Computational Statistics

Lecture notes

Mauricio B. García Tec

Instituto Tecnológico Autónomo de México

August 26, 2015

Contents

Chapter 1

Introduction

Computational statistics is a disciple lying within the interface of statistics and numerical analysis. The idea is to develop algorithms to compute, estimate or interact with statistical or mathematical objects of interest that would be hard to deal with analytically.

This course is not about learning how to programme or how to use a particular language. For expository purposes, however, we will give several examples using the software `R`. I assume the reader has some familiarity with this language, so in the examples I will often comment about efficient and advanced ways of using the tools available in `R` for implementing our algorithms. However, the reader can apply the algorithms we will study here on the programming language of their choice.

The core mechanism behind all the methods we shall cover is the use of randomness in our advantage. In the next chapter, we will learn how to simulate random numbers, and what we precisely mean by them. But suppose for the moment, however, that we know how to generate a sequence of *independent* random numbers that are taken uniformly from $(0,1)$[1]. The following example will show how to use these sequences to compute an approximation of the constant π.

Example 1.1 (Our first computation) The idea is the following: if you draw a quarter circle of radius 1 in the square $[0,1]\times[0,1]$, the area enclosed by this quarter circle is $\pi/4$. If we were to produce a sequence of random numbers uniformly distributed over this square, the fraction of points that would lie below the curve $f(x)=\sqrt{1-x^2}$ should be approximately $\pi/4$ after enough points have were considered.

In the following example we generate two sequences of uniform random numbers in $(0,1)$ using the `R` function `runif` and compute the fraction of points that lie below the quarter circle. Our final estimate of π is given by multipliying this fraction by four.

[1] Since we are working with computers, there is actually no such thing as a continuous uniform distribution in $(0,1)$; the machine precision constraints imply that there is actually only a finite quantity of numbers in $(0,1)$

```r
set.seed(110104) # Good practice!
nsim <- 1000 # The number of points we will generate
xcoord <- runif(nsim) # We store a random sequence for each coordinate
ycoord <- runif(nsim)
dist.orig <- sqrt(xcoord^2+ycoord^2) # Distance to (0,0)
hits <- dist.orig < 1 # Number of points lying below the circle of radius 1
area <- sum(hits)/nsim # Fraction below
4*area # Final Estimate

## [1] 3.112
```

Figure 1.1 shows an illustration of the method. The command to produce the figure is shown below.

```r
qplot(xcoord, ycoord, colour=hits)
```

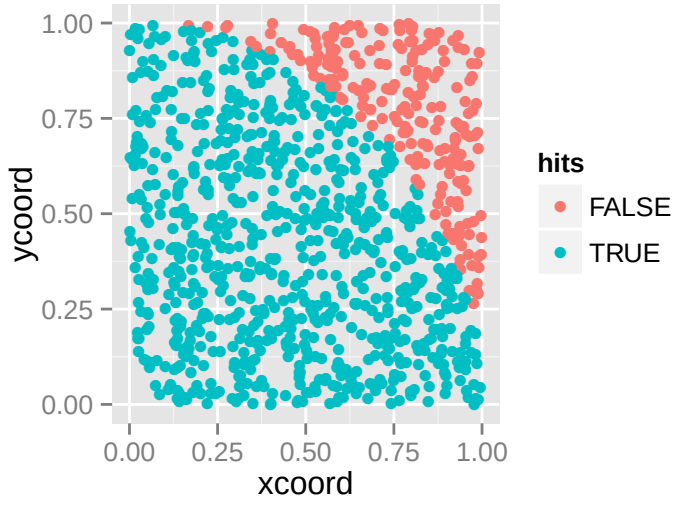

Figure 1.1: Hit and miss method for computing π

Later in this course we will look for ways of making these computations more efficient.■

Let us now give a different example to motivate our study of Computational Statistics. In fact, an example like this is what motivated the Monte-Carlo method, which will be one of the topics covered in this course. We will see a little bit of its history later, but these statistical algorithms were one of the first uses of modern computers dating back to the times of the ENIAC. The method was said to be devised by the great mathematician Stanislaw Ulam when failing to analytically compute a probability of a Solitaire game [Eck87].

Example 1.2 (Computing a hard probability) Imagine the following game. We start with five dice. We throw them and then we count the number of sixes appearing. If there are no sixes at all, then we have lost. If there is at least one six, we take away the dice having a six and throw again the remaining dice. Once more, if there are no sixes, we lose, otherwise, we keep throwing the remaining dice until either we run out of dice and win or we throw no sixes and lose. Analytically obtaining the exact probability of winning can be a tedious exercise. Instead, the following code in R simulates several runs of the game and calculates the fraction of games that are won. This fraction is an approximation of the probability of winning.

```
set.seed(110104)
simulate.game <- function(...){ # 1) Simulate one run of the game
  continue <- TRUE # Dummy to know if we must throw again
  win <- TRUE # Dummmy that will change to false if one throws no 6 at any point
  dice.left <- 6 # Dice to throw at next round
  while(continue){
    throw <- sample.int(6, dice.left, replace=TRUE)
    hits <- sum(throw == 6)
    if(hits==0){
      win <- FALSE
      continue <- FALSE
    } else{
      dice.left <- dice.left - hits
      if(dice.left==0) continue <- FALSE
    }
  }
  return(win)
}
compute.prob <- function(nsim){ # 2) Run several times and average
  results <- sapply(1:nsim, FUN=simulate.game)
  return(sum(results)/nsim)
}
c(compute.prob(10), compute.prob(100), compute.prob(1000),
compute.prob(1000), compute.prob(10000)) # Different approx. of the prob.

## [1] 0.0000 0.0000 0.0130 0.0200 0.0192
```

