Random Variable Generation

Michael Rose

November 18, 2018

Reader's Guide

In this chapter, practical techniques that produce random variables from both standard and nonstandard distributions are shown.

Given the availability of a uniform generator in R, we do not deal with the specific production of uniform random variables. The most basic techniques relate the distribution to be simulated to a uniform variate by a transform or a particular probabilistic property.

2.1 | Introduction

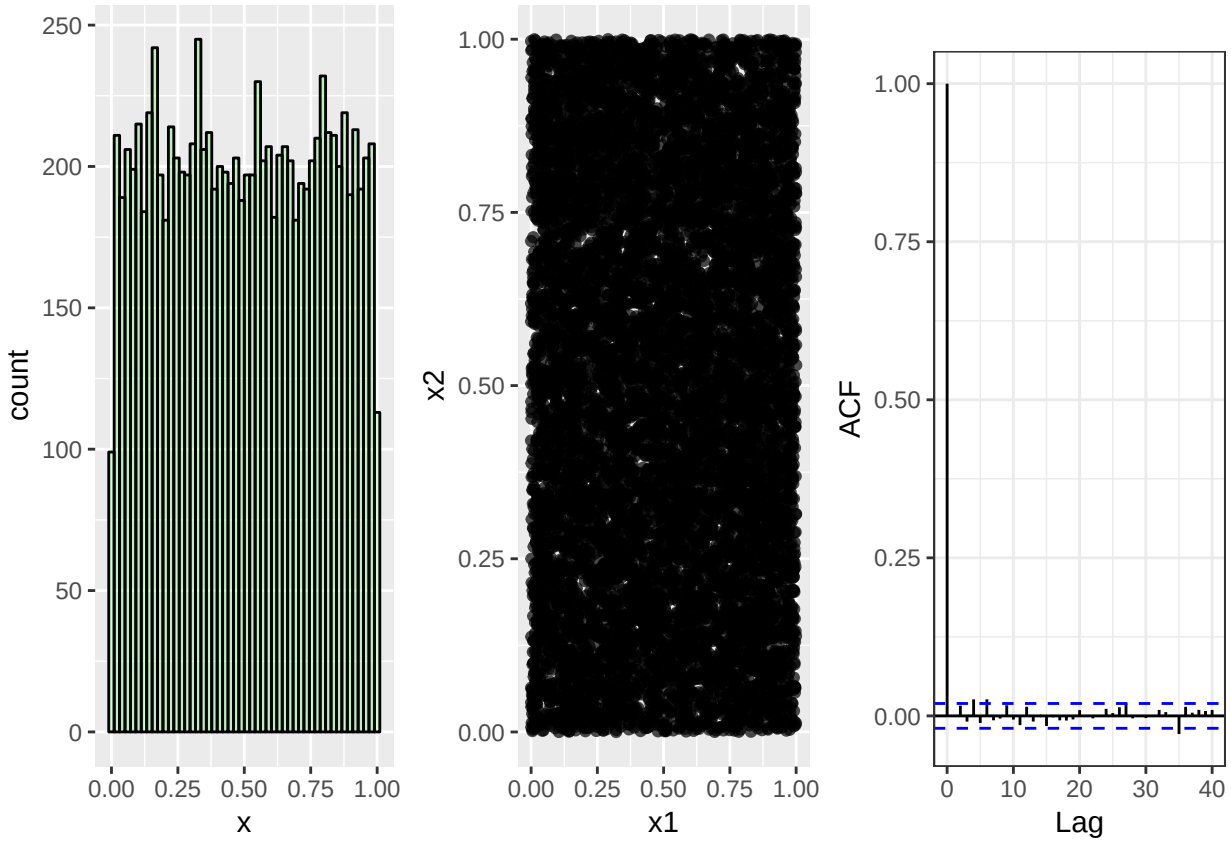
The methods in this book summarized under the denomination of *Monte Carlo Methods* mostly rely on the possibility of producting a supposedly endless flow of random variables for well-known or new distributions. Such a simulation is, in turn, based on the production of uniform random variables on the interval (0, 1). In a sense, the uniform distribution $\sim \mathrm{U}(0,1)$ provides the basic probabilistic representation of randomness on a computer and the generators for all other distributions require a sequence of uniform variables to be simulated.

2.1.1 | Uniform Simulation

```
# helper function to make ACF plots in ggplot
ggacf <- function(x, ci=0.95, type="correlation", xlab="Lag", ylab=NULL,
                  ylim=NULL, main=NULL, ci.col="blue", lag.max=NULL) {

    x <- as.data.frame(x)

    x.acf <- acf(x, plot=F, lag.max=lag.max, type=type)

    ci.line <- qnorm((1 - ci) / 2) / sqrt(x.acf$n.used)

    d.acf <- data.frame(lag=x.acf$lag, acf=x.acf$acf)

    g <- ggplot(d.acf, aes(x=lag, y=acf)) +
        geom_hline(yintercept=0) +
        geom_segment(aes(xend=lag, yend=0)) +
        geom_hline(yintercept=ci.line, color=ci.col, linetype="dashed") +
        geom_hline(yintercept=-ci.line, color=ci.col, linetype="dashed") +
        theme_bw() +
        xlab("Lag") +
        ggtitle(ifelse(is.null(main), "", main)) +
        if (is.null(ylab))
            ylab(ifelse(type=="partial", "PACF", "ACF"))
        else
            ylab(ylab)
```

```
    g
}
```

```
# generator 100 random uniform samples
hunnit <- runif(100, min = 2, max = 5)

# check properties of uniform generator

# create vector of randomly generated uniform vars
nsim <- 10^4
x <- runif(nsim)

# vectors to plot
x_1 <- x[-nsim]

# adjacent pairs
x_2 <- x[-1]

# place into dataframes
x <- tibble(x)
x_n <- tibble(x_1, x_2)

# plot
x_hist <- ggplot(x, aes(x)) + geom_histogram(bins = 50, fill = "green", color = "black", alpha = 0.2)
x_plot <- ggplot(x_n, aes(x_1, x_2)) + geom_point(color = "black", alpha = 0.7) + xlab("x1") + ylab("x2
x_acf <- ggacf(x)
gridExtra::grid.arrange(x_hist, x_plot, x_acf, ncol = 3)
```



From the simulation above, we see that runif is acceptable.

