Probability, Simulation, and Sampling

Christopher Brown

November 9, 2018

Introduction

These notes are written for the Math 331 Data Analysis With R course, and are a quick overview of three areas at the boundary between mathematics and computer science. The goals of these notes are:

- Introduce probability terminology and theory
- Introduce the notion of a simulation approach to problem solving and provide examples
- Give an "under-the-hood" glance at some techniques for algorithmically generating random samples.
- Provide some practical examples for using R for simulations and sampling

Probability

Terminology and Notation

An *event space* or *sample space* S is a nonempty set. An *event* is a subset of the given sample space. Visually, we can represent events in sample spaces as .

Probability is a complicated term. In these notes we give two rough definitions of probability, each of which is useful in some scenarios. For a more precise mathematical definition of probability see [1] and [2].

Frequentist definition: Roughly, we determine the *probability of an event* A by repeatedly performing an experiment in which the event A may or may not occur, and the probability is then the ratio of the number of times A occurs to the number of times the experiment is performed. That is, the probability of A is the relative frequency at which A occurs in an experiment repeated many times. For example, if we flip a fair coin 1000 times, we expect that the relative frequency with which we see heads will be about 1/2.

Bayesian definition: Roughly, we determine the *probability of an event* A by evaluating the credibility we have in the occurrence of A given our observations of the system. For example, if we have not flipped a coin, we might give equal credibility to Heads and Tails for the next flip. We base this on the fact that most coins in our experience seem to be fiar. On the other hand, if we have already flipped that coin 10 times and seen 10 heads, we might give more credibility to Heads than Tails for the next flip, because we suspect the coin may be biased.

Notation in probability theory is notoriously complex, informal, and nonstandardized. In these notes we'll try to be consistent and explain what we mean by our notation. There will still be times when informality is helpful.

We usually denote events by capital letters like A, B, and so on. We denote the *probability of event* A by $P(A)$.

When A and B are both events, the event "A and B" is the set $A \cap B$. We write $P(A \text{ and } B) = P(A \cap B)$. Many textbooks also write $P(A, B)$ for the probability of A and B; we won't use this notation but we mention it for the sake of connecting with other works.

Two events are *mutually exclusive* when $A \cap B = \emptyset$.

The event "A or B" is the set $A \cup B$ and we write $P(A \text{ or } B) = P(A \cup B)$.

The event "not A" is called the *complement of* A, and is written A^C. If we need to specify the sample space, we may use the set difference notation $S \setminus A$ instead of A^C.

Properties of Probability

Chapters 2 and 3 of [1] contain a standard presentation of the properties of probability. Chapter 1 of [2] contains a more advanced look at this material. Here, we recount only the axioms and the basic propositions at a basic level.

Suppose that S is a given sample space. Then a probability measure on S is a function P mapping events to real numbers, with P satisfying the following three axioms:

Axiom 1: For every event A, $0 \leq P(A) \leq 1$.

Axiom 2: $P(S) = 1$.

Axiom 3: For any sequence of mutually exclusive events A_1, A_2, ... A_n,...,

$$P\left(\bigcup_{i=1}^{\infty} A_i\right) = \sum_{i=1}^{\infty} P(A_i)$$

From these three axioms, we can then prove the following propositions:

Proposition 4 (Complementation Rule): $P(A^C) = 1 - P(A)$

Proposition 5 (P is Increasing): If $A \subseteq B$ then $P(A) \leq P(B)$.

Proposition 6 (Sum Rule): For all events A and B,

$$P(A \cup B) = P(A) + P(B) - P(A \cap B)$$

Example: Suppose that our experiment is to flip a fair coin twice and record the faces showing on each flip, in order. In this experiment, there are some events - like A="Heads, then Tails" - that are not further divisible into events, because there are no events that are proper nonempty subsets of A. Any event that has no proper nonempty subsets as events is called *atomic*. (There are other events that can be further subdivided, e.g. B="Flip Heads on at least one flip" has A as a proper subset.)

We could list the atomic events for this experiment (there are four, which we might briefly label as HH, HT, TH, and TT). Let's build a probability measure P by assigning $P(A) = 1/4$ for all atomic events A, and then requiring that P satisfy all the probability axioms. Two questions: (1) Can we *actually* require that this P satisfy the probability axioms, or will we encounter an inconsistency? and (2) Can we use this definition of P to compute probabilities of events?

For question (1): Yes, we can. This requires some proof, but the essential observation is that the atomic events are mutually exclusive and union to the sample space S.

For question (2): Yes. For the event B="flip heads on at least one flip", we can compute $P(B)$ in two ways:

$B = \{HH, HT, TH\}$ and so $P(B) = P(HH) + P(HT) + P(TH)$ from axiom 3, and so $P(B) = 1/4 + 1/4 + 1/4 = 3/4$ from our assignment of probabilities to atomic events.

Alternately, we have $B^C = \{TT\}$, so $P(B) = 1 - P(B^C) = 1 - 1/4 = 3/4$.

This is a fairly simple example, but the principle holds for much larger sample spaces. For example, if we are studying five card draw poker probabilities, we would want to consider the sample space with each poker hand as an atomic event. There are

```r
choose(52,5)
```

```
## [1] 2598960
```

possible hands, and so each hand would be assigned a probability of 1/2598960.

More generally, whenever we have a sample space that can be expressed as a union of N atomic events, assigning $P(A) = 1/N$ for all atomic events A and then requiring the probability axioms to hold gives a probability measure.

Conditional Probability

The notion of conditional probability is probably the most important concept in probability theory. In essence, we compute probabilities of events given different collections of information.

If A and B are events, then the *probability of A given that B occurred* is denoted $P(A|B)$. When we know that B occurred, this reduces our sample space to B. A few sketches with Venn diagrams should convince you that

$$P(A|B) = \frac{P(A \cap B)}{P(B)}$$

This expression can be rearranged to read, equivalently, $P(A \cap B) = P(A|B)P(B)$. Section 3.5 of [1] shows that the function $P(\cdot|B)$ mapping events to real numbers is also a probability measure, in that it satisfies the three axioms above.

Because $P(A \cap B) = P(B \cap A)$, we can now write

Proposition 7 (Bayes' Rule):

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

Proposition 8 (Conditioning on a Partition): If B_1, B_2,...,B_n is a collection of mutually exclusive events whose union is S (a *partition* of S), then

$$P(A) = \sum_{i=1}^{n} P(A|B_i)P(B_i)$$

Two events A and B are said to be *independent* if $P(A|B) = P(A)$. That is, A and B are independent if the computation of the probability of A does not change whether we know B happened or not.

Proposition 9: A and B are independent events if and only if $P(A \cap B) = P(A)P(B)$.

Random Variables

A *random variable* is a function mapping events in a sample space to real numbers. We often think of random variables as measurements of quantities of interest in an experiment. For example, if our experiment is to flip a fair coin 20 times, then one random variable might be X=number of Heads in the 20 flips. Here, the range of X is $\{0, \ldots, 20\}$.

In this example, we denote the event "we saw 1 Heads in the 20 flips" as $X = 1$. The notation $X = 1$ should be interpreted to mean "all of the events in the sample space S which, when measured by X, produce an output of 1". (For those of you with some advanced mathematics under your belt, $X = 1$ is one way to write the inverse image of the output 1 under the function X, i.e. the set $X^{-1}(1)$.) *It is very important to understand that when we write $X = 1$ or $X \leq 10$, these are just notations describing certain events, and we should treat them just as we would any other event!*

As a second example, if our experiment is to repeatedly flip a fair coin until the first occurrence of Heads, then a random variable might be X=number of Tails appearing before the first Heads appears. In this example, the range of X is $\{0, 1, \ldots\}$, and so the range of a random variable need not be finite.

