Statistical Computing (36-350)

Lecture 14: Simulation I: Generating Random Variables

Cosma Shalizi

14 October 2013

Agenda

- The basic random-variable commands
- Transforming uniform random variables to other distributions
  - The quantile method
  - The rejection method
- Where the uniform random numbers come from

Required Reading: Matloff, chapter 8

R Cookbook, chapter 8

Optional reading: Chambers, section 6.10

Stochastic Simulation

Why simulate?

Stochastic Simulation

Why simulate?

- We want to see what a probability model actually does

Stochastic Simulation

Why simulate?

- We want to see what a probability model actually does
- We want to understand how our procedure works on a test case

Stochastic Simulation

Why simulate?

- We want to see what a probability model actually does
- We want to understand how our procedure works on a test case
- We want to use a partly-random procedure

Stochastic Simulation

Why simulate?

- We want to see what a probability model actually does
- We want to understand how our procedure works on a test case
- We want to use a partly-random procedure

All of these require drawing random variables from distributions

Built-in Random Variable Generators

`runif`, `rnorm`, `rbinom`, `rpois`, `rexp`, etc. etc.

First argument is always `n`, number of variables to generate

Subsequent arguments are parameters to distribution, and vary with the distribution

Many Distributions at Once

Parameters are recycled:

```
> rnorm(n=4,mean=c(-1000,1000),sd=1)
[1]  -999.3637  1000.4710 -1000.4449  1000.1040
```

Each of the *n* draws can get its own parameters

sample

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

draw random sample of `size` points from `x`, optionally with replacement and/or weights

`x` can be anything where `length()` makes sense, basically

sample

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

draw random sample of `size` points from `x`, optionally with replacement and/or weights

`x` can be anything where `length()` makes sense, basically

`sample(x)` does a random permutation

sample

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

draw random sample of `size` points from `x`, optionally with replacement and/or weights
`x` can be anything where `length()` makes sense, basically
`sample(x)` does a random permutation
If `x` is a single number, treat it like `1:x`

```
> sample(5)
[1] 1 4 3 2 5
```

Why Would We Want to Do a Random Permutation?

```
sample(cats$Sex) # Randomly shuffle sexes among cats
```

Why Would We Want to Do a Random Permutation?

```
sample(cats$Sex) # Randomly shuffle sexes among cats

diff.in.means <- function(m.or.f) {
  mean.male <- mean(cats$Hwt[m.or.f=="M"])
  mean.female <- mean(cats$Hwt[m.or.f=="F"])
  return(mean.male-mean.female)
}
```

Why Would We Want to Do a Random Permutation?

```
sample(cats$Sex) # Randomly shuffle sexes among cats

diff.in.means <- function(m.or.f) {
  mean.male <- mean(cats$Hwt[m.or.f=="M"])
  mean.female <- mean(cats$Hwt[m.or.f=="F"])
  return(mean.male-mean.female)
}

> (obs.diff <- diff.in.means(cats$Sex))
[1] 2.120553
> null.diffs <- replicate(1000,diff.in.means(sample(cats$Sex)))
> summary(null.diffs)
     Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
-1.363000 -0.274400  0.010620  0.008788  0.307500  1.122000
> mean(null.diffs > obs.diff)
[1] 0
```

Why Would We Want to Do a Random Permutation?

```
sample(cats$Sex) # Randomly shuffle sexes among cats

diff.in.means <- function(m.or.f) {
  mean.male <- mean(cats$Hwt[m.or.f=="M"])
  mean.female <- mean(cats$Hwt[m.or.f=="F"])
  return(mean.male-mean.female)
}

> (obs.diff <- diff.in.means(cats$Sex))
[1] 2.120553
> null.diffs <- replicate(1000,diff.in.means(sample(cats$Sex)))
> summary(null.diffs)
     Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
-1.363000 -0.274400  0.010620  0.008788  0.307500  1.122000
> mean(null.diffs > obs.diff)
[1] 0
```

Most basic possible **permutation test**; get much trickier

Resampling

Define:

```
resample <- function(x) { sample(x, size=length(x), replace=TRUE) }
```

Resampling

Define:

```
resample <- function(x) { sample(x, size=length(x), replace=TRUE) }
```

Resample a data vector:

```
resample(cats$Hwt)
```

Resampling

Define:

```
resample <- function(x) { sample(x, size=length(x), replace=TRUE) }
```

Resample a data vector:

```
resample(cats$Hwt)
```

Resample the rows of a data frame:

```
cats[resample(1:nrow(cats),]
```

Resampling

Define:

```
resample <- function(x) { sample(x, size=length(x), replace=TRUE) }
```

