Lecture 9: Random Number Generation and Simulations

STAT GR5206 *Statistical Computing & Introduction to Data Science*

Linxi Liu
Columbia University

November 2, 2018

Last Time

Last week: `tidyverse`

- `tibble`: A `tibble`, or `tbl_df`, is a modern reimagining of the `data.frame`.
- `purrr`: enhancing R's functional programming (FP).
- `tidyr`: creating tidy data.
- `ggplot2`: creating advanced graphics

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.
  - Inverse transforms and the acceptance-rejection algorithm.
- **Monte Carlo Integration**. How to use simulation to approximate integrals.

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] 0.918700128 0.552530140 0.324911656 0.469969482
 [5] 0.296619171 0.061523237 0.988496977 0.003023173
 [9] 0.155961801 0.120277847
```

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] 0.918700128 0.552530140 0.324911656 0.469969482
 [5] 0.296619171 0.061523237 0.988496977 0.003023173
 [9] 0.155961801 0.120277847
```

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

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.

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.

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
```

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 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* when the cdf is invertible in closed form and the *acceptance-rejection method* otherwise.

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* when the cdf is invertible in closed form and the *acceptance-rejection method* otherwise.

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,
- 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)
```

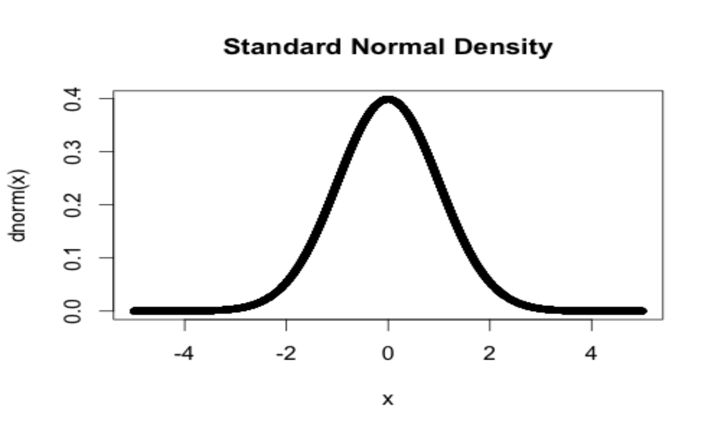
```
[1] 0.3989423
```

```
> 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!

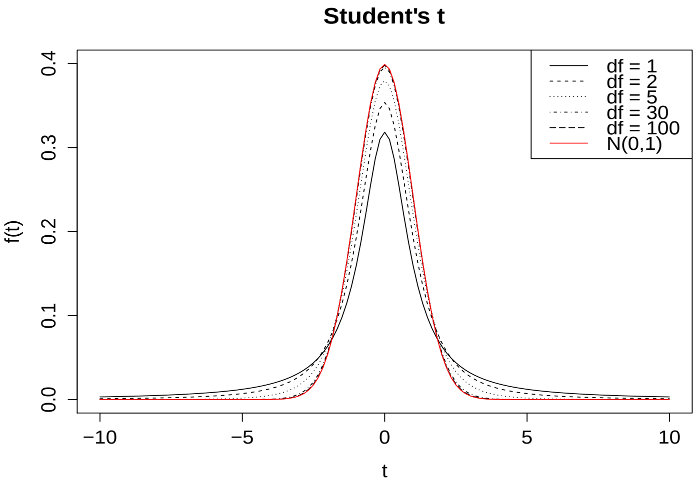
Recall that the student's t distribution converges to a standard normal distribution as $df \to \infty$.

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
f(t)
t
0.0
0.1
0.2
0.3
0.4
−10
−5
0
5
10
df = 1
df = 2
df = 5
df = 30
df = 100
N(0,1)

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

$$\int_2^{\infty} f(x|\alpha, \beta)\, dx$$

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

Check Yourself

Solutions

Want to calculate

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

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
```

