Simulating Random Numbers

Gordon Ross

Simulation

In statistics/probability we often wish to simulate random number from a probability distribution. Eg:

- We may wish to draw 10 realistions from a Normal(0,2) distribution.
- Or we may wish to draw 20 realistions from an Exponential(1) distribution.

How can we do this?

Simulation in R

Of course, R has built in functions which can do this for many distributions. E.g to simulate 10 realistions from a Normal(0,2) distribution we can type

```
rnorm(20,0,sqrt(2))
```

Or to simulate 20 realistions from an Exponential(1) distribution we can type:

```
rexp(20,1)
```

But what if we want to draw from a distribution that isnt implemented in R? And how is R managing to draw these numbers in the first place?

Simulation

So, how do we simulate values from a probability distribution?

Lets assume that we have some probability density $p(x)$ with distribution function F that we want to simulate values from.

For example, suppose that F is the Exponential distribution.

You may (or may not!) have previously came across the Probability Integral Transform theorem...

Probability Integral Transform

Theorem: Suppose X has distribution F_X. Define $U = F_X(X)$. Then, U has a Uniform(0,1) distribution.

Proof: We derive the distribution F_U of U and show that it is Uniform(0,1)

$$\begin{aligned} F_U(u) &= p(U \leqslant u) \\ &= p(F_X(X) \leqslant u) \\ &= p(X \leqslant F_X^{-1}(u)) \\ &= F_X(F_X^{-1}(u)) \\ &= u \end{aligned}$$

which is the CDF of a Uniform(0,1).

Essentially this theorem says that we can 'transform' an observation X from a random variable into a Uniform(0,1) variable by passing it through its own CDF function.

Direct Simulation

The Probability Integral Transform has a direct corollary which we get from inverting it. Suppose we start with a Uniform(0,1) variable U, and pass it through the inverse CDF F_X^{-1}. We end up with an observation from the distribution F_X.

Corollary: Suppose U has a Uniform(0,1) distribution. Let F_X be any **continuous** distribution function, and define $Y = F_X^{-1}(U)$. Then, X has distribution F_X

Proof: $p(X \leqslant x) = p(F_X^{-1}(U) \leqslant x) = p(U \leqslant F_X(x)) = F_X(x)$

The last step is true because $p(U \leqslant z) = z$ when U is Uniform on (0,1).

Direct Simulation

This means that we can simulate an observation from a continuous distribution F_X by first simulating a random Uniform(0,1) variable and passing it through the inverse CDF F_X^{-1}

Since it uses the inverse CDF, this is known as **inverse transform sampling**

Example - Exponential Distribution

Suppose for example we want to simulate values from an Exponential(λ) distribution. The density and distribution functions are:

$$f(x|\lambda) = \lambda e^{-\lambda x}, \quad F(x|\lambda) = 1 - e^{-\lambda x}$$

We start by deriving the inverse CDF. This is basic algebra, we set F to be equal to a constant and invert ('solve for y'):

$$\begin{aligned} 1 - e^{-\lambda x} &= u \\ \implies e^{-\lambda x} &= 1 - u \\ \implies x &= -(1/\lambda)\log(1-u) \\ \implies F_X^{-1}(x) &= -(1/\lambda)\log(1-u) \end{aligned}$$

Example - Exponential Distribution

So to sample n independent values from an Exponential(λ) distribution, we first sample n independent U(0,1) variables $U_1, \ldots, U_n$.

Then for each sampled U_i, we define:

$$X_i = F_X^{-1}(U_1) = -(1/\lambda)\log(1 - U_i)$$

The resulting $X_1, \ldots, X_n$ are i.i.d Exponential(λ)

R code

```
simulateExponential <- function(n,lambda) {
  u <- runif(n,0,1)
  x <- -(1/lambda) * log(1-u)
  return(x)
}
```

Example 2 - Kumaraswamy Distribution

The Kumaraswamy distribution is an (obscure!) distribution with 2 parameters a and b, and pdf:

$$f(y) = aby^{a-1}(1 - y^a)b - 1$$

for $y \in [0, 1]$ and 0 otherwise.

How do we simulate observations from this distribution?

Example 2 - Kumaraswamy Distribution

We need the inverse CDF, so we first need the CDF:

$$\begin{aligned} F(y) &= \int_0^y aby^{a-1}(1-y^a)b-1dy \\ &= \left[-(1-y^a)^b\right]_0^y \\ &= 1-(1-y^a)^b \end{aligned}$$

for $y \in [0,1]$.

