|  |
| --- |
| UCCS |
| Monte Carlo Markov Chains Hellinger Convergence Metrics |
| [Type the document subtitle] |

|  |
| --- |
| Broberg, Ronald  [Pick the date] |

Table of Contents

[1 Monte Carlo Integration 3](#_Toc444175449)

[1.1 Example: Approximating the integral 4](#_Toc444175450)

[1.2 Example: Approximating the expected value of the Beta distribution 5](#_Toc444175451)

[1.3 Monte Carlo Approximation for Optimization 7](#_Toc444175452)

[1.4 Example: Monte Carlo Optimization of 7](#_Toc444175453)

[1.5 Wrapping Up 8](#_Toc444175454)

[2 Markov Chains 8](#_Toc444175455)

[2.1 Example: Predicting the weather with a finite state-space Markov chain 10](#_Toc444175456)

[2.2 Continuous state-space Markov chains 12](#_Toc444175457)

[2.3 Example: Sampling from a continuous distribution using continuous state-space Markov chains 12](#_Toc444175458)

[2.4 Wrapping Up 13](#_Toc444175459)

[3 MCMC: The Metropolis Sampler 13](#_Toc444175460)

[3.1 Metropolis Sampling 14](#_Toc444175461)

[3.2 Example: Using the Metropolis algorithm to sample from an unknown distribution 15](#_Toc444175462)

[3.3 Reversibility of the transition operator 17](#_Toc444175463)

[4 MCMC: The Metropolis-Hastings Sampler 17](#_Toc444175464)

[4.1 Example: Sampling from a Bayesian posterior with improper prior 18](#_Toc444175465)

[4.2 Wrapping Up 22](#_Toc444175466)

[5 Hellinger distance 23](#_Toc444175467)

[5.1 Properties 23](#_Toc444175468)

[5.2 Examples 23](#_Toc444175469)

[1 Appendix: MCMC Monte Carlo Approximation of Integral Code 25](#_Toc444175470)

[2 Appendix: MCMC Monte Carlo Beta Expectation Code 26](#_Toc444175471)

[3 Appendix: MCMC Monte Carlo Optimization of Exponential Function Code 27](#_Toc444175472)

[4 Appendix: MCMC Markov Chain Finite State Transitions Code 28](#_Toc444175473)

[5 Appendix: MCMC Markov Chain Continous State Transitions Code 29](#_Toc444175474)

[6 Appendix: MCMC Metropolis Markov Sampler Code 30](#_Toc444175475)

[7 Appendix: MCMC Metropolis Hastings Priors and Posterior Code 32](#_Toc444175476)

# Monte Carlo Integration

<https://theclevermachine.wordpress.com/2012/09/22/monte-carlo-approximations/>

Using statistical methods we often run into integrals that take the form:

For instance, the expected value of a some function of a random variable .

and many quantities essential for Bayesian methods such as the marginal likelihood a.k.a “model evidence”

involve integrals of this form.  Sometimes (not often) such an integral can be evaluated analytically. When a closed form solution does not exist, numeric integration methods can be applied. However numerical methods quickly become intractable for any practical application that requires more than a small number of dimensions. This is where Monte Carlo approximation comes in. Monte Carlo approximation allows us to calculate an estimate for the value of Iby transforming the  integration problem into a procedure of sampling values from a tractable probability distribution and calculating the average of those samples. Here’s what I mean:

If the function fulfills two simple criteria, namely that the function is always positive on the interval

and that the integral of the function is finite

then we can define a corresponding probability distribution on the interval

Another way to think of it is that is a probability distribution scaled by a constant .

Using this link between probability distributions and, we can restate the original integration as

where samples are drawn independently from . This leads to a simple 4-Step Procedure for performing Monte Carlo approximation to the integral :

1. Identify
2. Identify and from it determine and .
3. Draw independent samples from
4. Evaluate

The larger the number of samples we draw, the better our approximation to the actual value of . This 4-step procedure is demonstrated in a toy examples below:

## Example: Approximating the integral

Say we want to calculate the integral:

We can calculate the closed form solution of this integral using integration by parts:

And

Or…we could calculate the Monte Carlo approximation of this integral.

**Step 1** we identify

**Step 2** we identify

and from this can also determine the probability distribution function . According to the definition expression for given above we determine to be:

**Step 3:** The expression on the right is the definition for the uniform distribution , which is easy to sample from using the MATLAB ***rand()*** (Notice too that the constant ).

**Step 4:** we calculate the Monte Carlo approximation as

where each is sampled from the standard uniform distribution . Below is some MATLAB code running the Monte Carlo Approximation for two different values of

SEE Appendix MCMC Monte Carlo Estimate of Integral

comparing the values of the variables ***Ihat1*** and ***Ihat2*** we see that the Monte Carlo approximation is better for a larger number of samples.

## Example: Approximating the expected value of the Beta distribution

Lets look at how the 4-step Monte Carlo approximation procedure can be used to calculate expectations. In this example we will calculate

Where and is the Beta function.

**Step 1:** we identify

**Step 2:** the function is simply the probability density function due the expression for above:

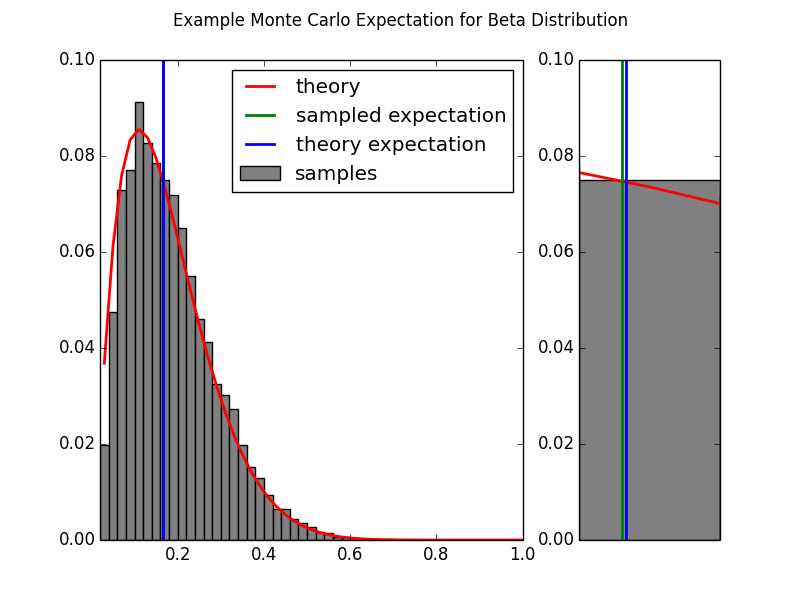
**Step 3** we can use MATLAB to easily draw independent samples using the function ***betarnd()***. And finally,

**Step 4** we approximate the expectation with the expression

below is some MATLAB code that performs this approximation of the expected value.

SEE Appendix MCMC Monte Carlo Beta Expectation

And the output of the code:



The analytical solution for the expected value of this Beta distribution:

is quite close to our approximation (also indicated by the small distance between the blue and green lines on the plot above).

## Monte Carlo Approximation for Optimization

Monte Carlo Approximation can also be used to solve optimization problems of the form:

If fulfills the same criteria described above (namely that it is a scaled version of a probability distribution), then (as above) we can define the probability function

This allows us to instead solve the problem

If we can sample from , the soution is easily found by drawing samples from and determining the location of the samples that has the highest density (Note that the solution is not dependent of the value of . The following example demonstrates Monte Carlo optimization:

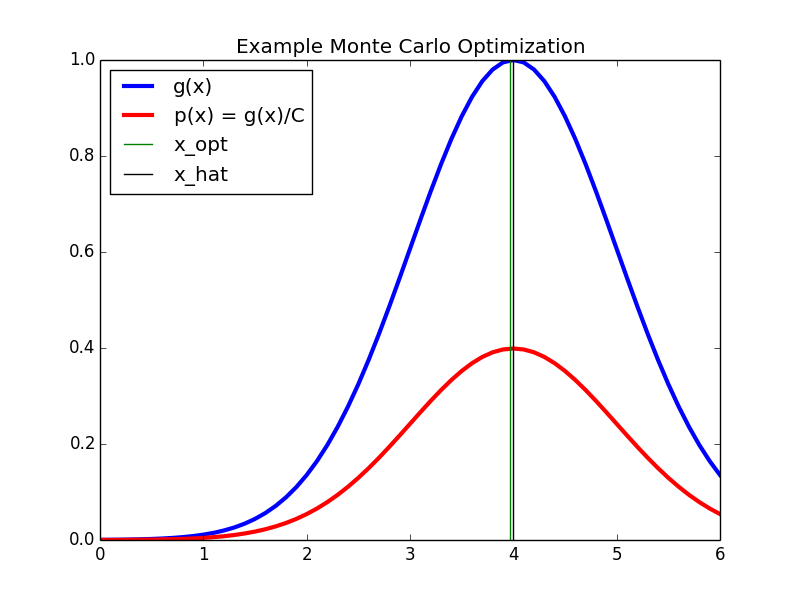
## Example: Monte Carlo Optimization of

Say we would like to find the value of which optimizes the function **.** In other words we want to solve the problem

We could solve for using standard calculus methods, but a clever trick is to use Monte Carlo approximation to solve the problem. First, notice that , is  a scaled version of a Normal distribution with mean equal to 4 and unit variance:

Where is the Normal distribution and . This means we can solve for by drawing samples from the normal distribution and determining where those samples have the highest density. The following chunk of matlab code solves the optimization problem in this way.