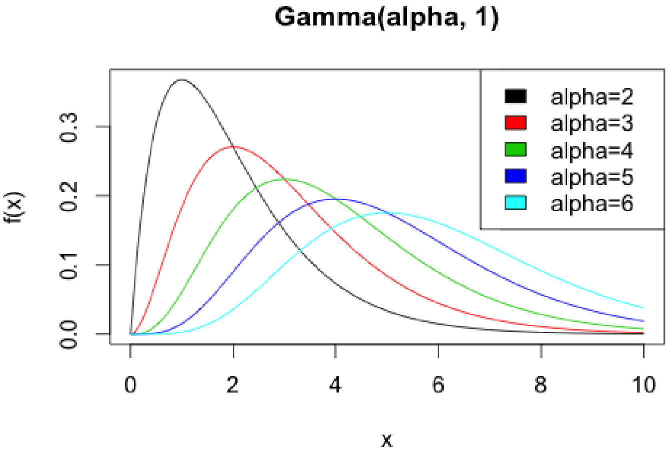
What about $Pr(X = 2)$?

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

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

```
> 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)
  ```

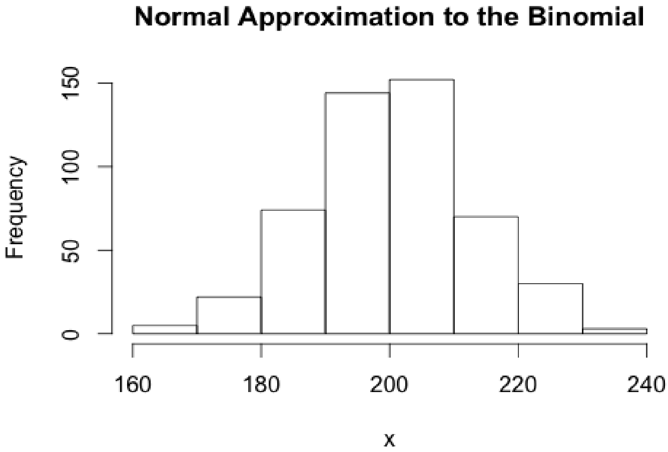
  ```
  [1] 0.2273564
  ```

  ```
  >  # 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



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* when the cdf is invertible in closed form and the *acceptance-rejection method* otherwise.

`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* when the cdf is invertible in closed form and the *acceptance-rejection method* otherwise.

Inverse Transform Method

Theorem

If X is a continuous random variable with cdf F, then $F(X) \sim U[0,1]$.

Inverse Transform Method

Theorem

If X is a continuous random variable with cdf F, then $F(X) \sim U[0,1]$.

Method

Generate u from $U[0,1]$, then $Y = F^{-1}(u)$ is a realization from F.

Why does this work?

$$\begin{aligned} P(Y \leq y) &= P(F^{-1}(U) \leq y) \\ &\overset{(a)}{=} P(F(F^{-1}(U)) \leq F(y)) \\ &= P(U \leq F(y)) \\ &= F(y), \end{aligned}$$

where (a) follows by monotonicity of F.

Inverse Transform Algorithm

1. Derive the inverse function F^{-1}. To do this:
   - Then solve $F(x) = u$ for x to find $x = F^{-1}(u)$.
2. Write a function to compute $x = F^{-1}(u)$.
3. For each realization:
   - Generate a random value u from Uniform(0,1).
   - Compute $x = F^{-1}(u)$

Inverse Transform Method

Example

Let's simulate exponential rvs (with $\lambda = 2$) using using the inverse transform method.

Inverse Transform Method

Example

Let's simulate exponential rvs (with $\lambda = 2$) using using the inverse transform method.

The pdf of the exponential distribution is $f(x) = \lambda e^{-\lambda t}$, so the cdf is

$$F(x) = \int_0^x f(t)dt = \int_0^x \lambda e^{-\lambda t} dt = 1 - e^{-\lambda x}.$$

Inverse Transform Method

Example

Let's simulate exponential rvs (with $\lambda = 2$) using using the inverse transform method.

The pdf of the exponential distribution is $f(x) = \lambda e^{-\lambda t}$, so the cdf is

$$F(x) = \int_0^x f(t)dt = \int_0^x \lambda e^{-\lambda t} dt = 1 - e^{-\lambda x}.$$

Now we invert the cdf.

$$u = 1 - e^{-\lambda x} \quad \rightarrow \quad x = -\frac{1}{\lambda}\log(1-u)$$

Inverse Transform Method

Example

Let's simulate exponential rvs (with $\lambda = 2$) using using the inverse transform method.