A random variable is *discrete* if its range is made up of numbers separated by gaps, with the smallest gap being positive. For example, if the range of X is $\{1, 2, 3\}$ or $\{2, 4, 6, 8, \ldots\}$, X is a discrete random variable. Otherwise, the random variable is said to be *continuous.*

Given a discrete random variable X, the *probability mass function* for X is the function $f(x) = P(X = x)$, and the *distribution function* is

$$F(x) = P(X \leq x) = \sum_{z \leq x} f(z)$$

We note in this case that if $x_1 < x_2$ and no value in the range of X is between x_1 and x_2 then we have $F(x_2) - F(x_1) = f(x_2)$. So, we can create the density function f from the distribution and vice versa.

Given a continuous random variable X, the *probability density function* for X is the function $f(x)$ such that

$$P(X \leq x) = \int_{-\infty}^{x} f(z)\, dz$$

and the *continuous distribution function* for X is the function

$$F(x) = P(X \leq x) = \int_{-\infty}^{x} f(z)\, dz$$

In the language of calculus, $F'(x) = f(x)$, so the rate of change of the distribution is the density. Thus we can create the density from the distribution and the distribution from the density.

In both the discrete and continuous cases, the distribution is a sort of sum of the density or mass, and the density or mass function is a change in the distribution.

Exercises

Simulation

When we *simulate* an experiment, we program the rules of an experiment and some initial conditions into a computer, and then we perform the experiment. Generally, simulated experiments are cheap and highly repeatable. However, they do require us to understand the rules for the experiment very well. So for many questions in probability, simulation can be a highly effective tool because in asking the probability question, we define the rules of the experiment! Some questions in the natural, physical, and social sciences can be approached by simulation experiments, but care must be taken in interpreting the results.

Random Number Generators in R

Probability simulations generally require a random number generator. Chapter 2 of [3] contains a wealth of information, both historical and technical, on random number generation in digital computers. This is a beautiful and absolutely essential topic at the boundary of mathematics and computer science.

We'll approach random number generation on a need-to-know basis. For now, we need to know how R can generate random numbers.

Generally, most computer programming languages have some sort of function that randomly selects a number uniformly distributed on the interval $[0, 1]$. R is "distribution-centric", and so names its function after the uniform distribution.

```r
runif(1)
```

```
## [1] 0.8134679
```

The argument to this function, 1, tells us that R will return a resulting vector of length 1, i.e. only one randomly generated number. If we need to draw several random numbers between 0 and 1, we can do so with a single command.

```r
runif(20)
```

```
##  [1] 0.90628946 0.47155705 0.79626561 0.82319376 0.60785518 0.06533200
##  [7] 0.12961262 0.04396236 0.15465939 0.32607599 0.36964603 0.63371613
## [13] 0.14280372 0.04430646 0.88168881 0.11607580 0.15458605 0.35089747
## [19] 0.09698659 0.55257931
```

Random number generation in R is quite fast. Try executing something like the following on your own.

```r
data <- runif(100000)
```

On my machine, generating these hundred thousand values required about half a second.

Random number generators work something like a roulette wheel; if we knew how fast the wheel was spun, how fast the ball was rolled, and all the little irregularities about how this particular roulette wheel spins, then we could use physics to correctly predict the outcome of the roulette spin. That is, roulette wheels are not really random, they are deterministic and have sensitive dependence on initial conditions; since it is nearly impossible to know the initial conditions precisely enough, the roulette wheel "feels" random. For our digital random number generators, we can set the initial conditions. This means that experiments can be made precisely repeatable. For example, if I execute

```
runif(5)
```

```
## [1] 0.2380025 0.3437795 0.3917654 0.9453424 0.6260722
```

and repeat it

```
runif(5)
```

```
## [1] 0.09258567 0.65489628 0.43063403 0.18077709 0.04113072
```

I don't necessarily get the same values. However, with the set.seed() function, I can set the initial condition - called the "seed" - to a certain value and guarantee a repeatable experiment. Round 1:

```
set.seed(1234)
runif(5)
```

```
## [1] 0.1137034 0.6222994 0.6092747 0.6233794 0.8609154
```

And round 2:

```
set.seed(1234)
runif(5)
```

```
## [1] 0.1137034 0.6222994 0.6092747 0.6233794 0.8609154
```

Other random number generators in R based on common distributions work in much the same way.

Another common task in simulation is to randomly choose some values from a set of values. For example, I may have the data set

```
data <- c('red','blue','green','yellow')
```

and I would like to choose three values randomly. This is called *sampling*, and can be done with or without replacement of values before the next draw. For example, without replacement we have

```r
set.seed(1234)
sample(data,3)
```

```
## [1] "red"   "blue"  "green"
```

and with replacement we have

```r
set.seed(1234)
sample(data,3,replace = TRUE)
```

```
## [1] "red"   "green" "green"
```

When we replace before drawing again, we can resample from the same small data set many times. For example, if I give a multiple choice exam with twenty questions, each of which has a correct answer in choice a, b, c, or d, I might want to generate the answers randomly to avoid human bias. So I can turn to R and sample:

```r
choices <- c('a','b','c','d')
sample(choices, 20, replace = TRUE)
```

```
##  [1] "c" "d" "c" "a" "a" "c" "c" "c" "c" "b" "d" "b" "d" "b" "b" "a" "a"
## [18] "b" "b" "a"
```

If for some reason I want to bias the selection towards choice a, I can increase the number of a's in the drawing pool.

```r
choices <- c('a','a','a','b','c','d')
sample(choices, 20, replace = TRUE)
```

```
##  [1] "a" "a" "c" "b" "d" "c" "a" "a" "a" "a" "b" "a" "c" "a" "a" "d" "c"
## [18] "b" "b" "a"
```

A Programming Aside: The Differences Among For Loops, While Loops, and Apply

In simulation, we will often need to repat an experiment or some action in an experiment many times. There are several common programming structures to accomplish this. In this section I'll take a moment to step to the side and consider these.

When deciding which programming structure to use to repeat an action ("loop"), we need to ask three questions: 1. Before beginning the loop, do we know how many times we need to run the loop? 2. Will any of the repetitions of the loop influence the computation in the *next* repetition of the loop? 3. Will any of the repetitions of the loop influence the computation in the *previous* repetition of the loop?

For us, an answer of *yes* to 3 takes us a little too deeply into a more advanced computer programming topic (recursively defined functions). Also, there are almost no situations we will run into that will give an answer of *yes* to 3. So, let's just assume we don't have to worry about 3! Now on to the more practical concerns…

If the answer to 1 is *no*, we will almost surely need a while loop.

If the answer to 1 is *yes* and the answer to 2 is *yes*, we will almost surely need a for loop.

If the answer to 1 is *yes* and the answer to 2 is *no*, we will be able to use an apply loop. This will happen very frequently for us; we'll see why in the examples in the next section!

Modularity and Simulation

In computer programming, *modularity* is the programming practice in which programs are broken down into small chunks by using functions and comments, in such a way that the chunks can be individually assessed for performance and the code can be more easily understood.

Example: Dice Throw Probability

Let's first simulate a probability we can compute symbolically, as a check: what is the probability of rolling a sum of 7 when rolling a pair of fair six-sided dice?

Generally, we can fit probability simulations into a common framework:

- Define constants for the experiment.
- Create one or more functions to perform one repetition of an experiment (in this case, one roll of the two dice).
- Create a function or code to perform many experiments and record the results of each experiment.
- Analyze the results of the collection of experiments.