Example 2 - Kumaraswamy Distribution

We then invert to find the inverse CDF:

$$u = 1 - (1 - y^a)^b$$

$$\implies 1 - y^a = (1 - u)^{1/b}$$

$$\implies y = \left(1 - (1 - u)^{1/b}\right)^{1/a}$$

$$\implies F_Y^{-1} = \left(1 - (1 - y)^{1/b}\right)^{1/a}$$

R code

```
simulateKumaraswamy <- function(n,a,b) {
  u <- runif(n,0,1)
  y <-  (1-(1-u)^(1/a))^(1/b)
  return(y)
}
```

Summary

We can summarise the pros and cons of the method of inversion as follows:

Advantages

- Produces independent samples from the distribution.
- Easy to implement if F^{-1} can be computed

Disadvantages

- Need to be able to calculate F and F^{-1}
- Many be difficult to extend to multivariate distributions depending if we can find F^{-1} for the marginals.

But lets take a step back...

So in theory, we can use inverse transform sampling to generate random numbers from a distribution, as long as we can invert its CDF function.

But to do this, we have to first generate random numbers from a Uniform distribution. We can do this in R using runif(). But how is this actually possible?

Computers are just deterministic electronic devices. How can a computer produce a number that is 'random'. In fact, how can any physical process produce randomness? Where are these random numbers actually coming from?

But lets take a step back...

Basic answer: the numbers produced by computers (eg using runif()) are not truly random – deterministic electronic devices cannot produce truly random numbers. This is not specific to R, it is just a basic fact about computers.

Instead, computers produce pseudo-random numbers. These are technically generated by fully deterministic processes, but the processes are so complex (in a mathematical sense) that the numbers produced can be viewed as being random for all intents and purposes.

Linear congruential generators

- Intuitive motivation: Suppose we take an integer, multiply it by some enormous factor, re-write it in base - 'something huge', and then throw away everything except for the digits after the decimal point. What is the result?
- Repeat the operation, feeding each step's output into the input for the next step, a random sequence might result.
- Formally, the pseudorandom sequence is defined by

$$X_{i+1} = (aX_i + c) \mod M$$

for some natural number M, with $a, c \in \{0, 1, \ldots, M-1\}$. The algorithm starts from an integer number X_0 which is called *the seed*. The X_i are integers but we can define

$$U_i = X_i / M$$

with the intuitive hope that U_i are well modelled by a Uniform$(0, 1)$.

Linear congruential generators

- Such a generator will repeat with period at most M.
- A property to try to achieve is *full period*. The values of M, a, c are chosen to maximise the period and speed of the generator, and the apparent randomness of the output.
- `RANDU` example (IBM)

$$X_{i+1} = (65539 X_i) \mod 2^{31}$$

RANDU R code

```
## RANDU
X     <- NULL
X[1] <- 1110
U     <- NULL
N     <- 10000
M     <- 2^(31)
a     <- 65539
c     <- 0
for(i in 2:N)
{
     X[i] <- (a*X[i-1] + c) %% M
     U[i] <- X[i]/M
}
```

Assessment of random number generators

- Not only is it important that the U_i are uniform, but also that they seem independent
- One way to do this is to consider k-tuples

$$(U_i, U_{i+1}, \ldots, U_{i+k-1})$$

of successive values as points in the set $(0, 1)^k$.

- R code

```
U1 <- U[1:(N-2)]
U2 <- U[2:(N-1)]
U3 <- U[3:N]
plot(U1); plot(U1,U2)
library(lattice); cloud(U1~U2*U3)
```

