theorem, where θ is a parameter and D is the data.

$$P(\theta|D) = \frac{P(D|\theta)P(\theta)}{D(D)} \quad (1)$$

The $P(\theta|D)$ term is the Bayesian posterior probability, which represents the probability of the parameter set, θ , given the data. The first term in the numerator describes the likelihood of obtaining the data, D , given the parameters, θ . The objective function for a parameter fitting problem, reformulated as a likelihood, is used here. The second term in the numerator describes the prior distribution of the parameters. This can vary depending on the problem, but the simplest case is a bounded uniform distribution to provide upper and lower bounds on θ . The denominator is a normalization constant that represents the probability of the data,

typically an unknown, but constant, term. This term is not necessary for MCMC simulation, as shown in Eq. (1).

To recover the Bayesian posterior distribution from a parameter fitting problem, MCMC needs to generate many samples from the parameter space in accordance to Eq. (1). The MCMC sampler stochastically proposes the next parameter set, $\theta_{proposed}$, based on the location of the current parameter set, $\theta_{current}$ (Doucet et al., 2000; Andrieu et al., 2003). The Metropolis-Hastings criterion is typically used to accept or reject $\theta_{proposed}$ by calculating the posterior probability, $P(\theta_{proposed}|\mathbf{D})$. If this value is higher than the posterior probability of the current parameter set, $P(\theta_{current}|\mathbf{D})$, then $\theta_{proposed}$ will be accepted with 100% probability. If the probability is lower, then $\theta_{proposed}$ may be accepted with the nonzero probability calculated by Eq. (4) (with $T = 1$). This step is repeated many times to allow the samplers to fully explore the parameter space.

Over the course of an MCMC simulation, all the values of θ (derived from each MCMC sample via the calculation of Eq. (1)) are preserved for post-hoc analysis. The posterior distribution can be recovered via histogram analysis or kernel density estimations from the chain of θ values. θ may be broken down into its individual parameters to obtain a marginal posterior probability distribution for each fitted parameter. These marginal distributions can yield generate parameter confidence intervals, identify inter-parameter correlations via joint probability distributions, and to identify the locations of multiple minima in the parameter space based on multi-modal marginal parameter distributions. Furthermore, irreducibility and aperiodicity properties of Markov Chains guarantees convergence of the posterior distribution (Sokal, 1997). The major benefit of these properties is that an MCMC simulation will yield the correct Bayesian posterior based on the data and model structure in finite time.

Despite the guarantee of convergence, it is impossible to test if a MCMC sampler has converged towards its equilibrium distribution or a temporary meta-stable distribution (Sokal, 1997). As a result, MCMC simulations typically sample a very large number of points. The initial samples that are taken prior to reaching the equilibrium distribution may bias the posterior distribution. The two ways possible ways to deal with this issue are: (i) discard some initial percentage of simulation samples or (ii) run a long simulation to minimize the bias. In either scenario, simulations take a long time because a large number of samples are needed for convergence.

Many variations have been proposed over the years in order to improve the efficiency of the sampling. One issue is that MCMC samplers may get trapped in local minima for a long time before escaping. An escape consists of several consecutive low probability parameter moves because it requires movement towards increasing deviation from the data. At any point, the MCMC sampler may change direction and move back into the local minima. Parallel tempering addresses this by employing multiple MCMC interacting simulations in parallel (Hogg, 2013; Earl and Deem, 2005). Each simulation is assigned a temperature and the Boltzmann distribution, Eq. (4), is used to accept a parameter proposal. The Bayesian posterior probability for the proposed and current parameter vectors are calculated as normal

(Eq. (1)), but are treated as energies (Eqs. (2) and (3)) in order to fit the formalism of the Boltzmann distribution.

$$E_1 = \log P(\theta_{\text{proposed}} | D) \quad (2)$$

$$E_0 = \log P(\theta_{\text{current}} | D) \quad (3)$$

$$\text{Parameter acceptance} = \min\left(1, \exp\left(\frac{E_1 - E_0}{T}\right)\right) \quad (4)$$

$$\text{Swap acceptance} = \min\left[1, \exp\left(-(E_i - E_j)\left(\frac{1}{T_i} - \frac{1}{T_j}\right)\right)\right] \quad (5)$$

These energies, E_1 and E_0 , are scaled by the simulation temperature before determining the proposal acceptance probability. In effect, the objective function is divided by temperatures so that samplers have higher probabilities of escaping local minima. Fig. 2 demonstrates how the energy “landscape” of a probability distribution can change given a high enough temperature. The higher temperature landscape at $T=3$ or $T=10$ demonstrates the relative ease at which an MCMC sampler can explore the entire parameter space compared to the lower temperature landscape where the sampler may be prone to getting trapped in local minima.

A unique feature of parallel tempering algorithms is that the tempered Monte Carlo chains are allowed to evolve independent of each other for some time. After a user-set hyperparameter, n_{steps} (a “swap interval”) has passed, an inter-chain swap of parameters is attempted to pass the information about potential energy minima (higher probability parameter locations) from higher temperature samplers to the lower temperature samplers. This swap probability is calculated with Eq. (5) by comparing the energy and temperature of chain i against those of chain j (Altekar et al., 2004). If a higher temperature sampler is in a parameter location with a larger energy value (higher Bayesian posterior and more favorable location), then that information will pass down to a lower temperature sampler with 100% probability. The opposite scenario, where a swap is proposed to provide a low temperature sampler with a lower energy parameter location (less favorable), is possible, but less probable (small, nonzero swap acceptance). The net effect is that high temperatures perform a broad search over the parameter landscape and pass locations of local minima to the lower temperatures, which conducts a depth search within these minima.