2.1.2 | The Inverse Transform

There is a simple, useful transformation, known as the probability integral transform which allows us to transform any random variable into a uniform variable and vice versa.

For example, if X has density f and cdf F, then we have the relation

$$F(x) = \int_{-\infty}^{x} f(t)dt$$

and if we set $U = F(X)$, then U is a random variable distributed from a uniform $U(0,1)$. This is because

$$P(U \leq u) = P[F(X) \leq F(x)] = P[F^{-1}(F(X)) \leq F^{-1}(F(x))] = P(X \leq x)$$

where we have assumed that F has an inverse.

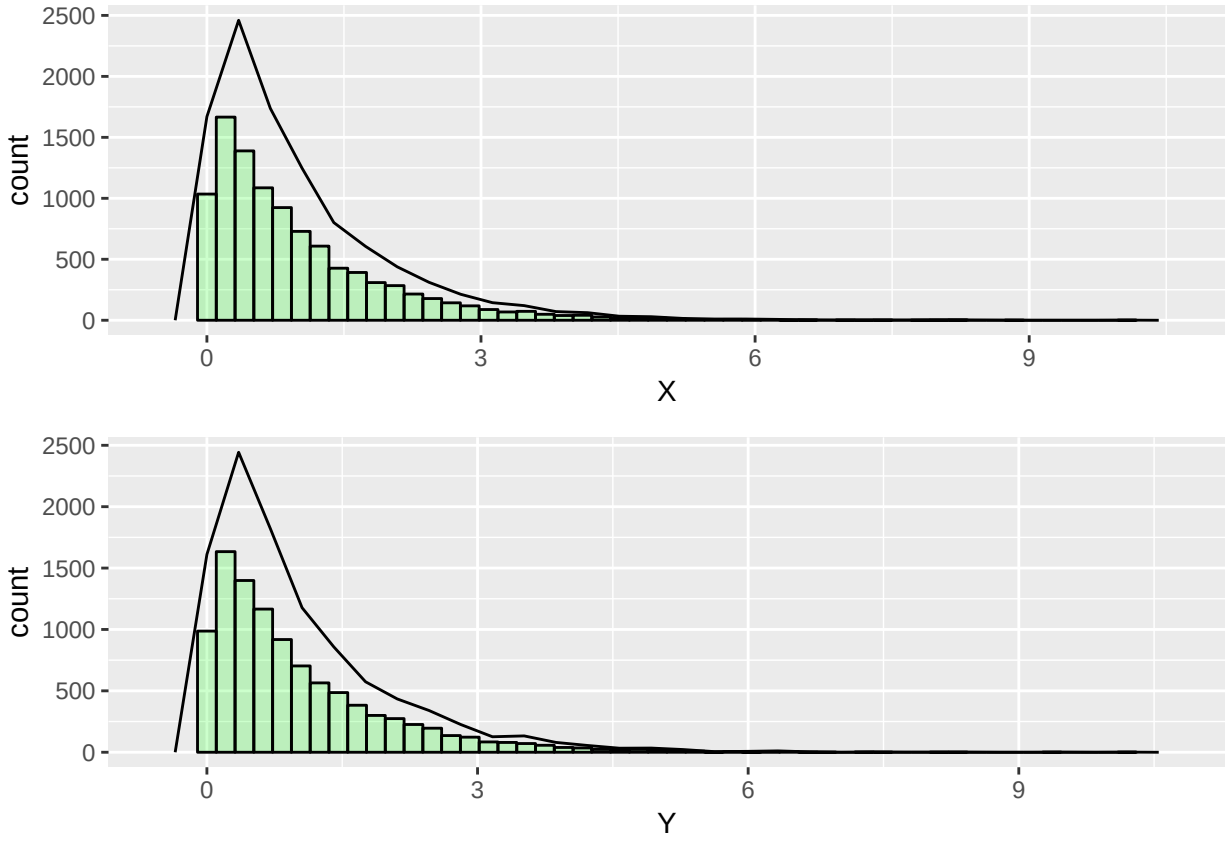
Example 2.1

If $X \sim \text{Exp}(1)$, then $F(x) = 1 - e^{-x}$. Solving for x in $u = 1 - e^{-x}$ gives us $x = -\log(1-u)$. Therefore, if $U \sim U(0,1)$, then $X = -\log U \sim \text{Exp}(1)$ as U and $1-U$ are both uniform.

```
# num random variables
nsim <- 10^4

U <- runif(nsim)

# transformation of uniforms
```

```r
X <- -log(U)

# exponentials
Y <- rexp(nsim)

# transform to data frames
X <- tibble(X)
Y <- tibble(Y)

# plot
unif_hist <- ggplot(X, aes(X)) + geom_histogram(bins = 50, fill = "green", color = "black", alpha = 0.2)
exp_hist <- ggplot(Y, aes(Y)) + geom_histogram(bins = 50, fill = "green", color = "black", alpha = 0.2)
gridExtra::grid.arrange(unif_hist, exp_hist, nrow = 2)
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



The plot above compares the outpit from the probability inverse transform with the output from rexp. The fits of both histograms to their exponential limit are not distinguishable.

2.2 | General Transformation Methods

When a distribution with density f is linked in a relatively simple way to another distribution that is easy to simulate, this relationship can often be exploted to construct an algorithm to simulate variables from f.