So the probability of winning is very low and its approximate value is 2%. Notice the use of the function `sapply` in the code. Using `sapply` instead of a usual `for` makes the computations in R a lot faster. The strategy is called vectorisation. ■

Chapter 2

Random Number Generation

As we have discussed. The idea of computational statistics is to use randomness to compute, estimate or sample from a statistical/mathematical object of interest. The first problem we must address is to solve what do we actually mean by randomness and how we can obtain a sequence of numbers that qualify as random.

In the first section we will learn how to generate a sequence of uniformly distributed random numbers in $(0, 1)$. Later we shall see how we can sample other probability distributions from our uniform sequences.

There are basically two strategies to produce the desired sequences: *Pseudo-Random Number Generation* and *True-Random Number Generation*.

As the name suggests, pseudo-random numbers are not entirely random, but they are produced by following a mathematical 'recipe' that guarantees that they behave similar enough to how observations of a true-random numbers do. True-random number generation is a very interesting topic in itself. The usual way in which true random numbers can be obtained is by taking into advantage the randomness of some physical phenomena; some classical examples include tossing a coin or throwing dice. Modern methods utilise quantum effects, thermal noise in electric circuits, the timing of radioactive decay, etc. An example of a provider of true-random numbers is `random.org`.

Example 2.1 The package `random` can be used in `R` to connect to `random.org` and obtain true random numbers. The outcome is not precisely uniformly in $(0, 1)$. One must specify a minimal and a maximal integer value and the outcome is a number or a sequence of numbers sampled from this range of integers. Evidently, one can obtain a sequence in $(0, 1)$ by dividing into the upper range from which the numbers were taken from. Unfortunately, I cannot run these example for these notes. But open an `R` console and type them yourself and try to figure out how they work!

1) To use the package `random` one must use the following instructions.

```
library(random)
rseq <- randomNumbers(n=10, min=1, max=10000) # Generate 10 true random numbers
# in between min and max
```

2) If, instead, you would like to use the connectivity properties of `R` and build your own program that extracts numbers from `random.org`, you can use the following code, which uses the simplest form of `html` syntax for this purpose.

```
library(XML)
library(RCurl)
true.random <- function(nsim, digits = 5){
  # Put together a valid URL address.
  url <- paste0('https://www.random.org/integers/',
          '?num=', nsim,'&max=', format(10^digits, scientific=FALSE),
          '&min=1&col=1&base=10&format=html&rnd=new')

  # Compile the code to extract the tag containing the numbers.
  # Take a look at the value of url to know why this works.
  html.code <- getURL(url)
  html <- htmlTreeParse(html.code, useInternal = TRUE)
  numbers.text <- xpathApply(html, "//pre[@class='data']", xmlValue)[[1]]
  numbers <- strsplit(numbers.text, split="\\n")[[1]]
  return(as.numeric(numbers)*10^(-digits)) # Normalise to (0,1)
}
```

For this course, it will not be necessary to use true random numbers. However, if you want to compare the performance of any algorithm we study, you can use one of this sequences.■

There is a vast amount of research dedicated to find efficient ways of generating true random numbers. In applications like cryptography, it is very important to have unpredictable sequences: if someone was to decipher the algorithm that is being used to encrypt data, then there could be really bad consequences. On the contrary, for many of the algorithms we will study in this course; all we need is that the sequence of numbers generated behave 'reasonably' similar to a sequence of random numbers in $(0, 1)$. When working with true random numbers, the result are unreproducible. When working with pseudo-random numbers instead, one often specifies a seed and one constructs the rest of the pseudo-random sequence based on this number; so the results are always repeatable. Reproducibility can be both an advantage or a disadvantage depending on the particular application.

2.1 Pseudo-Random Uniform Generation

The low efficiency of generating true random numbers is what motivates pseudo-random numbers. We will not do a seep survey of methods for generating pseudo-random numbers. In fact, it is not even advisable that you use the method presented here for your algorithms. There is a vast amount of research that has taken place in this field. So it is better to use well-tested methods. Having said this, the method we will describe now is surprisingly good given its simplicity.

Here is a short list of some things we will be looking for in a good random number generator:

1. It must have the moments and quantiles as the uniform distribution: mean 0.5, median 0.5, standard deviation $\sqrt{1/12}$, etc.

2. We would like to test its conformity with the uniform distribution by using some goodness-of-fit test (e.g., the χ^2 test).

3. Since we want to simulate from a sample of uniformly distributed independent variables in $(0,1)$, the produced numbers should look independent. For instance, we will not want that a number is correlated with its predecessors in the sequence.