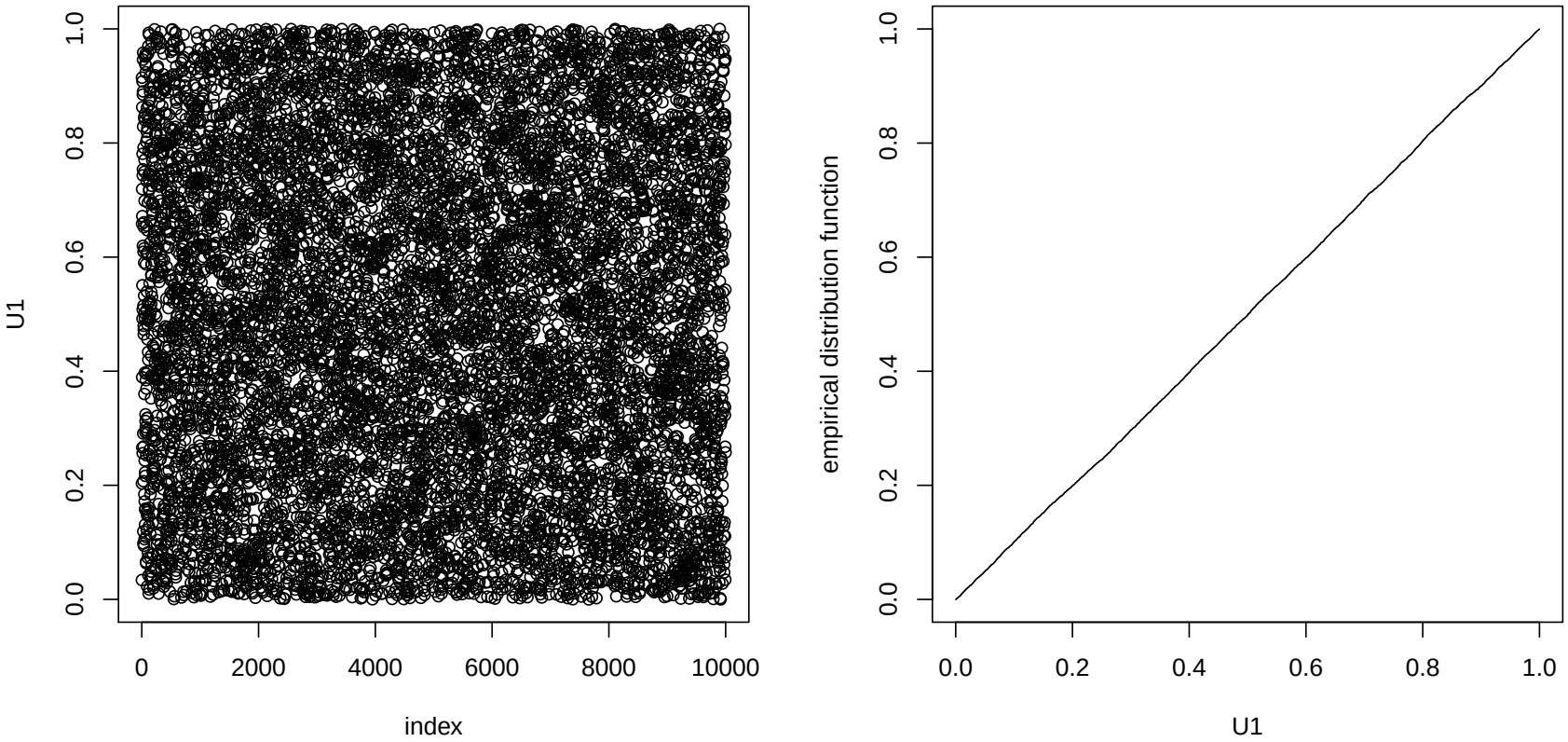


Figure: Left: Scatterplot of sequence $(U_1, \ldots, U_N)$ against index $i = 1, \ldots, N = 10000$. Right: empirical estimate of distribution function