```
> lambda <- 2
> n      <- 1000
> u      <- runif(n) # Simulating uniform rvs
> Finverse <- function(u, lambda) {
+   # Function for the inverse transform
+   return(ifelse((u<0|u>1), 0, -(1/lambda)*log(1-u)))
+ }
```

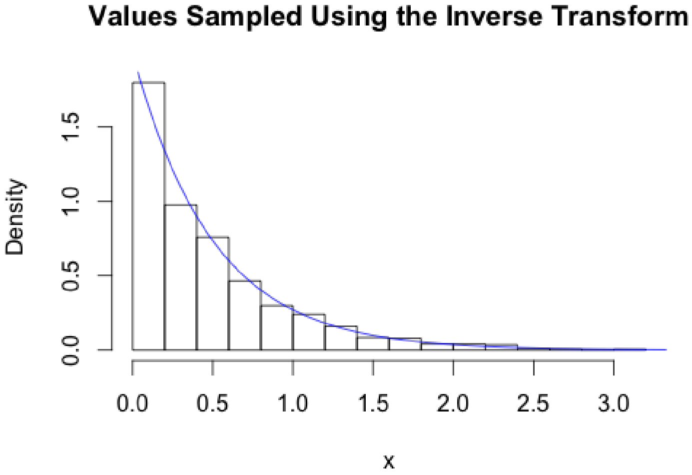
Inverse Transform Method

Example

Let's simulate exponential rvs (with $\lambda = 2$) using using the inverse transform method.

```
>  # x should be exponentially distributed
> x <- Finverse(u, lambda)
> hist(x, prob = TRUE, breaks = 15)
> y <- seq(0, 10, .01)
> lines(y, lambda*exp(-lambda*y), col = "blue")
```

Inverse Transform Method: Exponential



Check Yourself

Task

Simulate a random sample of size 1000 from the pdf $f_X(x) = 3x^2$, $0 \leq x \leq 1$.

- Find F.
- Find F^{-1}.
- Plot the empirical distribution (histogram) with the correct density overlayed on the plot.

Inverse Transform Method

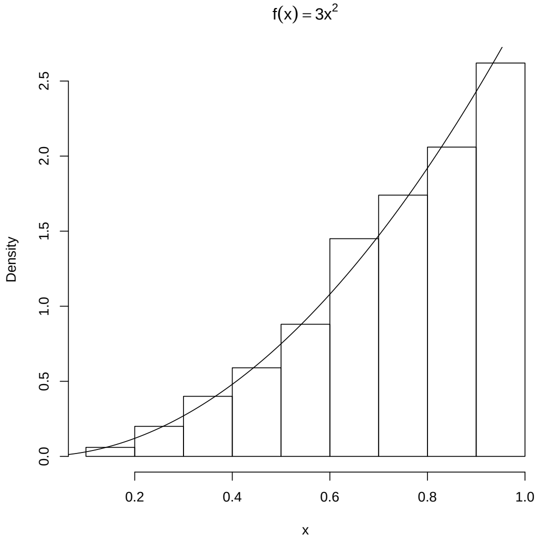
Solution

- $$F(x) = \int_0^x f(t)dt = \int_0^x 3t^2 dt = x^3.$$

- $$u = x^3 \quad \to \quad x = u^{1/3}$$

- ```
  > n         <- 1000
  > u         <- runif(n)
  > F.inverse <- function(u) {return(u^{1/3})}
  > x         <- F.inverse(u)
  > hist(x, prob = TRUE) # histogram
  > y         <- seq(0, 1, .01)
  > lines(y, 3*y^2) # density curve f(x)
  ```
  ```

Inverse Transform Method



Acceptance-Rejection Algorithm

So we've seen that generating rvs from a pdf f is easy if it's a standard distribution or if we have a nice, invertible CDF.

- What can we do if all we've got is the pdf f?