SEE Appendix Monte Carlo Optimization of Exponential Function Code



In the code output above we see the function ) we want to optimize in blue and the Normal distribution ) from which we draw samples in red. The Monte Carlo method provides a good approximation (green) to the real solution (black).

## Wrapping Up

In the toy examples above it was easy to sample from). However, for practical problems the distributions we want to sample from are often complex and operate in many dimensions. For these problems more clever sampling methods have to be used. Such sampling methods include Inverse Transform Sampling, Rejection Sampling, Importance Sampling, and Markov Chain Monte Carlo methods such as the Metropolis Hasting algorithm and the Gibbs sampler, each of which I plan to cover in separate posts.

# Markov Chains

<https://theclevermachine.wordpress.com/2012/09/24/a-brief-introduction-to-markov-chains/>

Markov chains are an essential component of Markov chain Monte Carlo (MCMC) techniques. Under MCMC, the Markov chain is used to sample from some target distribution. To get a better understanding of what a Markov chain is, and further, how it can be used to sample form a distribution, this post introduces and applies a few basic concepts.

A Markov chain is a stochastic process that operates sequentially (e.g. temporally), transitioning from one state to another within an allowed set of states.†

A Markov chain is defined by three elements:

1. A ***state space*** , which is a set of values that the chain is allowed to take
2. A ***transition operator*** that defines the probability of moving from state to .
3. An ***initial condition distribution*** which defines the probability of being in any one of the possible states at the initial iteration .

The Markov chain starts at some initial state, which is sampled from , then transitions from one state to another according to the transition operator .

A Markov chain is called ***memoryless***if the next state only depends on the current state and not on any of the states previous to the current:

(This memoryless property is formally known as the Markov property).

If the transition operator for a Markov chain does not change across transitions, the Markov chain is called **time homogenous**.  A nice property of time homogenous Markov chains is that as the chain runs for a long time and , the chain will reach an equilibrium that is called the chain’s **stationary distribution**:

We’ll see later how the stationary distribution of a Markov chain is important for sampling from probability distributions, a technique that is at the heart of Markov Chain Monte Carlo (MCMC) methods.

**Finite state-space (time homogenous) Markov chain**

If the state space of a Markov chain takes on a finite number of distinct values, and it is time homogenous, then the transition operator can be defined by a matrix , where the entries of are:

This means that if the chain is currently in the state, the transition operator assigns the probability of moving to the   state by the entries of row of (i.e.  each row of defines a conditional probability distribution on the state space). Let’s take a look at a finite state-space Markov chain in action with a simple example.

## Example: Predicting the weather with a finite state-space Markov chain

In Berkeley, CA, there are (literally) only 3 types of weather: sunny, foggy, or rainy (this is analogous to a state-space that takes on three discrete values). The weather patterns are very stable there, so a Berkeley weatherman can easily predict the weather next week based on the weather today with the following transition rules:

If it is *sunny* today, then

* it is highly likely that it will be *sunny*next week
* it is very unlikely that it will be *raining*next week
* and somewhat likely that it will *foggy*next week

If it is *foggy* today then

* it is somewhat likely that it will be *sunny*next week
* but slightly less likely that it will be *foggy*next week
* and fairly unlikely that it will be *raining*next week.

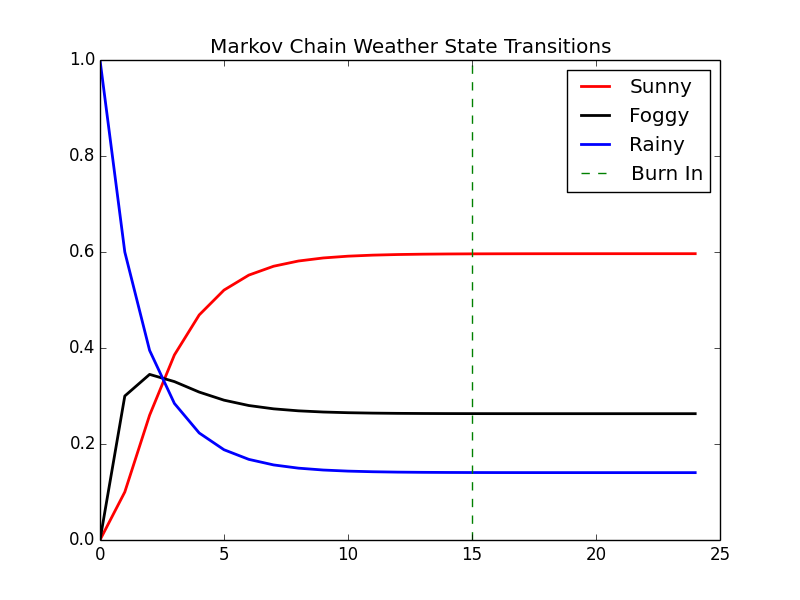
If it is *rainy* today then

* it is unlikely that it will be *sunny*next week
* it is somewhat likely that it will be *foggy*next week
* and it is fairly likely that it will be *rainy*next week

All of these transition rules can be instantiated in a single 3 x 3 transition operator matrix:

Where each row of corresponds to the weather at iteration , and each column corresponds to the weather the next week. Let’s say that it is rainy today, what is the probability it will be sunny next week, in two weeks, or in 6 months? We can answer these questions by running a Markov chain from the initial state of “rainy,” transitioning according to . The following chunk of MATLAB code runs the Markov chain.

SEE Appendix Markov Chain Finite State Transitions Code



Here we see that at week 1 the probability of sunny weather is 0.1. The next week, the probability of sunny weather is 0.26, and in 6 months, there is a 60% chance that it will be sunny. Also note that after approximately 15 weeks the Markov chain has reached the equilibrium/stationary distribution and, chances are, the weather will be sunny. This 15-week period is what is known as the **burn in** period for the Markov chain, and is the number of transitions it takes the chain to move from the initial conditions to the stationary distribution.

A cool thing about finite state-space time-homogeneous Markov chain is that it is not necessary to run the chain sequentially through all iterations in order to predict a state in the future. Instead we can predict by first raising the transition operator to the power, where is the iteration at which we want to predict, then multiplying the result by the distribution over the initial state, . For instance, to predict the probability of the weather in 2 weeks, knowing that it is rainy today (i.e. ):

np.dot(X[0,:],matpow(P,2)) = array([ 0.26 , 0.345, 0.395])

and in six months:

np.dot(X[0,:],matpow(P,24)) = array([ 0.59648855, 0.26315895, 0.1403525 ])

These are the same results we get by running the Markov chain sequentially through each number of transitions. Therefore we can calculate an approximation to the stationary distribution from by setting to a large number. It turns out that it is also possible to analytically derive the stationary distribution from (hint: think about the properties of eigenvectors).

## Continuous state-space Markov chains

A Markov chain can also have a continuous state space that exists in the real numbers . In this case the transition operator cannot be instantiated simply as a matrix, but is instead some continuous function on the real numbers. Note that the continuous state-space Markov chain also has a burn in period and a stationary distribution. However, the stationary distribution will also be over a continuous set of variables. To get a better understanding of the workings of a continuous state-space Markov chain, let’s look at a simple example.