Example 2.2

In example 2.1, we saw how to generate an exponential random variable starting from a uniform. Now we illustrate some of the RVs that can be generated starting from an exponential distribution.

For $X_i \sim \text{Exp}(1)$,

$$Y = 2\sum_{j=1}^{v} X_j \sim \chi^2_{2v},\ v \in \mathbb{N}^*$$

$$Y = \beta \sum j = 1^a X_j \sim G(a, \beta),\ a \in \mathbb{N}^*$$

$$Y = \frac{\sum_{j=1}^{a} X_j}{\sum_{j=1}^{a+b} X_j} \sim \beta(a, b),\ a, b \in \mathbb{N}^*$$

where $\mathbb{N}^* = \{1, 2, ...\}$.

```
# generate chi sq 6 dof
U <- runif(3*10^4)

# matrix for sums
U <- matrix(data = U, nrow = 3)

# uniform to exponential
X <- -log(U)

# sum up to get chi squares
X <- 2 * apply(X, 2, sum)
```

2.2.1 | A Normal Generator

One way to achieve normal random variable simulation using a transform is with the Box-Muller algorithm, devised for the generation of $N(0, 1)$ variables.

Example 2.3

If U_1 and U_2 are iid $\mathcal{U}(0, 1)$, the variables X_1, X_2 are defined by

$$X_1 = \sqrt{-2\log(U_1)}\cos(2\pi U_2),\ X_2 = \sqrt{-2\log(U_1)}\sin(2\pi U_2)$$

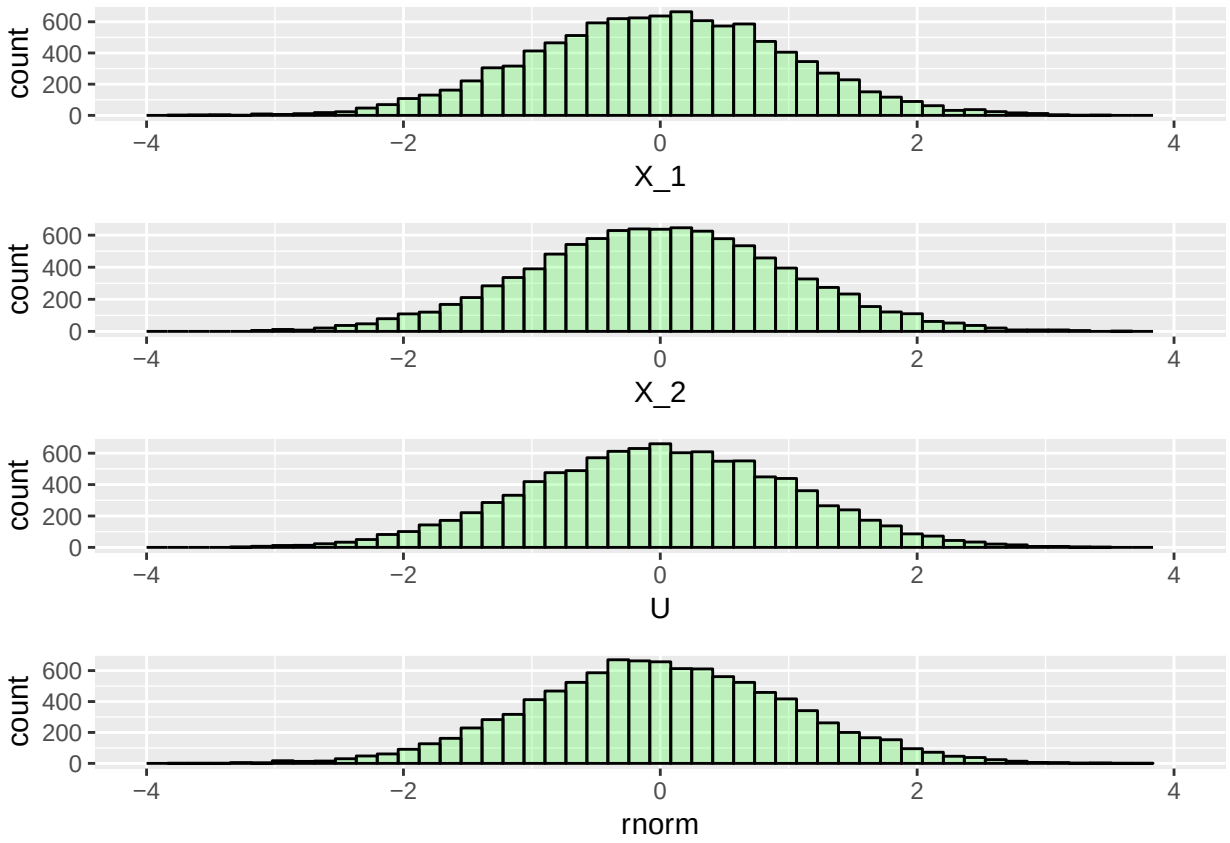
are then iid $\mathcal{N}(0, 1)$ by virtue of a change of variable argument.

```
# exercise 2.3, show that E[Z] = 0 and var(Z) = 1

# generate 12 unif rvs
U <- runif(n = 12, min = -0.5, max = 0.5)

# set z = sum U_i
Z <- sum(U)

# expected value
eU <- U * (1/12)
eU <- sum(eU)
```

```r
# or E[U_i] = (1/2 + (- 1/2)) / 2 = 0, therefore E[Z] = 0 * (1/12 * 12) = 0. Similarly, var(U_i) = (1/1
```

We see that the numerical value is less than the variance of 1/12.