Do not underestimate the last point above, as it will fail with our method if we do not choice the right initial parameters. The *linear congruency method*, introduced by David Lehmer in 1949, is defined as follows [Leh43]:

Definition 2.2 A *linear congruential generator* (LCG) with modulus $M \in \mathbb{N}$ (meant to be large), multiplier $a \in \{1,2,\ldots,M\}$, increment $c \in \{0,1\ldots,M\}$ and seed $x_0 \in \{0,1,\ldots,M-1\}$ is the sequence $(r_i)_{i=1}^{\infty}$ obtained by considering the sequence $(x_i)_{i=1}^{\infty}$ defined as

$$x_i = ax_{i-1} + c \mod M, \qquad i \geq 1,$$

and then setting $r_i := x_i/M$.

Observe the general framework for generating random numbers. We have a state space $S = \{0,1,\ldots,M-1\}$, which is the possible values our procedure output may take. Then we define a rule $T : S \to S$ which indicates how to produce the next number from the actual number. We input the x_0 and produce a sequence $(T^i(x_0))_{i=1}^{\infty}$. Finally, since the numbers take values on S and we want uniform numbers in $(0,1)$, we transform $(T^i(x_0))_{i=1}^{\infty}$ to meet our requirements. In this case, we simply divide by M.

One of example of choice of parameters is the NAG Fortran generator `G05CAF` uses $M = 259$ and $a = 1310$ and $c = 0$. More modern software uses much modern parameters. For example Borland C/C++ uses $M = 2^{32}$, $a = 22695477$ and increment $c = 1$. A bad choice of parameters can be really bad, as our later examples will show. It is not a good idea

to use your own parameters. If you want to use the LGC for your future algorithms, stick to well-tested initial paremeters. IBM's RANDU is a very famous example of a poor choice of initial parameters in the 70s.

Example 2.3 (Linear congruential generator) Below we will see how to implement a function using a linear congruential generator to create random numbers.

```
LCG <- function(nsim, M = 2^32, a = 22695477, c = 1, seed = 110104){
    X = c(seed, numeric(nsim-1)) # Preallocate space
    for(i in 1:(nsim-1)) X[i+1] <- ((a*X[i] + c)%% M) # Apply LCG rule
    return(X/M) # Apply transform
}
rseq <- LCG(1000)
head(rseq) # The first values

## [1] 2.563559e-05 8.118340e-01 6.844589e-01 2.424782e-01 9.725499e-01
## [6] 9.661459e-01

summary(rseq) # Quantiles and mean look good.

##      Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
## 0.0000256 0.2470000 0.5128000 0.5038000 0.7552000 0.9995000

sd(rseq) # Theoretical value is srt(12)~~0.2887

## [1] 0.2924752
```

The period of the method. The way the linear congruential method is designed, there is exactly M possible values which come out of the sequence $(T^i(x_0))_{i=1}^{\infty}$. So eventually, a number will repeat. We say that the method is *periodic*. A poor choice of parameters will give short periods for many seeds. A good choice of parameters is that which guarantees the maximum period possible for every seed. This is the content of the Hull-Dobell theorem [HD62] (a result which you really don't need to memorise).

Theorem 2.4 (Hull-Dobell) *Provided that the increment c is nonzero. Then the LCG has maximum period possible for every seed if and only if the three folowing conditions hold:*

1. *c and m are relative prime.,*
2. *$a-1$ divides all the prime factors of m,*
3. *4 divides $a-1$ if and only if 4 divides m.*

Example 2.5 (Short period for the LCG) Below we will see how to implement a function using a linear congruential generator to create random numbers.

```
LCG(25, M = 384, a = 5, c = 32, seed = 56)
```

```
##  [1] 0.1458333 0.8125000 0.1458333 0.8125000 0.1458333 0.8125000 0.1458333
##  [8] 0.8125000 0.1458333 0.8125000 0.1458333 0.8125000 0.1458333 0.8125000
## [15] 0.1458333 0.8125000 0.1458333 0.8125000 0.1458333 0.8125000 0.1458333
## [22] 0.8125000 0.1458333 0.8125000 0.1458333
```

Goodness-of-fit A goodness-of-fit test can be implemented using the χ^2 test. This test works like this. Suppose you have a theoretical distribution that can take the k values $x_1, x_2, \ldots, x_k$ each one with probability $p_1, p_2, \ldots, p_k$. Then e_i is the expected number of occurrences in the i-th category. Define the $\hat{\chi}^2$-statistic as

$$\hat{\chi}^2 = \sum_{i=1}^{k} \frac{(o_i - e_i^2)}{e_i}.$$

It is shown in a basic course in statistics that $\hat{\chi}$ approximately follows the distribution of a χ^2 random variable with $k-1$ degrees of freedom. To implement this in our case, we must first divide $[0,1]$ into k equally sized bins (k is arbitrary). Then e_i will be n/k where n is the number of simulations and o_i will be the number of observations in the k-th bin. We must then compare the obtained value in χ^2 for $k-1$ degrees of freedom. This is done in R using the function `pchisq`.

Example 2.6 We now show an implementation of the χ^2 goodness-of-fit test.

```
nsim <- 50
k = 5 # number of bins
rseq <- LCG(nsim, seed=110104)
ei <- rep(nsim/k, times=k)
oi <- table(trunc(k*rseq)/k) # Easy way to compute the oi
chi2 <- sum((oi-ei)^2/ei)
pchisq(chi2, df = k-1)
```

```
## [1] 0.9008146
```

The way in which the last number must be interpreted is as the probability that the generated sequence truly comes from a uniform distribution. In this case it is of about 90%, which is considered high. ■

Failure of independence Another important thing to check in random number generation to check that desired sequences truly seem like independently generated. There are number of things that can be done. The easier of which is to check the correlation with output values and its predecessors. Here we show with two example what can go wrong and how independent things would look like.

