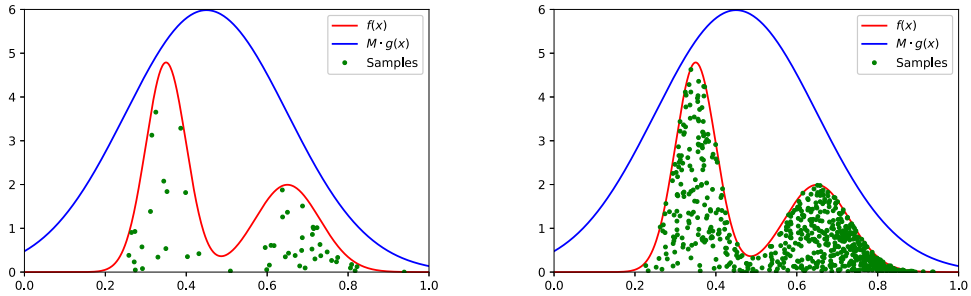


Figure 3.28: Monte Carlo results for rejection sampling of the bimodal Gaussian with $N = 10^3$ (left) and $N = 10^4$ samples (right).

```
M = 3 # The multiplication constant
N = 1000 # Number of samples
# The target probability density function
f = lambda x: 0.6 * norm.pdf(x, 0.35, 0.05) + 0.4 * norm.pdf(x, 0.65,
↪  0.08)
# The proposal probability density function
g = lambda x: norm.pdf(x, 0.45, 0.2)
# Draw N samples from the proposal
x_samples = M * np.random.normal(0.45, 0.2, (N,))
# Draw N uniform samples in the interval [0, 1]
u = np.random.uniform(0, 1, (N, ))
# Perform rejection sampling
samples = [(x_samples[i], u[i] * M * g(x_samples[i])) for i in
↪  range(N) if u[i] < f(x_samples[i]) / (M * g(x_samples[i]))]
# Plot results
fig, ax = plt.subplots(1, 1)
x = np.linspace(0, 1, 500) # x-coordinates
ax.plot(x, f(x), 'r-', label='$f(x)$') # target pdf
ax.plot(x, M * g(x), 'b-', label='$M \cdot g(x)$') # envelope pdf
# The samples found by rejection sampling
ax.plot([sample[0] for sample in samples], [sample[1] for sample in
↪  samples], 'g.', label='Samples')
plt.legend()
plt.show()
```

The results for $N = 10^3$ and $N = 10^4$ are shown in Figure 3.28. ∎

3.7.3 ▪ Markov Chain Monte Carlo (MCMC)

In contrast to Monte Carlo integration, where we need to uniformly sample a multi-dimensional domain of integration, in MCMC we want to visit a point $\mathbf{x}$ with a probability that is proportional to a desired probability density function $\pi(\mathbf{x})$. It is sufficient that the pdf be proportional to the desired target distribution, since it can always be normalized afterwards. We could, of course, obtain all the information by ordinary MC integration

over the region of interest by computing $\pi(\mathbf{x}_i)$ for every uniformly sampled point $\mathbf{x}_i$. But MCMC has the built-in capacity to automatically place its sampling points preferentially where the pdf is large, in direct proportion to it. In high-dimensional domains, or when the pdf is expensive to compute, both being prevalent in inverse parameter estimation and UQ problems, the MCMC will then have orders of magnitude less computational work to perform. MCMC works by constructing and simulating a Markov Chain whose equilibrium distribution is the distribution of interest.

Monte Carlo methods based on Markov Chains can thus be seen as a combination of sampling plus a step-wise search for large regions of high probability. This is performed in a framework that is guaranteed to produce the correct distribution in the limit, as the length of the chain increases.

There are three steps to understand the MCMC procedure:

1. Monte Carlo (see above).

2. Markov chain, $M\mu_n = \mu_{n+1}$, where M is the transition matrix, and invariant measure/distribution $M\mu = \mu$ of the process.

3. Combination of the two to generate a required measure/distribution—we want to construct a Markov chain whose invariant distribution is the desired/target distribution we seek to compute. By MCMC we will bypass the difficult problem of directly sampling from the target distribution, since we will construct a process with the desired limiting, target distribution. The famous "burn-in" process is necessary for the Markov Chain to reach its fixed point.

To understand how this is achieved, we need to briefly recall the necessary theory and definitions of Markov processes and Markov chains—see [204, 257] for fuller accounts. We begin by defining Markov chains and then explain the reasons why using a Markov chain facilitates the computation of an approximation to an otherwise intractable, high-dimensional joint probability distribution.

Markov processes are a category of stochastic processes with numerous applications in biology, physics, chemistry, economics, finance. This is the consequence of a fundamental determinism in many systems: the state of the system at a given time t_2 can be deduced from knowledge of its state at an earlier time, t_1, and does not depend on how the system arrived at t_1, i.e., on its history. A Markov Chain is a special case of this, that consists of a series of transitions between states having the property that the probability law of the future development of the chain (the next state), once we know its actual state, depends only on the state itself and not on how the process arrived in that state. We say that the future (state) is independent of the past (states), once the present (state) is known. The number of states can be either finite or countably infinite. A remarkable property of these processes, or chains, is that under certain conditions (quite often met) they will converge to an equilibrium, or stationary state, and then remain there for all time.

Example 3.50 (Convergence of a Simple Markov Chain). Consider the very simple case of a system that can be in two possible states, a and b. Suppose that the process moves from a to b with probability p, from b to a with probability q, and remains in state a or in b with probability $1-p$ or $1-q$ respectively. So the set of all possible states is $\Omega = \{a, b\}$, and the sequence of random states is $X_0, X_1, \ldots$. This model implies that the transitions between states are defined by the stochastic matrix (a matrix with positive elements less than or equal

to one, whose row sums are equal to one)

$$M = \begin{bmatrix} \mathrm{P}(a,a) & \mathrm{P}(a,b) \\ \mathrm{P}(b,a) & \mathrm{P}(b,b) \end{bmatrix} = \begin{bmatrix} 1-p & p \\ q & 1-q \end{bmatrix}$$

and that that $(X_0, X_1, \ldots)$ is a Markov chain with transition matrix M. Note that the first row of M is the conditional probability distribution (of a Bernoulli random variable—see Section 2.3.3) of X_{t+1} given $X_t = a$, and the second row is the conditional distribution of X_{t+1} given $X_t = b$. In other words,

$$\mathrm{P}\{X_1 = a \mid X_0 = a\} = 1 - p, \quad \mathrm{P}\{X_1 = b \mid X_0 = a\} = p;$$

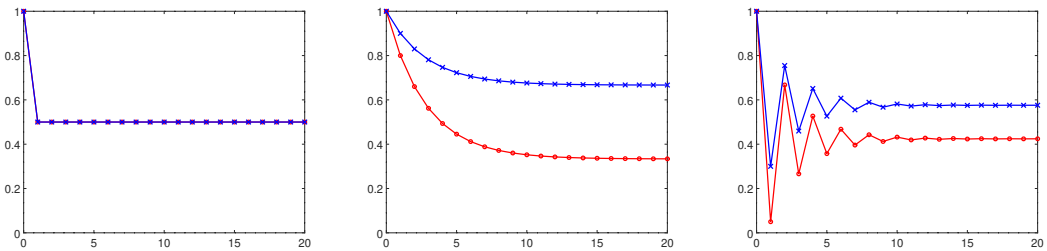
then in the next step we will have

$$\mathrm{P}\{X_2 = a \mid X_0 = a\} = (1-p)^2 + pq, \quad \mathrm{P}\{X_2 = b \mid X_0 = a\} = (1-p)p + p(1-q),$$

and so on. In fact, we see that the state transition probabilities at step n are given by

$$\mu_n = \mu_0 M^n,$$

where μ_0 is the initial distribution of the states, with $\mu_0 = [1\ 0]$ if $X_0 = a$ and $\mu_0 = [0\ 1]$ if $X_0 = b$. In Figure 3.29 we plot the $(1,1)$ and $(2,2)$ components of the matrix M as a function of the time step for the two initial states, $X_0 = a$ and $X_0 = b$, respectively, and with different values of p and q. We observe that the Markov chain converges to a stationary distribution in each case, and that the sum of the asymptotic values is equal to one. This is good news, since the stationary state must be either a or b and the total probability of the event (a or b) must sum to one. Note that we could start from any initial state, with arbitrary probability, since the initial state manifestly plays no role at all in the asymptotic. This will be very important for MCMC sampling. ∎