```r
# using histograms, compare this CLT normal generator with the Box-Muller algorithm. Pay particular att

nsim = 10^4
U_1 <- runif(nsim)
U_2 <- runif(nsim)

# transformed unif to norm
X_1 <- sqrt(-2 * log(U_1)) * cos(2 * pi * U_2)
X_2 <- sqrt(-2 * log(U_1)) * sin(2 * pi * U_2)

# rnorm for comparison
reg_norm <- rnorm(n = 10^4, 0, 1)

# placeholder array
U <- array(0, dim = c(nsim))

# fill array with simulated unifs
for (i in 1:nsim){
  U[i] <- sum(runif(12, -0.5, 0.5))
}

# coerce to dataframe
X_1 <- tibble(X_1)
X_2 <- tibble(X_2)
U <- tibble(U)
reg_norm <- tibble(reg_norm)

# plot
x1_hist <- ggplot(X_1, aes(X_1)) + geom_histogram(bins = 50, fill = "green", color = "black",  alpha = 
x2_hist <- ggplot(X_2, aes(X_2)) + geom_histogram(bins = 50, fill = "green", color = "black",  alpha = 
u_hist <- ggplot(U, aes(U)) + geom_histogram(bins = 50, fill = "green", color = "black",  alpha = 0.2) +
reg_norm_hist <- ggplot(reg_norm, aes(reg_norm)) + geom_histogram(bins = 50, fill = "green", color = "bl
gridExtra::grid.arrange(x1_hist, x2_hist, u_hist, reg_norm_hist, nrow = 4)
```



The simulation of a multivariate normal distribution $\mathcal{N}_p(\mu, \Sigma)$, where Σ is a $p \times p$ symmetric and positive-definite matrix, can be derived from the generic rnorm generator in that using a Cholesky decomposition of Σ ($\Sigma = AA^T$) and taking the transform by A of an iid normal vector of dimension p leads t a $\mathcal{N}_p(0, \Sigma)$ normal vector.

There is an R package that replicates these steps called rmnorm available from the mnormt library that allows the computation of the probability of hypercubes via the function sadmvn

```
# make a positive definite matrix for our multivariate norm. This is the identity matrix
identity_matrix <- diag(3)

# generative mvnorm density
mnormt::sadmvn(low = c(1, 2, 3), upper = c(10, 11, 12), mean = rep(0, 3), var = identity_matrix)
```

```
## [1] 4.87236e-06
## attr(,"error")
## [1] 2.367529e-22
## attr(,"status")
## [1] "normal completion"
```

2.2.2 | Discrete Distributions

To generate $X \sim P_\theta$, where P_θ is supported by the integers, we can calculate the probabilities:

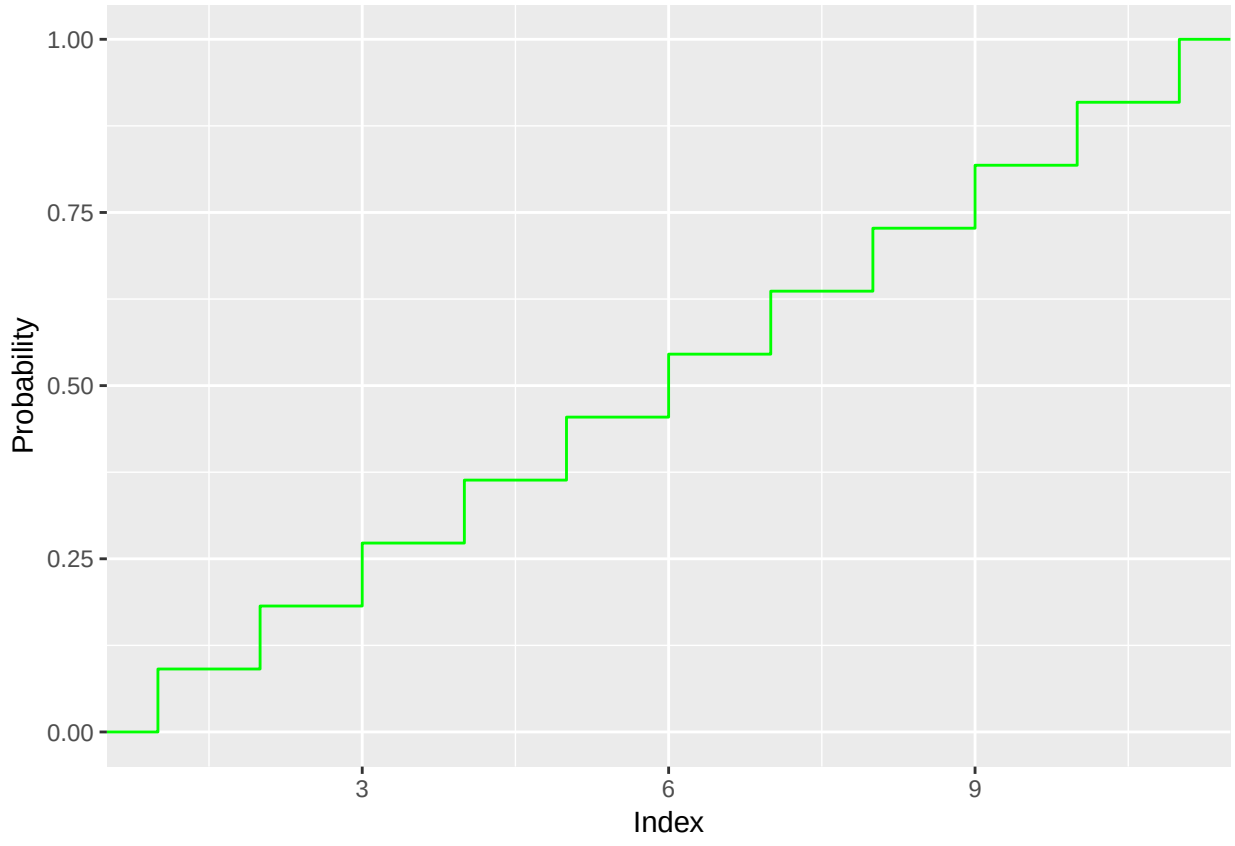
$$p_0 = P_\theta(X \leq 0), p_1 = P_\theta(X \leq 1), ...$$

and then generate $U \sim \mathcal{U}(0, 1)$ and take $X = k$ if $p_{k-1} < U < p_k$

Example 2.4

To generate $X \sim \mathcal{B}\backslash(10, 0.3)$, the probability values are obtained by pbinom(k, 10, 0.3)