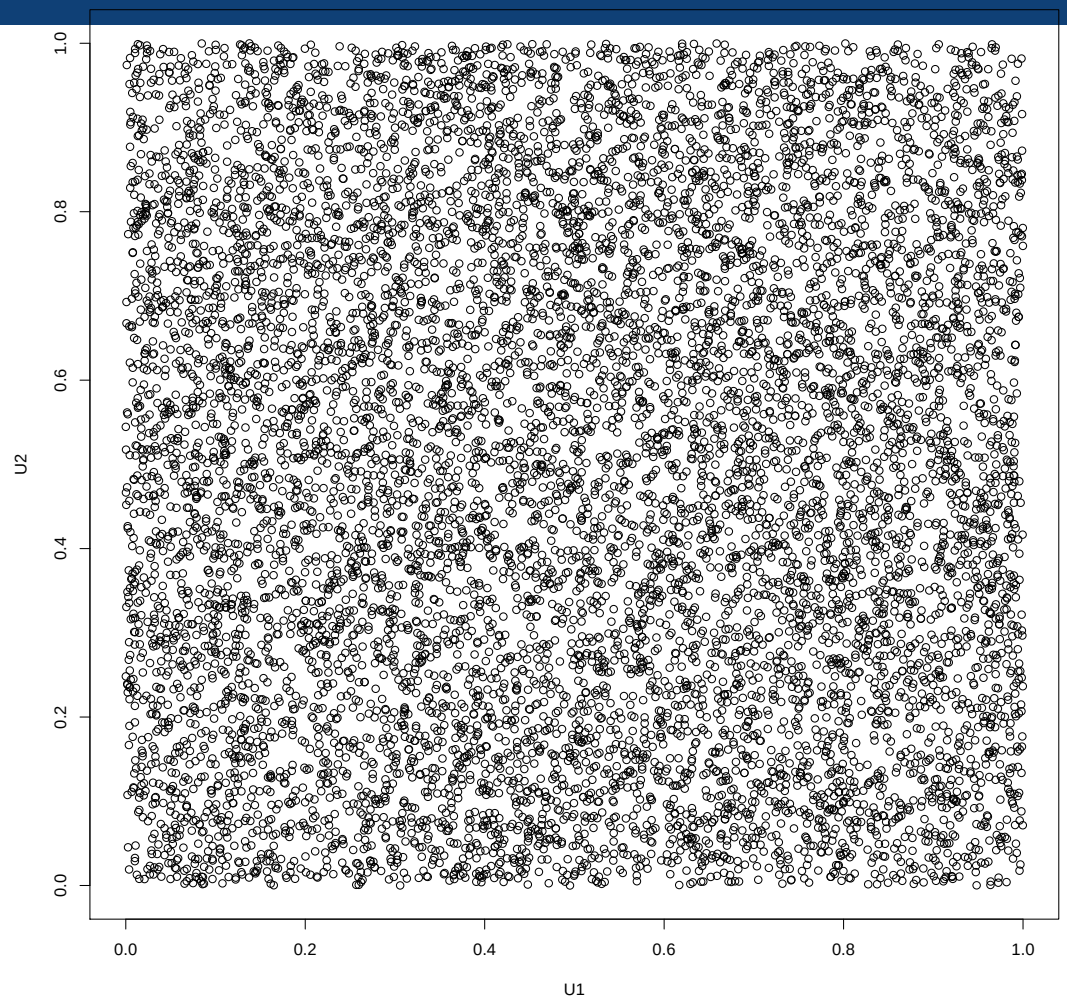


Figure: Left: Scatterplot of sequence $\boldsymbol{U}_1 = (U_1, \ldots, U_{N-2})$ against $\boldsymbol{U}_2 = (U_2, \ldots, U_{N-1})$