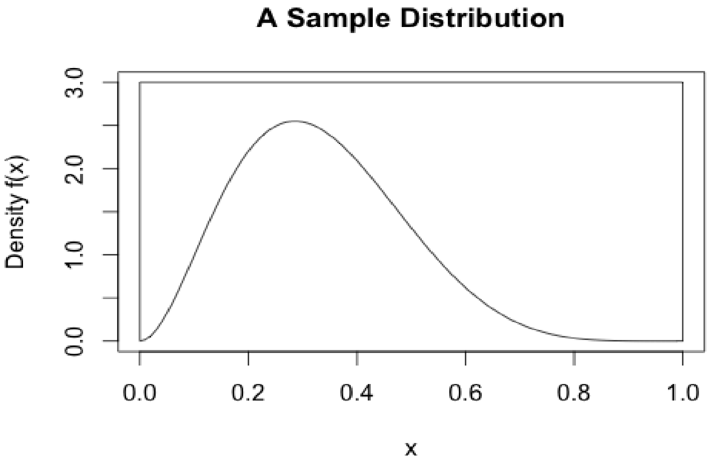
Acceptance-Rejection Algorithm

So we've seen that generating rvs from a pdf f is easy if it's a standard distribution or if we have a nice, invertible CDF.

- What can we do if all we've got is the pdf f?
- *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

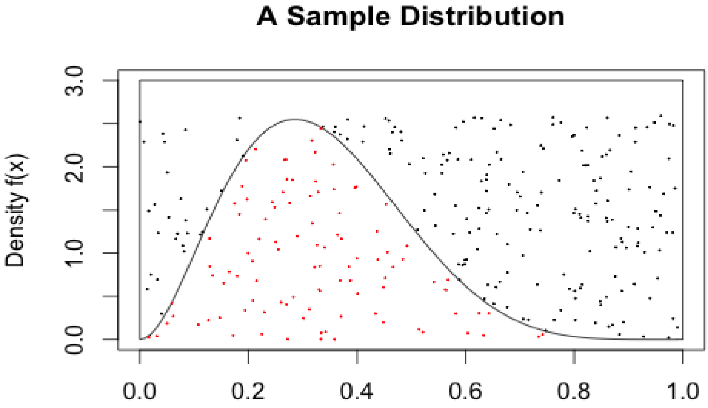
Suppose the pdf f is zero outside an interval $[c, d]$, and $\leq M$ on the interval.



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] 0.4166667
```

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

The Rejection Method

```
> mean(selected)
```

```
[1] 0.4166667
```

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

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

```
[1] 0.856
```

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

```
[1] 0.8554688
```

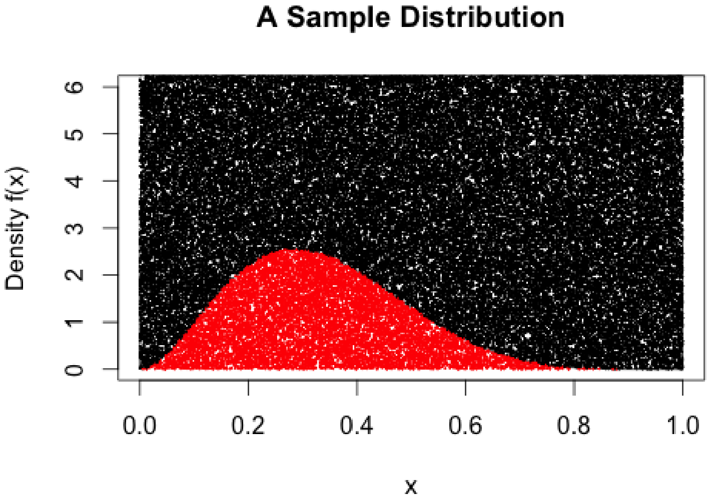
The Rejection Method

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] 0.10044
```

The Rejection Method



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.

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.