An issue that often arises from inverse problems that estimate ODE parameters is parameter correlation. Correlations cause parameter distributions to be highly anisotropic. Fig. 1A illustrates a highly anisotropic system that would be very difficult for MCMC to efficiently sample. The significant correlation between x_1 and x_2 can be clearly seen in the joint probability distribution in Fig. 1D. The majority of MCMC samples will lie outside regions of interest and provide no information that can be used to recover the probability distribution. As a result, to uncover the marginal distributions of x_1 and x_2 , as illustrated in Figs. 1B and C, a lengthy simulation would be required. To improve the sampling efficiency of anisotropic distributions, Goodman and Weare proposed an MCMC variant that utilizes an ensemble of samplers with affine invariance (Goodman and Weare, 2010). The affine invariance property of this algorithm means that sampling efficiency is unaffected by the anisotropic nature of a parameter topology. This means that the ensemble of samplers can sample from the distribution in Fig. 1A just as efficiently as it can sample from an isotropic distribution (e.g. a perfectly circular density plot).

ODE parameter fitting can greatly benefit from MCMC techniques, but several barriers of entries prevent it from being commonly adopted. Most existing packages suffer from shortcomings related to one of two categories. First, many packages exist in “prototyping language” such as Python, R, or MATLAB in order to facilitate the coding process. Prototyping an MCMC simulation is more convenient, but the user pays for it greatly with the lengthy simulation time. The second category of packages are those written in “fast languages” such as C(++) or FORTRAN. MCMC simulations compiled from these languages are considerably faster, but suffer from the opposite problem: difficulty of programming. A large barrier to entry needs to be overcome before research code can pass through the layers of abstraction into a “fast language.” APT-MCMC is a C++ implementation of MCMC with the aforementioned parallel tempering and affine-invariant ensemble of samplers (Hogg, 2013; Goodman and Weare, 2010). It aims to bridge the gap of programming accessibility by allowing the user to define their simulation solely within a Python package, which provides code conversion to compilable C++ code.

The rest of the article is structured as follows. First, a description of the APT-MCMC software is provided, including a brief user guide. Second, results of a standard series of optimization benchmark are presented. APT-MCMC is shown to converge to the optimal parameter set and are shown to be comparable to a popular Python MCMC package, *emcee*. Finally, generalizable MCMC hyperparameters are explored and tested against these optimization benchmarks; hyperparameter tuning heuristics are provided.

2. Methods

2.1. APT-MCMC package

APT-MCMC (Affine-invariant ensemble of samplers with Parallel Tempering Markov Chain Monte Carlo) began as a C++ implementation of MCMC merging PTempEst, a parallel tempering MCMC algorithm with Boltzmann distribution acceptance in MATLAB, and *emcee* (<http://dan.iel.fm/emcee/>), a (parallel tempering optional) affine invariant ensemble of samplers MCMC algorithm in Python (Foreman-Mackey et al., 2013; Hogg, 2013). APT-MCMC initializes with n_{shains} concurrent ensembles at temperatures arranged in a geometric

progression. The geometric step was taken from the source code of *emcee* (Foreman-Mackey et al., 2013). Within each ensemble, $n_{ensemble}$ samplers simultaneously generate the next proposed parameter step according to the “stretch-move” technique from Goodman and Weare (2010). In short, the stretch-move utilizes the location of another parameter set within the same ensemble to determine a movement direction. The proposed parameter is the result of a random movement along this direction and is determined by a step size hyperparameter. This value, z , allows movement between $\left[\frac{1}{2}, z\right]$ times the distance between the two parameters. Proposed parameter acceptance is governed by Eq. (4). Samplers within each temperature ensemble are allowed to move independently of other temperature ensembles for n_{steps} , after which, a parameter location swap is attempted between another sampler in a higher temperature ensemble. The probability of this swap is governed by Eq. (5). Table 1 summarizes the nomenclature used in this work.

Despite the impossibility of determining convergence in an MCMC simulation, APT-MCMC computes two statistical tests to provide evidence for convergence: integrated autocorrelation and the potential scale reduction factor (PSRF) of the simulation (Sokal, 1997; Nokedal and Wright, 1999). The PSRF is a value that compares the parameter distributions between and within ensembles. Values close to 1.0 are desired because this statistic provides evidence for convergence by indicating that the inter- and intra-ensemble distributions are identical (Nokedal and Wright, 1999). APT-MCMC calculates the PSRF value using the Welford’s variance algorithm. Integrated autocorrelation time, τ , measures the correlation of a signal at every possible time lag and is calculated according to Eqs. (6) and (7), where θ_t refers to the state of the Markov Chain (a parameter vector) at a point in time and M represents the length of the MCMC simulation.

$$C(t) = \frac{1}{M-t} \lim_{t' \rightarrow \infty} [\theta_{t+t'} - \langle \theta \rangle][\theta_{t'} - \langle \theta \rangle] \quad (6)$$

$$\tau = 1 + 2 \sum_{t=1}^{\infty} \frac{C(t)}{C(0)} \quad (7)$$

A property of Markov Chains is that each state depends solely on the one before it. As a result, a high autocorrelation indicates that the Markov Chain was correlated to multiple previous states, suggesting that the simulation was stuck in a certain parameter region. In contrast, a low autocorrelation indicates that the simulation sampled the parameter efficiently and provides evidence that the simulation has reached the true distribution of the system (Sokal, 1997). APT-MCMC uses Goodman’s Acor C code in order to calculate the integrated autocorrelation value of the simulation (Goodman and Weare, 2010).