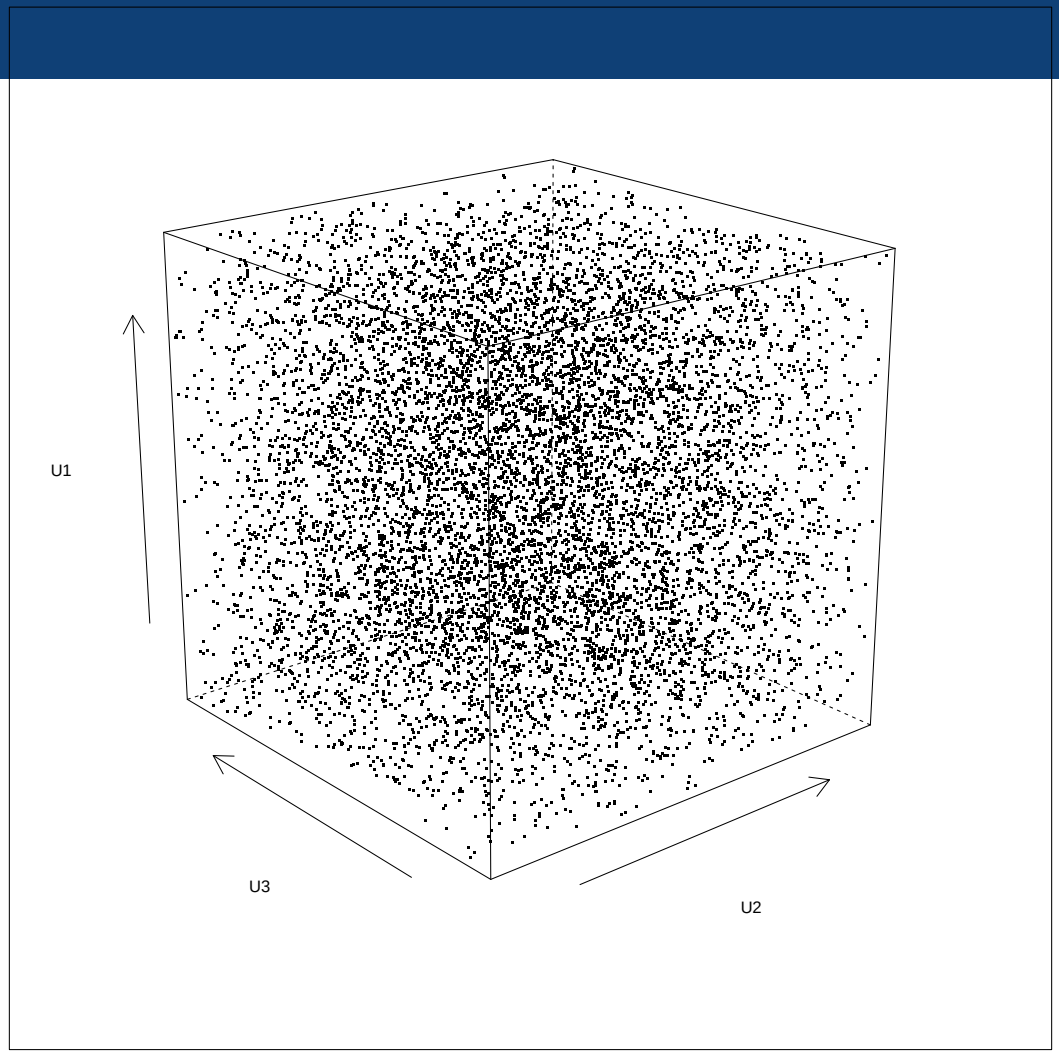


Figure: Left: Scatterplot of sequence $\boldsymbol{U}_1 = (U_1, \ldots, U_{N-2})$ against $\boldsymbol{U}_2 = (U_2, \ldots, U_{N-1})$ and $\boldsymbol{U}_3 = (U_3, \ldots, U_N)$

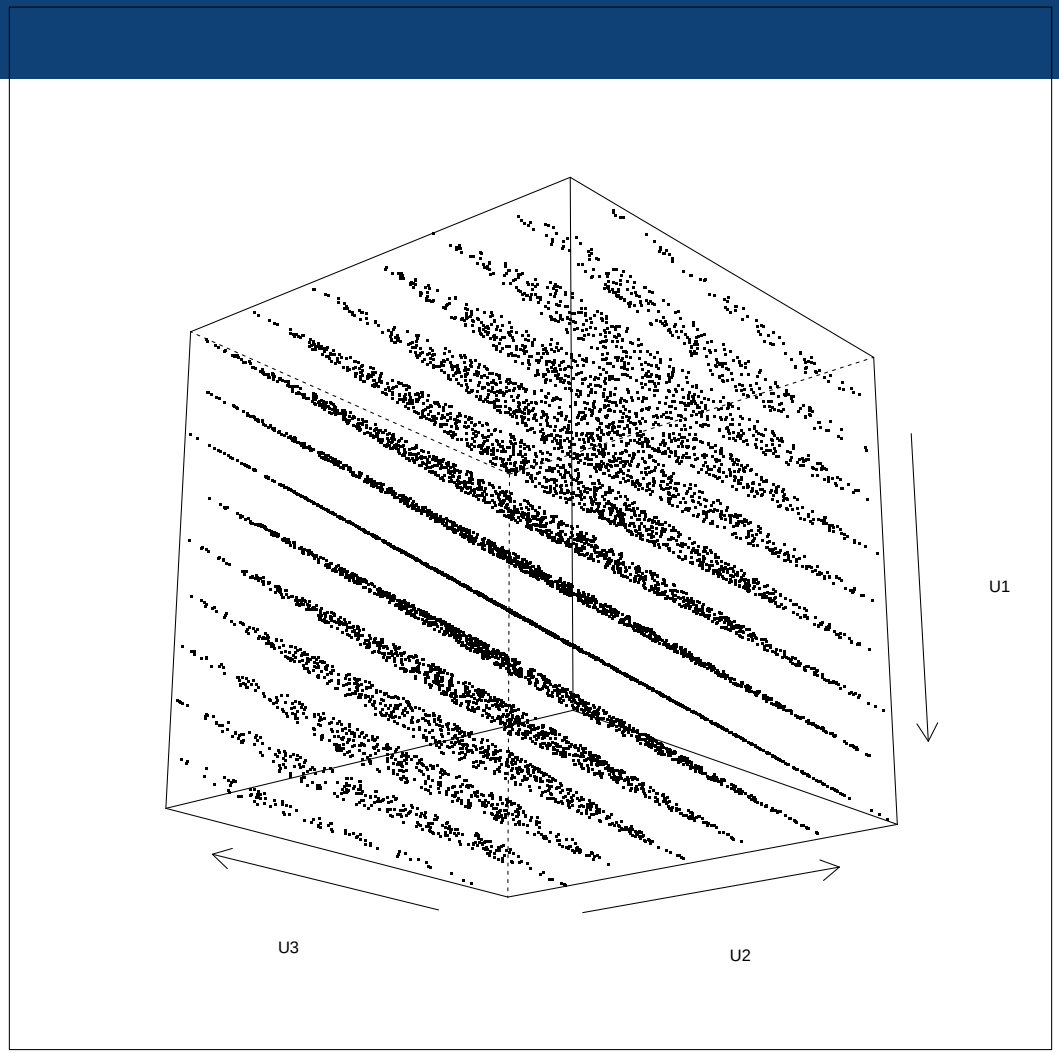


Figure: Left: Scatterplot of sequence $\boldsymbol{U}_1 = (U_1, \ldots, U_{N-2})$ against $\boldsymbol{U}_2 = (U_2, \ldots, U_{N-1})$ and $\boldsymbol{U}_3 = (U_3, \ldots, U_N)$

Comments

- If a pseudo-random number generation algorithm produces numbers which have patterns (e.g. dependence between U_1 and U_2) then this means that future random numbers can be predicted. This is a huge problem: most modern internet security is driven by random numbers to some degree, and predictability would result in exploitable vulnerabilities that hackers could use.

