Lecture 8: Random Number Generation and Simulations

STAT UN2102 *Applied Statistical Computing*

Gabriel Young
Columbia University

March 27th, 2018

Topics for Today

- **Random Number Generation**. Random numbers in `R` and the linear congruential generator.
- **Simulation**.
  - Simulating random variables using `R` base functions.
  - The `sample()` function to simulate discrete random variables.
  - Acceptance-rejection algorithm.

Random Number Generation

Random Number Generation

We've made references to random number generation throughout the course without understanding where they come from.

Random Number Generation

We've made references to random number generation throughout the course without understanding where they come from.

Today's Lecture

- How does R produce random numbers?
- It doesn't!
- R uses tricks that generate **pseudorandom numbers** that are indistinguishable from real random numbers.

Pseudorandom generators produce a deterministic sequence that is indistinguishable from a true random sequence if you don't know how it started.

Random Number Generation

Random Numbers in R

There are many ways to generate random numbers in R. Below we generate 10 random variables distributed uniformly over the unit interval.

```
> runif(10)
```

```
 [1] 0.2756871 0.5916548 0.2601711 0.3175934 0.4632539
 [6] 0.6562926 0.7128874 0.8610900 0.2065024 0.3168160
```

Random Number Generation

Random Numbers in R

There are many ways to generate random numbers in R. Below we generate 10 random variables distributed uniformly over the unit interval.

```
> runif(10)
```

```
 [1] 0.2756871 0.5916548 0.2601711 0.3175934 0.4632539
 [6] 0.6562926 0.7128874 0.8610900 0.2065024 0.3168160
```

On your machine, you'll see different random numbers.

Random Number Generation

Random Numbers in R

To recreate the same random numbers, use the function `set.seed()`.

```
> set.seed(10)
> runif(10)
```

```
 [1] 0.50747820 0.30676851 0.42690767 0.69310208 0.08513597
 [6] 0.22543662 0.27453052 0.27230507 0.61582931 0.42967153
```

Random Number Generation

Random Numbers in R

To recreate the same random numbers, use the function `set.seed()`.

```
> set.seed(10)
> runif(10)
```

```
 [1] 0.50747820 0.30676851 0.42690767 0.69310208 0.08513597
 [6] 0.22543662 0.27453052 0.27230507 0.61582931 0.42967153
```

Try it again.

```
> set.seed(10)
> runif(10)
```

```
 [1] 0.50747820 0.30676851 0.42690767 0.69310208 0.08513597
 [6] 0.22543662 0.27453052 0.27230507 0.61582931 0.42967153
```

Random Number Generation

Linear Congruential Generator (LCG)

A **Linear Congruential Generator (LCG)** is an algorithm that produces a sequence of pseudorandom numbers based on the recurrence relation formula:

$$X_n = (aX_{n-1} + c) \mod m$$

Random Number Generation

Linear Congruential Generator (LCG)

A **Linear Congruential Generator (LCG)** is an algorithm that produces a sequence of pseudorandom numbers based on the recurrence relation formula:

$$X_n = (aX_{n-1} + c) \mod m$$

Simulating from [0,1]

- The 1^{st} number is produced from a seed, and then used to generate the 2^{nd}. The 2^{nd} value is used to generate the 3^{rd}, and so on.
- Values are always between 0 and $m-1$, and the sequence repeats every m occurrences.
- Dividing by the m gives you uniformly distributed random numbers between 0 and 1 (but never quite hitting 1).
- The LCG algorithm motivates how we can simulate a sequence of pseudorandom numbers from the unit interval.

Random Number Generation

Linear Congruential Generator (LCG)

A **Linear Congruential Generator (LCG)** is an algorithm that produces a sequence of pseudorandom numbers based on the recurrence relation formula:

$$X_n = (aX_{n-1} + c) \mod m$$

Simulating from [0,1]

- The LCG is a *pseudorandom* number generator because after a while, the sequence in the stream of numbers will begin to repeat.
- More sophisticated variants of the LCD exist.

Simple Code Example

```
> seed <- 10
> new.random <- function(a = 5, c = 12, m = 16) {
+   out <- (a*seed + c) %% m
+   seed <<- out
+   return(out)
+ }
```

Random Number Generation

Simple Code Example

```
> seed <- 10
> new.random <- function(a = 5, c = 12, m = 16) {
+   out <- (a*seed + c) %% m
+   seed <<- out
+   return(out)
+ }
```

Remember function environments?

The symbol $<<-$ allows you to assign a new global variable in a local environment.

Random Number Generation

Simple Code Example

```
> seed <- 10
> new.random <- function(a = 5, c = 12, m = 16) {
+   out <- (a*seed + c) %% m
+   seed <<- out
+   return(out)
+ }
```

Modular Arithmetic

Modular arithmetic is performed using the symbol %%.

```
> 4 %% 4; 4 %% 3
```

```
[1] 0
```

```
[1] 1
```

Try it out..

```
> out.length <- 20
> variants   <- rep(NA, out.length)
> for (i in 1:out.length) {variants[i] <- new.random()}
> variants
```

```
 [1] 14  2  6 10 14  2  6 10 14  2  6 10 14  2  6 10 14  2
[19]  6 10
```

Random Number Generation

Try it out..

```
> out.length <- 20
> variants   <- rep(NA, out.length)
> for (i in 1:out.length) {variants[i] <- new.random()}
> variants
```

```
 [1] 14  2  6 10 14  2  6 10 14  2  6 10 14  2  6 10 14  2
[19]  6 10
```

- The generator shuffled some of the integers $0, 1, ..., m-1 = 15$ into an "unpredictable" order.
- Want the generator to shuffle all of these integers, but this generator only gives 4.