ODE solving is handled by Sundials CVODE v2.6.2 (Hindmarsh et al., 2005). Each temperature ensemble can be run in parallel and is handled by OpenMP. It is recommended to have at least one processor thread per temperature ensemble. The stretch-move step is

nested-parallelizable if additional threads are available (threads = 2 × number of temperature ensembles). Initialization code to set the sequence of temperatures for a simulation was ported from Python package *emcee* (Foreman-Mackey et al., 2013). Random number generation is handled by TRNG, a package designed for use with parallel Monte Carlo simulations (Bauke and Mertens, 2007). TRNG provides a leapfrog technique, which ensures statistically independent pseudorandom number streams for each parallel thread. Initial seeds for random number generators are provided by the current time plus the thread number.

As depicted in Fig. 3, the user is responsible for providing the parameters and their prior distributions, the likelihood/objective function (the $P(D|\theta)$ portion from Eq. (1)), fitting data, and MCMC options/hyperparameters to setup a simulation. This can be performed in C++ or by using the Python auto-generator package tailored for ODE inverse problems. APT-MCMC has been extensively tested on a Linux-64 platform, but it compiles and runs under OSX with the appropriate developer tools and under Windows 10 x64 under Cygwin and the Windows 10 Ubuntu Subsystem.

For user convenience, APT-MCMC comes with a Python 3 package to automatically generate the necessary C++ files to run a simulation. Eq. (8) shows the type of problems that the package generates.

$$\underset{p}{\text{minimize}} \sum_t [\log(\text{Data}(t)) - \log(X(t))]^2 \quad (8)$$

$$\begin{aligned} \text{subject to } \dot{X}(t) &= f(X, \theta, t, u(t)) \\ X(0) &= X_0 \end{aligned}$$

X represents states, p represents a parameter set, t represents time, u represents an input (optional). This encompasses most parameter-fitting usage cases. The user defines the parameters, ODE equations, fitting data, and MCMC options within an “apt-model” object. Data handling is done by defining experiment(s), where each experiment represents a separate dataset to simultaneously fit. This Python object will then generate readable and compilable C++ code. Advanced features such as implementation of ODE inputs, fitting unknown initial conditions, and adjusting CVODE integration options are supported within the Python interface. Within the objective function, the sum of squared error term is calculated after taking the natural log of the data and states to prevent numerical overflow. This addresses magnitude differences between data and model states, but may be toggled off within the Python code.

At runtime, APT-MCMC provides information on the status of each temperature ensemble, current energies, as well as the location of the best parameter set found since starting the run. Best parameter sets are identified to the θ that minimizes Eq. (8). Once the simulation is completed, all relevant simulation results and variables are saved in a binary file with a

“mcmc” extension. Python and MATLAB code is provided to read this file into a results object for post-processing and analysis.

2.2 APT-MCMC performance evaluation

To evaluate the performance of APT-MCMC, several optimization benchmark functions were tested. They were selected from the list provided by Jamil and Yang (2013). APT-MCMC was tested for its ability to recover the optimal parameter set for each function and for its runtime and memory usage. The functions tested were Ackley 1, Adjiman, Alpine 1, Bard, Beale, Bird, Bohachevsky 3, Booth, Bukin 6, Corana, Damavandi, Devilliers-Glasser, Eggholder, and Griewank (most functions are visualized in Fig. 4). They were chosen to represent a large variety of functional landscapes: basins, valleys, multiple optima, dimensionality, and differentiability. Function equations may be found in the Supplemental Material. Simulations were performed for each benchmark function in order to test if APT-MCMC reaches the optimum parameter set. All simulations were performed with the hyperparameter values noted in Table 3. Parameter histograms for each benchmark are available in the Supplemental Material. A bounded uniform prior was used to provide upper and lower bounds on the parameters specified by each benchmark function. Simulation autocorrelation and PSRF was calculated for each benchmark.

Speed and memory for the benchmarks were compared against the popular Python MCMC package *emcee*, which also features an MCMC algorithm using parallel tempering and sampler ensembles. However, several differences exist between the two packages. First, *emcee* attempts a temperature swap after every attempted stretch-move whereas APT-MCMC will make a user-specified number of “stretch-moves” before attempting a swap. Second, the “burn-in” phase in *emcee* is performed by running the MCMC simulation and initializing at the last location. Within APT-MCMC, this is handled by sampling from the parameter priors for a user-specified number of times and then initializing at the locations of the best samples. To ensure comparable testing, benchmark simulations were repeated with different settings from that of Table 3. n_{steps} was set to 1 in APT-MCMC to attempt a temperature swap after every single stretch-move. “Burn-in”, while not entirely similar between the two algorithms, was set to 1 in both *emcee* and APT-MCMC, which effectively initializes the MCMC simulation with a single sample from the prior. Simulation settings were set to four temperatures, eight processor threads, an ensemble size of 100, and a total swap length of $1e4$.

At the end of the tests, this swap length was determined to be sufficient for both algorithms to reach convergence for each benchmark. All of the aforementioned benchmarks were tested in triplicate for each algorithm. All tests were performed on the same machine with the following specifications: 2× Intel Xeon E5–2670 v3, 256GB DDR4 memory, and Ubuntu 14.04. APT-MCMC and *emcee* memory benchmarks were performed by requesting the resident memory usage from the Linux kernel via the *readproc* command.