Let's begin by defining constants for the experiment.

```r
set.seed(42) # Hitchhiker's Guide
number_of_experiments <- 100000
one_die <- 1:6
```

Next, let's define a function to roll two fair six-sided dice and sum the faces showing.

```r
one_roll <- function() {
  return(sum(sample(one_die,2,replace=TRUE)))
}
```

Before we go on to write the rest of our code, let's test this function.

```r
print(one_roll())
```

```
## [1] 12
```

And again:

```r
print(one_roll())
```

```
## [1] 7
```

And you can execute this a few more times to see how it behaves.

```r
sapply(1:10,function(z) one_roll())
```

```
##  [1]  8  6  9  8  8  9  7  7  7 12
```

This gives us a good idea of how we should proceed for our next phase: repeat the experiment many times and record the data.

```r
data <- sapply(1:number_of_experiments,
              function(z) one_roll())
```

At this point we need to compute the number of 7's we see in the data.

```r
number_of_7 <- length(which(data==7))
```

And now we can compute the probability that we roll a 7.

```r
number_of_7/number_of_experiments
```

```
## [1] 0.16853
```

Symbolically, we have six 7's out of 36 outcomes, i.e.

```r
6/36
```

```
## [1] 0.1666667
```

So our estimate is certainly good enough for government work. If we need to how can we improve our estimate?

Example: A Walk With a Coin

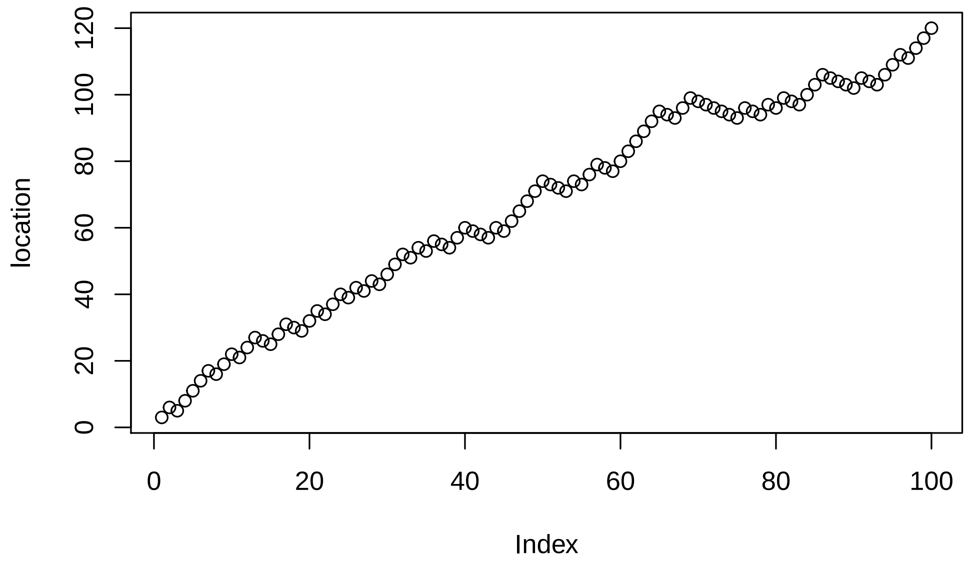
My son is standing on a long, straight sidewalk running east and west. I give him a fair coin and the following procedure:

Step 1: Take 1 step east. Step 2: Flip the coin. If Heads, take two more steps east. If Tails, take two steps west.

After 100 steps, where is he likely to be? We know that he could be anywhere from 100 steps west to 300 steps east, but how likely are any of those possibilities?

Let's model this with steps east being represented by positive numbers and steps west represented by negative numbers. We can simulate one trip fairly easily.

```r
set.seed(42)
coin_flips <- sample(c(-1,1),100,replace=TRUE)
extra_steps <- 2*coin_flips
location <- cumsum(1+extra_steps)
plot(location)
```



The final location is the data we really wish to collect.

```r
tail(location,1)
```

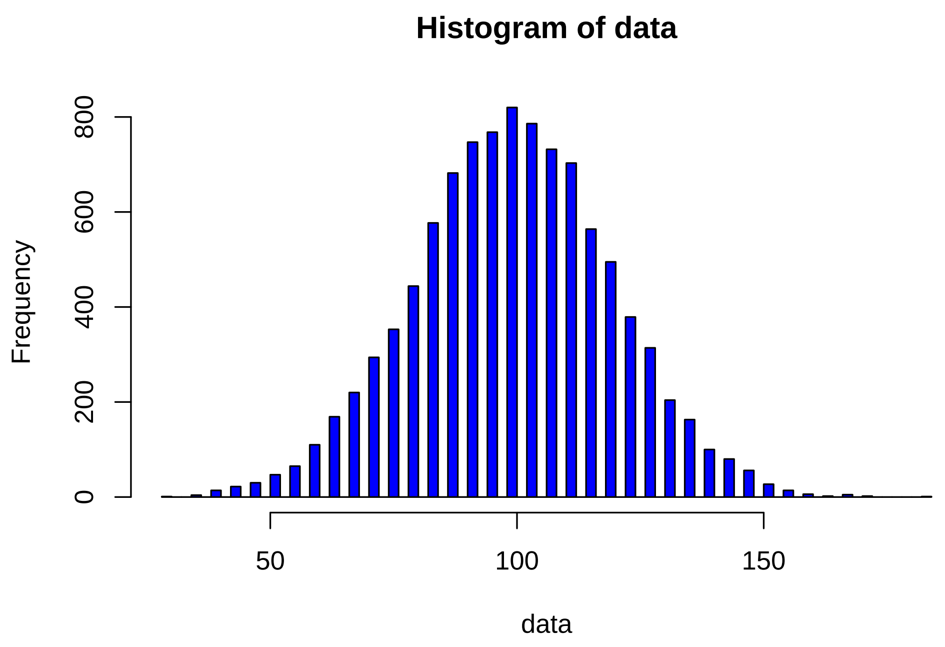
```
## [1] 120
```

Let's now write a function to perform one trip and then repeat that.

```r
set.seed(42)
number_of_trips <- 10000
one_trip <- function() {
  coin_flips <- sample(c(-1,1),100,replace=TRUE)
  extra_steps <- 2*coin_flips
  location <- cumsum(1+extra_steps)
  return(tail(location,1))
}
data <- sapply(1:number_of_trips,function(z) one_trip())
```

Let's examine the data.

```r
hist(data,breaks=60,col='blue')
```



How do we interpret this histogram?

Example: Professor Z's Office Hours

Professor Z tries to show up at his office for appointments on time. He really does! But students and faculty frequently stop him to ask questions or just chat, and he is a typical absent-minded professor, so sometimes he is late.

Estrella is trying to meet Professor Z at his office for an appointment. She really is! But between traffic and campus parking, Estrella might be late.

Professor Z and Estrella are both busy, so they can't wait long for the other person.

(a) Let's assume that Professor Z and Estrella arrive at the appointment at some time uniformly distributed between noon (time 0) and 1:00PM (time 1), and let's assume each will wait for the other for 15 minutes but not longer. Write a simulation to estimate the probability that Professor Z and Estrella actually meet for their appointment.

(b) Now let's assume that Professor Z and Estrella each arrive at the appointment at a time uniformly distributed between noon and 1PM. Let's also assume that each will wait for a uniformly distributed random amount of time between 0 and 20 minutes. What do you expect to happen to the probability the two meet in this scenario, as compared to that in part (a)? Write a simulation to estimate the probability that Professor Z and Estrella actually meet for their appointment.

Part (a)

Define our constants.

```r
set.seed(42)
number_of_experiments <- 100000
wait_time <- 15
```

Define functions to perform one repetition of this experiment.

```r
one_rep <- function() {
  arrival_time <- runif(2,min=0,max=60)
  abs(arrival_time[1]-arrival_time[2])<=15
}
```

How can we test the one_rep function?

Now repeat the experiment many times and record the data.

```r
data <- sapply(1:number_of_experiments,
       function(z) one_rep())
```

Finally we can analyze the data and compute the probability of interest.

```r
length(which(data))/number_of_experiments
```

```
## [1] 0.43483
```

Part (b)

Define our constants.

```
number_of_experiments <- 100000
```

Define functions to perform one repetition of this experiment.

```
one_rep <- function() {
  arrival_time <- runif(2,min=0,max=60)
  wait_time <- runif(1,min=0,max=20)
  abs(arrival_time[1]-arrival_time[2])<=wait_time
}
```

How can we test the one_rep function?