- The lesson is to use generators that have been carefully engineered by people with a good understanding of number theory.

- There are many tests which can be used to check if a pseudo-random number generator is producing numbers which seem truly random (i.e. which dont contain patterns). The 'Diehard' tests are a commonly used battery of such tests. We will explore some of these in the lab.

- Many of the algorithms in standard packages have been thoroughly tested, but it is wise to store the seed X_0 so that if necessary the sequence can be repeated, and to perform important calculations using two different generators (note: this is what the set.seed() function that I have been using in the labs does)

Comments

The Mersenne Twister algorithm for simulating pseudorandom numbers is the default choice in most cases. See

```
?.Random.seed
```

- `http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html`

Rejection Sampling

Rejection sampling is an alternative approach for when it is not easy to compute F^{-1} .

Suppose we wish to sample from an arbitrary probablity distribution $p(x)$ which may or may not integrate to 1.

Basic idea: we simulate values $x^1, \ldots, x^n$ from a different distribution $g(x)$ which we know how to simulate from easily. We then throw away ('reject') some of these values in a clever way so that the ones which remain are a sample from $p(x)$ rather than from $g(x)$.

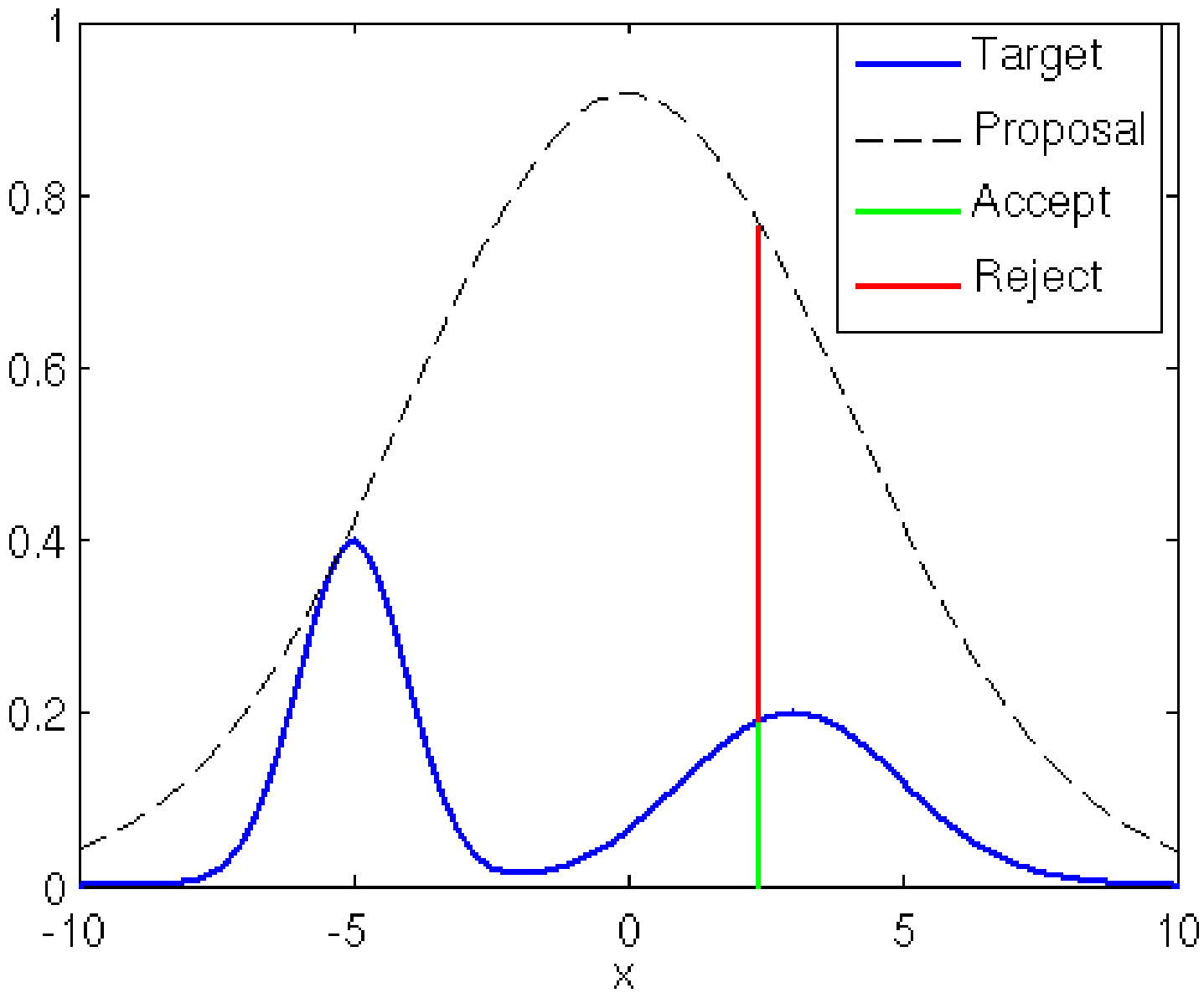
We hence 'convert' a sample from $g(x)$ (which we know how to simulate from) into a sample from $p(x)$ (which we do not know how to simulate from).