Random Number Generation

Try it again with different inputs...

```
> out.length <- 20
> variants   <- rep(NA, out.length)
> for (i in 1:out.length) {
+   variants[i] <- new.random(a = 131, c = 7, m = 16)
+ }
> variants
```

```
 [1]  5  6  9  2 13 14  1 10  5  6  9  2 13 14  1 10  5  6
[19]  9  2
```

Random Number Generation

Try it again with different inputs...

```
> out.length <- 20
> variants   <- rep(NA, out.length)
> for (i in 1:out.length) {
+   variants[i] <- new.random(a = 131, c = 7, m = 16)
+ }
> variants
```

```
 [1]  5  6  9  2 13 14  1 10  5  6  9  2 13 14  1 10  5  6
[19]  9  2
```

A bit better by making sure *c* and *m* are relatively prime.

One more try...

```
> out.length <- 20
> variants   <- rep(NA, out.length)
> for (i in 1:out.length) {
+   variants[i] <- new.random(a = 129, c = 7, m = 16)
+ }
> variants
```

```
 [1]  9  0  7 14  5 12  3 10  1  8 15  6 13  4 11  2  9  0
[19]  7 14
```

Random Number Generation

What Actually Gets Used...

```
> out.length <- 20
> variants   <- rep(NA, out.length)
> for (i in 1:out.length) {
+   variants[i] <- new.random(a=1664545, c=1013904223,
+                             m=2^32)
+ }
> variants/2^(32)
```

```
 [1] 0.2414938 0.4868097 0.9560252 0.1789021 0.8930807
 [6] 0.3094601 0.4947667 0.6213101 0.8339265 0.4841096
[11] 0.4813287 0.5115348 0.8728538 0.6784677 0.1766823
[16] 0.9381840 0.6604821 0.3395404 0.5585955 0.6441623
```

Random Number Generation

What Actually Gets Used...

```
> out.length <- 20
> variants   <- rep(NA, out.length)
> for (i in 1:out.length) {
+   variants[i] <- new.random(a=1664545, c=1013904223,
+                             m=2^32)
+ }
> variants/2^(32)
```

```
 [1] 0.2414938 0.4868097 0.9560252 0.1789021 0.8930807
 [6] 0.3094601 0.4947667 0.6213101 0.8339265 0.4841096
[11] 0.4813287 0.5115348 0.8728538 0.6784677 0.1766823
[16] 0.9381840 0.6604821 0.3395404 0.5585955 0.6441623
```

Type `?Random` to get more info on random number generators used in `R`.

Simulating Random Variables

Simulation

A stochastic model (probabilistic model) can give the distribution of some random variable Y. This random variable can be a complicated multivariate object with many independent components.

Why Do We Care About Simulation?

- To understand a model.
- To check a model.
- To fit a model.

Why Do We Care About Simulation?

To Understand a Model:

- Simulate model output. Simulate model accuracy and precision.
- Simulate how a hypothesis testing procedure behaves under H_0 and under H_A. Do the empirical results match the developed theory?
- Simulate the sampling distribution and variation of an estimator. Assume some parametric form on the model or use nonparametric methods such as the bootstrap procedure or permutation tests.

Why Do We Care About Simulation?

To Understand a Model:

- Simulate model output. Simulate model accuracy and precision.
- Simulate how a hypothesis testing procedure behaves under H_0 and under H_A. Do the empirical results match the developed theory?
- Simulate the sampling distribution and variation of an estimator. Assume some parametric form on the model or use nonparametric methods such as the bootstrap procedure or permutation tests.

To Check a Model:

- Cross-Validation.
- Simulated data from a stochastic model should resemble the real data.

Why Do We Care About Simulation?

To Understand a Model:

- Simulate model output. Simulate model accuracy and precision.
- Simulate how a hypothesis testing procedure behaves under H_0 and under H_A. Do the empirical results match the developed theory?
- Simulate the sampling distribution and variation of an estimator. Assume some parametric form on the model or use nonparametric methods such as the bootstrap procedure or permutation tests.

To Check a Model:

- Cross-Validation.
- Simulated data from a stochastic model should resemble the real data.

To Fit a Model:

- Markov Chain Monte Carlo Methods (MCMC).

Simulating from Probability Distributions

How do we simulate from a probability distribution?

Simulating from Probability Distributions

How do we simulate from a probability distribution?

There are many ways...

- **Common Distributions**: Use built-in `R` functions (normal, gamma, Poisson, binomial, etc..).
- **Uncommon Distributions**: Need to use simulation.

Simulating from Probability Distributions

How do we simulate from a probability distribution?

There are many ways...