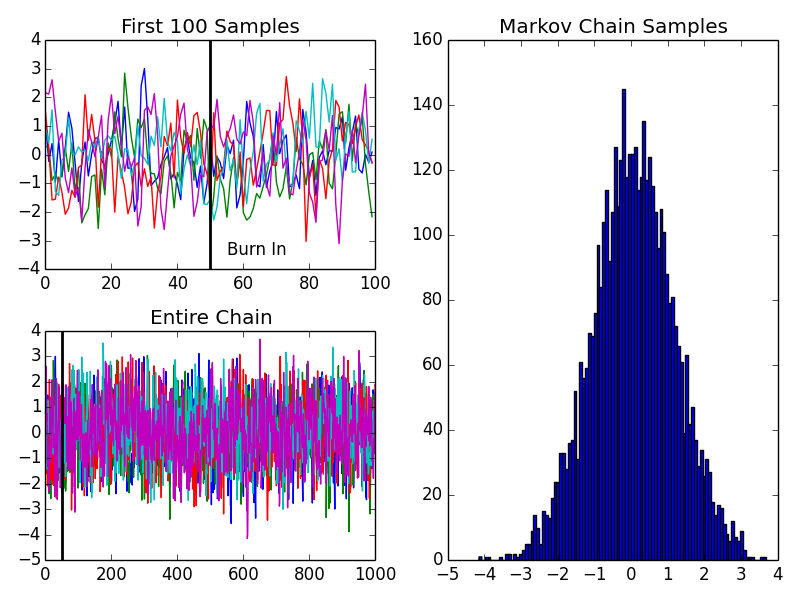
## Example: Sampling from a continuous distribution using continuous state-space Markov chains

We can use the stationary distribution of a continuous state-space Markov chain in order to sample from a continuous probability distribution: we  run a Markov chain for a sufficient amount of time so that it has reached its stationary distribution, then keep the states that the chain visits as samples from that stationary distribution.

In the following example we define a continuous state-space Markov chain. The transition operator is a Normal distribution with unit variance and a mean that is half the distance between zero and the previous state, and the distribution over initial conditions is a Normal distribution with zero mean and unit variance.

To ensure that the chain has moved sufficiently far from the initial conditions and that we are sampling  from the chain’s stationary distribution,  we will choose to throw away the first 50 burn in states of the chain. We can also run multiple chains simultaneously in order to sample the stationary distribution more densely. Here we choose to run 5 chains simultaneously.

SEE MCMC Markov Chain Continuous State Transitions Code



In the upper left panel of the code output we see a close up of the first 100 of the 1000 transitions made by the 5 simultaneous Markov chains; the burn in cutoff is marked by the black line. In the lower left panel we see the entire sequence of transitions for the Markov chains. In the right panel, we can tell from the sampled states that the stationary distribution for this chain is a Normal distribution, with mean equal to zero, and a variance equal to 1.3.

## Wrapping Up

In the previous example we were able to deduce the stationary distribution of the Markov chain by looking at the samples generated from the chain after the burn in period. However, in order to use Markov chains to sample from a specific target distribution, we have to design the transition operator such that the resulting chain reaches a stationary distribution that matches the target distribution. This is where MCMC methods like the Metropolis sampler, the Metropolis-Hastings sampler, and the Gibbs sampler come to rescue. We will discuss each of these Markov-chain-based sampling methods separately in later posts.

# MCMC: The Metropolis Sampler

<https://theclevermachine.wordpress.com/2012/10/05/mcmc-the-metropolis-sampler/>

As discussed in an earlier [post](https://theclevermachine.wordpress.com/2012/09/24/a-brief-introduction-to-markov-chains/), we can use a Markov chain to sample from some *target probability distribution* p(x)from which drawing samples directly is difficult. To do so, it is necessary to design a transition operator for the Markov chain which makes the chain’s stationary distribution match the target distribution. The Metropolis sampling algorithm  (and the more general Metropolis-Hastings sampling algorithm) uses simple heuristics to implement such a transition operator.

## Metropolis Sampling

Starting from some random initial state x^{(0)} \sim \pi^{(0)}, the algorithm first draws a possible sample x^*from a  **proposal distribution** q(x | x^{(t-1)}).  Much like a conventional transition operator for a Markov chain, the proposal distribution depends only on the previous state in the chain. However, the transition operator for the Metropolis algorithm has an additional step that assesses whether or not the target distribution has a sufficiently large density near the proposed state to warrant accepting the proposed state as a sample and setting it to the next state in the chain. If the density of p(x)is low near the proposed state, then it is likely (but not guaranteed) that it will be rejected. The criterion for accepting or rejecting a proposed state are defined by the following heuristics:

1. If p(x^*) \geq p(x^{(t-1)}),  the proposed state is kept x^*as a sample and is set as the next state in the chain (i.e. move the chain’s state to a location  where p(x)has equal or greater density).
2. If p(x^*) < p(x^{(t-1)})–indicating that p(x)has low density near x^*–then the proposed state may still be accepted, but only randomly, and with a probability \frac{p(x^*)}{p(x^{(t-1)})}

These heuristics can be instantiated by calculating the **acceptance probability** for the proposed state.

\alpha = \min \left(1, \frac{p(x^*)}{p(x^{(t-1)})}\right)

Having the acceptance probability in hand, the transition operator for the metropolis algorithm works like this: if a random uniform number uis less than or equal to \alpha, then the state x^*is accepted (as in (1) above), if not, it is rejected and another state is proposed (as in (2) above). In order to collect Msamples using  Metropolis sampling we run the following algorithm:

1. set t = 0
2. generate an initial state x^{(0)}from a prior distribution \pi^{(0)}over initial states
3. repeat until t = M

set t = t+1

generate a proposal state x^*from q(x | x^{(t-1)})

calculate the acceptance probability \alpha = \min \left(1, \frac{p(x^*)}{p(x^{(t-1)})}\right)

draw a random number ufrom \text{Unif}(0,1)

if u \leq \alpha, accept the proposal and set x^{(t)} = x^*

else  set x^{(t)} = x^{(t-1)}

## Example: Using the Metropolis algorithm to sample from an unknown distribution

Say that we have some mysterious function

p(x) = (1 + x^2)^{-1}

from which we would like to draw samples. To do so using Metropolis sampling we need to define two things: (1) the prior distribution \pi^{(0)}over the initial state of the Markov chain, and (2)  the proposal distribution q(x | x^{(t-1)}). For this example we define:

\pi^{(0)} \sim \mathcal N(0,1)

q(x | x^{(t-1)}) \sim \mathcal N(x^{(t-1)},1),

both of which are simply a Normal distribution, one centered at zero, the other centered at previous state of the chain. The following chunk of MATLAB code runs the Metropolis sampler with this proposal distribution and prior.

SEE Appendix: MCMC Metropolis Sampler

TODO: Output accept/reject examples

In the figure above, we visualize the first 50 iterations of the Metropolis sampler.The black curve represents the target distribution p(x). The red curve that is bouncing about the x-axis is the proposal distribution q(x | x^{(t-1)})(if the figure is not animated, just click on it). The vertical blue line (about which the bouncing proposal distribution is centered) represents the quantity p(x^{(t-1)}), and the vertical red line represents the quantity p(x^*), for a proposal state x^*sampled according to the red  curve. At every iteration, if the vertical red line is longer than the blue line, then the sample x^*is accepted, and the proposal distribution becomes centered about the newly accepted sample. If the blue line is longer, the sample is randomly rejected or accepted.

But why randomly keep “bad” proposal samples? It turns out that doing this allows the Markov chain to every-so-often visit states of low probability under the target distribution. This is a desirable property if we want the chain to adequately sample the entire target distribution, including any tails.