(a) $p = q = 0.5$ (b) $p = 0.2, q = 0.1$ (c) $p = 0.95, q = 0.7$

Figure 3.29: Convergence plots of the evolution of probabilities of a 2-state Markov Chain. Probability $\mathrm{P}\{X_t = a \mid X_0 = a\}$ in red, $\mathrm{P}\{X_t = b \mid X_0 = b\}$ in blue.

Definition 3.51 (Markov Chain).

1. *A Markov process $\{X_t\}$ is a stochastic process for which, given the value of X_t (the present), the values of X_s for any $s > t$ (the future) are independent of X_u for $u < t$ (the past).*

2. *A sequence of random variables $X_1, \ldots, X_t$ that can take its values in a discrete state space Ω is a* Markov Chain *if*

$$\mathrm{P}(X_{t+1} = x_{t+1} \mid X_0 = x_0, \ldots, X_t = x_t) = \mathrm{P}(X_{t+1} = x_{t+1} \mid X_t = x_t),$$

where the integer $t > 0$. If $\mathbf{X}_t \in \mathbb{R}^n$ is a random vector, then the right-hand side can be expressed as an $(n \times n)$ stochastic matrix, the transition matrix*, that we will denote by M. In general, the probability density function,*

$$p_{j,k}(r, s) = \mathrm{P}(X_s = k \mid X_r = j),$$

is called the transition probability function of the Markov Chain since it expresses the probability of moving from a given state r to a new state s.

3. *A time* homogeneous*, or stationary Markov chain has stationary transition probabilities that depend only on the difference $s - r$, and not on the time step itself, implying that*

$$p_{j,k}(s) = \mathrm{P}(X_{t+m} = k \mid X_t = j)$$

for any integer $m \geq 0$. As a result, we have

$$\mathrm{P}(X_{t+1} = x_{t+1} \mid X_t = x_t) = M, \ldots, \mathrm{P}(X_{t+1} = x_{t+1} \mid X_0 = x_0) = M^{t+1}.$$

4. *An* ergodic *Markov Chain is one that is both*

   - *irreducible, meaning that all transition probabilities are positive (we can get from any state to any other state), and*
   - *aperiodic, meaning the chain does not enter an endless cycle that repeatedly returns to its starting point.*

   The Law of Large Numbers can then be applied to the transition probabilities of such a chain.

5. *A distribution π is a* stationary distribution *of a Markov chain, with transition matrix M, if*

$$\pi M = \pi,$$

or

$$\pi(\mathbf{y}) = \sum_{\mathbf{x} \in \Omega} \mathrm{P}(\mathbf{y} \mid \mathbf{x}) \pi(\mathbf{x}).$$

Since M is a matrix, this is equivalent to saying that $\lambda = 1$ is an eigenvalue of M corresponding to the (left) eigenvector π. Note that π must be a row vector, due to the structure of the stochastic matrix whose rows sum to one. Clearly, if a Markov Chain has an initial distribution $\mu^{(0)} = \pi$, then it will remain at π for all time, that is $\mu^{(t)} = \pi$ for all $t > 0$.

We now have a theorem that guarantees the existence of a stationary distribution.

Theorem 3.52 (Existence of Stationary Distribution). *A finite, ergodic Markov Chain has a unique stationary distribution.*

Now, we require the notion of a *detailed balance equation* defined by

$$\mathrm{P}(x \mid y)\pi(y) = \mathrm{P}(y \mid x)\pi(x) \tag{3.102}$$

or in terms of the transition matrix, M,

$$\pi(x)M(x,y) = \pi(y)M(y,x) \tag{3.103}$$

for all x, $y \in \Omega$, which is a necessary condition for π to be stationary. A Markov Chain that satisfies the balance equation is said to be *reversible*. Then, if π satisfies this balance equation, it is precisely the stationary distribution of the MC, since

$$\begin{aligned}\sum_{y\in\Omega} \pi(y)M(y,x) &= \sum_{y\in\Omega} \pi(x)M(x,y) \\ &= \pi(x)\sum_{y\in\Omega} M(x,y) \\ &= \pi(x),\end{aligned}$$

where we have used the notation of the transition matrix M and the law of total probability that defines this stochastic matrix. Most Markov Chains encountered in MCMC are reversible, are derived from reversible components, or have reversible versions. Then this condition, expressed by the detailed balance equation, can be used to rigorously prove that the sampling methods below indeed produce an invariant, stationary, equilibrium distribution.

Convergence itself is based on the two notions of *total variation distance* and *mixing times*. Note that in implementations (see examples below) it is always necessary to check the convergence.

Definition 3.53 (Total Variation Distance). *The total variation distance between two probability distributions, μ and ν, is*

$$\begin{aligned}\|\mu - \nu\|_{\mathrm{TV}} &= \max_{A\subset\Omega} |\mu(A) - \nu(A)| \\ &= \frac{1}{2}\sum_{x\in\Omega} |\mu(x) - \nu(x)| \,.\end{aligned}$$

We can, finally, state the required convergence theorem.

Theorem 3.54 (Fundamental Theorem of Markov Chains). *Suppose that M is an ergodic Markov chain with equilibrium distribution π. Then, for all x, y*

$$\lim_{t\to\infty} M^t(x,y) = \pi(y).$$

In particular, for any $\epsilon > 0$ there exists a $t > 0$ such that

$$\left\|M^t(x,\cdot) - \pi\right\|_{\mathrm{TV}} \leq \epsilon.$$

Alternatively, there exist constants α and C, with $0 < \alpha < 1$ and $C > 0$, such that

$$\max_{x\in\Omega} \left\|M^t(x,\cdot) - \pi\right\|_{\mathrm{TV}} \leq C\alpha^t.$$

How long must we wait to attain a given precision?

Definition 3.55 (Mixing Time). *The mixing time $\tau_{\text{mix}}(\epsilon)$ is the time until the total variation distance to π is less than ϵ,*

$$\tau_{\text{mix}}(\epsilon) = \max_{x \in \Omega} \min \left\{ t \colon \left\| M^t(x, \cdot) - \pi \right\| \leq \epsilon \right\} \approx \ln \epsilon^{-1}.$$

A very comprehensive, recent account of mixing and convergence can be found in [165].

We recall that this approach of sampling from a given probability distribution is called Markov Chain Monte Carlo. Suppose that π is a probability distribution on Ω. If we can somehow construct a Markov chain (X_t) with stationary distribution π, then for a time t large enough, the distribution of X_t is guaranteed to be close to π. We now present the two principal algorithms for doing this: the Metropolis-Hasting (MH) algorithm and the Gibbs Sampling Method. They are widely-used as the building blocks for the methods found in the best MCMC software available, such as BUGS,[28] STAN,[29] PyMC,[30] etc. For this reason, it is important to understand exactly how they work, how they are programmed, and how they converge.