- **Common Distributions**: Use built-in `R` functions (normal, gamma, Poisson, binomial, etc..).
- **Uncommon Distributions**: Need to use simulation.
  - **Discrete random variables**: Often can use `sample()`.
  - **Continuous random variables**: Can use *inverse transform method* and the *acceptance-rejection method* otherwise. **Today we briefly discuss the acceptance-rejection method.**

Simulating from Probability Distributions

How do we simulate from a probability distribution?

There are many ways...

- **Common Distributions**: Use built-in `R` functions (normal, gamma, Poisson, binomial, etc..).
- **Uncommon Distributions**: Need to use simulation.
  - **Discrete random variables**: Often can use `sample()`.
  - **Continuous random variables**: Can use *inverse transform method* and the *acceptance-rejection method* otherwise. **Today we briefly discuss the acceptance-rejection method.**

Simulating from Probability Distributions

For common distributions, `R` has many built-in functions for simulating and working with random variables. These functions allow us to:

- Plot density functions (**look up pmfs and pdfs if needed**),
- Compute probabilities,
- Compute quantiles,
- Simulate random draws from the distribution.

R Commands for Distributions

R Commands

- `dfoo` is the probability density function (pdf) or probability mass function (pmf) of **foo**.
- `pfoo` is the cumulative probability function (cdf) of **foo**.
- `qfoo` is the quantile function (inverse cdf) of **foo**.
- `rfoo` draws random numbers from **foo**.

R Commands for Distributions

R Commands

- `dfoo` is the probability density function (pdf) or probability mass function (pmf) of **foo**.
- `pfoo` is the cumulative probability function (cdf) of **foo**.
- `qfoo` is the quantile function (inverse cdf) of **foo**.
- `rfoo` draws random numbers from **foo**.

Normal Density

```
> dnorm(0, mean = 0, sd = 1)
```

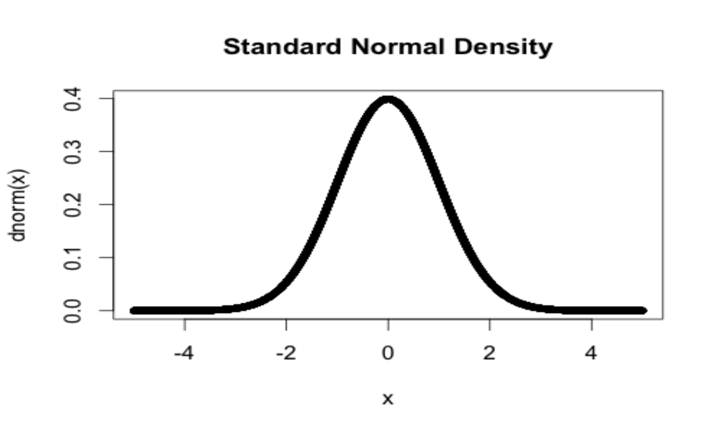
```
[1] 0.3989423
```

```
> 1/sqrt(2*pi)
```

```
[1] 0.3989423
```

Normal Density

```
> x <- seq(-5, 5, by = .001)
> plot(x, dnorm(x), main="Standard Normal Density", pch=20)
```



R Commands for Distributions

Normal CDF

```
> # P(Z < 0)
> pnorm(0)
```

```
[1] 0.5
```

```
> # P(-1.96 < Z < 1.96)
> pnorm(1.96) - pnorm(-1.96)
```

```
[1] 0.9500042
```

R Commands for Distributions

Normal Quantiles

```
> # P(Z < ?) = 0.5
> qnorm(.5)
```

```
[1] 0
```

```
> # P(Z < ?) = 0.975
> qnorm(.975)
```

```
[1] 1.959964
```

R Commands for Distributions

Draw Standard Normal RVs

```
> rnorm(1)
```

```
[1] 0.3897943
```

```
> rnorm(5)
```

```
[1] -1.2080762 -0.3636760 -1.6266727 -0.2564784  1.1017795
```

```
> rnorm(10, mean = 100, sd = 1)
```

```
 [1] 100.75578  99.76177 100.98744 100.74139 100.08935
 [6]  99.04506  99.80485 100.92552 100.48298  99.40369
```

R Base Distributions

Set I

Probability distribution	Functions
Beta	pbeta, qbeta, dbeta, rbeta
Binomial	pbinom, qbinom, dbinom, rbinom
Cauchy	pcauchy, qcauchy, dcauchy, rcauchy
Chi-Square	pchisq, qchisq, dchisq, rchisq
Exponential	pexp, qexp, dexp, rexp
F	pf, qf, df, rf
Gamma	pgamma, qgamma, dgamma, rgamma
Geometric	pgeom, qgeom, dgeom, rgeom
Hypergeometric	phyper, qhyper, dhyper, rhyper

- Access the R help documentation to look up all arguments for each function: `?pbeta`, `?qbeta`, `?dbeta`, `?rbeta`

R Base Distributions

Set II

Probability Distribution	Functions
Logistic	plogis, qlogis, dlogis, rlogis
Log Normal	plnorm, qlnorm, dlnorm, rlnorm
Negative Binomial	pnbinom, qnbinom, dnbinom, rnbinom
Normal	pnorm, qnorm, dnorm, rnorm
Poisson	ppois, qpois, dpois, rpois
Student T	pt, qt, dt, rt
Studentized Range	ptukey, qtukey, dtukey, rtukey
Uniform	punif, qunif, dunif, runif
Weibull	pweibull, qweibull, dweibull, rweibull

- Access the `R` help documentation to look up all arguments for each function: `?pt`, `?qt`, `?dt`, `?rt`

Student's t

Example

- Plot the density function of the student's t distribution with $df = 1, 2, 5, 30, 100$. Use different line types for the different degrees of freedom.
- Plot the standard normal density on the same figure. Plot this curve in red.