Example 2.7 (Failure of independence) We will do a simple plot comparing the output term r_i on the horizontal axis and r_{i+1} on the vertical axis. If the number were independent now pattern should be recognisable.

1) This is an example of something that looks reasonably independent.

```
library(ggplot2)
nsim <- 1000
rseq <- LCG(nsim)
qplot(rseq[-nsim], rseq[-1])
```

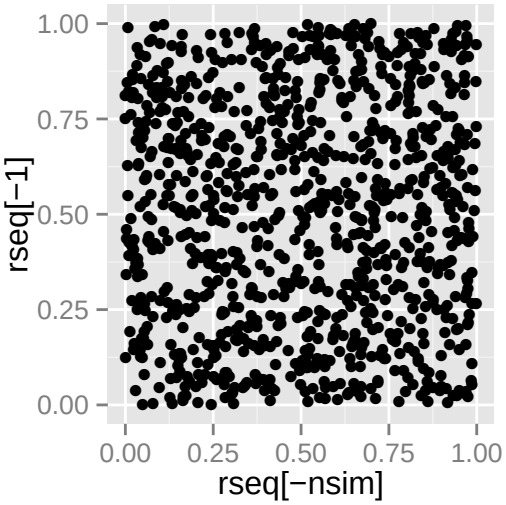

Figure 2.1: r_i independent of r_{i+1}

2) This is an example of something that definitely does not show independence.

```
library(ggplot2)
nsim <- 1000
rseq <- LCG(nsim, M = 574, a = 29, c = 466, seed = 11)
qplot(rseq[-nsim], rseq[-1])
```

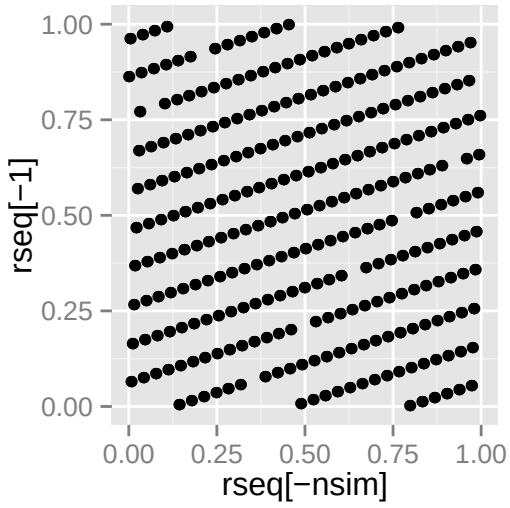

Figure 2.2: r_i extitnot independent of r_{i+1}

As a conclusion from this section it should be clear that is not a very good idea to come out with our own methods for creating random numbers. The method used by R is the *Mersenne Twister*. It is not really worth it to go into the details of this more complicated method for the purpose of this purpose. But one must know that it is a widely accepted reliable method.

2.2 The Inverse Function Method

Now that we now how to simulate a uniform a distribution in $[0,1]$ we will now see how to simulate an observation of another probabilistic distribution. What we will want to do is to design methods so that using an observation of a uniform random variable we obtain an observation of other distributions.

Example 2.8 (Simulating a Bernoulli) The simplest case of such a method is the simulation of a Bernoulli random variable with parameter p. Observe that if $U \sim \text{Unif}[0,1]$, then $\mathbb{P}(U \in [0,p]) = p$ and $\mathbb{P}(U \in (p,1]) = 1-p$. Hence the if we define a function $G(U)$ as

$$X = G(U) = \begin{cases} 1 & \text{if } U \in (0,p) \\ 0 & \text{if } U \in [p,1), \end{cases}$$

then X is has Bernoulli law with parameter p. ■

General case for discrete variables. Let us generalise the above example. Let $U \sim \text{Unif}[0,1]$ and suppose we would like X to take the values $x_1 < \ldots < x_n, \ldots$ with corresponding probabilities $p_1, \ldots, p_n, \ldots$. Define $F_n = \sum_{i=1}^{n} p_i$ and set $P_1 = [0, F_1]$ and $P_i = (F_{i-1}, F_i]$ for $i \geq 2$. Then $\mathbb{P}(U \in P_i) = p_i$. If we now set

$$X = G(U) = \begin{cases} x_i & \text{if } U \in P_i, \end{cases}$$

then $\mathbb{P}(X = x_i) = p_i$, so that X has the desired distribution. Let F_X be the cumulative distribution function of X, i.e., $F_X(x) = \mathbb{P}(X \leq x)$. Observe that G is like an 'inverse' of F_X in the sense that $F_X(G(F_i)) = F_i$.