An attractive property of the Metropolis algorithm is that the target distribution p(x)does not have to be a properly normalized probability distribution. This is due to the fact that the acceptance probability is based on the ratio of two values of the target distribution. I’ll show you what I mean. If p(x)is an unnormalized distribution and

p^*(x) = \frac{p(x)}{Z}

is a properly normalized probability distribution with normalizing constant Z, then

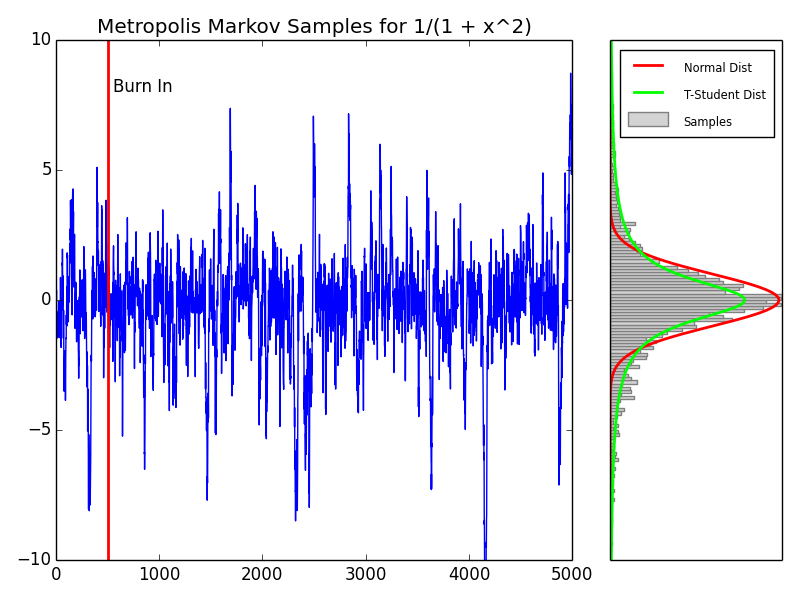
p(x) = Zp^*(x)

and a ratio like that used in calculating the acceptance probability \alphais

\frac{p(a)}{p(b)} = \frac{Zp^*(a)}{Zp^*(b)} = \frac{p^*(a)}{p^*(b)}

The normalizing constants Zcancel! This attractive property is quite useful in the context of Bayesian methods, where determining the normalizing constant for a distribution may be impractical to calculate directly. This property is demonstrated in current example. It turns out that the “mystery” distribution that we sampled from using the Metropolis algorithm is an unnormalized form of the Student’s-t distribution with one degree of freedom. Comparing p(x)to the definition of the definition Student’s-t, we see that p(x)is a Student’s-t distribution with degrees of freedom \nu=1, but missing the normalizing constant

Below is additional output from the code above showing that the samples from Metropolis sampler draws samples that follow a normalized Student’s-t distribution, even though p(x)is not normalized.



The left plot shows the progression of the Markov chain’s progression from state x^{(0)}( left) to state x^{(5000)}(right). The burn in period for this chain was chosen to be 500 transitions, and is indicated by the solid red line (~~for more on burnin see this previous~~ [~~post~~](https://theclevermachine.wordpress.com/2012/09/24/a-brief-introduction-to-markov-chains/)~~).~~

The right plot shows samples from the Markov chain in gray (with burn in samples removed). A normal distribution is overlain in red. The theoretical curve for the Student’s-t with one degree of freedom is overlain in green. We see that the states kept by the Metropolis sampler transition operator sample from values that follow the Student’s-t, even though the function p(x)used in the transition operator was not a properly normalized probability distribution.

## Reversibility of the transition operator

It turns out that there is a theoretical constraint on the Markov chain the transition operator in order for it settle into a stationary distribution (i.e. a target distribution we care about). The constraint states that the probability of the transition x^{(t)} \to x^{(t+1)}must be equal to the probability of the reverse transition x^{(t+1)} \to x^{(t)}. This reversibility property is often referred to as ***detailed balance***. Using the Metropolis algorithm transition operator, reversibility is assured if the proposal distribution q(x|x^{(t-1)})is symmetric. Such symmetric proposal distributions are the Normal, Cauchy, Student’s-t, and Uniform distributions.

However, using a symmetric proposal distribution may not be reasonable to adequately or efficiently sample all possible target distributions. For instance if a target distribution is bounded on the positive numbers 0 < x \leq \infty, we would like to use a proposal distribution that has the same support, and will thus be assymetric. This is where the **Metropolis-Hastings** sampling algorithm comes in. We will discuss in a later post how the Metropolis-Hastings sampler uses a simple change to the calculation of the acceptance probability which allows us to use non-symmetric proposal distributions.

# MCMC: The Metropolis-Hastings Sampler

<https://theclevermachine.wordpress.com/2012/10/20/mcmc-the-metropolis-hastings-sampler/>

*Note: Metropolis requires Pij = Pji (symmetric) but not the case for MH*

In an earlier [post](https://theclevermachine.wordpress.com/2012/10/05/mcmc-the-metropolis-sampler/) we discussed how the Metropolis sampling algorithm can draw samples from a complex and/or unnormalized target probability distributions using a Markov chain. The Metropolis algorithm first proposes a possible new state x^*in the Markov chain, based on a previous state x^{(t-1)}, according to the proposal distribution q(x^* | x^{(t-1)}). The algorithm accepts or rejects the proposed state based on the density of the the target distribution p(x)evaluated at x^*. (If any of this Markov-speak is gibberish to the reader, please refer to the previous posts on [Markov Chains](https://theclevermachine.wordpress.com/2012/09/24/a-brief-introduction-to-markov-chains/), MCMC, and the [Metropolis Algorithm](https://theclevermachine.wordpress.com/2012/10/05/mcmc-the-metropolis-sampler/) for some clarification).

One constraint of the Metropolis sampler is that the proposal distribution q(x^* | x^{(t-1)})must be symmetric. The constraint originates from using a Markov Chain to draw samples: a necessary condition for drawing from a Markov chain’s stationary distribution is that at any given point in time t, the probability of moving from x^{(t-1)} \rightarrow x^{(t)}must be equal to the probability of moving from x^{(t-1)} \rightarrow x^{(t)}, a condition known as ***reversibility*** or ***detailed balance***. However, a symmetric proposal distribution may be ill-fit for many problems, like when we want to sample from distributions that are bounded on semi infinite intervals (e.g. [0, \infty)).

In order to be able to use an asymmetric proposal distributions, the Metropolis-Hastings algorithm implements an additional correction factor c, defined from the proposal distribution as

c = \frac{q(x^{(t-1)} | x^*) }{q(x^* | x^{(t-1)})}

The correction factor adjusts the transition operator to ensure that the probability of moving from x^{(t-1)} \rightarrow x^{(t)}is equal to the probability of moving from x^{(t-1)} \rightarrow x^{(t)}, no matter the proposal distribution.

The Metropolis-Hastings algorithm is implemented with essentially the same procedure as the Metropolis sampler, except that the correction factor is used in the evaluation of acceptance probability \alpha.  Specifically, to draw Msamples using the Metropolis-Hastings sampler:

1. set t = 0
2. generate an initial state x^{(0)} \sim \pi^{(0)}
3. repeat until t = M

set t = t+1

generate a proposal state x^*from q(x | x^{(t-1)})

calculate the proposal correction factor c = \frac{q(x^{(t-1)} | x^*) }{q(x^*|x^{(t-1)})}

calculate the acceptance probability \alpha = \text{min} \left (1,\frac{p(x^*)}{p(x^{(t-1)})} \times c\right ) 

draw a random number ufrom \text{Unif}(0,1)

if u \leq \alphaaccept the proposal state x^*and set x^{(t)}=x^*

else set x^{(t)} = x^{(t-1)}

Many consider the Metropolis-Hastings algorithm to be a generalization of the Metropolis algorithm. This is because when the proposal distribution is symmetric, the correction factor is equal to one, giving the transition operator for the Metropolis sampler.

## Example: Sampling from a Bayesian posterior with improper prior

For a number of applications, including regression and density estimation, it is usually necessary to determine a set of parameters \thetato an assumed model p(y | \theta)such that the model can best account for some observed data y. The model function p(y | \theta)is often referred to as the likelihood function. In Bayesian methods there is often an explicit prior distribution p(\theta)that is placed on the model parameters and controls the values that the parameters can take.

The parameters are determined based on the posterior distribution p(\theta | y), which is a probability distribution over the possible parameters based on the observed data. The posterior can be determined using Bayes’ theorem:

p(\theta | y) = \frac{p(y | \theta) p(\theta)}{p(y)}

where, p(y)is a normalization constant that is often quite difficult to determine explicitly, as it involves computing sums over every possible value that the parameters and ycan take.