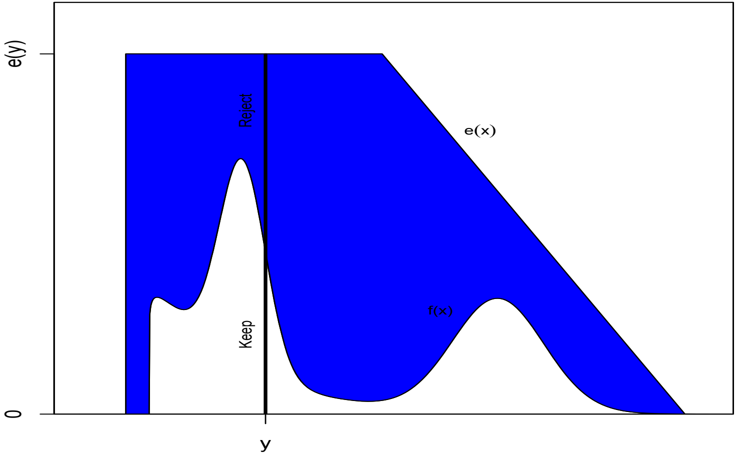
Why does it work?

$$P(X \leq y) = P\left(Y \leq y \middle| U \leq \frac{f(Y)}{e(Y)}\right) = \cdots = \int_{-\infty}^{y} f(z)dz$$

Exercise: Fill in the missing pieces using conditional distributions.

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 $f(x) = 60x^3(1-x)^2$, $0 \leq x \leq 1$.

- You can't invert f(x) analytically, so can't use the inverse transform method.

Example: Beta distribution

Beta(4,3) distribution

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

- You can't invert f(x) analytically, so can't use the inverse transform method.
- 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 $\alpha = 1/f.max$,

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

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)")
```

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

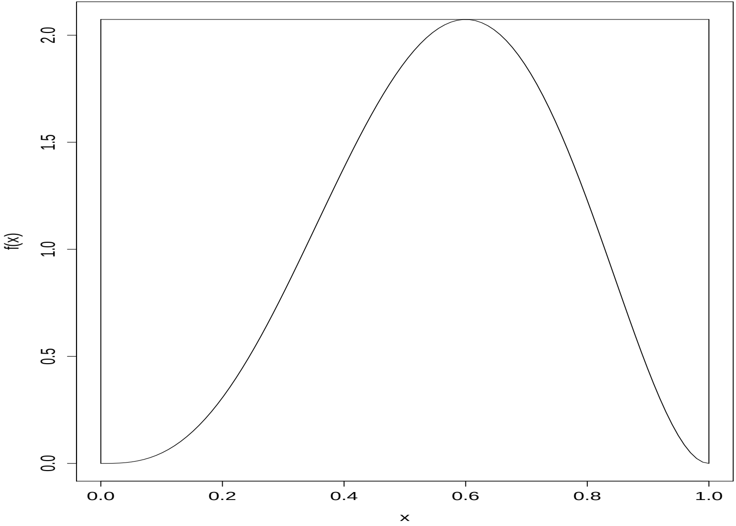
```
> xmax  <- 0.6
> f.max <- 60*xmax^3*(1-xmax)^2
```

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

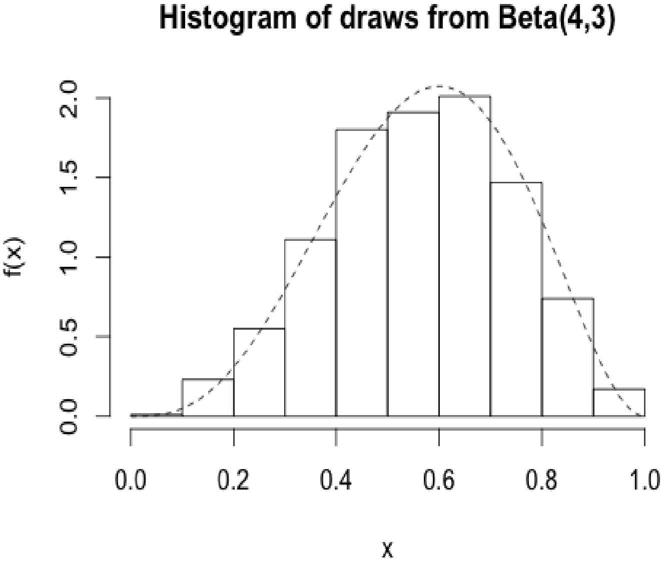


Example: Accept-Reject Algorithm for Beta distribution

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



Monte Carlo Integration

Numerical Integration

What is Numerical Integration?

- Often we need to solve integrals,

$$\int f(x)dx,$$

  but doing so can be hard.
- Even when we know the function f, finding a closed-form antiderivative may be difficult or even impossible.
- In these cases, we'd like to find good ways to approximate the value of the integral.
- Such approximations are generally referred to as **numerical integration**.

Common Techniques of Numerical Integration

There are many methods of numerical integration:

1. Riemann rule,
2. Trapezoid rule,
3. Simpson's rule,
4. Newton-Côtes Quadrature method (a generalization of the above three),
5. and others.

Numerical Integration

Common Techniques of Numerical Integration

There are many methods of numerical integration:

1. Riemann rule,
2. Trapezoid rule,
3. Simpson's rule,
4. Newton-Côtes Quadrature method (a generalization of the above three),
5. and others.

Today we study Monte Carlo integration.

Law of Large Numbers

Recall,

If $X_1, X_2, \ldots, X_n$ are iid with pdf p,

$$\frac{1}{n}\sum_{i=1}^{n} g(X_i) \to \int g(x)p(x)dx = \mathbb{E}_p[g(X)].$$

Law of Large Numbers

Recall,

If $X_1, X_2, \ldots, X_n$ are iid with pdf p,

$$\frac{1}{n}\sum_{i=1}^{n} g(X_i) \to \int g(x)p(x)dx = \mathbb{E}_p[g(X)].$$

The Monte Carlo Principle

To estimate $\int g(x)dx$, draw from p and take the sample mean of $f(x) = g(x)/p(x)$.

The Monte Carlo Principle

The Monte Carlo Principle

To estimate $\int g(x)dx$, draw from p and take the sample mean of $f(x) = g(x)/p(x)$.

By the Law of Large Numbers,

If $X_1, X_2, \ldots, X_n$ are iid with pdf p,

$$\frac{1}{n}\sum_{i=1}^{n}\frac{g(X_i)}{p(X_i)} \to \int g(x)dx.$$

Monte Carlo Integration

Let's Look at an Example

- Estimate the integral

$$\int_{-\infty}^{\infty} g(x)dx = \int_{-\infty}^{\infty} x^2 e^{-x^2} dx,$$

  using MC techniques.
- We know that this integral equals $\sqrt{\pi}/2$. (How?) Let's still perform the exercise.

Monte Carlo Integration

Solution

Estimate $\int g(x)dx$ by drawing standard normal rvs $X_1, X_2, \ldots$ and taking the sample mean of $g(x)/p(x)$ where $p(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}x^2}$ and

$$g(x)/p(x) = x^2 \cdot \sqrt{2\pi} e^{-\frac{1}{2}x^2}.$$