Now repeat the experiment many times and record the data.

```
data <- sapply(1:number_of_experiments,
       function(z) one_rep())
```

Finally we can analyze the data and compute the probability of interest.

```
length(which(data))/number_of_experiments
```

```
## [1] 0.29411
```

Does this probability agree with your expectation?

Example: Optimization

Suppose that we are trying to maximize the function $f(x) = x^5 e^{-2x}$ on the interval $[0, 5]$. We could take a calculus approach, but why do that when we have a computer in front of us? We could just evaluate $f(x)$ for LOTS of x values and see which gives the largest output.

```
set.seed(42)
inputs <- runif(10000,min=0,max=5)
f <- function(x) {
  return(x*(1-x))
}
outputs <- f(inputs)
```

The maximum value for this function on $[0, 5]$ is

```
max(outputs)
```

```
## [1] 0.2499996
```

which occurs at x-value

```
index <- which.max(outputs)
inputs[index]
```

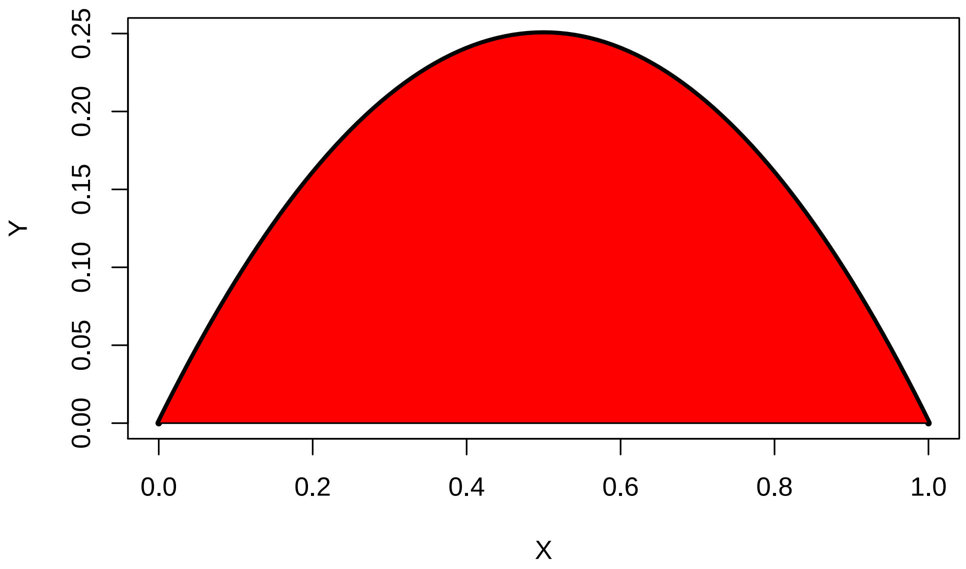
```
## [1] 0.5006234
```

Can you convince yourself that this is (approximately) the right answer?

Example: Area Under a Curve

Let's suppose that we want to compute the area under the curve $f(x) = x(1-x)$ on the interval $[0, 1]$. Graphically, we are trying to compute the area shown below:

```
X <- seq(from=0,to=1,by=0.01)
f <- Vectorize(
  function(x) {
    return(x*(1-x))
  }
)
Y <- f(X)
plot(X,Y,type='l',lwd=4)
XX <- c(X,rev(X))
YY <- c(Y,rep(0,length(X)))
polygon(XX,YY,col='red')
```



So, how can we do this? The thinking is to throw darts randomly at the graph above, compute the proportion of the darts that land in the area, and then use that to compute the area of the shaded region.

First, let's get the coordinates of our darts.

```
xmin <- 0
xmax <- 1
ymin <- 0
ymax <- 0.3
number_of_darts <- 10000
X <- runif(number_of_darts,min=xmin,max=xmax)
Y <- runif(number_of_darts,min=ymin,max=ymax)
```

Let's also compute the area of the image.

```
image_area <- (xmax-xmin)*(ymax-ymin)
image_area
```

```
## [1] 0.3
```

We now need to check to see which darts fall in the shaded region. A dart will fall in the region exactly when the y coordinate of the dart is smaller than the y coordinate of the graph at that x coordinate, i.e. exactly when $y \leq f(x)$.

```r
darts_in_region <- length(which(Y <= f(X)))
```

We now compute the proportion of darts in the shaded region.

```r
darts_in_region/number_of_darts
```

```
## [1] 0.5542
```

So about 55.4% of the darts fell in the shaded region, which tells us that the shaded region takes up about 55.4% of the image area 0.3. We can then estimate the area of the shaded region as

```r
darts_in_region/number_of_darts * image_area
```

```
## [1] 0.16626
```

It's worth noting that elementary calculus tells us that the exact answer is

```r
integrate(f,lower=0,upper=1)
```

```
## 0.1666667 with absolute error < 1.9e-15
```

So our darts estimate is quite good. Computationally, it does not beat the standard numerical integration techniques as long as we have, but it may be helpful in other ways.

Exercises

Sampling

In this section we will dig a little more deeply into how random number generators for various distributions can be programmed. This section is a little sparser, but contains links to lots of useful web content.

Distributions in R

Here is some information about the "built-in" distributions in R: https://stat.ethz.ch/R-manual/R-devel/library/stats/html/Distributions.html . In an introductory statistics class, you will probably have encountered the binomial, Poisson, normal, and chi-squared distributions.

The Monte Carlo Method

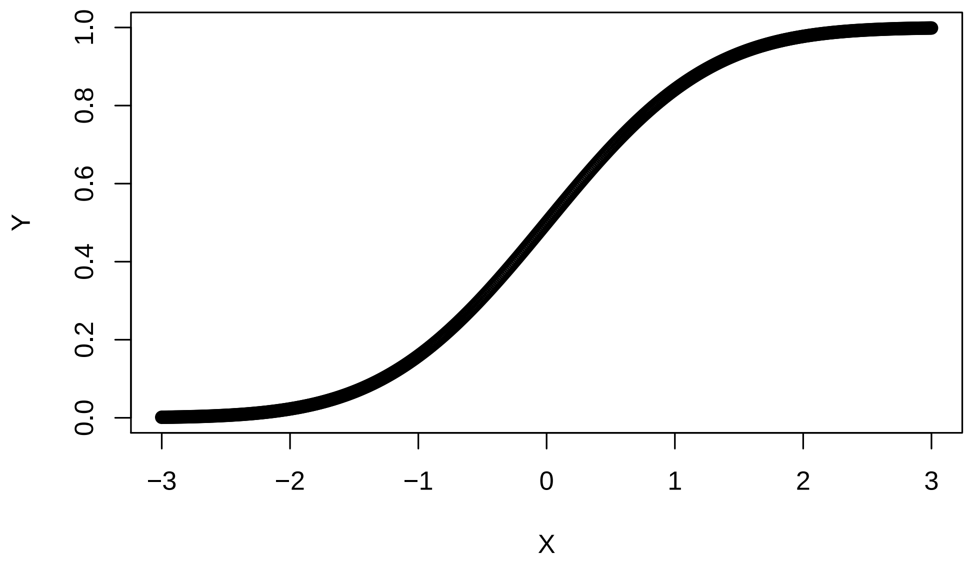
The Monte Carlo method is really not a method, but an entire class of computational algorithms that focus on answering questions through simulation. Read more here: https://en.wikipedia.org/wiki/Monte_Carlo_method . (As a point of interest, I was supported by Stan Ulaw's seminar and scholarship as a graduate student.) References [3] and [4] contain some excellent material discussing Monte Carlo methods. Reference [4] contains code and excellent exercises written in Matlab; Python users using the numpy and pylab packages will find this highly approachable. [5] is a more advanced mathematical text, aimed at clarifying the connections between Monte Carlo methods and the Bayesian approach to statistical analysis. [6] moves in the computational direction instead, clarifying the connections between Monte Carlo methods and the Bayesian approach using Python and the pymc3 package.