Let’s say that we assume the following model (likelihood function):

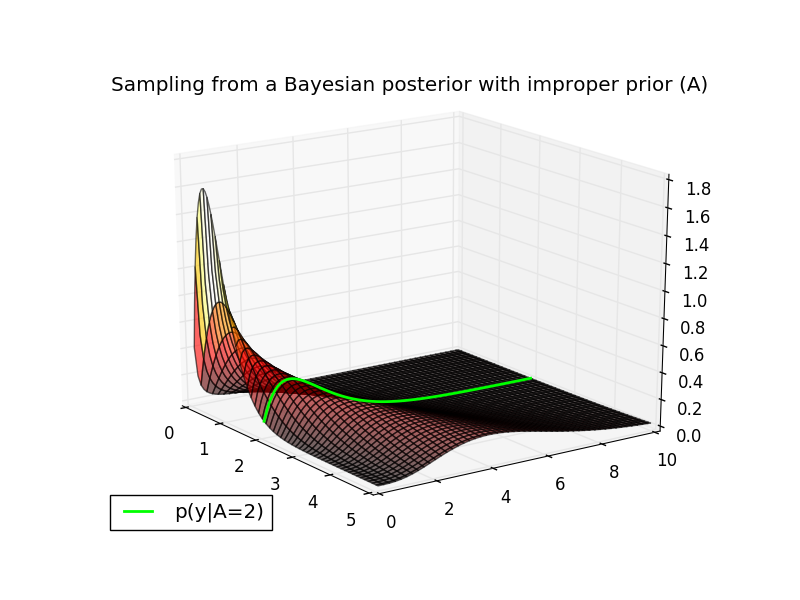
p(y | \theta) = \text{Gamma}(y;A,B), where

\text{Gamma}(y;A,B) = \frac{B^A}{\Gamma(A)} y^{A-1}e^{-By}, where

\Gamma( )is the [gamma function](http://en.wikipedia.org/wiki/Gamma_function). Thus, the model parameters are

\theta = [A,B]

The parameter Acontrols the shape of the distribution, and Bcontrols the scale. The likelihood surface for B = 1, and a number of values of Aranging from zero to five are shown below.



The conditional distribution p(y | A=2, B = 1)is plotted in green along the likelihood surface. You can verify this is a valid conditional in MATLAB with the following command:

|  |  |
| --- | --- |
| 1 | plot(0:.1:10,gampdf(0:.1:10,4,1)); % GAMMA(4,1) |

Now, let’s assume the following priors on the model parameters:

p(B = 1) = 1

and

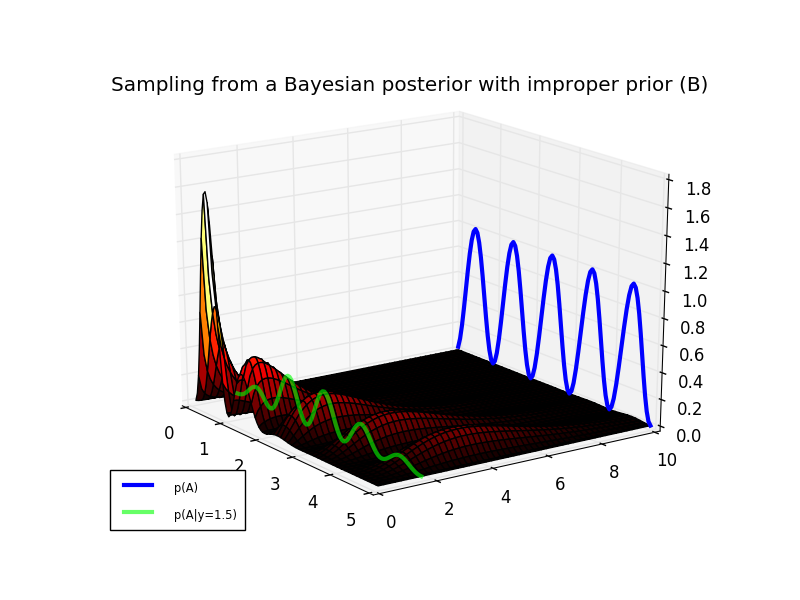
p(A) = \text{sin}(\pi A)^2

The first prior states that Bonly takes a single value (i.e. 1), therefore we can treat it as a constant. The second (rather non-conventional) prior states that the probability of Avaries as a sinusoidal function. (Note that both of these prior distributions are called ***improper priors*** because they do not integrate to one). Because Bis constant, we only need to estimate the value of A.

It turns out that even though the normalization constant p(y)may be difficult to compute, we can sample from p(A | y)without knowing p(x)using the Metropolis-Hastings algorithm. In particular, we can ignore the normalization constant p(x)and sample from the unnormalized posterior:

p(A | y) \propto p(y |A) p(A)

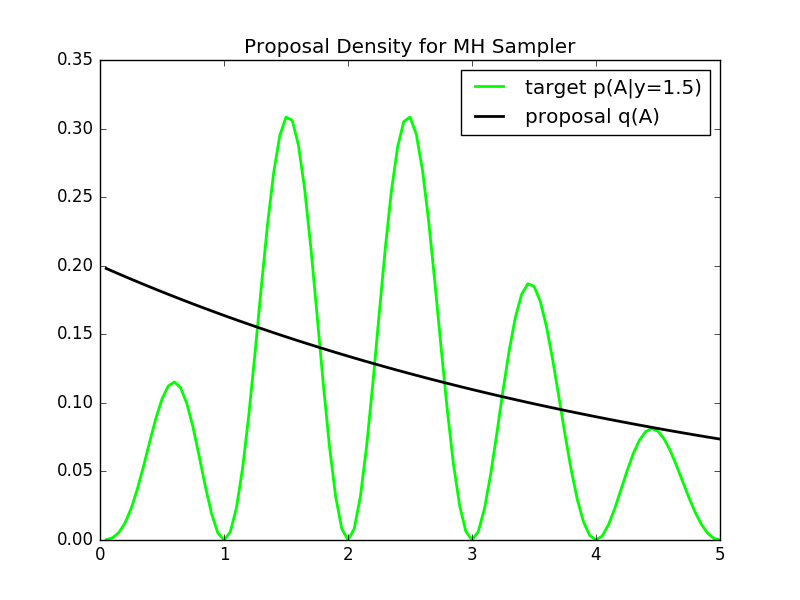
The surface of the (unnormalized) posterior for yranging from zero to ten are shown below. The prior p(A)is displayed in blue on the right of the plot. Let’s say that we have a datapoint y = 1.5and would like to estimate the posterior distribution p(A|y=1.5)using the Metropolis-Hastings algorithm. This particular target distribution is plotted in magenta in the plot below.



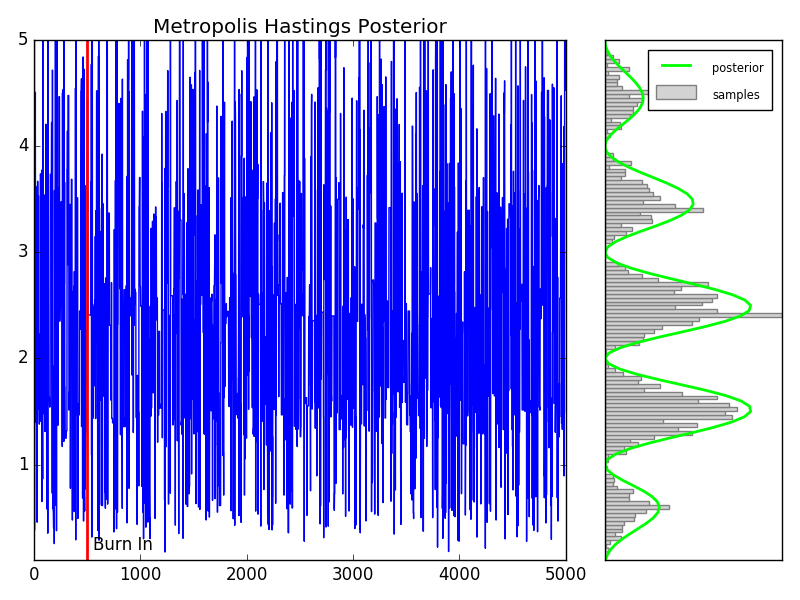
Using a symmetric proposal distribution like the Normal distribution is not efficient for sampling from p(A|y=1.5)due to the fact that the posterior only has support on the real positive numbers A \in [0 ,\infty). An asymmetric proposal distribution with the same support, would provide a better coverage of the posterior. One distribution that operates on the positive real numbers is the exponential distribution.

q(A) = \text{Exp}(\mu) = \mu e^{-\mu A},

This distribution is parameterized by a single variable \muthat controls the scale and location of the distribution probability mass. The target posterior and a proposal distribution (for \mu = 5) are shown in the plot below.