The principle on which all these algorithms are based is the following. We want to sample from a complicated, unknown distribution $p(x)$ that can be written as

$$p(x) = \frac{1}{Z_p} \tilde{p}(x),$$

where $\tilde{p}(x)$ can be easily evaluated for all x, and Z_p is the unknown, intractable part. To perform the sampling, we will employ the same three steps as for importance/rejection sampling:

1. Choose a *proposal* distribution, $q(x \mid x_t)$.
2. Sample a *candidate* point y from the proposal.
3. Choose a new point, x_{t+1}, according to an *acceptance* probability ratio, r, that depends on q and p.

In the acceptance ratio the hard, intractable part, Z_p, cancels. In MCMC, we choose the new point to ensure that we generate a Markov Chain that will converge to a stationary distribution π that is then a good approximation to *any* target distribution p. The good news is that this is valid universally, not only for all p, but also for any choice of the proposal q, though in practice we will tune q to improve convergence.

Metropolis-Hastings Algorithm There are several variants of the original Metropolis approach, but they all have the following basic structure.

- fix a tractable distribution $\tilde{p}(x)$ and choose a proposal distribution $q(x)$

[28] https://www.mrc-bsu.cam.ac.uk/software/bugs/
[29] https://mc-stan.org/
[30] https://docs.pymc.io/

Algorithm 3.7 Metropolis-Hasting

Given: an initial point, x_0, a tractable version of the target distribution, $\tilde{p}(x)$, and a proposal distribution, $q(x)$.
Compute: the stationary distribution $\pi(x)$ by generating an ergodic Markov chain.

1: $i = 0$, draw the starting point x_0 from $\tilde{p}(x)$ ▷ Initialize
2: **for** $i = 1$ to N **do** ▷ Loop to generate the chain
3: draw x_* from q ▷ Sample a candidate point
4: compute $\alpha_i = \min(1, r)$ ▷ Evaluate Acceptance ratio probability
5: draw $u \sim \mathcal{U}(0, 1)$ ▷ Generate a uniform random variable
6: **if** $u \leq \alpha_i$ **then** ▷ Chain update:
7: $x_i = x_*$ ▷ Next point is the candidate
8: **else**
9: $x_i = x_{i-1}$ ▷ Keep current point
10: **end if**
11: **end for**
12: Checkpoint for convergence ▷ Eventually, return to line 2
13: Output the stationary distribution.

- for each t
  - sample a candidate point y from the proposal distribution q
  - accept y with probability $\alpha = \min(1, r)$
- next t
- check convergence and output if satisfied

The most general, Metropolis-Hasting algorithm uses the acceptance ratio

$$r_{\text{MH}} = \frac{p(y)}{p(x)} \frac{q(x \mid y)}{q(y \mid x)}, \tag{3.104}$$

or in terms of the transition matrix

$$r_{\text{MH}} = \frac{p(y)}{p(x)} \frac{M(y, x)}{M(x, y)}. \tag{3.105}$$

We construct an ergodic Markov chain as detailed in Algorithm 3.7. To prove that the MH algorithm indeed converges to the stationary distribution $p(x)$, we just have to show that the detailed balance equation (3.102) or (3.103) is satisfied. Indeed, we find

$$\begin{aligned}
\underbrace{\alpha(y \mid x) q(y \mid x) p(x)}_{\mathrm{P}(y|x)} &= \min \left[\frac{p(y)}{p(x)} \frac{q(x \mid y)}{q(y \mid x)}, 1 \right] q(y \mid x) p(x) \\
&= \min \left[q(x \mid y) p(y), q(y \mid x) p(x) \right] \\
&= \min \left[1, \frac{p(x)}{p(y)} \frac{q(y \mid x)}{q(x \mid y)} \right] q(x \mid y) p(y) \\
&= \underbrace{\alpha(x \mid y) q(x \mid y) p(y)}_{\mathrm{P}(x|y)},
\end{aligned}$$

where we can consider $x = x_t$ and $y = x_{t+1}$.

There are other ways of performing the sampling in MH that are special cases of the general algorithm. The first is called Metropolis sampling and assumes that the proposal distribution is symmetric,

$$q(y \mid x) = q(x \mid y),$$

so that the acceptance ratio becomes just

$$r_{\mathrm{M}} = \frac{p(y)}{p(x)}. \tag{3.106}$$

This is not a real saving, since we still require and use q for selecting the candidate point.

A second, robust sampler for MH is based on a simple random walk approach,

$$X_{t+1} = X_t + \epsilon_t,$$

where $\epsilon_t \sim \mathcal{N}(0, 1)$. In this case the samples are i.i.d. and

$$q(x, y) = q(x - y).$$

We can thus use a (multivariate) normal distribution for the proposal q, without any loss of generality. The only parameter to be chosen, in this case, is the variance σ^2 of q, or its covariance matrix Σ in the multivariate case.

Two major questions must still be addressed:

1. How do we choose the proposal distribution q?
2. How do we determine whether we have reached the stationary distribution π?

These questions will be addressed in the examples below.

Gibbs Sampler This is a special case of the more general Metropolis-Hasting Algorithm particularly adapted to computing (or sampling from) high-dimensional multivariate distributions, since it proceeds by sequentially sampling from univariate conditional distributions, often available in explicit forms.

The Gibbs algorithm for MCMC sampling produces a Markov chain from a joint distribution of interest (the target) that must have a special form. Fixing all but one variable, the joint pdf must be of the same type and easy to simulate pseudo-random variables from it. The Gibbs algorithm then cycles through the coordinate variables, simulating each one in turn, conditional on all the others. As before, the algorithm requires a burn-in period for the Markov chain to converge sufficiently closely to its stationary distribution. To evaluate the convergence and the standard errors of the simulated values, one must run several independent Markov chains simultaneously.

Gibbs should not be used in general as it often exhibits very slow convergence. However, its simplicity compared to MH is often a determining factor. In any case, it should be used only if it seems to work well.

Other Algorithms

- Gelman's NUTS, based on Hamiltonian Monte Carlo (HMC) overcomes the slow mixing and improves the convergence of the Gibbs Sampler for more complex models. This is the method implemented in STAN.[31]

[31] https://mc-stan.org/

- Combinations of MH and Gibbs used as building blocks [86].
- Ensemble approach, as implemented in the `emcee` package.[32]

Mixing and Convergence The values of x that are generated by a Markov chain are statistically dependent. The higher the correlation, the longer it takes for the Markov chain to reach a stationary state. This implies, of course, the need for more iterations to obtain a good approximation of the posterior, or other quantities of interest. To obtain a Markov chain with low correlation, we require a proposal variance that is

- large enough to allow the chain to rapidly move around and explore the parameter space,
- but not so large that the proposals are rejected most of the time, thus slowing down the convergence.

In Gibbs sampling we cannot control the correlation, but in Metropolis, the correlation depends on the value of the width parameter δ in the proposal distribution. A careful selection of this parameter—usually by searching over a range of values—can decrease the correlation, increase the convergence rate and increase the effective sample size of the Markov chain. The Monte Carlo approximation to the posterior distribution, or any other quantity of interest, will thus also be improved.

An independent Monte Carlo sampler is perfectly mixing, since its autocorrelation is zero. This is not the case for MCMC, which will be poorly mixing, so we must seek to minimize this autocorrelation otherwise the chain will not explore all the parameter space in a reasonable time. To measure the amount of correlation in a chain, we compute the sample autocorrelation function. The lag-t autocorrelation function estimates the correlation between elements in the chain that are exactly t steps apart. For a sequence of values $\{\phi_1, \ldots, \phi_N\}$ the autocorrelation function is

$$a_t(\phi) = \frac{\frac{1}{N-t}\sum_{n=1}^{N-t}(\phi_n - \bar{\phi})(\phi_{n+t} - \bar{\phi})}{\frac{1}{N-1}\sum_{n=1}^{N}(\phi_n - \bar{\phi})^2}.$$

The higher the autocorrelation, the more MCMC samples are required to reach a given level of precision. The effective sample size is one way to estimate the size of the chain needed.