Example 2.9 (Simulating a Geometric distribution) The geometric distribution with parameter can be characterised as the number of attempts before a success with occurs of probability p in a sequence on identical independent events. If $X \sim \text{Geom}(p)$, then $\mathbb{P}(X = n) = (1-p)^{n-1}p$. We will show in R how use the method just described above to simulate observations of a geometric distribution out of observations of uniform random variables obtained using the function `runif`.

```
set.seed(110104)
# Function to simulate ------
rGeom.aux <- function(p){ # Simulate one draw
  U <- runif(1)
  # We now use a while loop to find to which partition U belongs to.
  k <- 1
  max.iter <- 10e5 # Security parameter to avoid infinite loops.
  # Good practice even when not necessary.
  Fk <- p
  while(U > Fk  &  k <= max.iter){
    k <- k + 1
    Fk <- Fk + (1-p)^(k-1)*p
  }
  return(k-1)
}
rGeom <- function(nsim, p){ # Simulate several draws
  replicate(nsim, rGeom.aux(p=p))
}
# Comparison with theoretical values ------
nsim <- 1000
p <- 0.45
geom.seq <- rGeom(nsim, p) # nsim simulations of Geom(p)
```

```r
max.obs <- max(geom.seq) # Max value to print in tables
observed <- table(factor(geom.seq, levels=0:max.obs)) # Observed frequencies
probs <- (1-p)^(0:max.obs)*p
expected <- round(probs*nsim)
data.frame(data.frame(observed), Expected=expected) # Observed vs expected
```

```
##    Var1 Freq Expected
## 1     0  426      450
## 2     1  255      248
## 3     2  148      136
## 4     3   69       75
## 5     4   51       41
## 6     5   24       23
## 7     6   12       12
## 8     7   10        7
## 9     8    1        4
## 10    9    2        2
## 11   10    1        1
## 12   11    1        1
```

General case for continuous variables. What made the previous easy cases work was that we found a function G such that $G(F_X(x)) = x$. Recall that $F(x) = \mathbb{P}(X \leq x)$ is always a non-decreasing function. In order to be invertible, we only need that F is strictly increasing. When F_X in invertible, we have the following result which can be used to generalise the previous cases.

Proposition 2.10 (The inverse function method) *Let $U \sim \text{Unif}[0,1]$ and X be a continuous random variable with cumulative distribution function F_X. Suppose that F_X is invertible. Then $F_X^{-1}(U)$ has the same law as X.*

Proof. In order to prove that X and $F_X^{-1}(U)$ have the same probability law it is sufficient to show that $\mathbb{P}(X \leq x) = \mathbb{P}(F_X^{-1}(U) \leq x)$, as this entire determines the probability law. We thus compute

$$\mathbb{P}(F_X^{-1}(U) \leq x) = \mathbb{P}(U \leq F_X(x)) = F_X(x) = \mathbb{P}(X \leq x),$$

where we used that F_X is strictly increasing and hence that $F_X(x_1) \leq F_X(x_2)$ if and only if $x_1 \leq x_2$. This proves the result. □

The proof above works whenever we can find G solving the problem $F_X(G(u)) = u$. If we define G as $G(u) := \inf\{x \mid F(x) \geq u\}$, then we recover the previous discrete cases.

Example 2.11 (Simulating an exponential distribution) Suppose we want to simulate a random variable having exponential distribution with parameter $\lambda > 0$. We say that $X \sim \text{Exp}\lambda$ if its cumulative distribution function is $F_X(x) = 1 - e^{-\lambda x}$. This function is easy to invert. Its inverse is $F_X^{-1}(u) = -\lambda^{-1}\ln(1-U)$.

```
set.seed(110104)
library(ggplot2)
nsim <- 1000
lambda <- 2
rExp <- function(nsim, lambda){
  return((-1/lambda)*log(1-runif(nsim)))
}
dat <- data.frame(Value=rExp(nsim, lambda))
ggplot(dat, aes(x=Value)) +
  geom_histogram(aes(y=..density..), binwidth= .2, colour="black", fill="white") +
  stat_function(fun = function(x) lambda*exp(-lambda*x),colour = "blue")
```

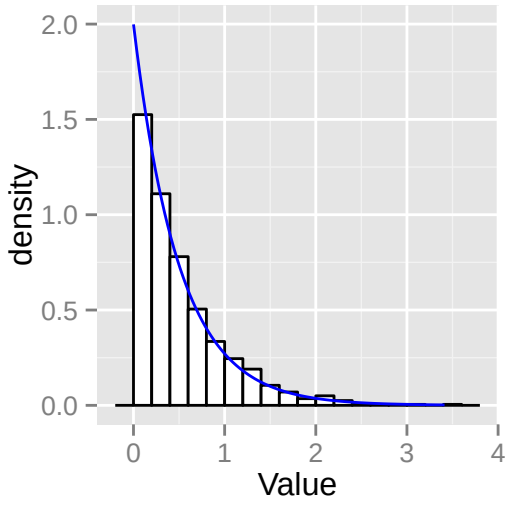

Figure 2.3: Density histogram of observations of simulated exponential random variables vs theoretical density Exp(λ) (in blue).

Observe that U and $1-U$ have identical distributions if $U \sim \text{Unif}(0,1)$. The above method would have works just as well if we had taken $G(u) = -\lambda^{-1}\ln(u)$ instead. ■

In practice, we rarely have a closed mathematical expression for the inverse of F_X (for instance, if we wanted to simulate the normal distribution). One solution is to try to use a

numerical software to estimate the inverse. Other approaches, sometimes more efficient, are given in the next sections.