The hyperparameters required by APT-MCMC were tested to provide heuristic guidelines. Specifically, APT-MCMC introduces certain hyperparameters that the user must set: n_{ensemble} , n_{chains} , step size, and n_{steps} . Each hyperparameter was tested over a range of values and each value was tested with 100 replicate simulations. For comparability, the

random number generator for each replicate simulation was seeded from a predetermined value for each replicate. All other hyperparameters were kept equal according to Table 3 (unless otherwise noted). To quantify the effects of each hyperparameter, two statistics were used: simulation parameter percent error (or deviation if optimum parameter is 0) and maximum parameter autocorrelation.

The ensemble size parameter was tested as a function of problem dimensionality. The Ackley benchmark was chosen for its dimensional scalability. A 5, 10, 20, and 30 dimensional problem was tested and ensemble size was set at $1\times$, $2\times$, $5\times$, $10\times$, and $20\times$ the number of parameters for each problem. In order to prevent ensemble sizes from biasing the number of simulation samples, swap length was adjusted each time to ensure that a total of 600,000 samples were taken per simulation. The step size parameter controls how far the MCMC samplers traverse the parameter space with each iteration. It was tested against a parameter fitting benchmark, the Bard function. The parameter dictating the number of moves per swap attempt controls the swap attempt frequency. This adjusts how long the lower temperature simulation is able to search a parameter location of interest before it potentially jumps to another optimum. This technique can greatly aid in highly multimodal problems and therefore the 10 dimension Ackley and the Griewank benchmarks were tested. The number of temperatures may be beneficial in highly multimodal problems due to the higher temperature ensemble's ability to identify optima. It was tested against the 10 dimension Ackley and the Bukin benchmarks.

APT-MCMC was tested against a sample ODE fitting problem using the classic Van de Vusse reaction scheme (see Eq. (9)). It was assumed that this reaction takes place within an isothermal continuously stirred tank reactor and the feed is pure A with the following properties: $\frac{F}{V} = 4/7 \text{ min}^{-1}$ and $C_{AI} = 10 \text{ mol/L}$. The resulting ODE system is presented in Eq. (10). Unsteady startup data for this system was generated for states $A(t)$, $B(t)$, $C(t)$, and $D(t)$ at 0.5 minute increments from 0 to 10 min and subject to a sampling noise of $N(\mu = 0, \sigma^2 = 0, 1)$. APT-MCMC fit the three kinetic parameters k_{AB} , k_{BC} , k_{AD} to concentration data of species. Simulation settings were set the those in Table 3 with the exception of swap length set at 10^3 . Each parameter was bounded within $[0, 10]$.

$$\begin{aligned}
\frac{dC_A(t)}{dt} &= \frac{F}{V}(C_{AI} - C_A(t)) - k_{AB}C_A(t) - k_{AD}C_A(t)^2 \\
\frac{dC_B(t)}{dt} &= -\frac{F}{V}C_B(t) - k_{BC}C_B(t) + k_{AB}C_A(t) \\
\frac{dC_C(t)}{dt} &= -\frac{F}{V}C_C(t) + k_{BC}C_B(t) \\
\frac{dC_D(t)}{dt} &= -\frac{F}{V}C_D(t) + \frac{1}{2}k_{AD}C_A(t)^2 \\
k_{AB} &= 0.833, k_{BC} = 1.667, k_{AD} = 0.167
\end{aligned} \tag{10}$$

APT-MCMC also tested a bioreactor system describing glucose to ethanol fermentation by *Saccharomyces cerevisiae* in a batch tank reactor (Fogler, 2013). The equations describing this reactor are shown in Eq. (11). The goal was to fit six parameters: $\mu_{\max}$, K_S , k_d , $Y_{S/C}$, m , $Y_{P/C}$ against time-series data for each state, $C_C(t)$ (yeast concentration), $C_S(t)$ (glucose concentration), $C_P(t)$ (ethanol concentration), at 0.5 h increments up to 14 h. Noise sampled from $n(\mu = 0, \sigma^2 = 2)$ was added to the data. The initial yeast, glucose, and product concentrations were set to 1.0 g/dm³, 250 g/dm³, and 0.0 g/dm³. Each parameter prior was set to a bounded uniform prior about [0, 20] to test APT-MCMC's ability to recover the parameters given a large uninformative prior.

$$\begin{aligned}
r_g(t) &= \mu_{\max} \left(1 - \frac{C_P(t)}{C_P^*}\right)^{0.52} \frac{C_C(t)C_S(t)}{K_S + C_S(t)} \\
\frac{dC_C(t)}{dt} &= r_g - k_d C_C(t) \\
\frac{dC_S(t)}{dt} &= -Y_{S/C} r_g - m C_S(t) \\
\frac{dC_P(t)}{dt} &= Y_{P/C} r_g \\
\mu_{\max} &= 0.33 \text{ h}^{-1}, K_S = 10.7 \text{ g/dm}^3, k_d = 0.01 \text{ h}^{-1} \\
Y_{S/C} &= 12.5 \text{ g/g}, m = 0.03 \text{ g/(g cells h)}, Y_{P/C} = 0.45 \text{ g/g}, C_P^* = 93 \text{ g/dm}^3
\end{aligned} \tag{11}$$

3. Results

3.1. Benchmarks

Table 2 shows the maximum likelihood parameter for each benchmark function. The swap length of simulations were arbitrarily chosen to be 10,000; however, the low values of the autocorrelation and PSRF statistics in Table 2 support that simulations converged over this run length. APT-MCMC returned accurate parameter estimates for most benchmarks. For example, the Damavandi benchmark contains a very large local minimum well near (7, 7)

(Fig. 4i), but APT-MCMC was able to locate the global minimum at (2, 2). The Bukin benchmark (Fig. 4h) contains a sharp ridge with multiple minima and APT-MCMC was able to get close to this point.