Student's t

Example

- Plot the density function of the student's t distribution with $df = 1, 2, 5, 30, 100$. Use different line types for the different degrees of freedom.
- Plot the standard normal density on the same figure. Plot this curve in red.

Fun fact!

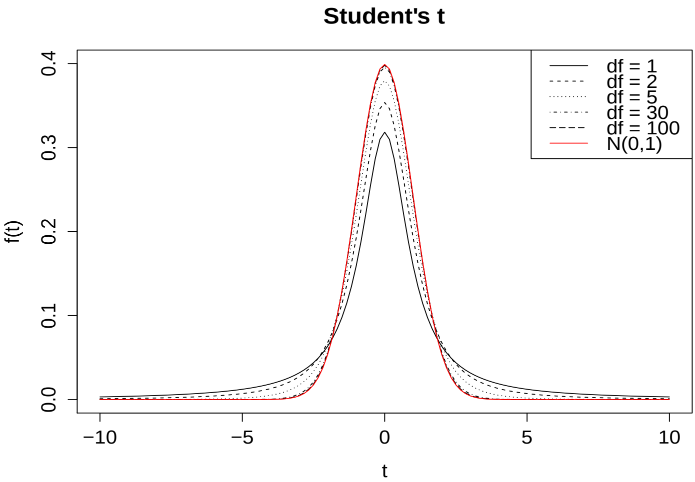
Recall that the student's t distribution converges to a standard normal distribution as $df \to \infty$.

Student's t

Solution

```
> t  <- seq(-10, 10, by = .01)
> df <- c(1, 2, 5, 30, 100)
> plot(t, dnorm(t), lty = 1, col = "red", ylab = "f(t)",
+      main = "Student's t")
> for (i in 1:5) {
+   lines(t, dt(t, df = df[i]), lty = i)
+ }
> legend <- c(paste("df=", df, sep = ""), "N(0,1)")
> legend("topright", legend = legend, lty = c(1:5, 1),
+        col = c(rep(1, 5), 2))
```

Student's t



Student's t
df = 1
df = 2
df = 5
df = 30
df = 100
N(0,1)
f(t)
t
0.0
0.1
0.2
0.3
0.4
−10
−5
0
5
10

Check Yourself

Tasks

Recall that the gamma density function is:

$$f(x|\alpha,\beta) = \frac{x^{\alpha-1}e^{-x/\beta}}{\Gamma(\alpha)\beta^{\alpha}}, \quad 0 < x < \infty, \quad \alpha > 0, \quad \beta > 0,$$

where α is the shape parameter and β is the scale parameter.

- For $\alpha = 2$ and $\beta = 1$ compute

$$Pr\,(X > 2) = \text{Area under curve from 2 to infinity}$$

- For the calculus savvy students:

$$Pr\,(X > 2) = \int_2^{\infty} f(x|\alpha,\beta)\,dx$$

- Plot the gamma density using shape parameters $\alpha = 2, 3, 4, 5, 6$.

Check Yourself

Solutions

Want to calculate

$$Pr\left(X > 2\right),$$

where $X \sim Gamma(\alpha = 2, \beta = 1)$.

```
> pgamma(2, shape = 2, rate = 1) # P(0 < X < 2)
```

```
[1] 0.5939942
```

```
> 1 - pgamma(2, shape = 2, rate = 1) # P(X > 2)
```

```
[1] 0.4060058
```

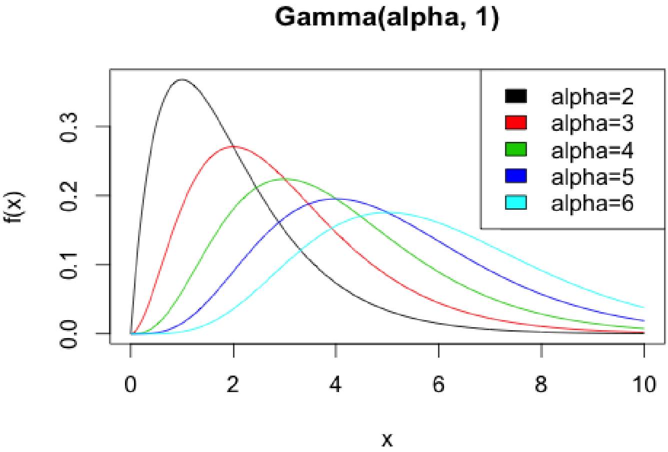
What about $Pr(X = 2)$?