```
> g.over.p <- function(x) {
+   return(sqrt(2*pi) * x^2 * exp(-(1/2)*x^2))
+ }
> mean(g.over.p(rnorm(10000))) # Try n = 10000
```

```
[1] 0.8873605
```

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

```
[1] 0.8862269
```

Monte Carlo Integration

By the Central Limit Theorem,

$$\frac{1}{n}\sum_{i=1}^{n}\frac{g(X_i)}{p(X_i)} \xrightarrow{d} \mathcal{N}\left(\int g(x)dx, \frac{\sigma^2_{g/p}}{n}\right).$$

- The Monte Carlo approximation is unbiased.
- The root mean square error is $\propto n^{-1/2}$, so if we just keep taking Monte Carlo draws, the error can get as small as you'd like, even if g or x are very complicated.

Monte Carlo Integration

How to Choose p?

In principle, any p which is supported on the same set as g could be used for Monte Carlo. In practice, we would like for p to be

- **Easy to simulate**.
- **Have low variance**. It generally improves efficiency to have the shape of $p(x)$ follow that of $g(x)$ such that $\sigma^2_{g/p}$ is small.
- **Takes a simple form**. It is often worth looking carefully at the integrand to see if a probability density can be factored out of it.

Monte Carlo Integration

Let's Look at an Example

Estimate the integral

$$\int_{-\infty}^{\infty} g(x)dx = \int_{-\infty}^{\infty} x^2 e^{-x^2} dx = \sqrt{\pi} \int_{-\infty}^{\infty} x^2 \left(\frac{1}{\sqrt{\pi}} e^{-x^2} \right) dx,$$

using MC techniques.

Monte Carlo Integration

Solution

Estimate $\int g(x)dx$ by drawing rvs $X_1, X_2, \ldots \sim \mathcal{N}(0, 1/2)$ and calculating the sample mean of $g(x) = \sqrt{\pi}x^2$.