In the examples that follow, given the importance of MCMC, we will cycle through the use of R, MATLAB/Octave and Python. The full understanding of the scripts below will facilitate the use of the more sophisticated MCMC packages PyMC3, STAN, and BUGS, which otherwise risk remaining just black boxes.

Example 3.56 (MH for a Known Gaussian Distribution). [123] This instructive example will be in two parts:

1. We will use the Metropolis-Hasting algorithm to compute the posterior distribution of the mean of a Gaussian distribution with known variance, from a sequence of given observations.
2. We will study the influence of the proposal distribution on the convergence of the Markov Chain generated in the first part.

[32] https://emcee.readthedocs.io/

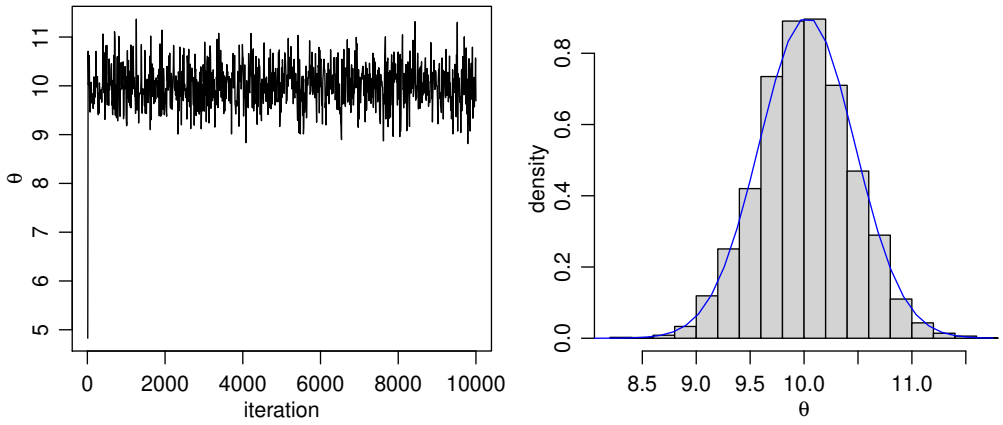


Figure 3.30: Metropolis-Hasting MCMC for a (known) Gaussian distribution. The resulting Markov chain (left) and a comparison between the computed posterior (histogram) and its exact value (solid line). Results computed by `mcMH.R`

In the first part, we will start with a proposal that is far from the reality, and the likelihood function will also be very approximate. In the second part, we will investigate more closely the behavior of the chain as the proposal distribution changes.

Recall Bayes' Law, introduced in Section 2.5 and extensively studied in Chapters 8, 9 and 11 in the context of inverse problems, data assimilation and uncertainty quantification. The law gives an expression for the posterior conditional probability of a parameter θ

$$p(\theta \mid y) = \frac{p(\theta)p(y \mid \theta)}{\int p(\theta')p(y \mid \theta')\,\mathrm{d}\theta'},$$

where y is the observation, $p(\theta)$ is the prior probability, $p(y \mid \theta)$ is the likelihood function and the denominator is a normalization factor representing the total probability of y. Very often, in practical applications, this denominator is intractable (impossible, or too expensive to compute), whereas the likelihood and prior are known. This is an ideal instance for MCMC, since we can simulate the posterior without having to know the normalizing factor.

We consider a simple problem, where we have a sequence of 5 measurements (or samples), $\{y_1, \ldots, y_5\}$, depending on a parameter θ for which we would like to obtain the probability distribution.

For the Metropolis-Hasting algorithm, we will use the Metropolis acceptance ratio (3.106), that is

$$r = \frac{p\left(\theta^* \mid y\right)}{p\left(\theta^{(t)} \mid y\right)} = \frac{p\left(y \mid \theta^*\right) p(\theta^*)}{p\left(y \mid \theta^{(t)}\right) p(\theta^{(t)})},$$

where θ^* is a candidate value for the chain drawn from the symmetric Gaussian proposal distribution $\mathcal{N}(0, \delta^2)$, the actual value in the chain is denoted $\theta^{(t)}$, the known prior is $\mathcal{N}(\mu, \tau^2)$, and the likelihood is computed assuming that $y_i \sim \mathcal{N}(\theta, \sigma^2)$. The exact expressions for the sample mean and variance are

$$\mu_n = \frac{(n/\sigma^2)\bar{y} + (1/\tau^2)\mu}{n/\sigma^2 + 1/\tau^2}$$

and

$$\tau_n^2 = \frac{1}{n/\sigma^2 + 1/\tau^2},$$

from Exercise 11.10—see also Section 11.5.

Results are plotted in Figure 3.30. We observe the following:

1. The chain converges very rapidly from the very approximate initial value, $\theta^{(0)} = 5$, and the burn-in period is very short.

2. The sampled posterior computed by the MH algorithm gives an excellent approximation to the exact expression, $p(\theta \mid y) \sim \mathcal{N}(\mu_n, \tau_n^2) = \mathcal{N}(10.03, 0.44)$.

■

```
#--- mcMH.R ---#
##### ---- Parameters and data
sig2 <- 1 ;          # likelihood variance
tau2 <- 10 ; mu <- 5 # Gaussian prior
theta<-0 ; delta<-2  # Gaussian proposal
N<-10000 ;           # length of chain
# observations
y<-c(9.37, 10.18, 9.16, 11.60, 10.33)
# initialize
THETA<-NULL ; set.seed(1)
#### ---- Metropolis algorithm
for(n in 1:N)
{
        theta.star <- rnorm(1,theta,sqrt(delta)) # candidate
        log.r <-( sum(dnorm(y,theta.star,sqrt(sig2),log=TRUE)) +
        dnorm(theta.star,mu,sqrt(tau2),log=TRUE) ) -
        ( sum(dnorm(y,theta,sqrt(sig2),log=TRUE)) +
        dnorm(theta,mu,sqrt(tau2),log=TRUE) )
        # acceptance test
        if(log(runif(1))<log.r) { theta<-theta.star }
        THETA<-c(THETA,theta) # update chain
}
##### ---- Output
# compute exact sample posterior
n      <-  5
y      <- round(rnorm(n,10,1),2)
mu.n   <- ( mean(y)*n/sig2 + mu/tau2 )/( n/sig2+1/tau2)
tau2.n <- 1/(n/sig2+1/tau2)
# set up
par(mfrow=c(1,2))
Nkeep<-seq(10,N,by=10)
# plot Markov Chain
plot(Nkeep,THETA[Nkeep],type="l",xlab="iteration",ylab=expression(theta))
# plot histogram of posterior distribution
hist(THETA[-(1:50)],prob=TRUE,main="",xlab=expression(theta),ylab="density")
th<-seq(min(THETA),max(THETA),length=100)
```