```
# make vector to store values
binoms <- c()

# calculate binomial probabilities
for (i in 0:10){
  binoms <- append(binoms, pbinom(i, 10, 0.3))
}

# make into table
binoms <- tibble("probability" = binoms)

# add row number, reorder rows
binoms <- binoms %>% mutate("index" = row_number()) %>% select("index", "probability")


ggplot(binoms,aes(x = binoms$index)) + geom_step(aes(x = binoms$index, y = binoms$probability), color =
```




```
binoms
```

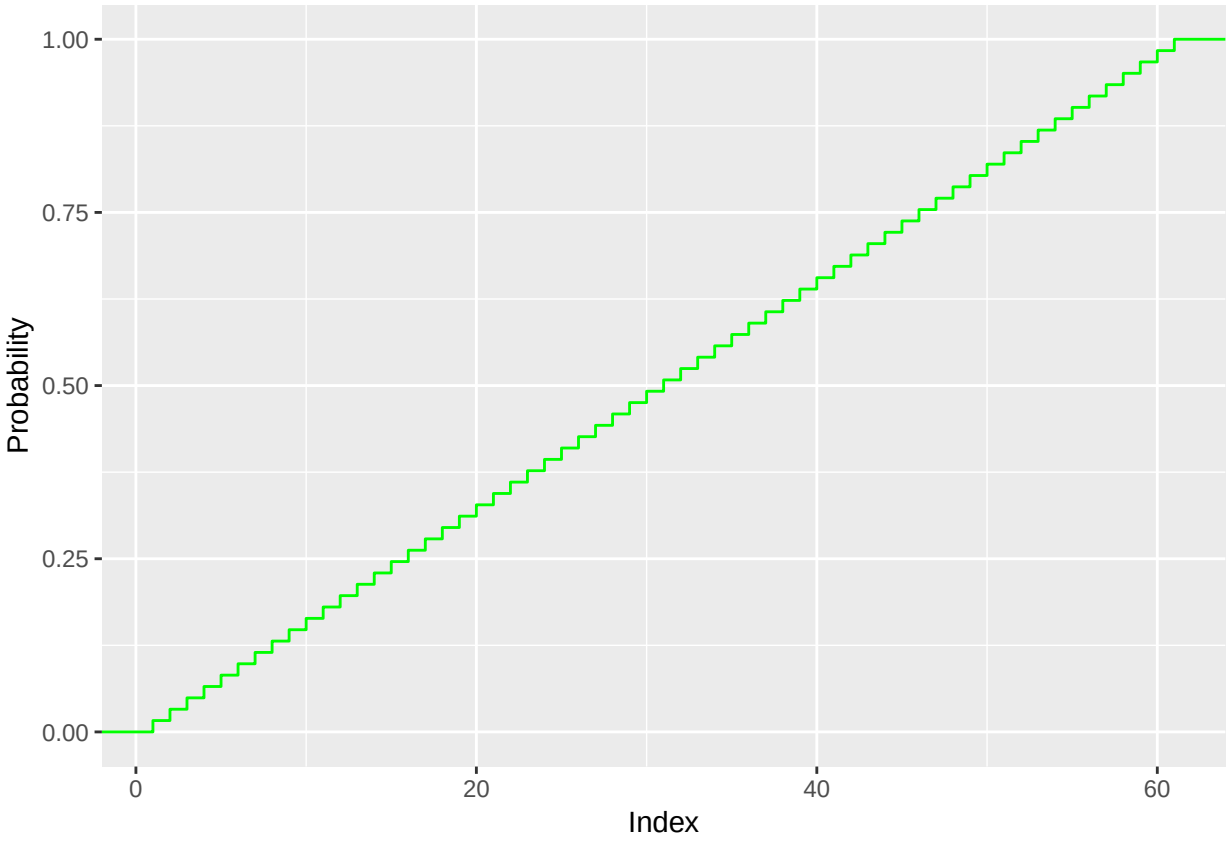
```
## # A tibble: 11 x 2
##    index probability
##    <int>       <dbl>
##  1     1      0.0282
```

```
##  2     2      0.149
##  3     3      0.383
##  4     4      0.650
##  5     5      0.850
##  6     6      0.953
##  7     7      0.989
##  8     8      0.998
##  9     9      1.000
## 10    10      1.000
## 11    11      1
```

If we wished to generate random variables from a Poisson distribution with mean $\lambda = 100$, the above algorithm would be inefficient. We expect most of our observations to be in the interval $\lambda = \pm 3\sqrt{\lambda}$ (λ is the variance and mean for the Poisson). For $\lambda = 100$, this interval is $(70, 130)$. Therefore, if we start at 0, we will almost always produce 70 tests of whether or not $p_{k-1} < U < p_k$ that are useless because they are almost certain to be rejected.

A remedy is to ignore what is outside of a highly likely interval such as $(70, 130)$, as $P(X < 70) + P(X > 130) = 0.00268$.