Resample a data vector:

```
resample(cats$Hwt)
```

Resample the rows of a data frame:

```
cats[resample(1:nrow(cats),]
```

Exercise: write an improved `resample` that works on vectors or data frames

Why Resample the Data?

```
diff.in.means2 <- function(df) {
  mean(df[df$Sex=="M","Hwt"]) - mean(df[df$Sex=="F","Hwt"])
}

resample.diffs <- replicate(1000,diff.in.means2(cats[resample(1:nrow(cats)),]))
```

Why Resample the Data?

```
diff.in.means2 <- function(df) {
  mean(df[df$Sex=="M","Hwt"]) - mean(df[df$Sex=="F","Hwt"])
}

resample.diffs <- replicate(1000,diff.in.means2(cats[resample(1:nrow(cats)),]))

> sd(resample.diffs)
[1] 0.316297
> quantile(resample.diffs,probs=c(0.025,0.975))
    2.5%    97.5%
1.459805 2.700566
```

Why Resample the Data?

```
diff.in.means2 <- function(df) {
  mean(df[df$Sex=="M","Hwt"]) - mean(df[df$Sex=="F","Hwt"])
}

resample.diffs <- replicate(1000,diff.in.means2(cats[resample(1:nrow(cats)),]))

> sd(resample.diffs)
[1] 0.316297
> quantile(resample.diffs,probs=c(0.025,0.975))
    2.5%    97.5%
1.459805 2.700566
```

Very close to *t*-test formulas. . .

Why Resample the Data?

```
diff.in.means2 <- function(df) {
  mean(df[df$Sex=="M","Hwt"]) - mean(df[df$Sex=="F","Hwt"])
}

resample.diffs <- replicate(1000,diff.in.means2(cats[resample(1:nrow(cats)),]))

> sd(resample.diffs)
[1] 0.316297
> quantile(resample.diffs,probs=c(0.025,0.975))
    2.5%    97.5%
1.459805 2.700566
```

Very close to *t*-test formulas. . .

Bootstrap standard error and confidence limits

Again, gets much more sophisticated

Biased Coins

Given: uniform random variable U, success probability p
Wanted: A Bernoulli(p) random variable

Biased Coins

Given: uniform random variable U, success probability p
Wanted: A Bernoulli(p) random variable
Return 1 if $U \leq p$, else return 0

Biased Coins

Given: uniform random variable U, success probability p
Wanted: A Bernoulli(p) random variable
Return 1 if $U \leq p$, else return 0

```
ifelse(runif(n) =< p, 1, 0)
```

or just

```
rbinom(n,size=1,prob=p)
```

Categorical or Discrete Variables

Given: uniform U, category probabilities $p_1, p_2, \ldots p_k$
Wanted: a categorical random variable with that p.m.f.

Categorical or Discrete Variables

Given: uniform U, category probabilities $p_1, p_2, \ldots p_k$
Wanted: a categorical random variable with that p.m.f.
If $U \leq p_1$, return 1
else if $U \leq p_1 + p_2$, return 2
etc.

Categorical or Discrete Variables

Given: uniform U, category probabilities $p_1, p_2, \ldots p_k$
Wanted: a categorical random variable with that p.m.f.
If $U \leq p_1$, return 1
else if $U \leq p_1 + p_2$, return 2
etc.

```
min(which(u < cumsum(p)))
```

(needs some thought to vectorize)

Categorical or Discrete Variables

Given: uniform U, category probabilities $p_1, p_2, \ldots p_k$
Wanted: a categorical random variable with that p.m.f.
If $U \leq p_1$, return 1
else if $U \leq p_1 + p_2$, return 2
etc.

```
min(which(u < cumsum(p)))
```

(needs some thought to vectorize)

```
rmultinoulli <- function(n,prob) {
  return(sample(1:length(prob),replace=TRUE,size=n,prob=prob))
}
```

(`rmultinom` gives counts, not a sequence)

The Quantile Transform Method

Given: uniform random variable U, CDF F
Claim: $X = F^{-1}(U)$ is a random variable with CDF F

The Quantile Transform Method

Given: uniform random variable U, CDF F
Claim: $X = F^{-1}(U)$ is a random variable with CDF F
Proof:

$$\mathbb{P}(X \leq a) = \mathbb{P}(F^{-1}(U) \leq a) = \mathbb{P}(U \leq F(a)) = F(a)$$

F^{-1} is the quantile function
$\therefore$ if we can generate uniforms and we can calculate quantiles, we can generate non-uniforms

Less Mathematically

To turn U into a coin-toss with bias p: is $U \leq p$ or not?