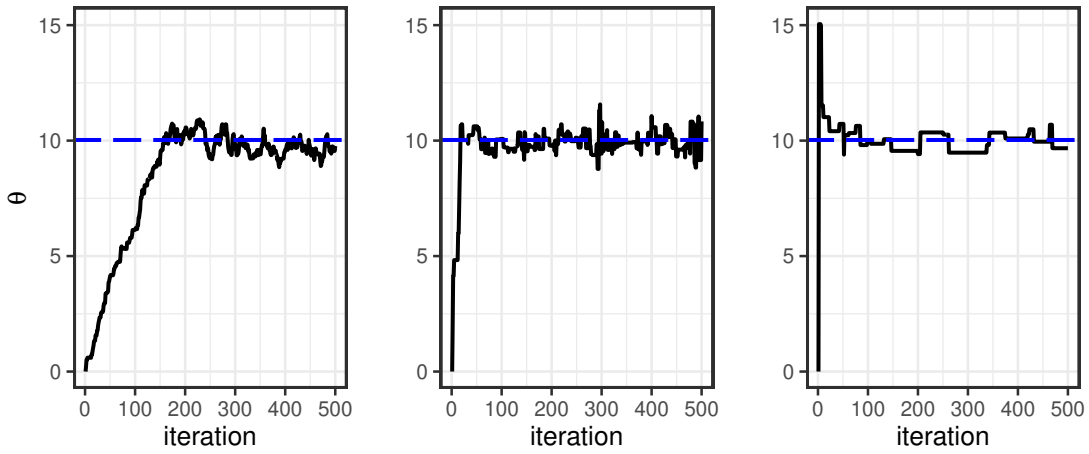


Figure 3.31: Convergence of Markov chains for different proposal distributions with values of $\delta^2 = 1/32$ (left), $\delta^2 = 2$ (center), and $\delta^2 = 64$ (right). Results were computed by `mcMH.R`

```
37 lines(th,dnorm(th,mu.n,sqrt(tau2.n)) )
```

CodeNotes

The code is quite self-explanatory. All the parameters are defined in lines 3 to 6.

- Line 8: the observations are realizations of the Gaussian distribution for which we seek the posterior law.
- Line 10: the vector `THETA` will accumulate and store the Markov chain.
- Line 14: the parameter `delta` is the important one for the mixing and convergence of the chain.
- Lines 25–28: use the exact expressions for the sample mean and variance of the Gaussian variable `y`.
- Line 31: this is just a subsampling to obtain clearer graphics.
- Line 36: we remove the burn-in period of the Markov chain, before plotting the histogram.

Example 3.57 (MH for Gaussian—Convergence Study). We pursue the previous example with a study of how the choice of the parameter δ in the proposal distribution affects the mixing, and hence the convergence of the simulated Markov chain. We will consider a sequence of values

$$\delta^2 \in \left\{ \frac{1}{32}, \frac{1}{2}, 2, 32, 64 \right\}.$$

The only change needed in the code above is the modification of `delta` in line 5. For diagnostic purposes, we also compute the autocorrelation function. The lag-1 autocorrelations,

computed with the command `acf`, for the above values of δ are

$$(0.98,\ 0.77,\ 0.69,\ 0.84,\ 0.86)$$

and the optimal value is obtained for $\delta^2 = 2$. This can be understood from the results for $\delta^2 = 1/32,\ 2$, and 64 plotted in Figure 3.31. We observe that,

1. For a small proposal variance, $\delta^2 = 1/32$ (left plot), the candidate value θ^* will be close to $\theta^{(t)}$ which implies that r will be approximately equal to one for most of the proposed values. It turns out that θ^* is accepted as $\theta^{(t+1)}$ in 87% of the iterations. This high acceptance rate guarantees that the chain moves, but these moves are always small and the convergence of the Markov chain is relatively slow.

2. For a large proposal variance, $\delta^2 = 64$ (right plot), the chain moves rapidly to the mode, but remains stuck at a constant value over long periods. Since the variance is large, the value of θ^* is mostly far from the posterior mode and proposals are only accepted in 5% of the iterations, so $\theta^{(t+1)}$ is set equal to $\theta^{(t)}$ in 95% of the iterations. This results in a highly correlated Markov chain.

3. The middle value, $\delta^2 = 2$, is the sweet spot here—we have rapid convergence, minimal autocorrelation, and good mixing.

In Chapter 11 we will present an example of the use of MCMC for Bayesian estimation of a linear regression problem. Probabilistic programming, described in Section 11.11, will be used.

Given the importance of MCMC, here is an example in Python where we simulate the Markov chain and then compare with a direct uniform sampling approach. As above, the MCMC part can easily be executed using a probabilistic programming approach, but this can turn into a black box method, which should be avoided.

Example 3.58 (MCMC for a Gaussian Posterior Using Python). A very didactic version of this example, entitled "MCMC Sampling for Dummies," can be found at https://twiecki.io/. Any reader who is completely new to MCMC will benefit from studying this presentation that concretely shows the MCMC in action, step-by-step.

Recall that we want to use a Markov chain to perform an improved Monte Carlo simulation. We can do this thanks to the aforementioned theorems that prove the convergence of the generated chain, under conditions of ergodicity and reversibility, to a stationary distribution. By design, this stationary distribution is the posterior distribution that we are trying to sample. The secret is that the MCMC performs better exploitation of zones with higher likelihood. This is achieved by using a proposal distribution q together with an acceptance ratio r that together ensure that the chain explores regions of higher posterior probability more frequently than zones of lower posterior probability.

First, we import the necessary modules.

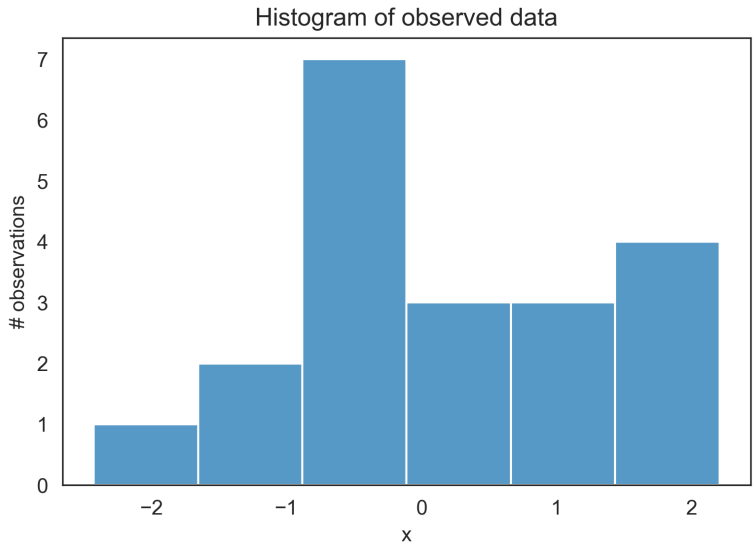
[1]:
```
%matplotlib inline

import numpy as np
import scipy as sp
import pandas as pd
```

```
import matplotlib.pyplot as plt
import seaborn as sns

from scipy.stats import norm

sns.set_style('white') # no gridlines
sns.set_context('talk')# thicker lines

np.random.seed(123)
```

To generate the data, we sample 20 points from a normal distribution, centered at zero. Our goal will be to estimate the posterior of the mean `mu` assuming that we know the standard deviation, equal to one.