We see that the proposal has a fairly good coverage of the posterior distribution. We run the Metropolis-Hastings sampler in the block of MATLAB code at the bottom. The Markov chain path and the resulting samples are shown in plot below.



As an aside, note that the proposal distribution for this sampler does not depend on past samples, but only on the parameter \mu(see line 88 in the MATLAB code below). Each proposal states x^*is drawn independently of the previous state. Therefore this is an example of an **independence sampler**, a specific type of Metropolis-Hastings sampling algorithm. Independence samplers are notorious for being either very good or very poor sampling routines. The quality of the routine depends on the choice of the proposal distribution, and its coverage of the target distribution. Identifying such a proposal distribution is often difficult in practice.

The MATLAB  code for running the Metropolis-Hastings sampler is below. Use the copy icon in the upper right of the code block to copy it to your clipboard. Paste in a MATLAB terminal to output the figures above.

## Wrapping Up

Here we explored how the Metorpolis-Hastings sampling algorithm can be used to generalize the Metropolis algorithm in order to sample from complex (an unnormalized) probability distributions using asymmetric proposal distributions. One shortcoming of the Metropolis-Hastings algorithm is that not all of the proposed samples are accepted, wasting valuable computational resources. This becomes even more of an issue for sampling distributions in higher dimensions. This is where Gibbs sampling comes in. We’ll see in a later post that Gibbs sampling can be used to keep all proposal states in the Markov chain by taking advantage of conditional probabilities.

# Hellinger distance

## Properties

The Hellinger distance forms a [bounded](https://en.wikipedia.org/wiki/Bounded_function) [metric](https://en.wikipedia.org/wiki/Metric_%28mathematics%29) on the [space](https://en.wikipedia.org/wiki/Function_space) of probability distributions over a given [probability space](https://en.wikipedia.org/wiki/Probability_space).

The maximum distance 1 is achieved when *P* assigns probability zero to every set to which *Q* assigns a positive probability, and vice versa.

Sometimes the factor 1/2 in front of the integral is omitted, in which case the Hellinger distance ranges from zero to the square root of two.

The Hellinger distance is related to the [Bhattacharyya coefficient](https://en.wikipedia.org/wiki/Bhattacharyya_distance) BC(P,Q)as it can be defined as

H(P,Q) = \sqrt{1 - BC(P,Q)}.

Hellinger distances are used in the theory of [sequential](https://en.wikipedia.org/wiki/Sequential_analysis) and [asymptotic statistics](https://en.wikipedia.org/wiki/Asymptotic_statistics).[[4]](https://en.wikipedia.org/wiki/Hellinger_distance#cite_note-4)[[5]](https://en.wikipedia.org/wiki/Hellinger_distance#cite_note-5)

## Examples

The squared Hellinger distance between two [normal distributions](https://en.wikipedia.org/wiki/Normal_distribution) \scriptstyle P\,\sim\,\mathcal{N}(\mu_1,\sigma_1^2)and \scriptstyle Q\,\sim\,\mathcal{N}(\mu_2,\sigma_2^2)is:


  H^2(P, Q) = 1 - \sqrt{\frac{2\sigma_1\sigma_2}{\sigma_1^2+\sigma_2^2}} \,  e^{-\frac{1}{4}\frac{(\mu_1-\mu_2)^2}{\sigma_1^2+\sigma_2^2}}.
  

The squared Hellinger distance between two [exponential distributions](https://en.wikipedia.org/wiki/Exponential_distribution) \scriptstyle P\,\sim \,\rm{Exp}(\alpha)and \scriptstyle Q\,\sim\,\rm{Exp}(\beta)is:


  H^2(P, Q) = 1 - \frac{2 \sqrt{\alpha \beta}}{\alpha + \beta}.
  

The squared Hellinger distance between two [Weibull distributions](https://en.wikipedia.org/wiki/Weibull_distribution) \scriptstyle P\,\sim \,\rm{W}(k,\alpha)and \scriptstyle Q\,\sim\,\rm{W}(k,\beta)(where  k is a common shape parameter and  \alpha\, , \beta are the scale parameters respectively):


  H^2(P, Q) = 1 - \frac{2 (\alpha \beta)^{k/2}}{\alpha^k + \beta^k}.
  

The squared Hellinger distance between two [Poisson distributions](https://en.wikipedia.org/wiki/Poisson_distribution) with rate parameters \alphaand \beta, so that \scriptstyle P\,\sim \,\rm{Poisson}(\alpha)and \scriptstyle Q\,\sim\,\rm{Poisson}(\beta), is:


  H^2(P,Q) = 1-e^{-\frac{1}{2}(\sqrt{\alpha} - \sqrt{\beta})^2}.
  

The squared Hellinger distance between two [Beta distributions](https://en.wikipedia.org/wiki/Beta_distribution) \scriptstyle P\,\sim\,\text{Beta}(a_1,b_1)and \scriptstyle Q\,\sim\,\text{Beta}(a_2, b_2)is:


H^{2}(P,Q) =1-\frac{B\left(\frac{a_{1}+a_{2}}{2},\frac{b_{1}+b_{2}}{2}\right)}{\sqrt{B(a_{1},b_{1})B(a_{2},b_{2})}}
  

where Bis the [Beta function](https://en.wikipedia.org/wiki/Beta_function).

# Appendix: MCMC Monte Carlo Approximation of Integral Code

# MONTE CARLO APPROXIMATION OF INT(xexp(x))dx

# FOR TWO DIFFERENT SAMPLE SIZES

import numpy as np

np.random.seed(271828)

runs=1000

# THE FIRST APPROXIMATION USING N1 = 100 SAMPLES

N1 = 100;

x = np.random.uniform(size=N1);

I\_hat\_1 = sum(x\*np.exp(x))/N1

I\_hat\_1 # 0.95254390652501486

# A SECOND APPROXIMATION USING N2 = 5000 SAMPLES

N2 = 5000;

x = np.random.uniform(size=N2);

I\_hat\_2 = sum(x\*np.exp(x))/N2

I\_hat\_2 # 1.0209454002831784

# Estimate variance for N1=100

est=[]

for r in range(runs):

x = np.random.uniform(size=N1);

est.append(sum(x\*np.exp(x))/N1)

np.var(est) # 0.0063657238770784439

# Estimate variance for N1=5000

est=[]

for r in range(runs):

x = np.random.uniform(size=N2);

est.append(sum(x\*np.exp(x))/N2)

np.var(est) # 0.00012223069067703404

# variance decreases linearly with number of runs

0.0063657238770784439/0.00012223069067703404

# 52.079586900955817

# Appendix: MCMC Monte Carlo Beta Expectation Code

# MONTE CARLO EXPECTATION

import numpy as np

import scipy.stats as stats

import matplotlib.pyplot as plt

from matplotlib import gridspec

import pylab

np.random.seed(271828)

alpha1 = 2;

alpha2 = 10;

N = 10000;

#x = betarnd(alpha1,alpha2,1,N);

x = stats.beta.rvs(alpha1,alpha2, size=N)

# MONTE CARLO EXPECTATION

expectMC = np.mean(x);

# ANALYTIC EXPRESSION FOR BETA MEAN

expectAnalytic = 1.\*alpha1/(alpha1 + alpha2);

plt.figure(figsize=(8, 6))

plt.hist(x[:,0])

plt.show()

# DISPLAY

steps=0.02

bins = np.arange(0,1.+steps,steps)

h=np.histogram(x,bins)

counts=h[0]

probSampled = 1.\*counts/sum(counts);

probTheory = stats.beta.pdf(bins,alpha1,alpha2);

fig = plt.figure(figsize=(8, 6))

gs = gridspec.GridSpec(1, 2, width\_ratios=[3, 1])

ax0 = plt.subplot(gs[0])

ax0.bar(bins[1:],probSampled,width=steps,color='grey',label="samples")

ax0.plot(bins[1:]+0.5\*steps,probTheory[1:]/sum(probTheory[1:]),'r',linewidth=2,label="theory")

ax0.axvline(x=expectMC,color='g',linewidth=2,label="sampled expectation")

ax0.axvline(x=expectAnalytic,color='b',linewidth=2,label="theory expectation")

ax0.set\_xlim([0.02,1.00])

leg=ax0.legend(loc='upper right')

ltext = leg.get\_texts()

llines = leg.get\_lines()

frame = leg.get\_frame()plt.setp(ltext, fontsize='x-small')

fig.suptitle("Example Monte Carlo Expectation for Beta Distribution")

ax1 = plt.subplot(gs[1])

ax1.bar(bins[1:],probSampled,width=steps,color='grey')

ax1.plot(bins[1:]+0.5\*steps,probTheory[1:]/sum(probTheory[1:]),'r',linewidth=2)

ax1.axvline(x=expectMC,color='g',linewidth=2)

ax1.axvline(x=expectAnalytic,color='b',linewidth=2)

ax1.set\_xlim([0.16,0.18])

ax1.xaxis.set\_major\_locator(pylab.NullLocator())