2.3 The Acceptance-Rejection Method

The *acceptance-rejection rejection method* is an alternative to sample/simulate from probability distribution when we do not have an easy way to compute F_X^{-1}. It is a little bit complicated to graps the intuition of the method at first sight. For that reason, let us first give a simpler case.

Uniform rejection sampling. A uniform density function on a (measurable) set E is a function f such that $f(x) = c \geq 0$ is constant for all $x \in E$ and $c \int_E \mathrm{d}x = 1$. Evidently, the function $f_X(x) = \mathbb{1}_{[0,1]}$,the indicator function of the interval $[0,1]$, defines a uniform density on $[0,1]$.

Proposition 2.12 *Let $D \subset [0,1]$ be a subset with $\int_D \mathrm{d}x > 0$. Define a random variable X as follows.*

1. *Generate $U \sim \mathrm{Unif}[0,1]$.*
2. *If $U \in D$, set $X = U$, otherwise, go to the previous step and generate a new independent value of U.*

Then X is distributed uniformly on D.

Proof. Here is a short simple argument. We first observe that the above recipe is well-defined. For this, we must only check that the method eventually concludes. Let N be the minimal iteration such that $U \in D$. Observe that $\mathbb{P}(N > n) = (1 - \mathbb{P}(U \in E))^n$, which converges to zero as $n \to \infty$ since $\mathbb{P}(U \in E) > 0$.

Let now $E \subset D$ be any (measurable) set. Then by construction we have that $\mathbb{P}(X \in E) = \mathbb{P}(U \in E \mid U \in D)$ since each draw of U is independent from the past and we set $X = U$ until we know that $U \in D$. But this implies that X has density function $(1/\mathbb{P}(U \in D))\mathbb{1}_D$, since

$$\mathbb{P}(X \in E) = \mathbb{P}(U \in E \mid U \in D) = \frac{\mathbb{P}(\{U \in E\} \cap \{U \in D\})}{\mathbb{P}(U \in D)} = \int_E \frac{1}{\mathbb{P}(U \in D)} \mathrm{d}x,$$

and this concludes the proof. □

Acceptance-Rejection method. Suppose we want to simulate from a probability law determined by a density function f but we only know how draw from a density g with the property that $f(y) \leq Mg(y)$ for all y with $M > 0$ some real number (such g is called a *majorising* function for f). Then there is a marvellous trick to draw from f.

Suppose for the moment that $M = 1$, here is the intuition of the method. Suppose that we are 'situated' at a train station y so that $g(y)$ measures the density of arrivals of passengers

at the station y. Suppose further that each arriving passenger is accepted with a probability $p(y)$ depending on y. The density of accepted passengers is now $p(y)g(y)$. If we want to simulate acceptances according to the density f, then we better had $p(y) = f(y)/g(y)$, so that the density of accepted passengers at y becomes $f(y)$. How can we achieve this? We can simply generate a uniform random variable U in $[0,1]$ and accept an arrival if $U \leq f(y)/g(y)$ (here we use that $f(y) \leq g(y)$) and reject it otherwise.

Proposition 2.13 *Let f and g be density function such that there exists $M \geq 1$ such that $f \leq Mg$. Define X according to the following algorithm*

1. *Generate $U \sim \mathrm{Unif}[0,1]$ and Y independently according to the density g,*
2. *If $U \leq f(Y)/(Mg(Y))$ then accept and set $X = Y$. Otherwise, go back to the previous step and start again.*

Then the resulting X is distributed according to the density function f.

Proof. The proof is just a computation that uses the law of total probability.

$$\begin{aligned}\mathbb{P}(X \leq x) &= \mathbb{P}\Big(Y \leq x \mid U \leq \tfrac{f(Y)}{Mg(Y)}\Big) \\ &= \frac{\mathbb{P}\Big(Y \leq x, U \leq \tfrac{f(Y)}{Mg(Y)}\Big)}{\mathbb{P}\Big(U \leq \tfrac{f(Y)}{Mg(Y)}\Big)}\end{aligned}$$

where the first equality holds by construction, the second by the definition of conditional probability. We now use the law of total probability two compute the term in the numerator.

$$\begin{aligned}\mathbb{P}\Big(Y \leq x, U \leq \tfrac{f(Y)}{Mg(Y)}\Big) &= \int_{\infty}^{\infty} \mathbb{P}\Big(Y \leq x, U \leq \tfrac{f(Y)}{Mg(Y)} \mid Y = y\Big) g(y)\,\mathrm{d}y \\ &= \int_{\infty}^{x} \mathbb{P}\Big(U \leq \tfrac{f(Y)}{Mg(Y)} \mid Y = y\Big) g(y)\,\mathrm{d}y \\ &= \int_{\infty}^{x} \mathbb{P}\Big(U \leq \tfrac{f(y)}{Mg(y)}\Big) g(y)\,\mathrm{d}y \\ &= \frac{1}{M}\int_{-\infty}^{x} \frac{f(y)}{g(y)} g(y)\,\mathrm{d}y \\ &= \frac{1}{M}\int_{-\infty}^{x} f(y)\,\mathrm{d}y.\end{aligned}$$