[2]:
```
data = np.random.randn(20)
```

[3]:
```
ax = plt.subplot()
sns.histplot(data, kde=False, ax=ax)
_ = ax.set(title='Histogram of observed data', xlabel='x', ylabel='#␣
↪observations');
```



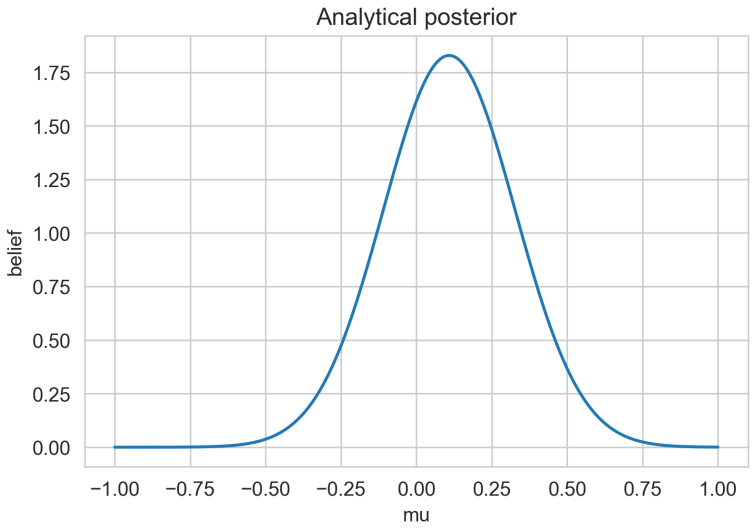
Model definition In this simple case, we will assume that the data are normally distributed, i.e., the likelihood of the model is normal. For simplicity, we have assumed $\sigma = 1$ is known and we infer the posterior for μ. For each parameter we want to infer, we have to choose a prior. For simplicity, we also take a normal distribution as the prior for μ. Thus, our model is

$$\mu \sim \mathcal{N}(0, 1), \qquad x|\mu \sim \mathcal{N}(x; \mu, 1).$$

For this model, we actually can compute the posterior analytically, since, for a normal likelihood with known standard deviation, the normal prior for `mu` is conjugate—this will be

explained in Chapter 11—and the posterior then has the same distribution as the prior, with suitably modified expectation and variance.

[4]:
```
def calc_posterior_analytical(data, x, mu_0, sigma_0):
    sigma = 1.
    n = len(data)
    mu_post = (mu_0/sigma_0**2 + data.sum()/sigma**2)/(1./sigma_0**2 + n/
↪sigma**2)
    sigma_post=(1./sigma_0**2 + n/sigma**2)**-1
    return norm(mu_post, np.sqrt(sigma_post)).pdf(x)

sns.set_style('whitegrid')
ax = plt.subplot()
x = np.linspace(-1, 1, 500)
posterior_analytical = calc_posterior_analytical(data, x, 0., 1.)
ax.plot(x, posterior_analytical)
ax.set(xlabel='mu', ylabel='belief', title='Analytical posterior');
```



This shows our quantity of interest, the probability of μ's values after having seen the data, taking our prior information into account. Now we assume, however, that our prior was not conjugate and we could not solve this analytically, which is usually the case. In these cases, we naturally turn to MCMC sampling.

Explanation of MCMC Sampling with Code We demonstrate the MCMC sampling logic. At first, we need a starting parameter position that can be randomly chosen, but we fix it arbitrarily to say

```
mu_current = 0.5
```

Then, we propose to move (jump) from that position somewhere else—this is the Markov part. Different strategies are possible for the proposal. The Metropolis sampler is the simplest

and just takes a sample from a normal distribution, having no relationship to the normal distribution we assumed for the model, centered around the current `mu` value (i.e., `mu_current`) with a certain standard deviation, known as the `proposal_width`, that will determine how far the proposal jumps. For this Brownian step, we use `scipy.stats.norm`.

```
proposal = norm(mu_current, proposal_width).rvs()
```

Next, we evaluate whether the proposal is a good place to jump to or not. If the resulting normal distribution with the proposed `mu` explains the data better than the old `mu`, we definitely want to go there. What does "explains the data better" mean? We quantify fit by computing the probability of the data, given the likelihood (normal) with the proposed parameter values—proposed `mu` and a fixed `sigma = 1`. This can easily be computed by calculating the probability for each data point using `scipy.stats.normal(mu, sigma).pdf(data)` and then multiplying the individual probabilities, i.e., computing the likelihood. Note that usually we would use log probabilities for this so that the product is converted into a sum.

```
likelihood_current = norm(mu_current, 1).pdf(data).prod()
likelihood_proposal = norm(mu_proposal, 1).pdf(data).prod()

# Compute prior probability of current and proposed mu
prior_current = norm(mu_prior_mu, mu_prior_sd).pdf(mu_current)
prior_proposal = norm(mu_prior_mu, mu_prior_sd).pdf(mu_proposal)

# Numerator of Bayes formula
p_current = likelihood_current * prior_current
p_proposal = likelihood_proposal * prior_proposal
```

Up until now, we essentially have a hill-climbing algorithm that would just propose movements in random directions and only accept a jump if the `mu_proposal` has higher likelihood than `mu_current`. Eventually we will attain `mu = 0` (or close to it) from where no more moves will be possible. However, we want to compute a posterior, so we will also have to sometimes accept moves into the other direction. The key to this is that by dividing the two probabilities,

```
p_accept = p_proposal / p_current
```

we get an acceptance probability. We can already see that if `p_proposal` is larger than `p_current`, this acceptance probability will be greater than one and we will definitely accept. However, if `p_current` is larger, say twice as large, there will be a 50% chance of moving there. To compute this, we draw a $\mathcal{U}[0, 1]$ random value and compare it to the acceptance probability,

```
accept = np.random.rand() < p_accept

if accept:
    # Update position
    cur_pos = proposal
```

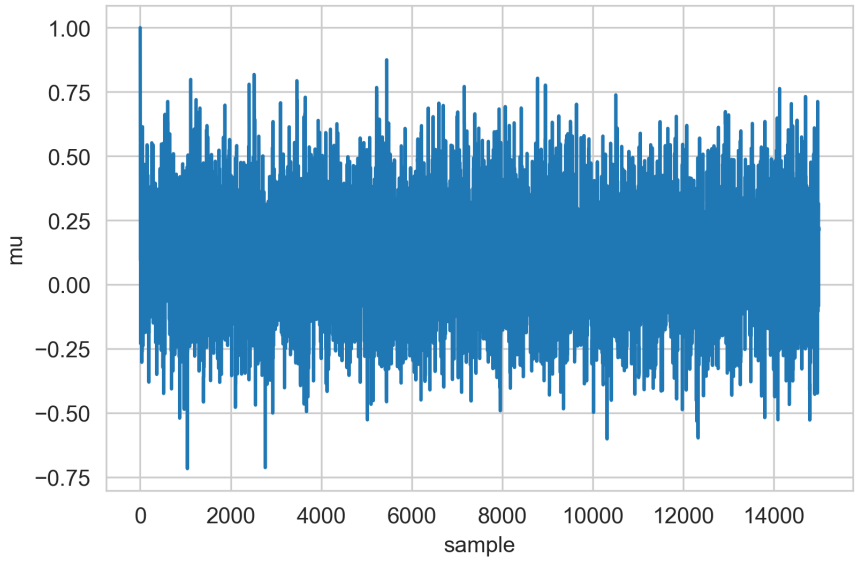
This simple procedure, the veritable core of the Markov chain, gives us samples from the posterior.

Putting It All Together Here is the complete function that performs the MCMC sampling.