```
> g <- function(x) {sqrt(pi)*x^2}
> mean(g(rnorm(10000, sd = 1/sqrt(2)))) # Try n = 10000
```

```
[1] 0.9082105
```

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

```
[1] 0.8862269
```

Check Yourself

Tasks

- Estimate $P(X < 3)$ where X is an exponentially distributed random variable with $rate = 1/3$. HINT: Let $f(x)$ be the pdf of the exponential density with $rate = 2$.

$$P(X < 3) = \mathbb{E}_f[\mathbb{I}(X < 3)] = \int_{-\infty}^{\infty} \mathbb{I}(x < 3) f(x) dx,$$

  where $\mathbb{I}(x < 3)$ is the indicator function, meaning it equals 1 if $x < 3$ and 0 otherwise.
- Use built-in `R` functions to find the exact probability.

Check Yourself

Solution

- ```
  > n <- 10000
  > mean(rexp(n, rate = 1/3) < 3)
  ```

  ```
  [1] 0.624
  ```

- ```
  > pexp(3, rate = 1/3)
  ```

  ```
  [1] 0.6321206
  ```

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 Some Common Distributions from Unif(0,1)

Simulating Some Common Distributions from Unif(0,1)

How do we simulate some common distributions only using the uniform distribution?

- Use Inverse Transforms
- Use Acceptance-Rejection
- Use Transformations

Simulating Some Common Distributions from Unif(0,1)

Common Continuous Transformations

- $X \sim Unif(a, b)$; draw $U \sim Unif(0, 1)$, then $X = a + (b - a)U$
- $X \sim Cauchy(\alpha, \beta)$, Draw $U \sim Unif(0, 1)$, then $X = \alpha + \beta \tan(\pi(U - 1/2))$.
- $X \sim N(0, 1)$; draw U_1, U_2 iid $Unif(0, 1)$, then $X_1 = \sqrt{-2 \log U_1} cos(2\pi U_2)$ and $X_2 = \sqrt{-2 \log U_1} sin(2\pi U_2)$ are independent $N(0, 1)$.
- $X \sim N(\mu, \sigma^2)$; draw $Z \sim N(0, 1)$, then $X = \sigma Z + \mu$.
- Multivariate $N(\mu, \Sigma)$; generate standard multivariate vector Z, then $X = \Sigma^{1/2} Z + \mu$.

Simulating Some Common Distributions from Unif(0,1)

Similar methods can be extended to discrete random variables.

Common Discrete Transformations

- $Poiss(\lambda)$; draw $U_1, U_2, \ldots, \sim$ iid $Unif(0,1)$; then $X = j - 1$, where j is the lowest is the lowest index for which $\Pi_{i=1}^{j} U_j < e^{-\lambda}$.
- $Bernoulli(p)$; draw $U \sim Unif(0,1)$, then $X = \mathbb{I}(U < p)$ is distributed $Bernoulli(p)$.
- $Binomial(p)$; The sum of n independent $Bernoulli(p)$ draws has a $Binomial(p)$ distribution.

Simulating a Binomial

Example

Simulate a random sample of size 1000 from $Binomial(n = 10, p = .3)$ using $Unif[0, 1]$.

Check Yourself

Solution

```
> R <- 1000
> n <- 10
> binom.list <-NULL
> for (i in 1:R) {
+   U <- runif(n)
+   binom.list[i] <-sum(U<.3)
+ }
> # Compare
> mean(binom.list);var(binom.list)
```