```r
# generate Poisson RVs for large values of lambda

numsims <- 10^4
lambda <- 100
spread <- 3 * sqrt(lambda)

# generate a sequence of integer values in the range around the mean
lambda_sequence <- round(seq(max(0, lambda - spread), lambda + spread, 1))

# generate probabilities of getting lambda_sequence values
probs <- ppois(lambda_sequence, lambda)

# check to see what interval the uniform RV fell in and assign the corrent Poisson value to X
rvs <- rep(0, numsims)

# notice how it only assigns values to the earliest RVs
for (i in 1:numsims){
  u <- runif(1)
  rvs[i] <- lambda_sequence[i] + sum(probs < u)
}

# plot
probs <- probs %>% tibble("probability" = probs)
probs <- probs %>% mutate("index" = row_number()) %>% select("index", "probability")
ggplot(probs, aes(x = probs$index)) + geom_step(aes(x = probs$index, y = probs$probability), color = "g
```



A more formal remedy to the inefficiency of starting the cumulative probabilities at p_0 is to start instead from the mode of the discrete distribution P_θ and to explore the neighboring values until the cumulative probability is 1 up to an approximation error. The p_k's are then indexed by the visited values rather than by the integers, but the validity of the method remains complete.

2.2.3 | Mixture Representations

Sometimes a probability distribution can be represented as a *mixture distribution*, that is, of the form:

$$f(x) = \int_\gamma g(x|y)p(y)dy \text{ or } f(x) = \sum_{i\in\gamma} p_i f_i(x)$$

depending on whether the auxillary space γ is continuous or discrete, and g and p are standard distributions that are easily simulated.

To generate a RV X using such a representation, we can first generate a variable Y from the mixing distribution and then generate X from the selected conditional distribution. For example:

$$\text{if } y \sim p(y) \text{ and } X \sim f(x|y), \text{ then } X \sim f(x) \text{ (if continuous) if } \gamma \sim P(\gamma = i) = p_i \text{ and } X \sim f_\gamma(x), \text{ then } X \sim f(x) \text{ (if discrete)}$$

For instance, we can write the Student's t density with v degrees of freedom T_v as a mixture where :

$$X|y \sim \mathcal{N}(0, \tfrac{v}{y}) \text{ and } Y \sim \chi^2_v$$

Generating from a T_v distribution could then amount to generating from a χ^2_v distribution, and then from the corresponding normal distribution.

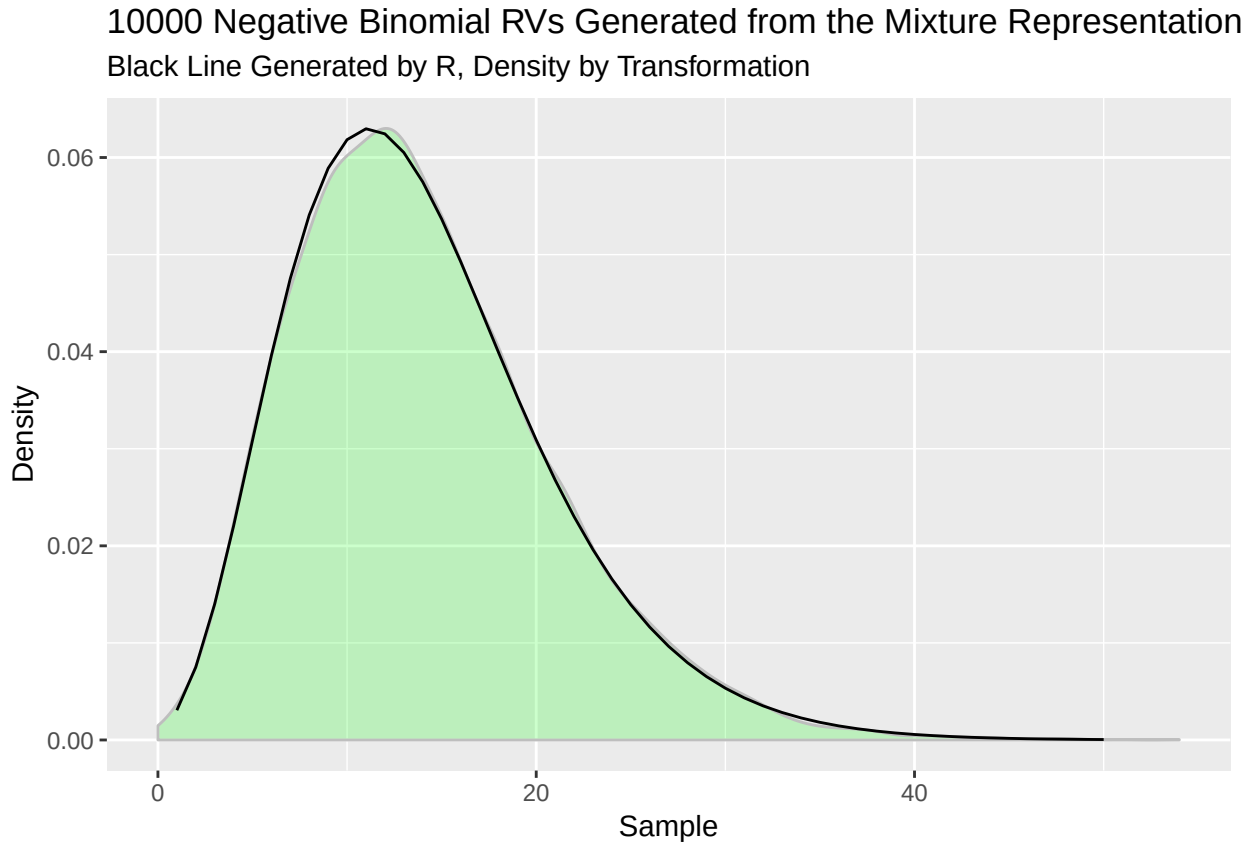
Example 2.6

If X is a negative binomial random variable, $X \sim \mathcal{N}]\}(n, p)$, then X has a mixture representation

\begin{center} $X|y \sim P(y)$ and $Y \sim \mathcal{G}(n, \beta)$\$ \end{center}

where $\beta = \frac{1-p}{p}$.