The notable exception is with the DeVilliers-Glasser function, represented in Eq. (12). This benchmark is a sum of squares parameter fitting problem and, despite not being previously reported, appears to contain multiple global minima. The highly multimodal histogram (see Supplemental Material) for parameter x_4 and the bimodality of x_1 suggest that this problem has multiple solutions. The hyperbolic tangent is an odd function and its symmetry explains the bimodal distribution at $x_1 \pm 54$. The cyclical nature of $\sin(x)$ explains the multimodal nature of x_4 .

$$\sum_{i=1}^{24} \left[x_1 x_2^{t_i} \tanh[x_3 t_i + \sin(x_4 t_i)] \cos(t_i e^{x_5}) - y_i \right] \quad (12)$$

$$t_i = 0.1(i - 1)$$

$$y_i = (53.81) \left(1.27^{t_i} \right) \tanh[3.012 t_i + \sin(2.13 t_i)] \cos(t_i e^{0.507})$$

$$x_i \in [-500, 500] \quad i = 1, 2, \dots, 5$$

The head-to-head comparison tests between APT-MCMC and *emcee* is summarized in Table 4. For each benchmark tested, APT-MCMC was many times faster than the Python counterpart. This speed came with the cost of using 14% more memory during simulations. The reported times for APT-MCMC do not include generation time (time required for Python to generate C++ code) nor do they include C++ compilation time, but the inclusion of those times (roughly an extra 5–7 s) will not significantly change Table 4. These results are expected due to Python overhead as an interpreted language. While useful for prototyping, the slower execution speed is less desirable MCMC simulations. As a result, APT-MCMC may lack some of the flexibility of *emcee* but succeeds in being the faster tool at runtime. Lastly, the symmetric mean absolute percentage error was computed for each benchmark simulation instead of percentage error to deal with division by zero issues. The error calculations demonstrate that APT-MCMC is able to achieve similar a computational performance with *emcee*.

3.2. Hyperparameters

The $n_{ensemble}$ hyperparameter dictates how many simultaneous samplers exist for each temperature ensemble. These samplers utilize each other's locations to generate directions of search for the next state. Test results for $n_{ensemble}$ are shown in Fig. 5. Autocorrelation increases with problem dimensionality due to the increasing amounts of local minima that are introduced. Parameter error was unaffected by ensemble size. In general, the higher the ensemble sizes (as a function of problem dimensionality) resulted in lower integrated autocorrelation values. Larger ensemble sizes reduced autocorrelation while utilizing the

same amount of computational resources (simulation times were similar among each problem dimensionality). The reduced autocorrelation is because larger ensembles offer more varied and robust search directions. For smaller problems such as the 5 parameter case, there were no distinguishable effects due to the ease of convergence. For large problems, preference should be given towards increasing ensemble size rather than swap length. As a reference, the source code behind *emcee* enforces the minimum ensemble size at $2\times$ the problem parameter dimensionality (Foreman-Mackey et al., 2013).

The n_{chains} hyperparameter in a simulation helps the system explore highly modal parameter spaces by running many tempered simulations simultaneously. Samplers in high temperature ensembles can locate alternative energetically favorable regions (potential locations of local minima) with higher probabilities and pass this information to the lower temperature samplers for minima identification. Fig. 6a shows the results for the Bukin benchmark. There were diminishing decreases of autocorrelation at higher number of chains, which indicate that this benchmark function had local minima that were relatively easy to escape from. In contrast, Fig. 6b shows the results for the 10-dimension Ackley benchmark and demonstrates a linear relationship between autocorrelation and the number of chains. The 10-dimension Ackley benchmark contains more local minima than the Bukin benchmark and benefits more from the additional chains. n_{chains} cannot be set lower than 2 because parameter swapping would not be possible with a single temperature Markov Chain. Increasing n_{chains} increases the magnitude of the temperature sequence and higher magnitudes benefit from the aforementioned favorable region exchange. This information exchange leads to faster exploration of the parameter space and therefore lower autocorrelation values. While the number of tempered chains increases computational cost, they can in parallel and scale well via OpenMP. As a result, this hyperparameter can be set to equal the number of CPU threads available on the machine running the simulation. APT-MCMC is able to utilize up to the number of detected CPU threads (logical cores) on the machine it runs on. OpenMP synchronization locks in the APT-MCMC OpenMP code ensure that all chain movements are completed prior to a swap attempt. Setting n_{chains} to a value that exceeds the number of CPU threads will greatly increase simulation time because some threads have to calculate multiple tempered chains while others are forced to idle. Computational resources are better spent on longer simulations with an n_{chains} equivalent to the number of cores. It is recommended to set n_{chains} to as many available CPU cores as possible because simulation efficiency improves and parallelization offsets any computational trade-offs.

The n_{steps} hyperparameter describe how much a sampler ensemble explores a local region of the parameter space before it is potentially swapped to a different parameter location. The effect of this hyperparameter is shown in Fig. 7a for the Griewank benchmark. Both autocorrelation and parameter error decreases with increasing n_{steps} . In contrast, simulation time increased. Fig. 7b shows the results for the 10-dimension Ackley function, which demonstrate similar trends. Larger n_{steps} allow for the ensemble of samplers to explore an region of interest for longer before they are potentially transferred to a different region. Very small values may prevent the samplers from exploring optima sufficiently. As n_{steps} became larger, autocorrelation was unaffected, but simulation time increased. For both benchmark functions, there were diminishing returns in autocorrelation decreases compared to the

increases in simulation time. Values within 20–30 offered trade-offs between autocorrelation, simulation time, and parameter accuracy.