Check Yourself

Solutions

```
> alpha <- 2:6
> beta  <- 1
> x     <- seq(0, 10, by = .01)
> plot(x, dgamma(x, shape = alpha[1], rate = beta),
+      col = 1, type = "l", ylab = "f(x)",
+      main = "Gamma(alpha, 1)")
> for (i in 2:5) {
+   lines(x, dgamma(x, shape = alpha[i], rate = beta),
+         col = i)
+ }
> legend <- paste("alpha=", alpha, sep = "")
> legend("topright", legend = legend, fill = 1:5)
```

Check Yourself



Check Yourself

Tasks

Let $X \sim Binom(n, p)$. For large n, recall the normal approximation to the binomial distribution:

$$P(X \leq x) \approx \Phi\left(\frac{x + .5 - np}{\sqrt{np(1-p)}}\right),$$

where $\Phi(z)$ is the cdf of the standard normal distribution.

- Let $X \sim Binom(n = 1000, p = 0.20)$. Using the normal approximation to the binomial distribution, compute the approximate probability $P(X \leq 190)$.
- Calculate the exact probability $P(X \leq 190)$.
- Let $X \sim Binom(n = 1000, p = 0.20)$. Simulate 500 realizations of X and create a histogram (or bargraph) of the values.

Check Yourself

Solution

- The approximation is given by

$$P(X \le 190) \approx \Phi\left(\frac{190 + .5 - (1000)(0.20)}{\sqrt{(1000)(0.20)(0.80)}}\right),$$

```
> val <- 190
> n   <- 1000
> p   <- 0.20
> correction <- (val + 0.5 - n*p)/(sqrt(n*p*(1-p)))
> pnorm(correction) # P(Z < correction)
```

```
[1] 0.226314
```

Check Yourself

Solution

- ```
  > # P(X <= 190)
  > pbinom(val, size = n, prob = p)
  ```

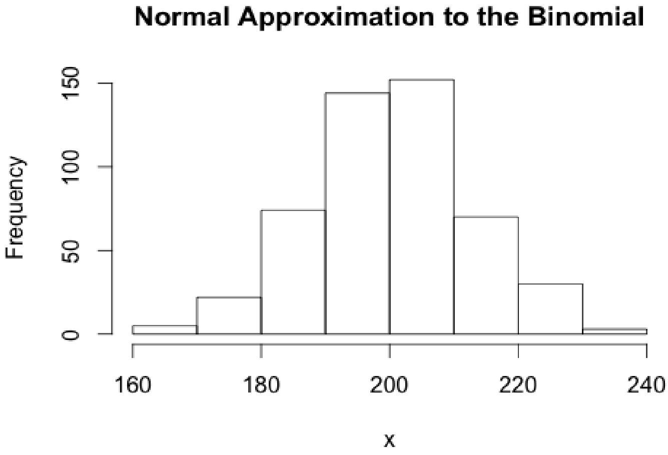
  ```
  [1] 0.2273564
  ```

  ```
  >  # P(x = 0) + P(X = 1) + ... + P(X = 190)
  > sum(dbinom(0:val, size = n, prob = p))
  ```

  ```
  [1] 0.2273564
  ```

- ```
  > x <- rbinom(500, size = n, prob = p)
  > hist(x, main = "Normal Approximation to the Binomial")
  ```

Check Yourself



Check Yourself

Tasks

Draw the following random variables. In each case calculate their sample mean, sample variance, and range (max minus min). Are the sample statistics (mean, variance, range) what you'd expect?

- 5000 normal random variables, with mean 1 and variance 8
- 4000 t random variables, with 5 degrees of freedom
- 3500 Poisson random variables, with mean 4
- 999 chi-squared random variables, with 11 degrees of freedom
- 2000 uniform random variables, between $-\sqrt{12}/2$ and $\sqrt{12}/2$

Repeat the above. This is just to emphasize the (obvious!) point: each time you generate random numbers in `R`, you get different results.

Simulating from Probability Distributions

How do we simulate from a probability distribution?

There are many ways...

- **Common Distributions**: Use built-in `R` functions (normal, gamma, Poisson, binomial, etc..).
- **Uncommon Distributions**: Need to use simulation.
  - **Discrete random variables**: Often can use `sample()`.
  - **Continuous random variables**: Can use *inverse transform method* and the *acceptance-rejection method* otherwise. **Today we briefly discuss the acceptance-rejection method.**

`sample()` Function

We use of the `sample()` function to sample from

1. The discrete uniform distribution.
2. Uncommon discrete distributions (by specifying the probabilities)

Form: `sample(x, size, replace = FALSE, prob = NULL)`

`sample()` Function

We use of the `sample()` function to sample from

1. The discrete uniform distribution.
2. Uncommon discrete distributions (by specifying the probabilities)

Form: `sample(x, size, replace = FALSE, prob = NULL)`

Recall,

We used the sample function in the **bootstrap** procedure.

`sample()` Function

We'd like to generate rvs from the following discrete distribution:

x	1	2	3
$f(x)$	0.1	0.2	0.7

sample() Function

We'd like to generate rvs from the following discrete distribution:

x	1	2	3
$f(x)$	0.1	0.2	0.7

```
> n <- 1000; p <- c(0.1, 0.2, 0.7)
> x <- sample(1:3, size = n, prob = p, replace = TRUE)
> head(x, 10)
```

```
 [1] 3 3 3 3 3 3 2 2 3 3
```

```
> rbind(p, p.hat = table(x)/n)
```

```
          1     2     3
p     0.100 0.200 0.700
p.hat 0.094 0.201 0.705
```

Check Yourself

Tasks

- Use `sample()` to simulate 100 fair die rolls.
- Use `runif()` to simulate 100 fair die rolls. You may also want to use something like `round()`.

