evolMC: a package for Monte-Carlo simulation

W Burchett E Roualdes G Weyenberg

November 25, 2013

Abstract

Sampling from multi-modal target distributions proves difficult for basic MCMC methods. Borrowing from the evolutionary based algorithms of optimization theory, Liang and Wong [2000] introduced evolutionary Monte Carlo (EMC). EMC begins with parallel tempering and then adds to it a step in which sequences intelligently evolve by learning from each other. A comparative simulation highlights the effectiveness of EMC over other sampling methods.

Population based MCMC methods are becoming increasingly common in applied statistics. Parallel tempering, for instance, starts by simulating multiple sequences from a heated target distribution to encourage a more thorough global search of the support space. Liang and Wong [2000] introduced evolutionary MC that builds upon, and generalizes, parallel tempering by blending with it theory from evolutionary algorithms. Evolutionary Monte Carlo (EMC) is a population based Monte Carlo method that intelligently evolves at each iteration. [citation needed]

The discussion of population based Monte Carlo methods necessarily begins by introducing some terminology. The following recounts the terminology given in Chapter 5 of Liang *et al.* [2011]: We wish to sample from a distribution

$$f(x|t) \propto \exp\{-H(x)/t\}$$

where $t \geq 1$ is called the *temperature* and $H(x)$, called the *fitness function*, corresponds to the negative log-density of x, up to a constant. A *population* $\mathbf{x}$ consists of N *individuals* x_i, $i = 1, \ldots, N$, and we also define an

associated set of temperatures $\mathbf{t} = (t_1, \ldots, t_N)$. ($x_i \in \mathbb{R}^d$ and the t_i are in descending order.) Each individual x_i is independently sampled from the distribution $f_i(x_i) \propto f(x_i|t_i)$. For any such distribution, the *partition function* $Z(t) = \int_{\mathcal{X}} f(x|t)dx$ is the inverse of the normalizing constant. The *Boltzmann distribution* of the population $\mathbf{x}$ is the product of the individual distributions, $f(\mathbf{x}) \propto \exp\{-\sum_{i=1}^{N} H(x_i)/t_i\}$, with $Z(\mathbf{t}) := \prod_{i=1}^{N} Z(t_i)$ as the inverse of the normalizing constant.

Iterations of the EMC algorithm consist of three operations: mutation, crossover, and exchange. Each type of operation makes a specific type of modification to the current population, and the change is carried forward to the next generation with a probability governed by a Metropolis-type rule.

In general terms, a mutation operation adds noise to the current population. This can be analogous to a basic random walk type proposal, but more elaborate methods have been proposed. A crossover operator consists of randomly selecting two individuals and generating replacement individuals by somehow mixing the parent values, analogous to the Chromosomal crossovers that sometimes occur during Meiosis. An exchange step proposes a switch of two randomly selected individuals without switching their associated temperatures.

In the remainder of this paper we compare the performance of the EMC algorithm to parallel tempering and a naïve Metropolis sampler in the context of a 3 dimensional distribution with 20 modes. Section two describes in greater detail the algorithms used and the simulation setup. While results of the simulation are presented and discussed in sections 3 and 4.

2 Methods

The target distribution for our study is a 20-part mixture of trivariate Gaussian distributions. Each part of the mixture is $\mathcal{N}_3(\mu_i, 0.1I_3)$, and each vector μ_i is located uniformly in $[-10, 10] \times [-10, 10] \times [-10, 10]$. This distribution possesses numerous modes that are separated by relatively large regions of low probability, making it challenging for a basic Metropolis sampler to mix between the modes.

2.1 Algorithms

The EMC algorithm we have implemented is described briefly in Algorithms 1–4. The mutate step uses a simple random-walk approach, where the variance of the noise distribution depends on the temperature of the individual being modified. The crossover we used is a simple single-splice one, where the parents are selected with weights based on their fitness.

[Algorithm 1 about here.]

[Algorithm 2 about here.]

[Algorithm 3 about here.]

[Algorithm 4 about here.]

2.2 Simulations

We began our investigation of the EMC algorithm by attempting to determine good values for the mutation proposal distributions.

Parallel tempering is a special case of EMC, where the mutation probability, $p_m = 1$. A n-th simulation varied the mutation probability.

3 Results

3.1 Software

We present the R package evolMC, which is a framework for doing Markov-Chain Monte Carlo simulations. The evolMC package includes implementations of several MCMC algorithms, including Metropolis-Hastings, Gibbs, parallel tempering, and the evolutionary updating methods. The package is available from `http://github.com/grady/evol-mc/`.

The package aims to provide a general framework for constructing a chain, leaving it up to the user to provide the specific pieces specific to the problem. The user must first define a *state object* and then implement a number of functions which act on this state object. In many traditional cases this state object can simply be a vector of reals, but any R object (e.g. a phylogenetic tree object defined by a 3rd-party package) can be used. The functions provided by evolMC will eventually produce a *chain* object, which is simply an R list where each element is an instance of the state object.