```
# generate student t with v dof mixture

# parameter assignment
numsims <- 10^4
n <- 6
p <- 0.3

# generate RVs
y <- rgamma(numsims, n, rate = p / (1-p))
x <- rpois(numsims, y)

# generate negative binom for comparison
negbin <- dnbinom(1:50, n, p)

# place in tibble
x <- tibble("sample" = x)
x <- x  %>% mutate("index" = row_number()) %>% select("index", "sample")

negbin <- tibble("probability" = negbin)
negbin <- negbin %>% mutate("index" = row_number()) %>% select("index", "probability")

# plot
ggplot(x, aes(x = x$sample)) + geom_density(fill = "green", color = "gray",  alpha = 0.2) + geom_line(da
  ggtitle("10000 Negative Binomial RVs Generated from the Mixture Representation", subtitle = "Black Li
```



2.3 | Accept-Reject Methods

There are many distributions for which the inverse transform method and even general transformations will fail to be able to generate the required random variables. For these cases, we turn to indirect methods.

These indirect methods are called **Accept-Reject Methods** and they only require us to know the functional form of the density f of interest (called the target density) up to a multiplicative constant. We will use a simpler to simulate density g, called the *instrumental* or *candidate* density to generate the random variable for which the simulation is actually done.

We impose the following constaints on g:

f and g have compatible supports (i.e. g(x) $>$ 0 when f(x) $>$ 0)

There is a constant M with $\frac{f(x)}{g(x)} \leq M$ for all x

In this case, X can be simulated as follows:

$$\text{Generate } Y \sim g,\ U \sim \mathcal{U}(0,1) \text{ Accept } X = Y \text{ if } U \leq \frac{1}{M}\frac{f(Y)}{g(Y)} \text{ Return to 1 otherwise}$$

Here is a generalized implementation, in which randg() is a function that delivers generations from the density g.

```
# u <- runif(1) * M
# y <- randg(1)

# while(u > f(y)/g(y)){
#   u <- runif(1) * m
```

```
#   y <- randg(1)
# }
```

Which produces a single generation y from f

Example 2.7

Suppose we wish to simulate $\mathcal{B}\rceil\sqcup\dashv(\alpha,\beta)$ random variables where the instrumental distribution is the $\mathcal{U}(0,1)$ distribution and α, β are both larger than 1 (the rbeta function does not impose this restriction).

The upper bound M is then the maximum of the beta density

```
# find max of beta
M_const <- optimize(f = function(x){dbeta(x, 2.7, 6.3)}, interval = c(0, 1), maximum = TRUE)
objective_M <- M_const$objective
```

Since the candidate density g is equal to 1, the proposed value Y is accepted if $M \times U$ is under the beta density f at that realization.

Note that generating $U \sim \mathcal{U}(0,1)$ and multiplying it by M is equivalent to generating $U \sim \mathcal{U}(0,M)$.

For $\alpha = 2.7$, $\beta = 6.3$, an alternative R implementation of the Accept-Reject algorithm is

```
# declare parameters
numsims <- 10^5
a <- 2.7
b <- 6.3

# generate distributions

# uniform over (0, M)
u <- runif(numsims, max = objective_M)

# generation from g
y <- runif(numsims)

# accepted subsample
x <- y[u < dbeta(y, a, b)]

# percentage accepted
length(x) / 10^5
```

```
## [1] 0.37448
```

In the implementation above, our numsims is fixed. This makes our accepted values $\sim \text{Bin}(n, \frac{1}{M})$. Instead of this, in most cases the number of accepted values is fixed. This can be exploited as in

```
x <- NULL

while (length(x) < numsims){
  y <- runif(numsims * objective_M)
  x <- c(c, y[runif(numsims * objective_M) * objective_M < dbeta(y, a, b)])
}