```
[1] 2.963
```

```
[1] 1.989621
```

Check Yourself (Capstone Example)

Task

Generate 1000 draws from a bivariate normal distribution by starting with independent uniform random variables U_1 and U_2. Let the bivariate normal have mean and covariance matrix

$$\mu = \begin{pmatrix}\mu_X & \mu_Y\end{pmatrix}^T = \begin{pmatrix}5 & 10\end{pmatrix}^T$$

and

$$\Sigma = \begin{pmatrix}\sigma_X^2 & \rho\sigma_X\sigma_Y \\ \rho\sigma_X\sigma_Y & \sigma_Y^2\end{pmatrix} = \begin{pmatrix}4 & -3 \\ -3 & 9\end{pmatrix}.$$

Simulating Some Common Distributions from Unif(0,1)

SVD

The singular-value decomposition of a square matrix Σ is the factorization

$$\Sigma = UDV^T,$$

where U and V are orthogonal matrices and D is a diagonal matrix of Σ's eigenvalues.

Square root of a matrix

Define the square root of covariance matrix Σ by

$$\Sigma^{1/2} = UD^{1/2}V^T$$

Simulating Some Common Distributions from Unif(0,1)

SVD

The singular-value decomposition of a square matrix Σ is the factorization

$$\Sigma = UDV^T,$$

where U and V are orthogonal matrices and D is a diagonal matrix of Σ's eigenvalues.

Square root of a matrix

Define the square root of covariance matrix Σ by

$$\Sigma^{1/2} = UD^{1/2}V^T$$

$$(\mathbf{UD^{1/2}V^T})(\mathbf{UD^{1/2}V^T}) = \mathbf{UD^{1/2}D^{1/2}V^T} = \mathbf{UDV^T}$$

Check Yourself (Capstone Example)

SVD in R

```
> Sigma <- matrix(c(4,-3,-3,9),nrow=2)
> svd(Sigma)
```

```
$d
[1] 10.405125  2.594875

$u
           [,1]      [,2]
[1,] -0.4241554 0.9055894
[2,]  0.9055894 0.4241554

$v
           [,1]      [,2]
[1,] -0.4241554 0.9055894
[2,]  0.9055894 0.4241554
```

Check Yourself (Capstone Example)

$$\Sigma^{1/2} = UD^{1/2}V^T$$

Square Root of a Matrix

```
> Sigma <- matrix(c(4,-3,-3,9),nrow=2)
> Sigma
```

```
     [,1] [,2]
[1,]    4   -3
[2,]   -3    9
```

```
> Sq.Sigma <- (svd(Sigma)$u)%*%sqrt(diag(svd(Sigma)$d))%*%t(svd(Sigma)$v)
> Sq.Sigma
```

```
           [,1]       [,2]
[1,]  1.9013832 -0.6202757
[2,] -0.6202757  2.9351760
```

```
> Sq.Sigma%*%Sq.Sigma
```

```
     [,1] [,2]
[1,]    4   -3
[2,]   -3    9
```

Check Yourself (Capstone Example)

Finish the example

Generate 1000 draws from a bivariate normal distribution by starting with independent uniform random variables U_1 and U_2. Let the bivariate normal have mean and covariance matrix

$$\mu = \begin{pmatrix} \mu_X & \mu_Y \end{pmatrix}^T = \begin{pmatrix} 5 & 10 \end{pmatrix}^T$$

and

$$\Sigma = \begin{pmatrix} \sigma_X^2 & \rho\sigma_X\sigma_Y \\ \rho\sigma_X\sigma_Y & \sigma_Y^2 \end{pmatrix} = \begin{pmatrix} 4 & -3 \\ -3 & 9 \end{pmatrix}.$$

Note

- $X \sim N(0,1)$; draw U_1, U_2 iid $Unif(0,1)$, then $X_1 = \sqrt{-2\log U_1} cos(2\pi U_2)$ and $X_2 = \sqrt{-2\log U_1} sin(2\pi U_2)$ are independent $N(0,1)$.
- Multivariate $N(\mu, \Sigma)$; generate standard multivariate vector Z, then $X = \Sigma^{1/2} Z + \mu$.

Optional Reading

- Chapter 5 (Simulation) in Advanced Data Analysis from an Elementary Point of View.
- Chapter 6 (Simulation and Monte Carlo Integration) in Computational Statistics.