Less Mathematically

To turn U into a coin-toss with bias p: is $U \leq p$ or not?

To turn U into a binomial: start with $X = 0$; if $U \leq F(X)$, stop, otherwise add 1 to X and check again

Less Mathematically

To turn U into a coin-toss with bias p: is $U \leq p$ or not?

To turn U into a binomial: start with $X = 0$; if $U \leq F(X)$, stop, otherwise add 1 to X and check again

Tedious do this iteratively

No next value for continuous random variables

Less Mathematically

To turn U into a coin-toss with bias p: is $U \leq p$ or not?

To turn U into a binomial: start with $X = 0$; if $U \leq F(X)$, stop, otherwise add 1 to X and check again

Tedious do this iteratively

No next value for continuous random variables

Quantiles solve both difficulties

Quantile functions often don't have closed form, and don't have nice numerical solutions

But we know the probability density function — can we use that?

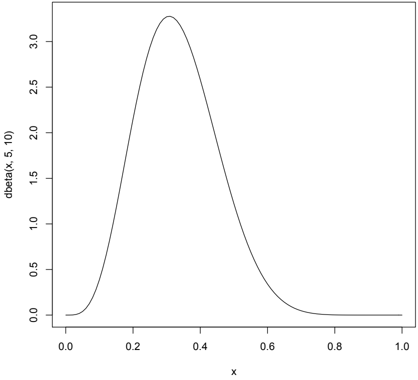
Suppose the pdf f is zero outside an interval $[c, d]$, and $\leq M$ on the interval

Draw the rectangle $[c, d] \times [0, M]$, and the curve f

Area under the curve $= 1$

Area under curve and $x \leq a$ is $F(a)$

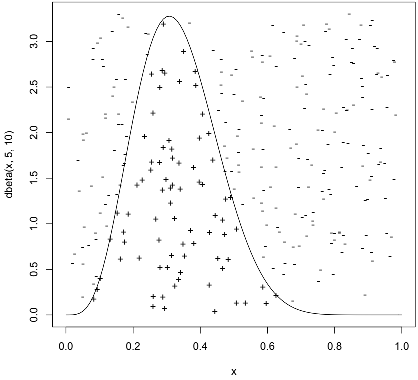
How can we uniformly sample area under the curve?



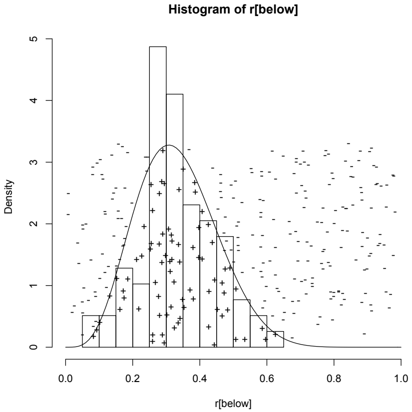

```
M <- 3.3; curve(dbeta(x,5,10),from=0,to=1,ylim=c(0,M))
```

We sample uniformly from the *box*, and take the points under the curve

We sample uniformly from the *box*, and take the points under the curve
$R \sim \mathrm{Unif}(c, d)$
$U \sim \mathrm{Unif}(0, 1)$
If $MU \leq f(R)$ then $X = R$, otherwise try again




```
r <- runif(300,min=0,max=1); u <- runif(300,min=0,max=1)
below <- which(M*u <= dbeta(r,5,10))
points(r[below],M*u[below],pch="+"); points(r[-below],M*u[-below],pch="-")
```




```
hist(r[below],xlim=c(0,1),probability=TRUE); curve(dbeta(x,5,10),add=TRUE)
points(r[below],M*u[below],pch="+"); points(r[-below],M*u[-below],pch="-")
```

If f doesn't go to zero outside $[c, d]$, try to find another density ρ where

- ρ also has unlimited support
- $f(a) \leq M\rho(a)$ everywhere
- we can generate from ρ (say by quantiles)

Then $R \sim \rho$, and accept when $MU\rho(R) \leq f(R)$
(Uniformly distributed on the area under ρ)

Need to make multiple "proposals" R for each X

e.g., generated 300 for figure, only accepted 78

Important for efficiency to keep this ratio small

Ideally: keep the proposal distribution close to the target

Where Do the Uniforms come From?