In the simulation section of these notes we have seen examples of estimating probabilities through simulation and optimizing functions through simulation. However, we have not really seen simulations used to generate samples from distributions of continuous random variables. How do computer programs work "under the hood"? How does R generate 10000 draws from a beta distribution? We'll tackle this in the next two sections.

Inversion Sampling

Suppose that X is a continuous random variable with distribution function $F(x) = P(X \leq x)$. Then F is an increasing function of x, and will have an "s shape",similar to the following.

```r
X <- seq(from=-3,to=3,by=0.01)
Y <- pnorm(X)
plot(X,Y)
```



These curves are sometimes called *s-shaped* curves, informally.

Let's note three facts about this distribution function F. First,

$$\lim_{x \to -\infty} F(X) = 0$$

Second,

$$\lim_{x \to \infty} F(x) = 1$$

And third, F is an invertible function.

Inversion sampling uses these three facts and the random number number generator for the uniform distribution on $[0, 1]$. The process is as follows:

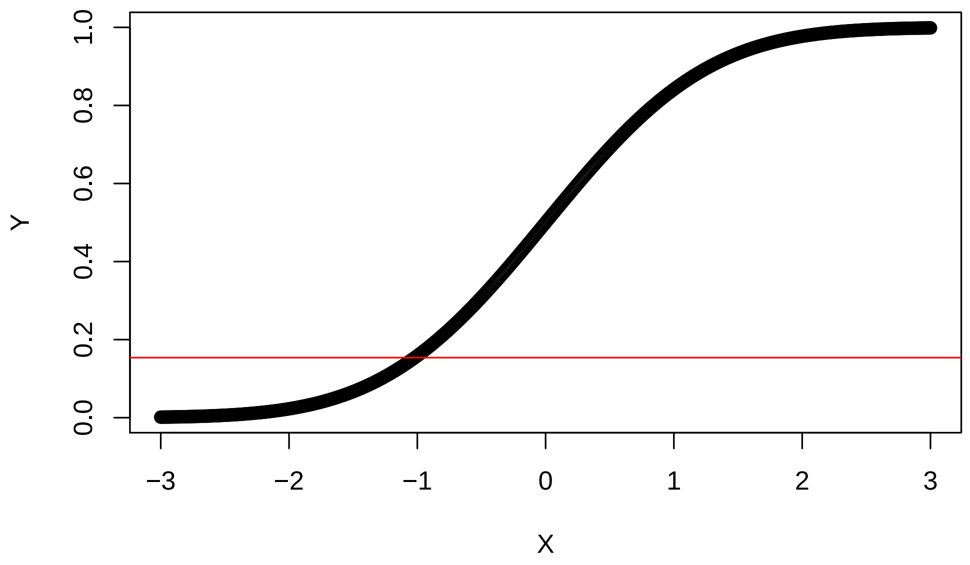
Set-up: You'll need your distribution function in some format. For this example I'll take mine to be written in the vectors of X and Y coordinates used to generate the graph above.

Step 1: Draw a random number from the uniform distribution on $[0, 1]$. This will represent a random output of your distribution function, so I will call mine y.

```
y <- runif(1)
```

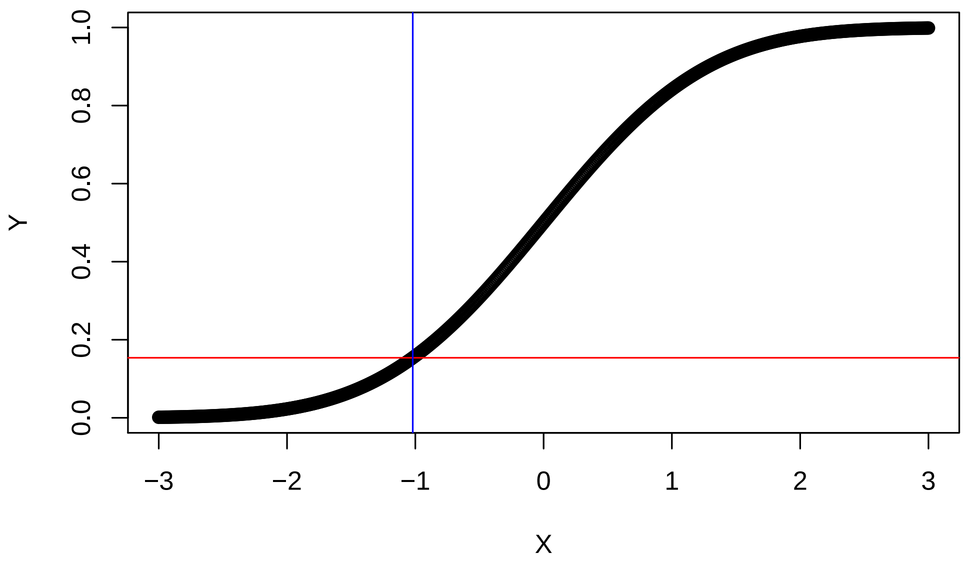
Step 2: We think of plotting y on the vertical axis in the graph above and then finding the x value corresponding to it.

```r
set.seed(42)
plot(X,Y)
abline(h=y,col='red')
```



Typically this involves some sort of search through a list. We can visualize this as finding the x-coordinate in the distribution corresponding to the y coordinate we drew.

```r
location <- which.min(abs(Y-y))
the_draw <- X[location]
plot(X,Y)
abline(h=y,col='red')
abline(v=the_draw,col='blue')
```

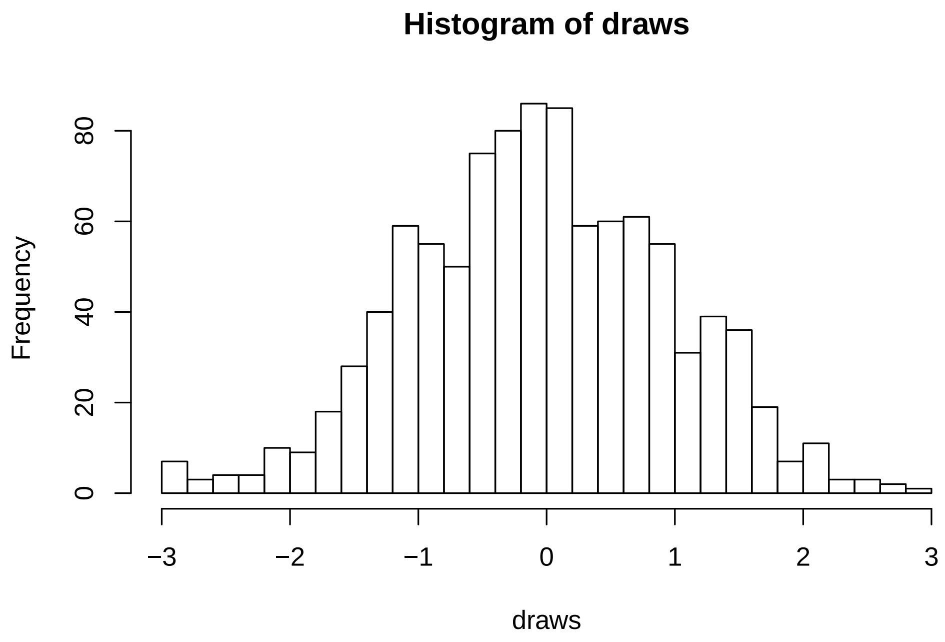


Here we have drawn a sample value of -1.02. If we repeat this many times, we generate many draws from the distribution.

```r
set.seed(42)
y <- runif(1000)
find_X <- function(y) {
  location <- which.min(abs(Y-y))
  return(X[location])
}
draws <- sapply(y,find_X)
```

Our data is in the variable draws; what does it look like?

```r
hist(draws,breaks=40)
```



Our sample seems to fall roughly into the shape of a normal distribution. Because our random variable was normal, this seems like a good confirmation of this sampling technique.