Check Yourself

Solution

- ```
  > n     <- 100
  > rolls <- sample(1:6, n, replace = TRUE)
  > table(rolls)
  ```

  ```
  rolls
   1  2  3  4  5  6
  21 12 22 15 16 14
  ```

- ```
  > rolls <- floor(runif(n, min = 0, max = 6))
  > table(rolls)
  ```

  ```
  rolls
   0  1  2  3  4  5
  21 12  7 15 18 27
  ```

Simulating from Probability Distributions

How do we simulate from a probability distribution?

There are many ways...

- **Common Distributions**: Use built-in `R` functions (normal, gamma, Poisson, binomial, etc..).
- **Uncommon Distributions**: Need to use simulation.
  - **Discrete random variables**: Often can use `sample()`.
  - **Continuous random variables**: Can use *inverse transform method* and the *acceptance-rejection method* otherwise. **Today we briefly discuss the acceptance-rejection method.** .

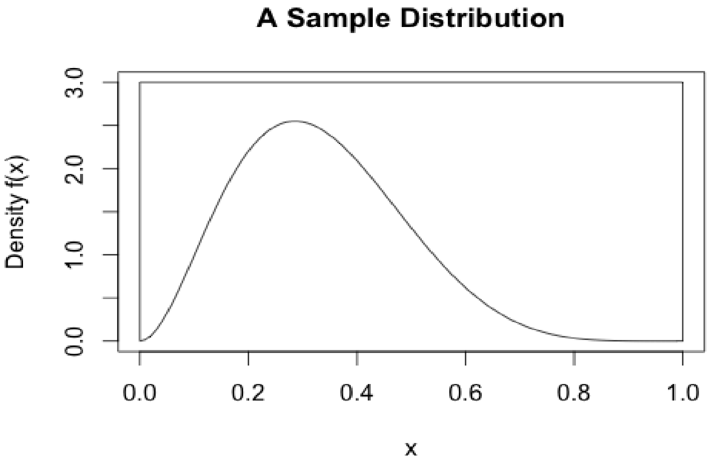
Acceptance-Rejection Algorithm

Sample from a Not-Common Distribution?

- How do we sample from a not-common distribution?
- E.g., not normal, not binomial, not gamma, etc..
- Suppose we do have the pdf $f(x)$.
- *Rejection* sampling obtains draws exactly from the target distribution.
- How? By sampling candidates from an easier distribution then correcting the sampling probability by randomly rejecting some candidates.

The Rejection Method

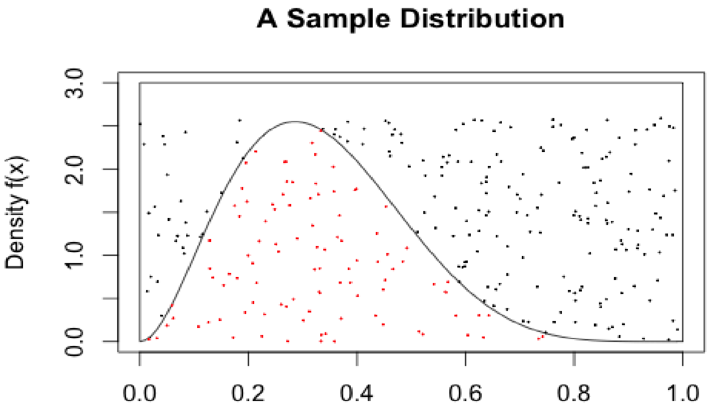
Suppose the pdf f is zero outside an interval $[c, d]$, and $\leq M$ on the interval.



The Rejection Method

We can draw from uniform distributions in any dimension. Do it in two:

```
> x1 <- runif(300, 0, 1); y1 <- runif(300, 0, 2.6)
> selected <- y1 < dbeta(x1, 3, 6)
```



The Rejection Method

```
> mean(selected)
```

```
[1] 0.3366667
```

```
> accepted.points <- x1[selected]
```

The Rejection Method

```
> mean(selected)
```

```
[1] 0.3366667
```

```
> accepted.points <- x1[selected]
```

```
> # Proportion of sample points less than 0.5.
> mean(accepted.points < 0.5)
```

```
[1] 0.8118812
```

```
> # The true distribution.
> pbeta(0.5, 3, 6)
```

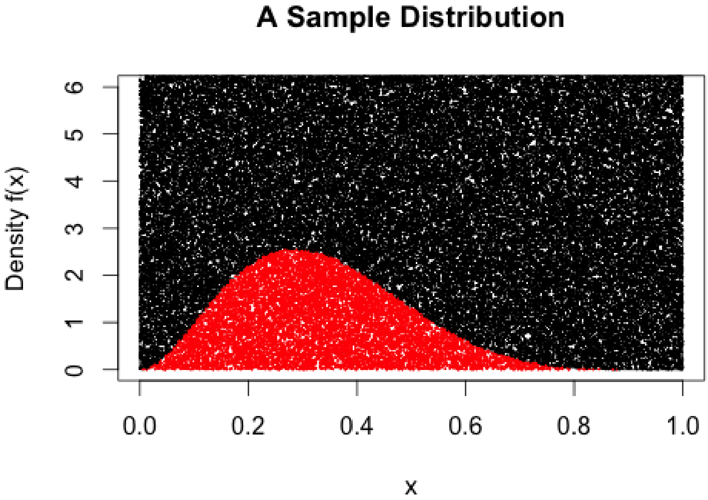
```
[1] 0.8554688
```

The Rejection Method

For this to work efficiently, we have to cover the target distribution with one that sits close to it.

```
> x2       <- runif(100000, 0, 1)
> y2       <- runif(100000, 0, 10)
> selected <- y2 < dbeta(x2, 3, 6)
> mean(selected)
```

```
[1] 0.1006
```