Rejection Sampling

Basic idea: we choose a distribution $g(x)$ such that $g(x) \geqslant p(x)$ for all x.

We then simulate x^* from $g(x)$ and evaluate the ratio $p(x^*)/g(x^*)$. Intuitively, we want to keep x^* if this ratio is close to 1, and reject it if the ratio is close to 0

Rejection Sampling



Rejection Sampling

More specifically, let $q = p(x^*)/g(x^*)$. We keep x^* with probability q, and reject it with probability $(1 - q)$

For rejection sampling to work, we need $g(x) \geqslant p(x)$ for all x. In practice we will not be able to find distributions with this property.

Instead, we can multiply $g(x)$ by a positive constant $M \geqslant 1$ and instead require that $Mg(x) \geqslant p(x)$ for all x.

Note: another requirement is that we need $p(x)/g(x)$ to be finite for all x (otherwise we cannot find an M that works).

Rejection Sampling

This leads to the standard rejection sampling algorithm:

1. Sample x^* from $g(x)$.
2. Sample $u \sim Uniform(0, 1)$
3. If $u \leqslant p(x^*)/Mg(x)*$ then return x^*, otherwise discard it and return to Step 1

Repeat this n times to get an independent sample of n observations from $p(x)$.

Note: $p(x)$ does not need to be normalised for this to work!

Example

Suppose we wish to draw values from a Normal($\mu = 3$, $\sigma^2 = 1$) distribtution using rejection sampling. The density is:

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{\left(\frac{(x-\mu)^2}{2\sigma^2}\right)}$$

The maximum value of $p(x)$ occurs at the mode, which is $p(3) = 0.4$.

We will use rejection sampling using a Uniform distribution as the proposal.

Example

Suppose we choose a proposal distribution $g(x) = Uniform(a, b)$. Note that this has finite support while $p(x)$ has infinite support. However if we choose $[a, b]$ to be wide enough, then $p(x)$ is essentially 0 outside this region.

Say we take $g(x)$ as Uniform(-5,5). Then the density of $g(x)$ is:

$$g(x) = \begin{cases} 1/10 & \text{if } a \leqslant x \leqslant b \\ 0 & \text{otherwise} \end{cases}$$

Example

We need to choose M such that $Mg(x) \geqslant p(x)$ for all x. So, we need $M \geqslant \max p(x)/g(x)$,

We said that $\max p(x) = 0.4$ and $\max g(x) = 0.1$. So we take $M = 4$. The rejection sampler is hence:

1. Sample x^* from $Uniform(-5, 5)$.
2. Sample $u \sim Uniform(0, 1)$
3. If $u \leqslant p(x^*)/Mg(x^*)$ then return x, otherwise discard it and return to Step 1

Filling in the values, the last step is:

If $u \leqslant N(x^*; 3, 1)/(4 * 0.1)$ then return x, otherwise discard it and return to Step 1

Where $N(x^*; 3, 1)$ is the $N(3, 1)$ distribution evaluated at x^*

Example - R Code

```
f <- function(n) {
  samples <- numeric(n)
  numSoFar <- 0
  for (i in 1:n) {
    while(TRUE) {
      x <- runif(1,-5,5)
      u <- runif(1,0,1)
      if (u < dnorm(x,3,1)/0.4) {
        numSoFar <- numSoFar + 1
        samples[numSoFar] <- x
        break
      }
    }
  }
  return(samples)
}

plot(density(f(1000)),type='l')
```

Proof of Rejection Sampling

I will now give a formal proof that rejection sampling works, i.e. that the resulting samples are indeed draws from $p(x)$. To keep things simple, assume that $p(x)$ integrates to 1, i.e that it is a density (although the argument for an arbitrary unnormalised $p(x)$ is similar).