[5]:
```
def sampler(data, samples=4, mu_init=.5, proposal_width=.5,␣
↪mu_prior_mu=0, mu_prior_sd=1.):
    mu_current = mu_init
    posterior = [mu_current]
    for i in range(samples):
        # suggest new position
        mu_proposal = norm(mu_current, proposal_width).rvs()

        # Compute likelihood by multiplying prob.'s of each data point
        likelihood_current = norm(mu_current, 1).pdf(data).prod()
        likelihood_proposal = norm(mu_proposal, 1).pdf(data).prod()

        # Compute prior probability of current and proposed mu
        prior_current = norm(mu_prior_mu, mu_prior_sd).pdf(mu_current)
        prior_proposal = norm(mu_prior_mu, mu_prior_sd).pdf(mu_proposal)

        p_current = likelihood_current * prior_current
        p_proposal = likelihood_proposal * prior_proposal

        # Accept proposal?
        p_accept = p_proposal / p_current

        accept = np.random.rand() < p_accept

        if accept:
            # Update position
            mu_current = mu_proposal

        posterior.append(mu_current)

    return np.array(posterior)
```

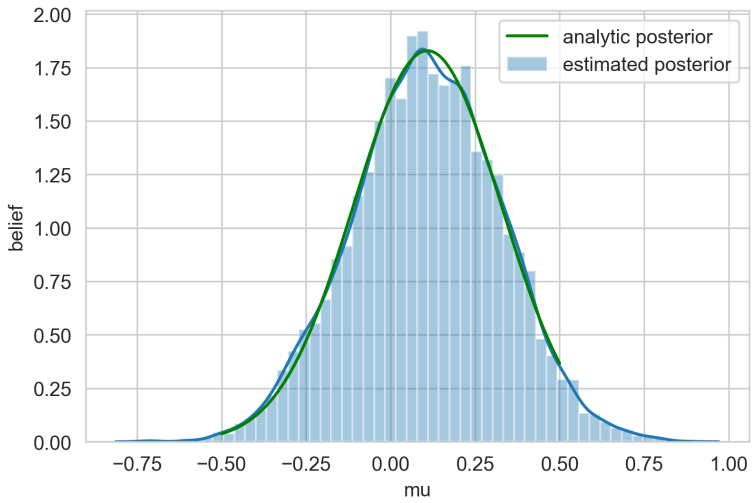
MCMC works by repeating this sampling procedure over a long time; then the samples generated are guaranteed, by the theorems seen above, to come from the posterior distribution of our model. To visualize this, we can draw a large number of samples and plot them.

[6]:
```
posterior = sampler(data, samples=15000, mu_init=1.)
fig, ax = plt.subplots()
ax.plot(posterior)
_ = ax.set(xlabel='sample', ylabel='mu');
```



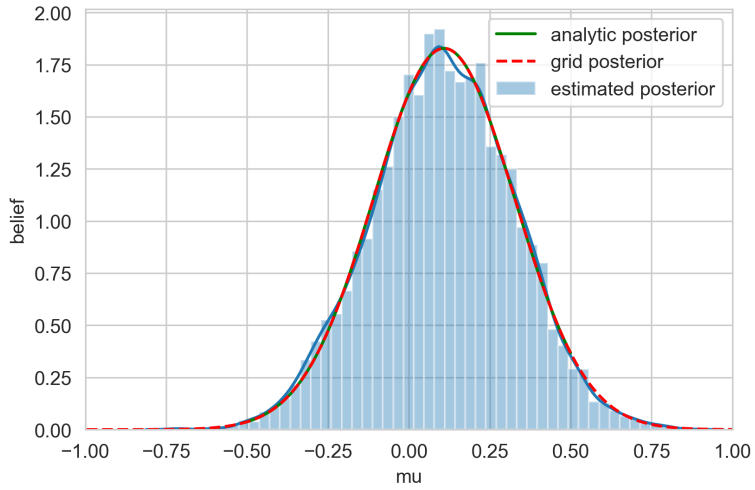
This plot is usually called the trace. To obtain an approximation of the posterior from it, we simply compute the histogram of this trace. It is important to keep in mind that, although this looks similar to the data we sampled above to fit the model, the two are completely separate. The next plot represents our *belief*, in the Bayesian sense, in `mu`. In this conjugate case it just happens to also be normal, but for a different model it could have a completely different shape than the likelihood or prior.

[7]:
```
ax = plt.subplot()

sns.distplot(posterior[500:], ax=ax, label='estimated posterior')
x = np.linspace(-.5, .5, 500) # exclude burn-in samples (first 500)
post = calc_posterior_analytical(data, x, 0, 1)
ax.plot(x, post, 'g', label='analytic posterior')
_ = ax.set(xlabel='mu', ylabel='belief');
ax.legend();
```



Alternative Approach: Grid Sampling When the dimension of the parameter vector is small, we can simply sample the range of possible values for μ and compute the posterior by multiplication. Finally we display the histogram. However, this will not be tractable as soon as we have more than 4 or 5 parameters to estimate, due to the curse of dimensionality.

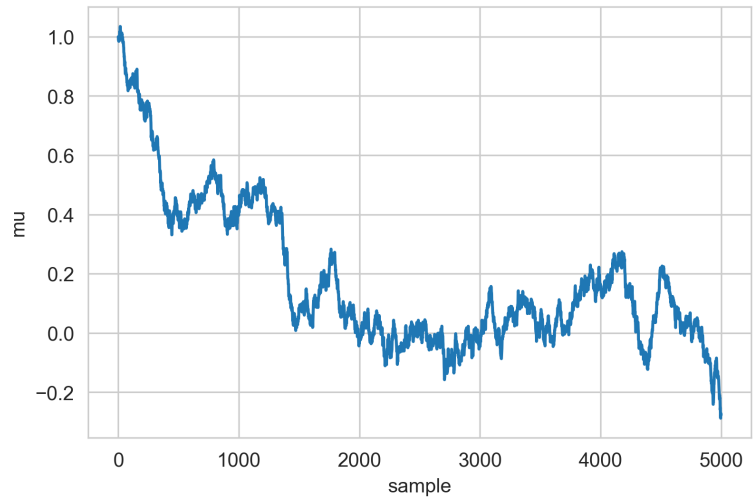
[8]:
```
# Exhaustive grid sampling

mu_prior_mu=0
mu_prior_sd=1.

gr_posterior = list()

mu_is = np.linspace(-3, 3, 601)
for mu_i in mu_is:
    prior_i = norm(mu_prior_mu, mu_prior_sd).pdf(mu_i)
    likelihood_i = norm(mu_i, 1).pdf(data).prod()
    gr_posterior.append(likelihood_i * prior_i)

# Calculate normalizing constant:
gr_posterior_sum = sum(gr_posterior)/100. # divide by 100 since every bin␣
 ↪in the prior histogram (prior_i) has width 0.01

# Final posterior is:
gr_posterior = gr_posterior / gr_posterior_sum
# Plot and compare
ax1 = plt.subplot()

sns.distplot(posterior[500:], ax=ax1, label='estimated posterior')
x = np.linspace(-.5, .5, 500)
post = calc_posterior_analytical(data, x, 0, 1)
ax1.plot(x, post, color='g', label='analytic posterior')
x = np.linspace(-3, 3, 601)
ax1.plot(x, gr_posterior, color='r', linestyle='--', label='grid␣
 ↪posterior')
_ = ax1.set(xlabel='mu', ylabel='belief');
ax1.legend();
plt.xlim([-1, 1])
```



We see that by following the above procedure, we get samples from the same distribution as the one that was derived analytically, as well as the distribution computed by MCMC.

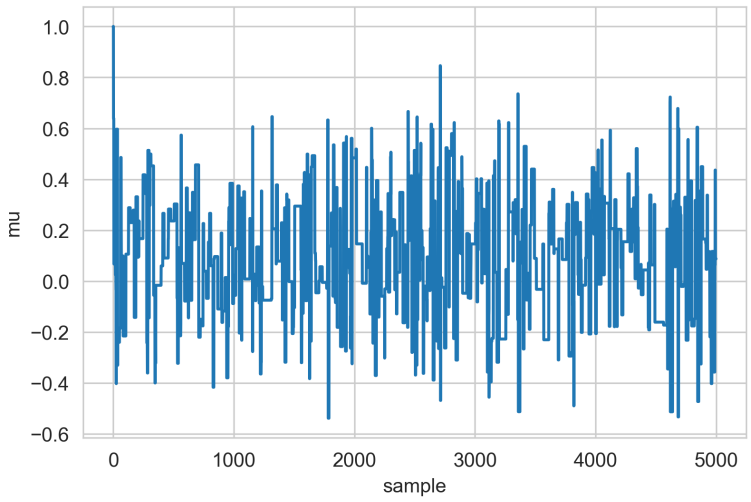
Proposal Width Above, we set the proposal width to `0.5`, which turned out to be a reasonably good value. In general we do not want the width to be too narrow because the sampling will be inefficient as it takes a long time to explore the whole parameter space and shows the typical random-walk behavior. Let us compute the chain with a small width of 0.01.