#ax1.yaxis.set\_major\_locator(pylab.NullLocator())

# plt.show()

plt.savefig('mcmc-monte-carlo-beta-expectation.png')

# Appendix: MCMC Monte Carlo Optimization of Exponential Function Code

# MONTE CARLO OPTIMIZATION OF exp(x-4)^2

import numpy as np

import scipy.stats as stats

import matplotlib.pyplot as plt

from matplotlib import gridspec

import pylab

np.random.seed(271828)

def g(x):

return np.exp(-0.5\*(x-4)\*\*2)

# INITIALZIE

N = 100000

step=0.1

x = np.arange(0,6+step,step)

C = np.sqrt(2\*np.pi)

y = stats.norm.pdf(4,1,x)

# CALCULATE MONTE CARLO APPROXIMATION

#x = normrnd(4,1,1,N);

n = np.random.normal(4,1,size=N)

h=np.histogram(n,100)

counts=h[0]

bins=h[1]

optIdx = np.argmax(counts)

x\_hat = bins[optIdx];

# OPTIMA AND ESTIMATED OPTIMA

# ph = plot(x,g(x)/C,'r','Linewidth',3); hold on

# gh = plot(x,g(x),'b','Linewidth',2); hold on;

# oh = plot([4 4],[0,1],'k');

# hh = plot([xHat,xHat],[0,1],'g');

plt.plot(x,g(x),color='blue',linewidth=3,label="g(x)")

plt.plot(x,g(x)/C,color='red',linewidth=3,label="p(x) = g(x)/C")

plt.axvline(x\_hat, color='green',linewidth=1,label="x\_opt")

plt.axvline(4, color='black',linewidth=1,label="x\_hat")

leg=plt.legend(loc='upper left')

ltext = leg.get\_texts()

llines = leg.get\_lines()

frame = leg.get\_frame()

plt.title("Example Monte Carlo Optimization")

#plt.show()

plt.savefig('mcmc-monte-carlo-optimization-exp.png')

# Appendix: MCMC Markov Chain Finite State Transitions Code

# FINITE STATE-SPACE MARKOV CHAIN EXAMPLE

import numpy as np

import scipy.stats as stats

import matplotlib.pyplot as plt

from matplotlib import gridspec

import pylab

# TRANSITION OPERATOR

# S = Sunny

# F = Foggy

# R = Rainy

# S->S F->S R->S

# S->F F->F R->F

# S->R F->R R->R

P = np.array([[ 0.8 , 0.15, 0.05],

[ 0.4 , 0.5 , 0.1 ],

[ 0.1 , 0.3 , 0.6 ]])

nWeeks = 25

# time series of state vectors

X = np.zeros((nWeeks,3))

#INITIAL STATE IS RAINY

X[0,:] = [0,0,1];

# RUN MARKOV CHAIN

for iB in range(nWeeks-1):

X[iB+1,:] = np.dot(X[iB,:],P)

# DISPLAY

plt.plot(X[:,0],color='r',linewidth=2,label='Sunny')

plt.plot(X[:,1],color='k',linewidth=2,label='Foggy')

plt.plot(X[:,2],color='b',linewidth=2,label='Rainy')

plt.axvline(15, color='g',ls='dashed',label='Burn In')

leg=plt.legend(loc='upper right')

ltext = leg.get\_texts()

llines = leg.get\_lines()

frame = leg.get\_frame()

plt.title("Markov Chain Weather State Transitions")

#plt.show()

plt.savefig('mcmc-markov-chain-finite-state.png')

# Appendix: MCMC Markov Chain Continous State Transitions Code

# EXAMPLE OF CONTINUOUS STATE-SPACE MARKOV CHAIN

import numpy as np

import scipy.stats as stats

import matplotlib.pyplot as plt

from matplotlib import gridspec

import pylab

# INITIALIZE

np.random.seed(271828)

nBurnin = 50; # BURNIN

nChains = 5; # MARKOV CHAINS

# DEFINE TRANSITION OPERATOR

def P(x,n):

return np.random.normal(0.5\*x,1,size=n)

nTransitions = 1000;

x = np.zeros((nTransitions,nChains));

x[0,:] = np.random.normal(1,1,size=nChains);

# RUN THE CHAINS

for iT in range(nTransitions-1):

x[iT+1,:] = P(x[iT],nChains);

plt.close('all')

fig = plt.figure()

ax1 = plt.subplot(221) # Burn In

ax2 = plt.subplot(223) # Full Chains for 5 runs

ax3 = plt.subplot(122) # Histogram of Full without Burnin

ax1.plot(x[0:100,:])

ax1.axvline(50,color='k',linewidth=2,label="Burn In")

ax1.text(50+5,np.floor(np.min(x[:100,:])) + 0.5, r'Burn In')

ax1.set\_title("First 100 Samples")

ax2.plot(x[:,:])

ax2.axvline(50,color='k',linewidth=2,label="Burn In")

ax2.set\_title("Entire Chain")

h=ax3.hist(np.ndarray.flatten(x[100:,:]),bins=100)

ax3.set\_title("Markov Chain Samples")

plt.tight\_layout()

#plt.show()

plt.savefig('mcmc-markov-chain-continuous-state.png')

# Appendix: MCMC Metropolis Markov Sampler Code

# METROPOLIS SAMPLING EXAMPLE

import numpy as np

import scipy.stats as stats

import matplotlib.pyplot as plt

from matplotlib import gridspec

import pylab

np.random.seed(271828)

# DEFINE THE TARGET DISTRIBUTION

def p(x):

return 1./(1.+x\*\*2)

# INITIALIZE CONSTANTS

nSamples = 5000;

burnIn = 500;

nDisplay = 30;

sigma = 1;

minn = -20;

maxx = 20;

step=0.1

xx = np.arange(3.\*minn,3.\*maxx+step,step);

target = p(xx);

pauseDur = .8;

# INITIALZE SAMPLER

x = np.zeros((1,nSamples));

x[0,0]= np.random.normal()

# RUN SAMPLER

for t in range(nSamples-1):

# SAMPLE FROM PROPOSAL

xStar = np.random.normal(x[0,t],sigma);

proposal = stats.norm.pdf(xx,x[0,t],sigma);

# CALCULATE THE ACCEPTANCE PROBABILITY

alpha = min([1., p(xStar)/p(x[0,t])]);

# ACCEPT OR REJECT?

u = np.random.uniform()

if u < alpha:

x[0,t+1] = xStar;

str = 'Accepted';

else:

x[0,t+1] = x[0,t];

str = 'Rejected';

#end

# DISPLAY SAMPLING DYNAMICS

# to do

# DISPLAY RESULTS

# generate some data

a = np.arange(1,nSamples+1,1)

# plot it

fig = plt.figure(figsize=(8, 6))

gs = gridspec.GridSpec(1, 2, width\_ratios=[3, 1])

# DISPLAY MARKOV CHAIN

ax0 = plt.subplot(gs[0])

ax0.set\_ylim([-10,10])

ax0.plot(a, x[0,:])

ax0.axvline(x=burnIn,color='r',linewidth=2)

ax0.text(burnIn+50, 8, r'Burn In')

plt.title('Metropolis Markov Samples for 1/(1 + x^2)')

# DISPLAY SAMPLES

ax1 = plt.subplot(gs[1])

ax1.set\_ylim([-10,10])

ax1.xaxis.set\_major\_locator(pylab.NullLocator())

ax1.yaxis.set\_major\_locator(pylab.NullLocator())

h=ax1.hist(x[0,burnIn:],bins=200,orientation="horizontal",color='lightgrey',edgecolor = 'grey',label="Samples")

b=np.arange(-10,10,0.1)

n=stats.norm.pdf(b)

t=stats.t.pdf(b,1)

plt.plot(n\*nSamples/sum(n),b,color='r',linewidth=2,label="Normal Dist")

plt.plot(t\*nSamples/sum(n),b,color='lime',linewidth=2,label="T-Student Dist")