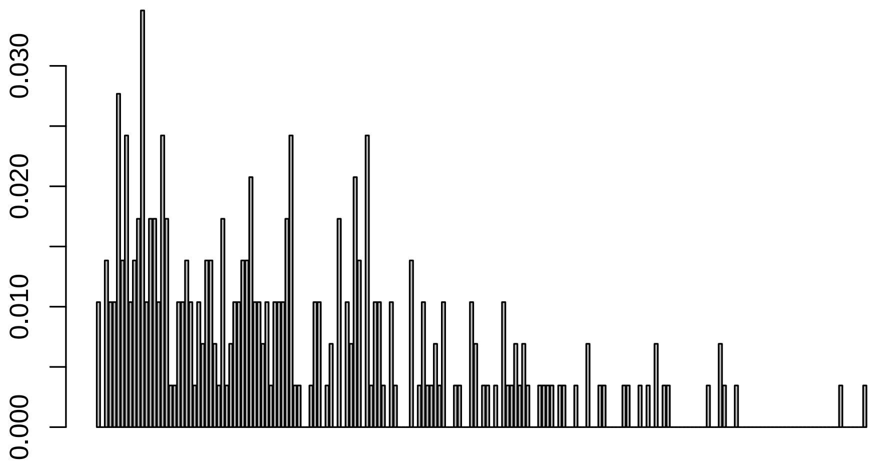
How does this work for discrete random variables and for data? As an example, let's look at the yearly sunspot data set in R.

```
data(sunspot.year)
spots <- ceiling(sunspot.year)
head(spots)
```

```
## [1]  5 11 16 23 36 58
```

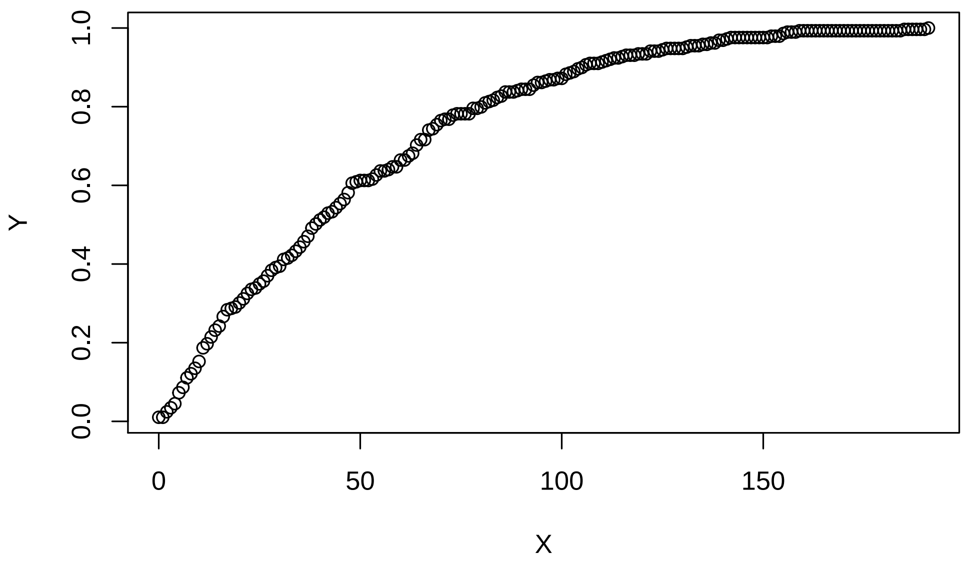
This is just the number of sunspots recorded each year, in order from 1700 to 1988. Let's first find the mass function.

```
countem <- function(data,x) {
  sum(data==x)
}
X <- 0:max(spots)
temp <- sapply(X,function(z) countem(spots,z))
f <- temp/sum(temp)
barplot(f)
```



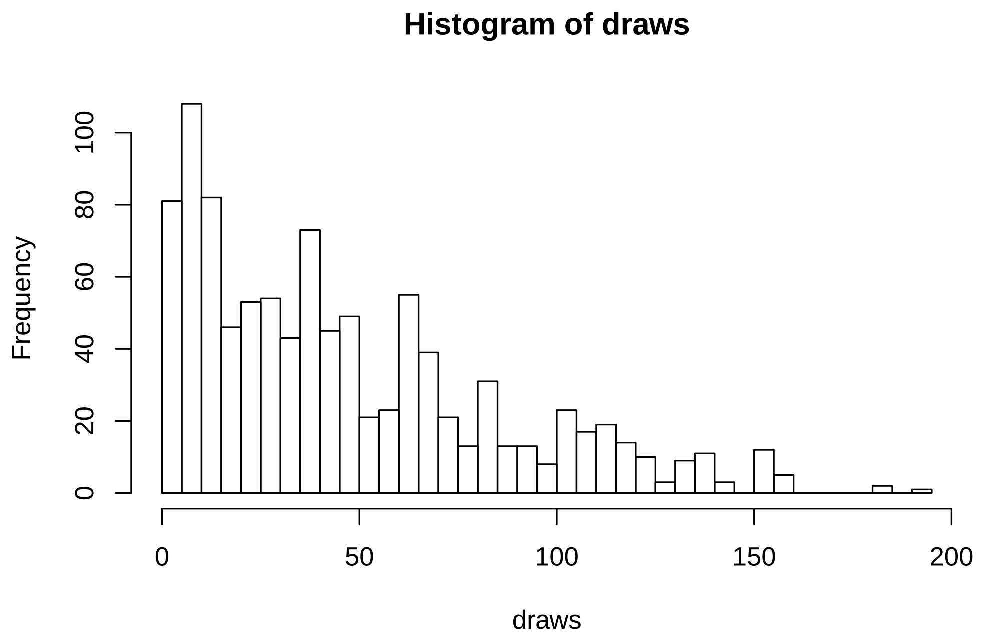
We can now find the distribution function by summing.

```
Y <- cumsum(f)
plot(X,Y)
```



And now that we have the distribution we can perform inversion sampling as we did for the normal distribution.

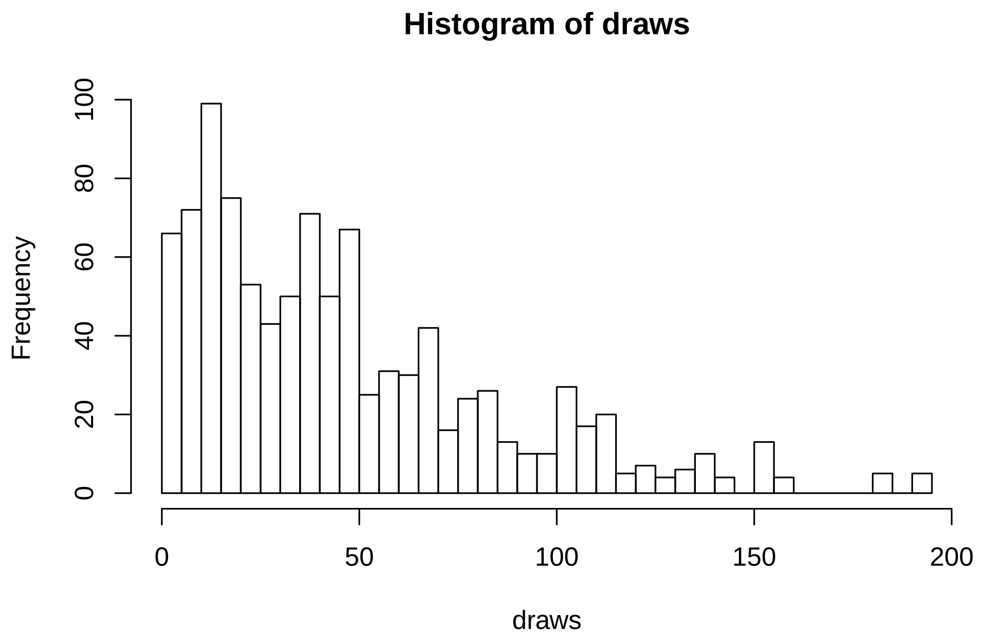
```
set.seed(42)
y <- runif(1000)
find_X <- function(y) {
  location <- which.min(abs(Y-y))
  return(X[location])
}
draws <- sapply(y,find_X)
hist(draws,breaks=60)
```



This looks similar to our original distribution of sunspots, so it seems good.