x <- x[1:numsims]
```

Some key properties of the Accept-Reject algorithm which should always be considered when using it are the following:

** Only the ratio $\frac{f}{M}$ is needed, so the algorithm does not depend on the normalizing constant ** The bound $f \leq Mg$ need not be tight; the algorithm remains valid (if less efficient) when M is replaced with any larger constant ** The probability of acceptance is $\frac{1}{M}$, so M should be as small as possible for a given computational effort.

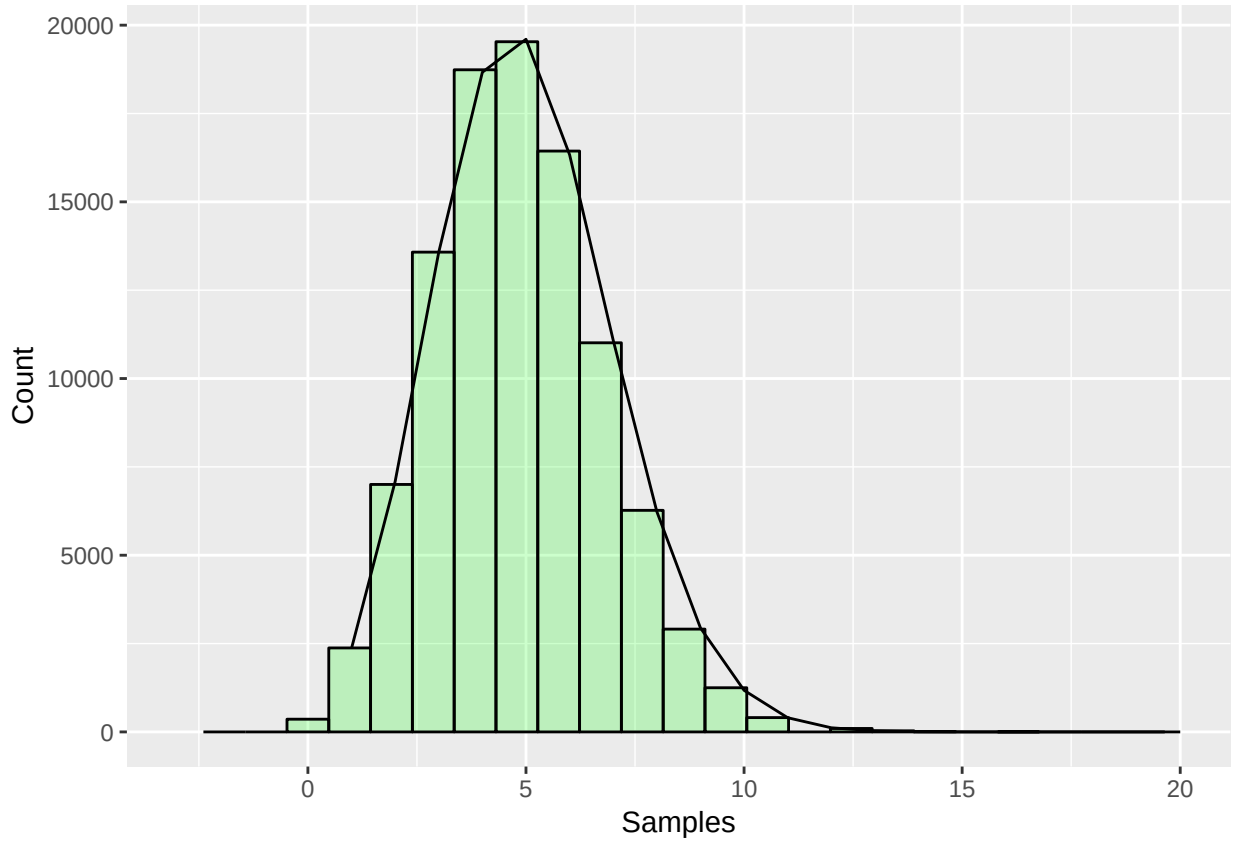
2.4 | Additional Exercises

2.11

In both questions, the comparison between generators is understood in terms of efficiency via the system.time function.

A Generate a binomial $\mathcal{B}\backslash(n, p)$ random variable with $n = 25$, $p = 0.2$. Plot a histogram for your simulated sample and compare it with the binomial mass function. Compare your generator with the R binomial generator.

```
# declare parameters
numsims <- 10^5
n <- 25
p <- 0.2

# binomial RV
binoms <- pbinom(c(0:n), n, p)

# placeholder array
genned_x <- array(0, numsims)

# generate cdf
for (i in 1:numsims){
  u <- runif(1)
  genned_x[i] <- sum(binoms < u)
}

genned_x <- tibble("sample" = genned_x) %>% mutate("index" = row_number()) %>% select("index", "sample"

comparison_binom <- dbinom(1:n, n, p)
comparison_binom <- tibble("probability" = comparison_binom) %>% mutate("index" = row_number()) %>% sel

ggplot(genned_x, aes(x = genned_x$sample)) + geom_histogram(bins = 25, fill = "green", color = "black",
```



2.15

The Poisson Distribtion $P(\lambda)$ is connected to the exponential distribution through the Poisson process in that it can be simulated by generating exponential random variables until their sum exceeds 1. That is, if $X_i \sim \mathcal{E}\S(\lambda)$ and if K is the first value for which $\sum_{i=1}^{K+1} X_i > 1$, then $K \sim P(\theta)$. Compare this algorithm with rpois and the algorithm of example 2.5 for both small and large values of λ

```
# poisson 1 through transformations
pois_1 <- function(numsims, lambda){
  spread <- 3 * sqrt(lambda)
  lambda_samples <- round(seq(max(0, lambda - spread), lambda + spread, 1))
  pois_probs <- ppois(lambda_samples, lambda)
  approx <- rep(0, numsims)
  for (i in 1:numsims){
    u <- runif(1)
    approx[i] = max(lambda_samples[1], 0) + sum(pois_probs < u) - 1
  }
  return(approx)
}

# poisson 2 through exponential generators
pois_2 <- function(numsims, lambda){
  approx <- rep(0, numsims)
  for (i in 1:numsims){
```

```
    sum <- 0
    k <- 1
    sum <- sum + rexp(1, lambda)
    while (sum < 1){
      sum <- sum + rexp(1, lambda)
      k <- k + 1
    }
    approx[i] <- k
  }
  return(approx)
}

# test functions against each other
numsims <- 10^4
lambda <- 10

system.time(pois_1(numsims, lambda))
```

```
##    user  system elapsed
##   0.067   0.000   0.066
```

```
system.time(pois_2(numsims, lambda))
```

```
##    user  system elapsed
##   0.428   0.000   0.429
```

```
system.time(rpois(numsims, lambda))
```

```
##    user  system elapsed
##   0.002   0.000   0.002
```