An almost identical computation for the denominator yields

$$\begin{aligned}\mathbb{P}\left(U \leq \frac{f(Y)}{Mg(Y)}\right) &= \int_{\infty}^{\infty} \mathbb{P}\left(U \leq \frac{f(Y)}{Mg(Y)} \mid Y = y\right) g(y)\,\mathrm{d}y \\ &= \int_{-\infty}^{\infty} \frac{f(y)}{Mg(y)} g(y)\,\mathrm{d}y \\ &= \frac{1}{M}\int_{-\infty}^{\infty} f(y)\,\mathrm{d}y \\ &= \frac{1}{M}.\end{aligned}$$

Combining the two simplifications above, we conclude that $\mathbb{P}(X \leq x) = \int_{-\infty}^{x} f(y)\,\mathrm{d}y$, which is to say that f is a density function associated to the law of X. This concludes the proof. □

Example 2.14 (Generating a Beta distribution) The Beta distribution is a probability law that has applications for Bayesian inference as well models in which a probabilistic phenomena is supported in an interval of finite length. We say that X has is distributed Beta with parameters α, β if it has density function

$$f_X(x) = \frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\Gamma(\beta)} x^{\alpha-1}(1-x)^{\beta-1}\mathbb{1}_{[0,1]}$$

where $\Gamma(x) = \int_0^{\infty} x^{t-1}e^{-x}\,\mathrm{d}x$ is the gamma function. One can show that the maximum valued of f is attained at $(\alpha-1)/(\alpha+\beta-2)$. Hence we can use set $g(x) = \mathbb{1}_{[0,1]}$ and $M = f((\alpha-1)/(\alpha+\beta-2))$.

```
alpha = 10 # Some random values for this example.
beta = 5
f <- function(x, alpha, beta){
  gamma(alpha + beta)/(gamma(alpha)*gamma(beta))*x^(alpha-1)*(1-x)^(beta-1)
}
g <- function(x) {1}
M <- f((alpha-1)/(alpha+beta-2), alpha, beta)
ggplot(data.frame(x=c(0,1)), aes(x)) +
  stat_function(fun = f, arg=list(alpha=alpha, beta=beta), aes(colour = "f")) +
  stat_function(fun = function(x) g(x)*M, aes(colour = "M*g")) +
  scale_colour_manual("Function", values = c("red", "blue")) + ylab("density")
```

```
set.seed(999)
nsim <- 1000
rBeta.aux <- function(alpha, beta, verbose=FALSE){
```

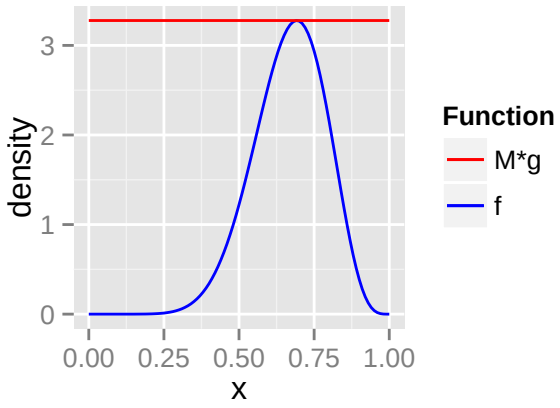

Figure 2.4: Acceptance-rejection for a Beta distribution: comparison between f and a majorising function with parameters $(\alpha, \beta) = (10, 5)$.

```
  d <- data.frame() # To keep track of acceptance and rejection.
  accepted <- FALSE
  max.iter <- 10e6 # When we use whiles, we set a max.iter paramater for security.
  iter <- 1
  while(!accepted & iter<=max.iter){
    U <- runif(1)
    Y <- runif(1) # In our case, g is uniform [0,1]. So this is sampling from g.
    if(U <= f(Y, alpha, beta)/(g(Y)*M)){
      accepted <- TRUE
      X <- Y
    }
    d <- rbind(d, data.frame(Y=round(Y,4), U=round(U,4),
        f.over.Mg=round(f(Y, alpha, beta)/(g(Y)*M),4), Accepted=accepted))
    iter <- iter + 1
  }
  if(verbose==TRUE){
    return(d) # If we indicate verbose we return a table with outcomes.
  } else{
    return(X)
  }
}
rBeta <- function(nsim, alpha, beta){
  replicate(nsim, rBeta.aux(alpha=alpha, beta=beta))
```

```
}
hist(rBeta(nsim, alpha, beta), breaks=20, main="", prob=TRUE)
```

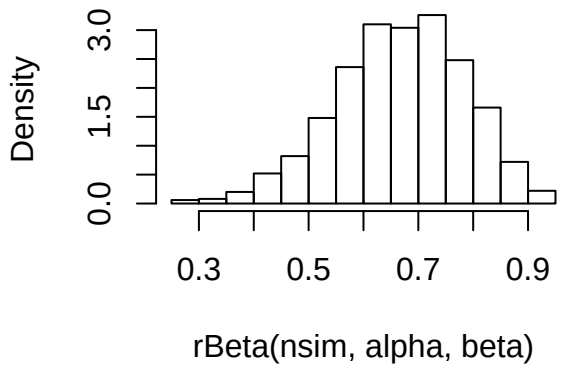

Figure 2.5: Acceptance-rejection for a Beta distribution: histogram of simulated observations.

```
rBeta.aux(alpha, beta, verbose=TRUE)

##         Y      U f.over.Mg Accepted
## 1 0.0119 0.5266    0.0000    FALSE
## 2 0.0990 0.6736    0.0000    FALSE
## 3 0.5218 0.5151    0.4578    FALSE
## 4 0.9931 0.0698    0.0000    FALSE
## 5 0.2994 0.6724    0.0142    FALSE
## 6 0.8710 0.5923    0.2441    FALSE
## 7 0.1621 0.4379    0.0001    FALSE
## 8 0.5788 0.1415    0.7007     TRUE
```