Let x^* be an observation simulated using rejection sampling from a proposal distribtion $g(x)$. We need to show that:

$$p(x^* \leqslant x | \text{x* is accepted}) = \int_{-\infty}^{x} p(s)ds$$

i.e that x^* is a sample from the distribution function corresponding to $p(x)$,

Proof of Rejection Sampling

Bayes theorem tells us that:

$$p(x^* \leqslant x | \text{x* is accepted}) = \frac{p(x^* \leqslant x \text{ and } x^* \text{ is accepted})}{p(\text{x* is accepted})}$$

Note that $p(\text{x* is accepted})$ is **not conditional on any particular value of** x^*. It is the unconditional probability of drawing an arbitrary value from $g(x)$ and having it be accepted. It is hence equal to the proportion of the area under $Mg(x)$ which is also under $p(x)$

Proof of Rejection Sampling

$$p(\text{x* is accepted}) = \frac{\int_{-\infty}^{\infty} p(x^*)dx^*}{\int_{-\infty}^{\infty} Mg(x^*)dx^*} = \frac{1}{M}$$

Similarly for the numerator:

$$p(x^* \leqslant x \text{ and } x^* \text{ is accepted}) =$$

$$= \int_{-\infty}^{x} g(x^*)\frac{p(x^*)}{Mg(x^*)}dx^* = \frac{\int_{-\infty}^{x} p(x^*)dx^*}{M}$$

(note: this is true since x is fixed and $p(x^* \leqslant x)$ is computed with respect to the distribution of $g(\cdot)$ since this is being used to simulate x^*).

Proof of Rejection Sampling

Substituting in:

$$p(x^* \leqslant x | \text{x* is accepted}) = \frac{\frac{\int_{-\infty}^{x} p(x^*)dx^*}{M}}{1/M}$$

$$= \int_{-\infty}^{x} p(x^*)dx^*$$

As required.

Practicalities

As part of this proof, we saw that the probability of accepting a sample was equal to $1/M$, so the probability of rejection is $1 - 1/M$.

This means that on average, we will need to draw M samples before one is accepted. So if we want to simulate n observations from $p(x)$ we will on average need to draw nM observations from $g(x)$.

Remember that $M \geqslant 1$.

1. We want M to be as small as possible in order to minimise the number of samples
2. If M is too large then rejection sampling may not be practical

Practicalities - Choosing M

So, we want M to be as small as possible. We also need (by definition) to have $Mg(x) \geqslant p(x)$ for all x.

By rearranging, this means we should choose M to be the smallest value such that:

$$M \geqslant \max_x \frac{p(x)}{g(x)}$$

So ideally:

$$M = \max_x \frac{p(x)}{g(x)}$$

Example

Suppose $p(x) = Beta(3, 2)$ and $g(x) = Uniform(0, 1)$. Define:

$$f(x) = \frac{p(x)}{g(x)} = \frac{\Gamma(5)}{\Gamma(3)\Gamma(2)} x^2(1-x)$$

We want to choose M to be the maximum value of $f(x)$. We hence differentiate, set equal to 0, etc.

Example

$$f'(x) = \frac{\Gamma(5)}{\Gamma(3)\Gamma(2)}(2x - 3x^2) = 0$$

$$\Rightarrow x = 2/3$$

So maximum occurs at $x = 2/3$. The maximum value is hence $f(2/3) = 16/9$

So we choose $M = 16/9$

Example - R Code

```
f <- function(n) {
  samples <- numeric(n)
  numSoFar <- 0
  for (i in 1:n) {
    while(TRUE) {
      x <- runif(1,0,1)
      u <- runif(1,0,1)
      if (u < dbeta(x,3,2)/(16/9)) {
        numSoFar <- numSoFar + 1
        samples[numSoFar] <- x
        break
      }
    }
  }
  return(samples)
}

plot(density(f(1000)),type='l')
```

Rejection Sampling

The uncondiontal probability of a sample being accepted is $1/M$. When M is very large, this means that it will take many samples in order for one to be accepted.

Since the 'best' M is essentially determined by the shape of $g(x)$, this means we need to choose a good $g(x)$ which (ideally) has a similar shape to $p(x)$.

In practice the uniform distribution is unlikely to have this property, particularly when $p(x)$ is extremely peaked (eg a Cauchy)

Note that some distributions $g(x)$ might not work at all, since we need $p(x)/g(x)$ to be finite for all x, otherwise we cant find an M that works.

Rejection Sampling

Limitation of rejection sampling: sometimes it can be hard to find a good $g(x)$, particular when $p(x)$ is multivariate

As such, rejection sampling tends to break down in high dimensions. it may take trillions or more draws from $g(x)$ before one is accepted

Summary

Advantages:

1. Very general method, can be used to sample from any distribution.
2. Easy to implement and code.
3. Dont need to know normalising constant for $p(x)$
4. Efficient if we can find $g(x)$ for which M is small.

Disdvantages:

1. Can be hard to find a good $g(x)$, particularly in high dimensions.
2. If a poor choice of $g(x)$ results in M being large, can be very inefficient