The step size hyperparameter determines how far a proposed parameter can move in accordance to the stretch-move scheme. Small step sizes prevent proper parameter space exploration. High step sizes allow potential movements farther away from the rest of the ensemble samplers. Fig. 8 shows the results for the Bard benchmark. The suggested value for this hyperparameter is between 2 and 3 where there is a clear minimum in autocorrelation. As a reference, Goodman and Weare suggested a value of 2 for this hyperparameter (Goodman and Weare, 2010).

3.3 Van de Vusse reaction scheme

Fig. 9 illustrates the Bayesian posterior results from an ODE parameter fitting problem involving the Van de Vusse reaction scheme. Parameter histograms are shown along with a fitted kernel density estimation. The peaks of each parameter's probability density correspond with the true parameters used to generate the data for the Van de Vusse reactions. Furthermore, the best parameter vector from the simulation (associated with the highest likelihood) demonstrated 1–3% difference from the true parameter vector. The fit from the best parameter set is shown in Fig. 10.

3.4 Bioreactor

Fig. 11 illustrates the Bayesian posterior results from the ODE fitting of a bioreactor. Histogram peaks for most of the parameters matched the true values. The histogram peaks for the $Y_{S/C}$ and $Y_{P/C}$ were under-predicted. This was likely due to the difficulty the algorithm had in fitting K_S , a Michaelis-Menten half-max concentration term, as indicated by the high variance within the K_S distribution. As a result, the biomass growth term, r_g , differed from the true growth rate and the two yield parameters were used to compensate. Michaelis-Menten half-maximum constants are difficult to fit without carefully characterized reaction rate data that are often experimentally challenging to collect. If such data were available, the parameter estimates would be significantly improved based on the tightening of the bounds on the K_S parameter. Regardless, this example highlights a major benefit of APT-MCMC. Compared with traditional nonlinear least squares fitting algorithms such as the trust region reflective or Levenburg-Marquardt, APT-MCMC does not require a good set of initial conditions to achieve a promising model fit; instead, the use of broad parameter bounds is sufficient to estimate parameter values and distributions, which is especially useful when some parameters may prove difficult to identify from the available data. The fit from the best parameter set is shown in Fig. 12.

3.5. Limitations

One current limitation is associated with the Python autogeneration package. It assumes uniform priors on all parameters and a sum of squared error likelihood function. However, the C++ code can be easily modified to overcome these limitations. Should the user require additional features for advanced fitting procedures, the user can auto generate the files with Python and then modify the appropriate C++ functions to their needs. Any problem that can

be posed as a maximum likelihood can be programmed into APT-MCMC, but C++ programming of Usermodel.cpp is required.

Another limitation is the lack of a termination criterion, which is inherent to MCMC techniques. While APT-MCMC seeks to alleviate this issue by calculating the autocorrelation and PSRF values, MCMC statistics are necessary but not sufficient tests of convergence. Practically, the user should set the simulation to a very long swap length. While the simulation proceeds, the user should analyze the partial APT-MCMC results as they become available and make a stopping decision based on the statistical tests and in determining if the distributions of the parameter posterior histograms are still changing. MCMC simulations may also get stuck in a meta-stable convergence state (which is not testable either), but a popular solution is to run multiple MCMC simulations from distinct initial conditions (Sokal, 1997). The second simulation can be run with tighter bounds on each parameter based on the posteriors from the first simulation and one can verify if simulations converge in the same parameter location.

4. Conclusion

APT-MCMC is presented as a fast MCMC platform tailored for use in solving parameter-fitting problems. User-convenience was prioritized during the creation of APT-MCMC. Traditionally, MCMC packages are provided in either easy-to-prototype, but slow, or difficult-to-learn, but fast, languages. The Python auto-generation package serves to provide users with an easy way to prototype and set up simulations while retaining the speed associated with a compiled and static language (C++). Additionally, MCMC hyperparameters are provided based on tests using common optimization benchmarks. APT-MCMC is ready for use and exists as an open source project on GitLab.

Supplementary Material

Refer to Web version on PubMed Central for supplementary material.

Acknowledgments

Funding

This work was supported by the National Institutes of Health [R01-GM-105728] and the United States Department of Education [GAANN P200A120195].

References

- Altekar G, Dwarkadas S, Huelsenbeck JP, Ronquist F, 2004 Parallel metropolis coupled Markov chain Monte Carlo for Bayesian phylogenetic inference. *Bioinformatics* 20, 407–415. doi:10.1093/bioinformatics/btg427. [PubMed: 14960467]
- Andrieu C, de Freitas N, Doucet A, Jordan MI, 2003 An introduction to MCMC for machine learning. *Mach. Learn.* 50, 5–43. doi:10.1023/A:1020281327116.
- Bauke H, Mertens S, 2007 Random numbers for large-scale distributed Monte Carlo simulations. *Phys. Rev. E* 75, 1–14. doi:10.1103/PhysRevE.75.066701.
- Diaconis P, 2009 The Markov Chain Monte Carlo revolution. *Bull. Am. Math. Soc.* 46, 179–205. doi:10.1090/S0273-0979-08-01238-X.