A Sample Distribution
Density f(x)
0
1
2
3
4
5
6
0.0
0.2
0.4
0.6
0.8
1.0
x

Acceptance-Rejection Algorithm

Formally,

- We'd like to sample from a pdf, f.
- Suppose we know how to sample from a pdf g and we can easily calculate $g(x)$.
- Let $e(\cdot)$ denote an *envelope*, with the property

$$e(x) = g(x)/\alpha \geq f(x),$$

for all x for which $f(x) > 0$ for a given constant $0 < \alpha \leq 1$.

Acceptance-Rejection Algorithm

Formally,

- We'd like to sample from a pdf, f.
- Suppose we know how to sample from a pdf g and we can easily calculate $g(x)$.
- Let $e(\cdot)$ denote an *envelope*, with the property

$$e(x) = g(x)/\alpha \geq f(x),$$

  for all x for which $f(x) > 0$ for a given constant $0 < \alpha \leq 1$.
- Sample $Y \sim g$ and $U \sim Unif(0, 1)$ and if $U < f(Y)/e(Y)$, accept Y, otherwise reject it.

Acceptance-Rejection Algorithm

Formally,

- We'd like to sample from a pdf, f.
- Suppose we know how to sample from a pdf g and we can easily calculate $g(x)$.
- Let $e(\cdot)$ denote an *envelope*, with the property

$$e(x) = g(x)/\alpha \geq f(x),$$

  for all x for which $f(x) > 0$ for a given constant $0 < \alpha \leq 1$.
- Sample $Y \sim g$ and $U \sim Unif(0, 1)$ and if $U < f(Y)/e(Y)$, accept Y, otherwise reject it.

Note

- α is the expected proportion of candidates that are accepted.
- Draws accepted are iid from the target density f.

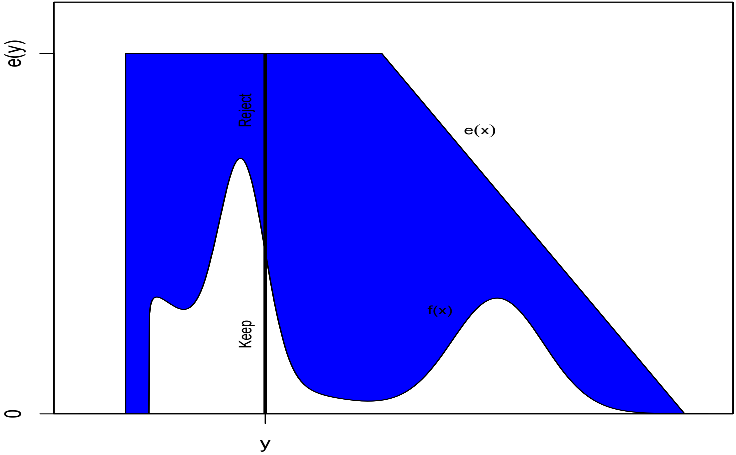
Acceptance-Rejection algorithm

First, find a suitable density g and envelope e. Then the algorithm proceeds as follows:

1. Sample $Y \sim g$.
2. Sample $U \sim$ Unif(0,1).
3. If $U < f(Y)/e(Y)$, accept Y. Set $X = Y$ and consider X to be an element of the target random sample. Equivalent to sampling $U|y \sim U(0, e(y))$ and keeping the value if $U < f(y)$.
4. Repeat from step 1 until you have generated your desired sample size.

Illustration of Acceptance-Rejection Sampling

Illustration of acceptance-rejection sampling for a target distribution, f, using a rejection sampling envelope e.



Envelope

Good envelopes have the following properties:

1. Envelope exceeds the target everywhere $e(x) > f(x)$ for all x.
2. Easy to sample from g.
3. Generate few rejected draws.

A simple approach to finding the envelope:

Determine $\max_x\{f(x)\}$, then use a uniform distribution as g, and $\alpha = 1/\max_x\{f(x)\}$.

Example: Beta distribution

Beta(4,3) distribution

Goal: Generate a RV with pdf $f(x) = 60x^3(1-x)^2$, $0 \le x \le 1$.

Example: Beta distribution

Beta(4,3) distribution

Goal: Generate a RV with pdf $f(x) = 60x^3(1-x)^2$, $0 \leq x \leq 1$.