However, I have to stress that this is definitely not the most efficient way to sample from data representing a discrete random variable; it's a "hypothetically, we could…" discussion. The easiest approach is to just use the sample function on your original data set! The procedure we ran through is more or less what R does under the hood with the sample function.

```
draws=sample(spots,1000,replace=TRUE)
hist(draws,breaks=60)
```

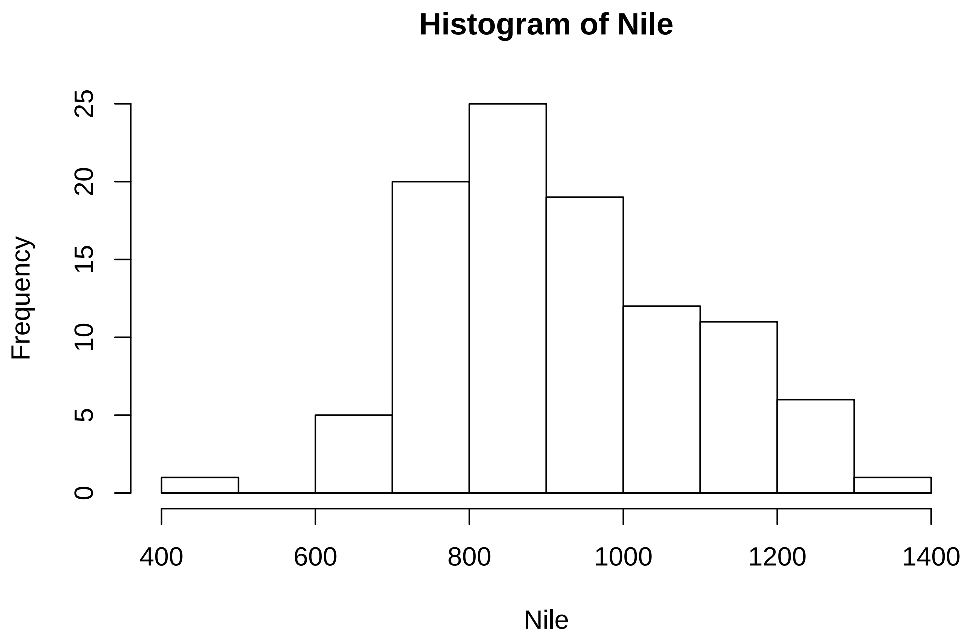


Okay, but what if our random variable is continuous and we wish to sample from it? Let's examine the Nile River water flow data set.

```r
data(Nile)
head(Nile)
```

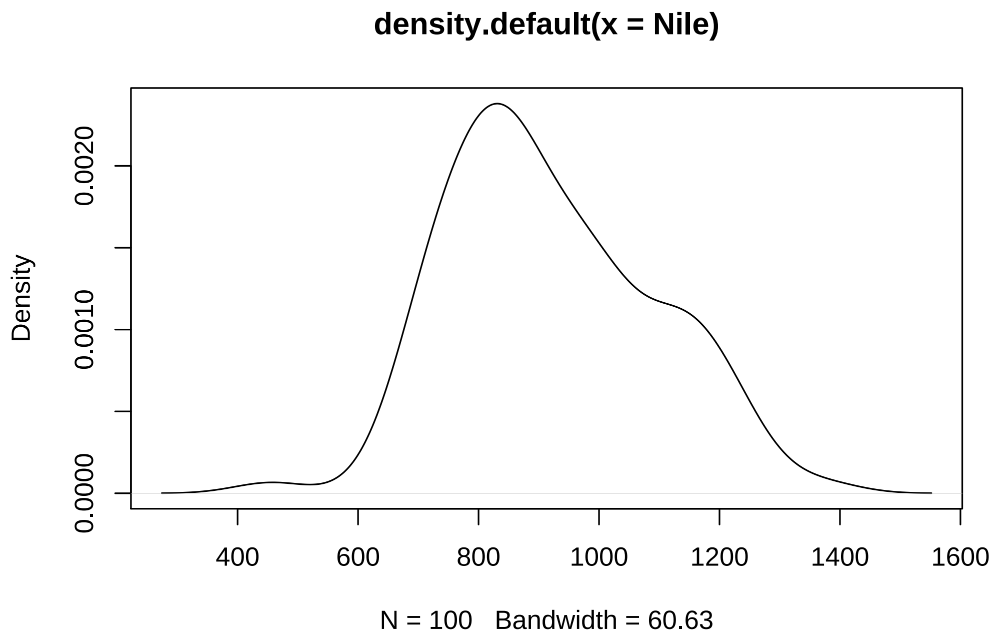
```
## [1] 1120 1160  963 1210 1160 1160
```

```r
hist(Nile)
```

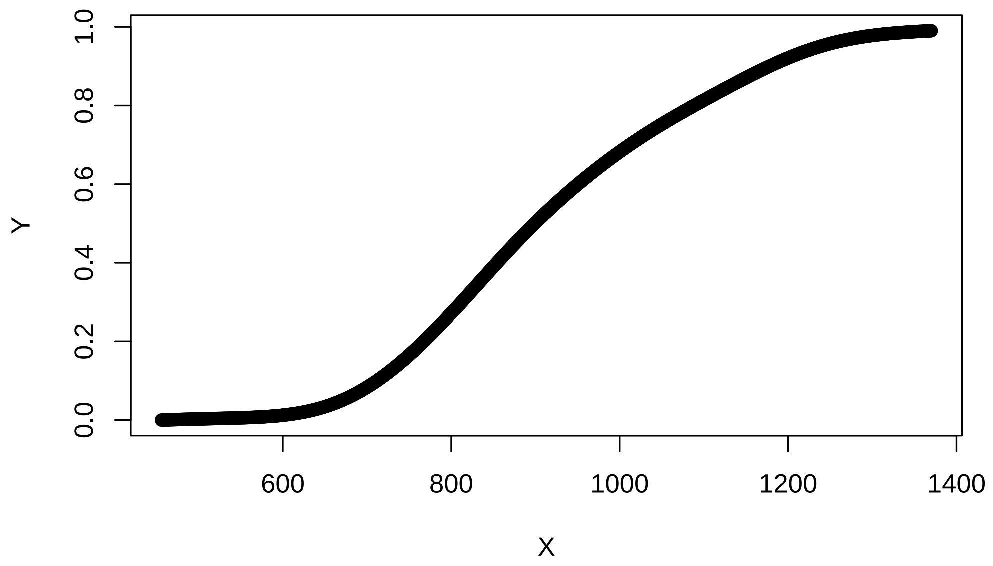


We can fit a smooth density function to the Nile data. (Use ?density in R to read more about this function.)

```
Nile_density <- density(Nile)
plot(Nile_density)
```