- Doucet A, Godsill S, Andrieu C, 2000 On sequential Monte Carlo sampling methods for Bayesian filtering. *Stat. Comput.* 10, 197–208. doi:10.1023/A:1008935410038.
- Earl DJ, Deem MW, 2005 Parallel tempering: theory, applications, and new perspectives. *Phys. Chem. Chem. Phys.* 7, 3910. doi:10.1039/b509983h arXiv: 0508111. [PubMed: 19810318]
- Fogler HS, 2013 *Bioreactors Elements of Chemical Reaction Engineering*, fourth ed. Pearson Prentice Hall, Upper Saddle River, NJ 07458, pp. 432–434 (chapter 7).
- Foreman-Mackey D, Hogg DW, Lang D, Goodman J, 2013 emcee: the MCMC Hammer. *Publ. Astronom. Soc. Pac.* 125, 306–312. doi:10.1086/670067 arXiv: 1202.3665.
- Goodman J, Weare J, 2010 Ensemble samplers with affine invariance. *Commun. Appl. Math. Comput. Sci.* 5, 65–80. doi:10.2140/camcos.2010.5.65.
- Hindmarsh AC, Brown PN, Grant KE, Lee SL, Serban R, Shumaker DE, Woodward CS, 2005 SUNDIALS: suite of nonlinear and differential/algebraic equation solvers. *ACM Trans. Math. Softw.* 31, 363–396. doi:10.1145/1089014.1089020.
- Hogg J, 2013 *Advances in Rule-based Modeling: Compartments, Energy, and Hybrid Simulation, with Application to Sepsis and Cell Signaling*, (Ph.D. thesis) University of Pittsburgh.
- Jamil M, Yang XS, 2013 A literature survey of benchmark functions for global optimization problems. *Int. J. Math. Modell. Numer. Opt.* 4, 150–194. doi:10.1504/IJMMNO.2013.055204 arXiv: 1308.4008.
- Nocedal J, Wright SJ, 1999 *Numerical Optimization*. Springer-Verlag, New York.
- Sokal A, 1997 Monte Carlo methods in statistical mechanics: foundations and new algorithms In: DeWitt-Morette C, Cartier P, Folacci A (Eds.), *Functional Integration: Basics and Applications*. Springer US, pp. 131–192 10.1007/978-1-4899-0319-8_6.

Fig. 1. MCMC results from exploring a highly correlated probability function. (A) Contour plot for an anisotropic distribution. Epsilon was set to 0.001. (B) and (C) MCMC results yields marginal probability distributions for x_1 and x_2 . (D) These distributions can be combined into a joint distribution plot to identify parameter correlations.

Fig. 2. Effects of varying temperatures on the Boltzmann distribution. Shown is a single parameter, θ , and the system's energy at that parameter value. The steep local minima are difficult to escape because the escape probability as defined by Metropolis-Hastings criterion becomes exponentially smaller. Higher temperatures modulate the energy landscape such that local minima become shallow and easier to escape.

Fig. 3.

Schematic of generating an APT-MCMC simulation in C++. Users need to specify their parameter-fitting problem using `Usermodel.cpp` and set MCMC options in `options.hpp`. The Python 3 package can autogenerate these files by providing the ODE states and their equations, parameter lower and upper bounds, and the dataset(s) for fitting.

Fig. 4. Visualizations for the benchmarks tested. Bard, Corana, Devilliers-Glasser are not shown due to high parameter dimensionality.

Fig. 5.

100 repeated simulations were used to test the effects of the $n_{msemble}$ hyperparameter by using the dimension-scalable Ackley benchmark function. The value of $n_{mem}b_{le}$ was set as a function of the dimensionality of the Ackley function and denoted by #x nPar (multiplier x # of parameters). Mean $\pm$ SEM of best parameter error and integrated autocorrelation shown. Raw parameter error is reported because the optimum parameters are located at 0. Problems benefit from larger ensemble sizes, but this comes at the cost of increased computation time.

(a) 2-dimension Bukin benchmark function

(b) 10-dimension Ackley benchmark function

Fig. 6.

100 repeated simulations were used to test the effects of the n_{chains} hyperparameter by using benchmark functions Bukin and Ackley. Mean $\pm$ SEM of integrated autocorrelation shown and parameter error shown. Additional temperature ensembles decreased simulation autocorrelation, although there were diminishing returns after 4. A higher number of temperatures results in improved parameter mixing and simulation efficiency. There was no significant effect on the accuracy of the maximum likelihood parameter vector.

Fig. 7.

100 repeated simulations were used to test the effects of the n_{steps} hyperparameter by using benchmark functions. Mean $\pm$ SEM of best parameter error and integrated autocorrelation shown and parameter error shown. Increasing the steps per swap decreased maximum autocorrelation but increased the simulation time. Parameter error decreased only in the Griewank case.

Fig. 8.

100 repeated simulations were used to test the effects of the step size hyperparameter by using the 2-dimension Bard benchmark function. Mean $\pm$ SEM of best parameter percent error and integrated autocorrelation shown. The minimum autocorrelation was achieved at a clear optimum at 2.5–3. There was no effect on the accuracy of the maximum likelihood parameter vector.

Fig. 9.

Parameter histogram from simulation to fit 3 parameters to Van de Vusse reaction data. The vertical line in each subplot represents the true parameter values used to generate the data. The smooth line in each subplot is a kernel density estimation of the histogram.

Fig. 10.

Data and best parameter fit for the Van de Vuse reaction scheme system. Simulated state data (with noise) are shown as dots in either graph. Concentration trajectories from the best parameter fit are shown as lines for each states.

Fig. 11. Parameter histogram from simulation to fit 6 parameters in a batch bioreactor. The vertical line in each subplot represents the true parameter values used to generate the data. The smooth line in each subplot is a kernel density estimation of the histogram.

Fig. 12. Data and best parameter fit for the bioreactor system. Simulated state data (with noise) are shown as dots in either graph. Concentration trajectories from the best parameter fit are shown as lines for each states.