[9]:
```
posterior_small = sampler(data, samples=5000, mu_init=1., proposal_width=.
↪01)
fig, ax = plt.subplots()
ax.plot(posterior_small);
_ = ax.set(xlabel='sample', ylabel='mu');
```



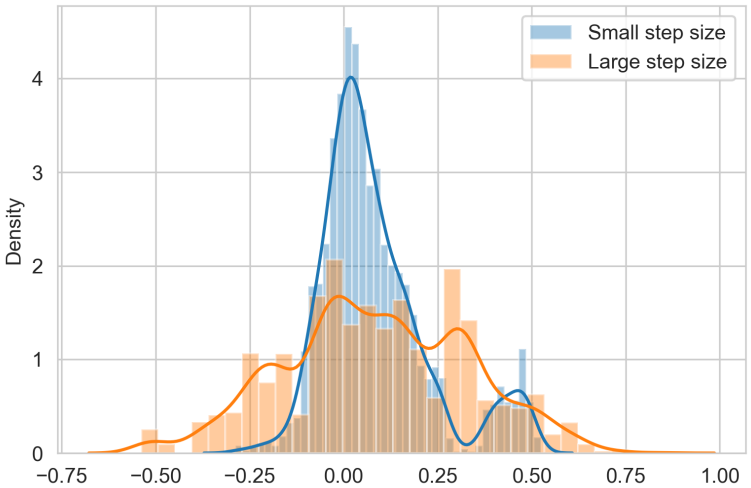
Now, we contrast with a larger width, 3, that should not be so large that we hardly ever accept a jump.

[10]:
```
posterior_large = sampler(data, samples=5000, mu_init=1.,␣
↪proposal_width=3.)
fig, ax = plt.subplots()
ax.plot(posterior_large); plt.xlabel('sample'); plt.ylabel('mu');
_ = ax.set(xlabel='sample', ylabel='mu');
```



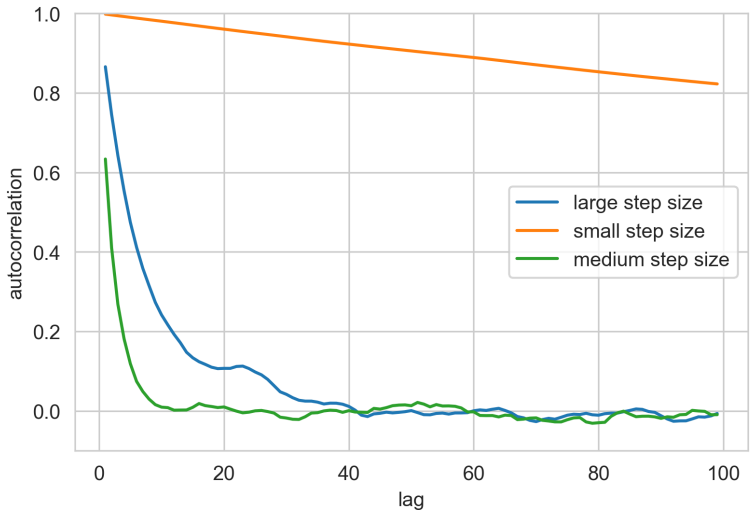
Note, however, that we are still sampling from our target posterior distribution here, as guaranteed by the mathematical proof, just less efficiently, as can be clearly seen in this comparison plot.

[11]:
```
sns.distplot(posterior_small[1000:], label='Small step size')
sns.distplot(posterior_large[1000:], label='Large step size');
_ = plt.legend();
```



With more samples this will eventually look like the true posterior. The key is that we want our samples to be independent of each other, which clearly is not the case with a large proposal width. Thus, a common metric to evaluate the efficiency of our sampler is the autocorrelation, i.e., how correlated sample `i` is to sample `i-1`, `i-2`, etc.:

[12]:
```
from pymc3.stats import autocorr
lags = np.arange(1, 100)
fig, ax = plt.subplots()
ax.plot(lags, [autocorr(posterior_large, l) for l in lags], label='large␣
↪step size')
ax.plot(lags, [autocorr(posterior_small, l) for l in lags], label='small␣
↪step size')
ax.plot(lags, [autocorr(posterior, l) for l in lags], label='medium step␣
↪size')
ax.legend(loc=0)
_ = ax.set(xlabel='lag', ylabel='autocorrelation', ylim=(-.1, 1))
```



The medium step case, with proposal width of 0.5, produces the sharpest decrease in autocorrelation, which is a sign of independence of the successive steps. We would like to have a criterion for determining the good step width automatically. One common method is to keep adjusting the proposal width so that roughly 50% of the proposals are rejected.

Extending to More Complex Models Now we could also add a `sigma` parameter for the standard-deviation and follow the same procedure for inferring this second parameter. In this case, we would be generating proposals for both `mu` and `sigma`, but the algorithm logic would be nearly identical. Or, we could have data from a very different distribution, like a Binomial, and still use the same algorithm and get the correct posterior. Here we could definitely benefit from probabilistic programming, where we just define the model we want and let MCMC take care of the inference. This is discussed in Section 11.11 of Chapter 11.

For example, the model below can be written in PyMC3 quite easily. Below we also use the Metropolis sampler, which automatically tunes the proposal width, and see that we get identical results.

```
import pymc3 as pm

with pm.Model():
    mu = pm.Normal('mu', 0, 1)
    sigma = 1.
    returns = pm.Normal('returns', mu=mu, sd=sigma, observed=data)

    step = pm.Metropolis()
    trace = pm.sample(15000, step)

sns.distplot(trace[2000:]['mu'], label='PyMC3 sampler');
sns.distplot(posterior[500:], label='Hand-written sampler');
plt.legend();
```

3.8 ▪ Stochastic Differential Equations

What are SDEs? Naively, they are ordinary differential equations driven by some random, white noise. But since white noise is nowhere differentiable by definition (see below), we need a special stochastic calculus to define and analyze these equations.

Why SDEs? All dynamical systems evolve under the influence of stochastic forces. These range from intrinsic uncertainties at molecular or mesoscopic levels, to large-scale effects of complex systems. Often the uncertainties come from external, applied forces, from initial and from boundary conditions, even from the geometry of the domain. Finally, material properties are an important source of randomness. All of these can be modeled by suitable stochastic parameters and stochastic processes and these will modify fundamentally the behavior of the dynamical system.

3.8.1 ▪ ODE Driven by Noise

Consider the simplest Cauchy problem for an ordinary differential equation,

$$\begin{aligned} \dot{\mathbf{x}}(t) &= a\left(\mathbf{x}, t\right), \quad t > 0, \\ \mathbf{x}(0) &= \mathbf{x}_0, \end{aligned} \tag{3.107}$$

where the initial condition $\mathbf{x}_0 \in \mathbb{R}^n$, the right-hand side $a\colon \mathbb{R}^n \to \mathbb{R}^n$ is a smooth vector field, and $\mathbf{x}(t)$ is the system state at time t. Under reasonable Lipschitz continuity conditions on a, we know that a unique solution exists for the Cauchy problem (3.107). Now suppose that the system is subject to random perturbations. We could then formally[33] modify the

[33] Mathematically, this means that the step is not necessarily rigorous, nor proven to be such.