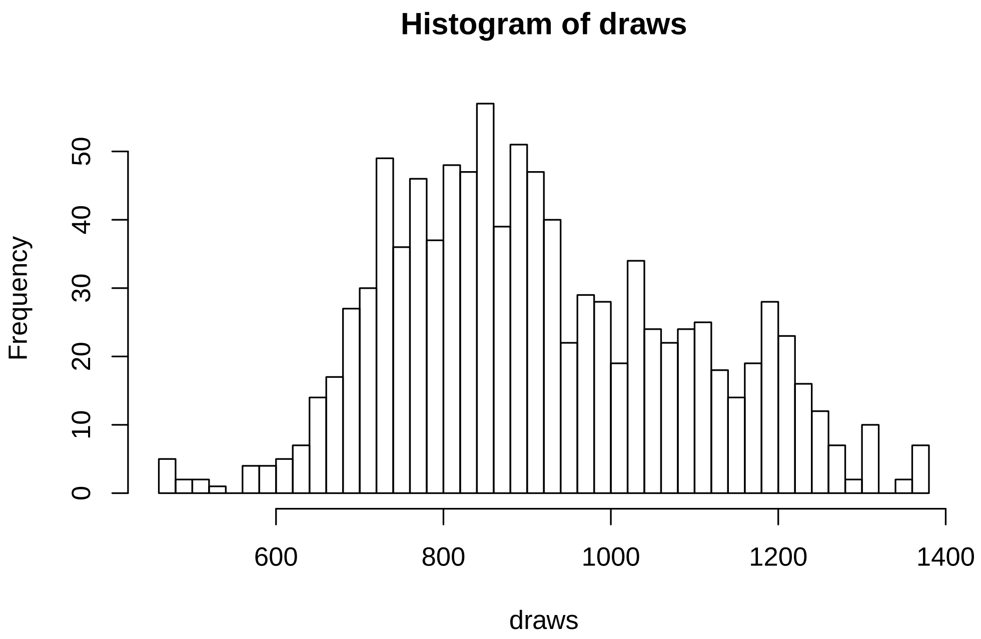


```
Nile_approx <- approxfun(Nile_density)
f <- function(x){
  if (x<min(Nile) | x>max(Nile)) {
    r <- 0
  } else {
    r <- Nile_approx(x)
  }
  return(r)
}
f <- Vectorize(f)
X <- min(Nile):max(Nile)
Y <- sapply(X,function(z) integrate(f,lower=-Inf,upper=z,stop.on.error = FALSE)$value)
plot(X,Y)
```



We can now sample from this distribution function using inversion sampling.

```r
set.seed(42)
y <- runif(1000)
find_X <- function(y) {
  location <- which.min(abs(Y-y))
  return(X[location])
}
draws <- sapply(y,find_X)
hist(draws,breaks=60)
```



The benefit of this approach over sampling from the original data set is that the approximating density function has smoothed out the data. We now get some data points in the gaps in between the values in the original data set. This fits our mental model of flow rates of the Nile; there's no reason that the river may have a flow rate of 702 one year and 714 another year, but never 710. If we sample from the original data set we'll never draw a sample value of 710, but with the smoothed density, we might.

Rejection Sampling

Since we already have a sampling method, why introduce another? Well, inversion sampling works quite efficiently for a single random variable but not for multivariable scenarios.

Instead, we'll use a technique called *rejection sampling* (or sometimes "acceptance-rejection sampling"). This technique is quite similar to the darts-throwing technique we used to estimate integrals earlier. The idea is that we will throw a "dart" into a region with a density function. If the dart falls under the graph of the density function, we will accept the appropriate coordinates of the dart as a sample point, and if the dart fals above the graph of the density function we will reject the dart for our sample and move on to another.

Let's begin by defining a density function. We start with a shape.

```
f_temp <- function(x1,x2) {
  if (x1>=0 & x1<=2 & x2>=0 & x2<=2) {
    return(5*exp(-x1-x2))
  } else {
    return(0)
  }
}
```

We want the volume under this to be 1. Let's throw darts to find the current volume!

```
set.seed(42)
number_of_darts <- 100000
X1 <- runif(number_of_darts,min=0,max=2)
X2 <- runif(number_of_darts,min=0,max=2)
Y <- runif(number_of_darts,min=0,max=5)
outputs <- sapply(1:length(X1),function(j) f_temp(X1[j],X2[j]))
volume <- length(which(Y<=outputs))/number_of_darts * 2*2*5
volume
```

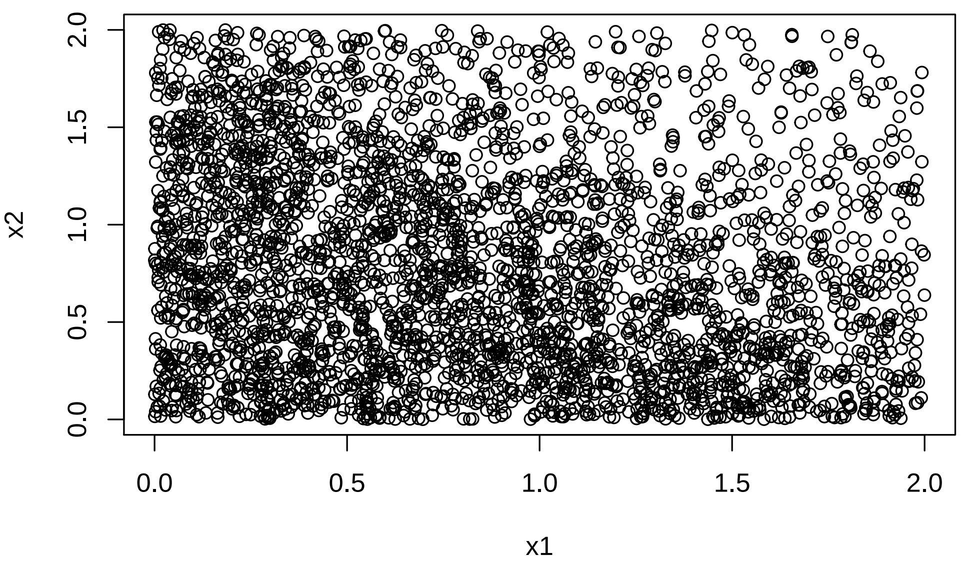
```
## [1] 3.7334
```

And now we can scale our function so that it is a proper density function, i.e. integrates to 1.

```
f <- function(x1,x2) {
  return(f_temp(x1,x2)/volume)
}
```

At this point we have a density function and we can perform rejection sampling. Much of this code will look similar to the code we used to create darts. Pay attention to the y values, though! Let's draw a sample of 1000 points.

```
set.seed(42)
number_of_darts <- 5000
X1 <- runif(number_of_darts,min=0,max=2)
X2 <- runif(number_of_darts,min=0,max=2)
Y <- runif(number_of_darts,min=0,max=1)
outputs <- sapply(1:length(X1),function(j) f_temp(X1[j],X2[j]))
keepers <- which(Y<=outputs)
x1 <- X1[keepers]
x2 <- X2[keepers]
plot(x1,x2)
```



Note the heavier clustering near $(0, 0)$. The density function is maximal there, and so we should expect tighter clustering in our sample.

There are a number of ways to improve a rejection sampling scheme. Chapter 2 in [3] has more details on rejection sampling, and chapter 2 in [4] has several examples of designing sampling algorithms using inversion schemes and in some cases and rejection sampling in others.

Exercises

References

[1] Ross. **A First Course in Probability: Fifth Edition**. Prentice-Hall, 1998.

[2] Grimmett, Stirzaker. **Probability and Random Processes: Third Edition**. Oxford University Press, 2001.

[3] Rubinstein and Kroese. **Simulation and the Monte Carlo Method: Third Edition**. Wiley, 2017.

[4] Shonkwiler, Mendivil. **Explorations in Monte Carlo Methods**. Springer, 2009.

[5] Gamerman, Lopes. **Markov Chain Monte Carlo: Stochastic Simulation for Bayesian Inference, Second Edition**. Chapman and Hall/CRC, 2006.

[6] Pilon. **Probabilistic Programming and Bayesian Methods for Hackers**. Find this at https://github.com/CamDavidsonPilon/Probabilistic-Programming-and-Bayesian-Methods-for-Hackers.