#plt.ylabel('samples')

leg=ax1.legend(loc='upper left')

ltext = leg.get\_texts()

llines = leg.get\_lines()

frame = leg.get\_frame()

plt.setp(ltext, fontsize='x-small')

plt.tight\_layout()

#plt.show()

plt.savefig('mcmc-metropolis-sampler.png')

# Appendix: MCMC Metropolis Hastings Priors and Posterior Code

# METROPOLIS-HASTINGS BAYESIAN POSTERIOR

import numpy as np

#import scipy.stats as stats

import matplotlib.pyplot as plt

from matplotlib import gridspec

import pylab

from math import gamma

# INITIALIZE

np.random.seed(271828)

# PRIOR OVER SCALE PARAMETERS

B = 1.;

# DEFINE LIKELIHOOD

# likelihood = inline('(B.^A/gamma(A)).\*y.^(A-1).\*exp(-(B.\*y))','y','A','B');

def likelihood(y,A,B):

return (B\*\*A/gamma(A))\*y\*\*(A-1.)\*np.exp(-(B\*y))

# CALCULATE AND VISUALIZE THE LIKELIHOOD SURFACE

# yy = linspace(0,10,100);

# AA = linspace(0.1,5,100);

# avoid infinite edges

yy=np.arange(0.1,10.1,0.1)

AA=np.arange(0.05,5.05,.05)

likeSurf = np.zeros((yy.size,AA.size));

for iA in range(AA.size):

likeSurf[:,iA]=likelihood(yy[:],AA[iA],B)

from mpl\_toolkits.mplot3d import axes3d

from matplotlib import cm

#PLOT

fig = plt.figure()

ax = fig.add\_subplot(111, projection='3d')

axs=ax.get\_axes()

axs.azim=-35.

axs.elev=20.

#axs.dist=

Ys, As = np.meshgrid(yy, AA)

ax.plot\_surface(As.T,Ys.T,likeSurf, rstride=2, cstride=2, alpha=0.6,cmap=cm.hot)

ax.set

# plot A=2

ax.plot(list(As[39]),list(yy), list(likeSurf[:,39]), color='lime', linewidth=2,label='p(y|A=2)')

leg=plt.legend(loc='lower left')

ltext = leg.get\_texts()

llines = leg.get\_lines()

frame = leg.get\_frame()

plt.title("Sampling from a Bayesian posterior with improper prior (A)")

#plt.show()

plt.savefig('mcmc-metropolis-hastings-improper-prior-A.png')

# DEFINE PRIOR OVER SHAPE PARAMETERS

#prior = inline('sin(pi\*A).^2','A');

def prior(A):

return np.sin(np.pi\*A)\*\*2

# DEFINE THE POSTERIOR

# p = inline('(B.^A/gamma(A)).\*y.^(A-1).\*exp(-(B.\*y)).\*sin(pi\*A).^2','y','A','B');

def p(y,A,B):

return (B\*\*A/gamma(A))\*y\*\*(A-1.)\*np.exp(-(B\*y))\*np.sin(np.pi\*A)\*\*2

# CALCULATE AND DISPLAY THE POSTERIOR SURFACE

postSurf = np.zeros(likeSurf.shape);

for iA in range(AA.size):

postSurf[:,iA]=p(yy[:],AA[iA],B)

fig = plt.figure()

ax = fig.add\_subplot(111, projection='3d')

axs=ax.get\_axes()

axs.azim=-35.

axs.elev=20.

Ys, As = np.meshgrid(yy, AA)

ax.plot\_surface(As.T,Ys.T,postSurf, rstride=2, cstride=2, alpha=1.0,cmap=cm.hot)

ax.set

# prior

ax.plot(AA,np.ones((1,AA.size)).T\*10,prior(AA),color="blue",linewidth=3,label="p(A)")

# posterior (not shadowed properly)

ax.plot(As[:,14],np.ones((1,AA.size)).T\*1.6,postSurf[14,:],color="lime",alpha=0.6,linewidth=3,label="p(A|y=1.5)")

leg=plt.legend(loc='lower left')

ltext = leg.get\_texts()

llines = leg.get\_lines()

frame = leg.get\_frame()

plt.setp(ltext, fontsize='x-small')

plt.title("Sampling from a Bayesian posterior with improper prior (B)")

#plt.show()

plt.savefig('mcmc-metropolis-hastings-improper-prior-B.png')

# INITIALIZE THE METROPOLIS-HASTINGS SAMPLER

# DEFINE PROPOSAL DENSITY

# q = inline('exppdf(x,mu)','x','mu');

def exppdf(x,a=1.0):

return 1./a \* np.exp(-x/a)

def q(x,mu):

return exppdf(x,mu)

# MEAN FOR PROPOSAL DENSITY

mu = 5.;

fig = plt.figure()

plt.plot(AA,postSurf[14,:],color='lime',linewidth=2,label='target p(A|y=1.5)')

plt.plot(AA,q(AA,mu),color='black',linewidth=2,label='proposal q(A)')

leg=plt.legend(loc='upper right')

ltext = leg.get\_texts()

llines = leg.get\_lines()

frame = leg.get\_frame()

plt.title("Proposal Density for MH Sampler")

#plt.show()

plt.savefig('mcmc-metropolis-hastings-proposal-density.png')

# DISPLAY TARGET AND PROPOSAL

# SOME CONSTANTS

nSamples = 5000;

burnIn = 500;

minn = 0.1; maxx = 5.;

# INTIIALZE SAMPLER

x = np.zeros((1 ,nSamples));

x[0,0] = mu;

t = 0;

y=1.5

# RUN METROPOLIS-HASTINGS SAMPLER

for t in range(nSamples-1):

# SAMPLE FROM PROPOSAL

xStar = np.random.exponential(mu);

# CORRECTION FACTOR

c = q(x[0,t],mu)/q(xStar,mu);

# CALCULATE THE (CORRECTED) ACCEPTANCE RATIO

alpha = np.min([1., p(y,xStar,B)/p(y,x[0,t],B)\*c]);

# ACCEPT OR REJECT?

u = np.random.rand();

if u < alpha:

x[0,t+1] = xStar;

else:

x[0,t+1] = x[0,t];

# xStar = np.random.exponential(mu); c = q(x[0,t],mu)/q(xStar,mu); p(y,xStar,B)/p(y,x[0,t],B)\*c;

# DISPLAY RESULTS

# x-axis steps (t)

step=1

a = np.arange(1,nSamples+step,step)

# plot it

fig = plt.figure(figsize=(8, 6))

gs = gridspec.GridSpec(1, 2, width\_ratios=[3, 1])

# DISPLAY MARKOV CHAIN

ax0 = plt.subplot(gs[0])

ax0.set\_ylim([minn,maxx])

ax0.plot(a, x[0,:])

ax0.axvline(x=burnIn,color='r',linewidth=2)

ax0.text(burnIn+50, 0.2, r'Burn In')

plt.title('Metropolis Hastings Posterior')

# DISPLAY SAMPLES

ax1 = plt.subplot(gs[1])

ax1.set\_ylim([minn,maxx])

ax1.xaxis.set\_major\_locator(pylab.NullLocator())

ax1.yaxis.set\_major\_locator(pylab.NullLocator())

h=ax1.hist(x[0,burnIn:],bins=200,orientation="horizontal",color='lightgrey',edgecolor = 'grey',label="samples")

#b=np.arange(-10,10,0.1)

#n=stats.norm.pdf(b)

#t=stats.t.pdf(b,1)

#plt.plot(n\*nSamples/sum(n),b,color='r',linewidth=2,label="Normal Dist")

#plt.plot(t\*nSamples/sum(n),b,color='lime',linewidth=2,label="T-Student Dist")

plt.plot(postSurf[14,:]\*(nSamples-burnIn)/sum(postSurf[14,:]),AA, color='lime', linewidth=2,label="posterior");

#plt.ylabel('samples')

leg=ax1.legend(loc='upper right')

ltext = leg.get\_texts()

llines = leg.get\_lines()

frame = leg.get\_frame()

plt.setp(ltext, fontsize='x-small')

plt.tight\_layout()

#plt.show()

plt.savefig('mcmc-metropolis-hastings-posterior.png')