Table 1

Overview of the nomenclature used.

Nomenclature	Meaning
n_{steps}	Hyperparameter Number of MCMC sampler steps to take before a parameter location swap is attempted between tempered chains
$n_{ensemble}$	Hyperparameter Size of MCMC ensemble of samplers within each tempered chain
n_{chains}	Hyperparameter Number of tempered chains to simultaneously explore the problem
Swap Length	Hyperparameter Number of proposed parameter location swaps among tempered chains; determines simulation length
τ	Statistical Measure Integrated autocorrelation, measure the how well an MCMC simulation explored the parameter space (lower is better)
PSRF	Statistical Measure Potential scale reduction factor, a measurement of simulation variance between tempered chains and within-chain ensembles (lower is better)

Table 2

Overview of results from several benchmark functions. Benchmark functions are provided in Supplemental Materials. For each benchmark, the integrated autocorrelation time(τ), potential scale reduction factor (PSRF), the best parameter set, and the true optimum in parenthesis are reported. Each simulation was performed using the hyperparameters in Table 3.

Benchmark name	τ	PSRF	Param 1	Param 2	Param 3	Param 4	Param 5
Ackley	1.16	1.00	$5.58 * 10^{-3}$ (0)	$-1.53 * 10^{-2}$ (0)	$9.78 * 10^{-1}$ (0)		
Adjiman	1.85	1.03	2.00 (2)	$9.60 * 10^{-2}$ (0.106)			
Alpine	2.34	1.00	$6.80 * 10^{-2}$ (0)	0.143 (0)	$3.19 * 10^{-2}$ (0)	$1.52 * 10^{-2}$ (0)	$2.67 * 10^{-2}$ (0)
Bard	1.85	1.20	$8.11 * 10^{-2}$ ($8.24 * 10^{-2}$)	1.09 (1.13)	2.38 (2.34)		
Beale	1.95	1.06	3.00 (3)	0.500 (0.5)			
Bird	4.05	1.02	-1.58 (-1.58)	-3.13 (-3.13)			
Bohachevsky	1.18	1.00	$1.86 * 10^{-3}$ (0)	$1.33 * 10^{-3}$ (0)			
Booth	1.85	1.02	1.00 (1)	3.00 (3)			
Bukin	1.85	1.05	-10.56 (-10)	0.753(1)			
Corana	1.93	1.00	$3.86 * 10^{-2}$ (0)	$8.42 * 10^{-3}$ (0)	$3.29 * 10^{-2}$ (0)	$4.76 * 10^{-2}$ (0)	
Damavandi	1.85	1.00	2.00 (2)	2.00 (2)			
Devilliers	1.95	1.18	53.9 (53.81)	1.27 (1.27)	2.95 (3.01)	65.0 (2.13)	.507 (.507)
Eggholder	1.85	1.35	512 (512)	404 (404.23)			
Griewank	1.10	1.00	$-4.05 * 10^{-2}$ (0)	$3.58 * 10^{-2}$ (0)			

Table 3

Simulation settings for benchmark tests.

Hyperparameter	Setting
n_{chains}	4
$n_{ensemble}$	100
n_{steps}	25
Burn-in phase	10 0 0
Swap length	$10^4{}^a$

^aDevilliers Glasser swap length set to 10^6 .

Table 4

Comparison of computing resources used by APT-MCMC and Python package *emcee*. Results shown are averaged from $n = 3$ test simulations. Memory was the physical resident memory in mebibytes, as reported by the Linux kernel. Each benchmark function simulation was set according to the hyperparameters in Table 3 with the exception of $n_{steps} = 1$ (for comparability between APT-MCMC and *emcee*). Burn-in for both algorithms were set to 1 iteration. APT-MCMC was faster and used less memory for each case. The error columns represent the calculated symmetric mean absolute percentage error averaged over all parameters and across replicates.

Benchmark name	<i>emcee</i>			APT-MCMC		
	Time (s)	Memory (MiB)	Error	Time (s)	Memory (MiB)	Error
Ackley	118	189	$9.63 * 10^{-01}$	2	193	$9.63 * 10^{-01}$
Adjiman	125	159	$9.73 * 10^{-03}$	2	194	$9.50 * 10^{-03}$
Alpine	110	251	$9.63 * 10^{-01}$	3	255	$9.63 * 10^{-01}$
Bard	118	190	$7.05 * 10^{-01}$	2	193	$1.86 * 10^{-02}$
Beale	128	160	$2.58 * 10^{-04}$	2	193	$5.34 * 10^{-04}$
Bird	124	157	$1.14 * 10^{-05}$	2	193	$6.67 * 10^{-01}$
Bohach	116	158	$9.63 * 10^{-01}$	2	193	$9.63 * 10^{-01}$
Booth	116	158	$2.06 * 10^{-04}$	2	193	$1.98 * 10^{-04}$
Bukin	115	158	$4.89 * 10^{-01}$	2	193	$1.01 * 10^{-01}$
Corana	118	220	$9.63 * 10^{-01}$	3	255	$9.63 * 10^{-01}$
Damavandi	116	157	$5.35 * 10^{-01}$	2	193	$2.86 * 10^{-04}$
deVilliersGlasser	187	252	$8.90 * 10^{-01}$	9	255	$2.88 * 10^{-01}$
Eggholder	114	157	$6.48 * 10^{-06}$	2	193	$1.40 * 10^{-04}$
Griewank	116	158	$9.63 * 10^{-01}$	2	193	$9.63 * 10^{-01}$
Average	123	180	$5.32 * 10^{-01}$	2	206	$4.21 * 10^{-01}$