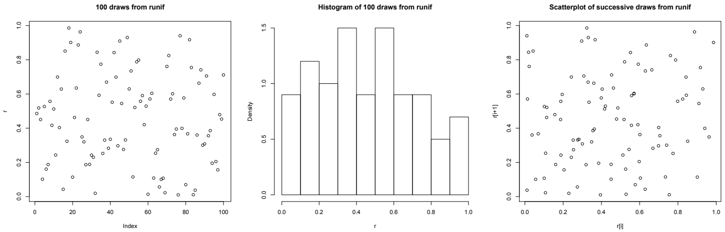
Uniform numbers come from finite algorithms, so really only **pseudo-random**

We want:

- Number of U_i in $[a, b] \subseteq [0, 1]$ is $\propto (b - a)$
- No correlation between successive U_i
- No detectable dependences in larger or longer groupings

Modern pseudo-random generators are now very good at all three

Too involved to go into here, but will show a simpler cousin




```
plot(r,main="100 draws from runif")
plot(hist(r),freq=FALSE,main="Histogram of 100 draws from runif")
plot(r[-100],r[-1],xlab="r[i]",ylab="r[i+1]",
  main="Scatterplot of successive draws from runif")
```

Rotations

Take

$$U_{i+1} = U_i + \alpha \bmod 1$$

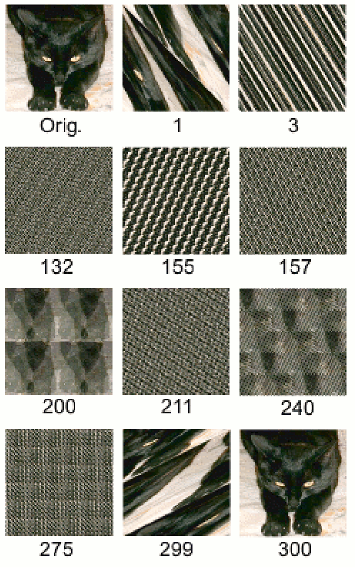
If α is irrational, this never repeats and is uniformly distributed
If α is rational but the denominator is very large, the period is very long, and it is uniform on those points

More Complicated Dynamics

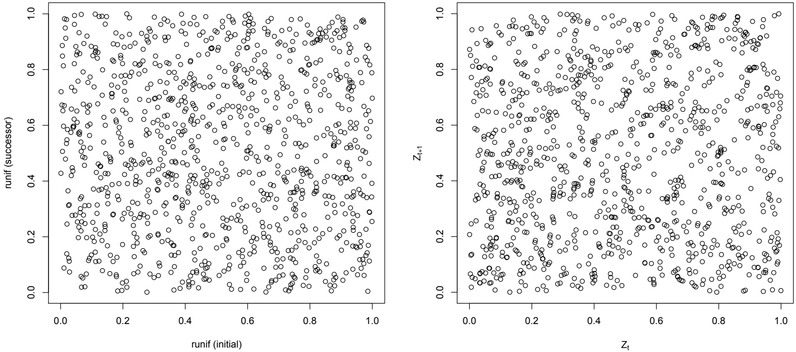
Arnold Cat Map:

$$\begin{aligned} U_{t+1} &= U_t + \phi_t \bmod 1 \\ \phi_{t+1} &= U_t + 2\phi_t \bmod 1 \end{aligned}$$

If we report only U_t, the result is uniformly distributed and hard to predict



Wikipedia, s.v. "Arnold's cat map"



successive values from `runif` vs. Arnold cat map

Similar ideas are built into the random-number generator in R (with more internal dimensions)

Generally: Long periods, rapid divergence of near-by points (unstable), uniform distribution, low correlation

Using the default generator is a very good idea, unless you really know what you are doing

Setting the Seed

The sequence of pseudo-random numbers depends on the initial condition, or **seed**

Stored in `.Random.seed`, a global variable

To reproduce results exactly, set the seed

```
> old.seed <- .Random.seed # Store the seed
> set.seed(20010805)  # Set it to the day I adopted my cat
> runif(2)
[1] 0.1378908 0.7739319
> set.seed(20010805)  # Reset it
> runif(2)
[1] 0.1378908 0.7739319
> .Random.seed <- old.seed  # Restore old seed
```

See Chambers, §6.10, for some subtleties about working with external programs

Summary

- Unstable dynamical systems give us something very like uniform random numbers
- We can transform these into other distributions when we can compute the distribution function
    - The quantile method when we can invert the CDF
    - The rejection method if all we have is the pdf
- The basic R commands encapsulate a lot of this for us