3.2 Simulations

Verbiage

4 Discussion

The winner is...P

References

Faming Liang and Wing Hung Wong. Evolutionary monte carlo: Applications to cp model sampling and change point problem. *Statistica sinica*, 10(2):317–342, 2000.

Faming Liang, Chuanhai Liu, and Raymond Carroll. *Advanced Markov chain Monte Carlo methods: learning from past samples*, volume 714. Wiley. com, 2011.

A evolMC

The evolMC package includes a vignette demonstrating a few basic use cases, with example code. Here we briefly discuss the architectural vision underlying the package and a few of the most important functions.

evolMC was designed to be as flexible as possible, and hence it operates on a quite abstract level. It is written in a very functional style of programming; most of the important functions in evolMC accept functions as arguments and return new functions. An important goal was for the user to be free to use whatever object they wish to represent a realization of the random element in question. This departs from many existing R MCMC packages, which require vector- or array-type structures for the random elements in question.

The main job of the user of evolMC is to define a few basic functions which can interact in specified ways with whatever object is selected to represent states of the chain. For example, the user will often need to implement a function which evaluates a log-density for a given state. Using the routines

provided by evolMC, one will then manipulate these functions, eventually producing a function which implements some desired MCMC algorithm.

The core function in evolMC is `iterate(n,f,x)`. This function is also among the least interesting, because it simply creates a list where the $(k+1)$-th element is $f^k(x)$. (Function exponentiation means iterated application here.) If the provided function `f` is "carefully selected" then the resulting chain might have "desirable properties". Of course, we are likely interested in the case where the desirable property in question is that the resulting list represents a sample from some target distribution. Most other functions in the package exist to help one in constructing a function `f` with specific desirable properties.

Perhaps the most basic MCMC algorithm is the Gibbs sampler. Suppose we want to sample from $[x, y]$. If we can sample from $[x|y]$ and $[y|x]$, then we can implement a Gibbs sampler. To do this one need only implement functions `rxy` and `ryx`, which do the respective sampling. These functions should accept the current state as their first argument, and return a new state object with the relavant changes made.

Once one has these functions, the gibbs function can be used to produce a new updating function which will implement the Gibbs algorithm. The basic signature is `gibbs(...)`. The ... argument allows one to provide any number of updating functions; in our case we have two, `rxy` and `ryx`. The result of `gibbs(rxy,ryx)` is a new "carefully selected" function with signature `function(state)` which acts as a Gibbs updater. If called repeatedly (e.g. by iterate) it will alternate between using `rxy` and `ryx` to update the state.[1]

A second important algorithm is the Metropolis algorithm. The function `metropolis` helps one to construct an updating function which implements Metropolis-Hastings. To use this function one must first implement: the log-density function for the target distribution (`ln.d`); a proposal generator function (`r.prop`); and (if required) a log-density function for the proposal generator (`ln.d.prop`). A call `metropolis(ln.d, r.prop, ln.d.prop)` will return a "carefully selected" function (with signature `function(state)`), which implements the Metropolis-Hastings scheme you have defined.

These three basic building blocks can be used to construct a surprising number of more advanced MCMC algorithms, and indeed many of the more

[1]The sequential choice of updating functions is the default mode of operation for the function returned by `gibbs`. It can also stochastically select an updating function according to provided weights.

advanced algorithms provided by evolMC are constructed using only these pieces.

Algorithm 1 Evolutionary Monte Carlo

procedure EMC
  with prob p_m ▷ p_m is the *mutation probability*.
    MUTATE
  otherwise
    CROSSOVER
  end
  EXCHANGE
end procedure

Algorithm 2 A random-walk *mutation*.

procedure MUTATE
  Copy the current population to x.
  for all individuals in x **do**
    $y \leftarrow \mathcal{N}_d(x_i, t_i\sigma^2 I)$
    with prob $\min\{1, \exp(\cdots)\}$
      $x_i \leftarrow y$
    end
  end for
  Set current population to x.
end procedure

Algorithm 3 The fitness-weighted *crossover*.

procedure CROSSOVER
  Copy the current population to x.
  for all individuals in x **do**
    $w_i \leftarrow \exp(-H(x_i))$
  end for
  Select k uniformly from $\{1{:}\, d\}$.
  Select i from $\{1{:}\, N\}$ with weights proportional to $\{w_\cdot\}$.
  Select j uniformly from $\{1{:}\, N\}\backslash\{i\}$.
  In x, swap elements $k{:}\, d$ of individuals i and j.
  with prob $\min\{1, \exp(-H(x_i)/t_i + H(x_j)/t_j + \cdots)\}$
    Set the current population to x.
  end
end procedure

Algorithm 4 The *exchange* attempts to swap individuals between neighboring temperature states.

procedure EXCHANGE
  Copy the current population to x.
  Select i uniformly from $\{1{:}\, N\}$.
  Select j uniformly from $\{i \pm 1\} \cap \{1{:}\, N\}$.
  Swap individuals i, j of x.
  with prob $\min\{1, \exp(-H(x_i)/t_i + H(x_j)/t_j)\}$
    Set the current population to x.
  end
end procedure