- We'll take g to be the uniform distribution on $[0,1]$. Then, $g(x) = 1$.
- Let $f.max = max_{x \in [0,1]} f(x)$, then we form envelope with $\alpha = 1/f.max$,

$$e(x) = g(x)/\alpha = f.max \geq f(x).$$

Example: Beta pdf and envelope

Solution Part I

```
> f <- function(x) {
+   return(ifelse((x < 0 | x > 1), 0, 60*x^3*(1-x)^2))
+ }
> x <- seq(0, 1, length = 100)
> plot(x, f(x), type="l", ylab="f(x)")
```

$$f'(x) = 180x^2(1-x)^2 - 120x^3(1-x) = 0 \quad \rightarrow \quad x = 0.6.$$

Example: Beta pdf and envelope

Solution Part I

```
> f <- function(x) {
+   return(ifelse((x < 0 | x > 1), 0, 60*x^3*(1-x)^2))
+ }
> x <- seq(0, 1, length = 100)
> plot(x, f(x), type="l", ylab="f(x)")
```

$$f'(x) = 180x^2(1-x)^2 - 120x^3(1-x) = 0 \quad \rightarrow \quad x = 0.6.$$

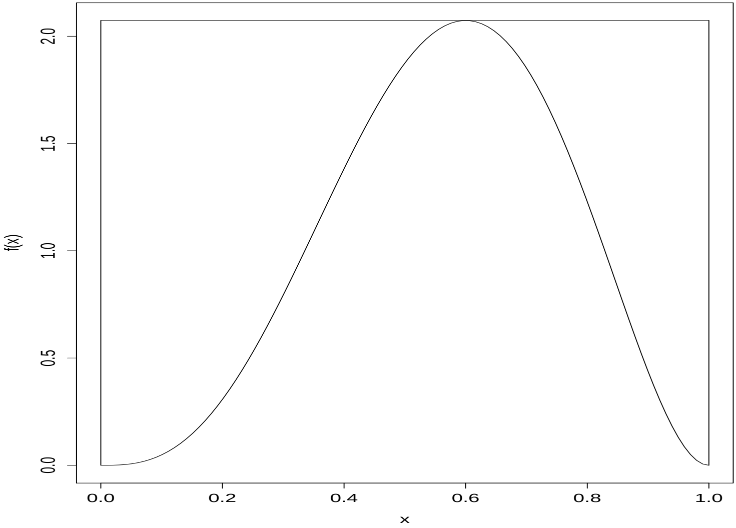
```
> xmax  <- 0.6
> f.max <- 60*xmax^3*(1-xmax)^2
```

Example: Beta pdf and envelope

Solution Part I

```
> e <- function(x) {
+   return(ifelse((x < 0 | x > 1), Inf, f.max))
+ }
> lines(c(0, 0), c(0, e(0)), lty = 1)
> lines(c(1, 1), c(0, e(1)), lty = 1)
> lines(x, e(x), lty = 1)
```

Example: Beta pdf and Envelope

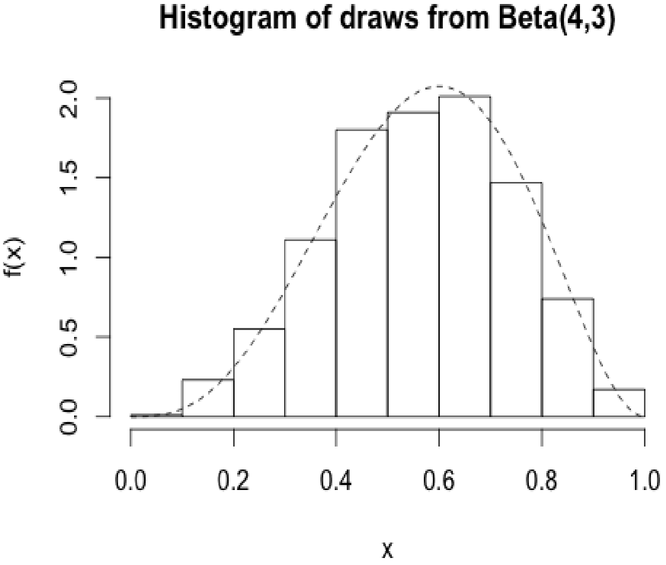


Example: Accept-Reject Algorithm for Beta distribution

Solution Part II

```
> n.samps <- 1000   # number of samples desired
> n       <- 0                         # counter for number samples acc
> samps   <- numeric(n.samps) # initialize the vector of output
> while (n < n.samps) {
+   y <- runif(1)    #random draw from g
+   u <- runif(1)
+   if (u < f(y)/e(y)) {
+      n        <- n + 1
+      samps[n] <- y
+   }
+ }
> x <- seq(0, 1, length = 100)
> hist(samps, prob = T, ylab = "f(x)", xlab = "x",
+         main = "Histogram of draws from Beta(4,3)")
> lines(x, dbeta(x, 4, 3), lty = 2)
```

Example: Accept-Reject Algorithm for Beta distribution



Check Yourself

Tasks

Draw the following random variables. In each case calculate their sample mean, sample variance, and range (max minus min). Are the sample statistics (mean, variance, range) what you'd expect?

- 5000 normal random variables, with mean 1 and variance 8
- 4000 t random variables, with 5 degrees of freedom
- 3500 Poisson random variables, with mean 4
- 999 chi-squared random variables, with 11 degrees of freedom
- 2000 uniform random variables, between $-\sqrt{12}/2$ and $\sqrt{12}/2$

Repeat the above. This is just to emphasize the (obvious!) point: each time you generate random numbers in `R`, you get different results.