This of course, was the simplest case when g can be taken constant. However, other options could had been used. Observe we cannot a constant g if our objective density f is supported in a set of infinite length. ■

Exercise 2.1 Use the acceptance rejection method to simulate observations of $Z = |X|$ where X is a standard normal random variable. *Hint:* Deduce that the density function of Z is twice the density function of X due to symmetry. Set $g(x) = e^{-x}$ and find an appropriate constant M.

Observe the for the acceptance-rejection method to work efficiently we better had M as small as possible so that the majorising conditions still holds. Otherwise, the acceptance probability would be too low.

Exercise 2.2 (Von Neumann bias correction) Suppose we are throwing a biased coin with probability of observing heads $p > 1/2$. Try to device a method based on the ideas similar to the ones in this section to obtain an unbiased sample from the biased sample. *Note:* sometimes true random numbers have a natural biased and a method like this, called von Neumann bias correction, is used to obtain unbiased samples.

2.4 The Box-Müller method

The Box-Muller method is a great trick for easily generating sample of a random normal distribution. It is *only* useful for the normal distribution and no other distribution. It is widely used for its simplicity of application. The whole idea is simply an observation coming from multivariate calculus.

The change of variable formula. One of the first results studied in multivariate calculus is the change of variable formula. It is a generalisation of integration by substitution. Suppose we have a invertible transform $T = (T_1, \dots T_n) \colon \mathbb{R}^n \to \mathbb{R}^n$. Suppose that $x_1, \dots, x_n$ are coordinates of some integrable function f. Then the change of variable formula states how we can integrate f in terms of the new coordinates $T_1(x_1, \dots, x_n), \dots, T_n((x_1, \dots, x_n)$. This is given as follows:

$$\begin{aligned}&\int_{T(E)} f(x_1, \dots, x_n)\, \mathrm{d}x_1 \dots \mathrm{d}x_n \\ &\qquad = \int_E f(x_1(T_1, \dots, \dots, T_n), \dots, x_n(T_1, \dots, \dots, T_n))\, |\mathrm{D}T|\, \mathrm{d}T_1 \dots \mathrm{d}T_n.\end{aligned}$$

Here, $|\mathrm{D}T|$ is the determinant of the jacobian of the transform T.

Example 2.15 (Integration using polar coordinates) One of the most usual applications of the change of variable formula is the use of polar coordinates. in $\mathbb{R}^2$. Define a transformation $T \colon \mathbb{R}^2 \to \mathbb{R}^2$ from polar coordinates $\{(r, \theta) \mid r \in (0, \infty), \theta \in [0, 2\pi)\}$ to rectangular coordinates $\{(x, y) \mid x \in \mathbb{R}, r \in \mathbb{R}\}$ as

$$\begin{cases} T(r, \theta) = (T_1(r, \theta), T_2(r, \theta)) = (r \cos(\theta), r \sin(\theta)). \end{cases}$$

The Jacobian of T is the linear transform represented by the matrix

$$\begin{bmatrix} \frac{\partial T_1}{\partial r} & \frac{\partial T_1}{\partial \theta} \\ \frac{\partial T_2}{\partial r} & \frac{\partial T_2}{\partial \theta} \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -r \sin(\theta) \\ \sin(\theta) & r \cos(\theta) \end{bmatrix}.$$

The determinant of this matrix is easily seen to be r. Thus the change of variable formula becomes

$$\int_{T(E)} f(x,y)\,\mathrm{d}x\,\mathrm{d}y = \int_E f(r\cos(\theta), r\sin(\theta))r\,\mathrm{d}r\,\mathrm{d}\theta.$$

In what follows we will see how to use polar coordinates to find an easy way to generate random variables ■

The Box-Müller trick consists of realising that if we take the joint density function of two independent standard normal random variables X and Y, then the transformation to polar coordinates yields a nice expression easy to work with. The joint density function $f_{X,Y}$ of two independent standard normal random variables X and Y is

$$f_{X,Y}(x,y) = f_X(x)f_Y(y) = \frac{1}{\sqrt{2\pi}}e^{-\frac{1}{2}x^2}\frac{1}{\sqrt{2\pi}}e^{-\frac{1}{2}y^2} = \frac{1}{2\pi}e^{-\frac{1}{2}(x^2+y^2)}.$$

Then by the change of variable formula, the joint density of (R,Θ), where R and Θ are defined by the transformation $X = R\cos(\Theta)$ and $Y = R\sin(\Theta)$ is given by

$$f_{R,\Theta}(r,\theta) = \frac{1}{2\pi}e^{-\frac{r}{2}}r\mathbb{1}_{[0,2\pi)}(\theta)\mathbb{1}_{[0,\infty)}(r) = \left(\frac{1}{2\pi}\mathbb{1}_{[0,2\pi)}(\theta)\right)\left(\frac{1}{2\pi}e^{-\frac{r}{2}}\mathbb{1}_{[0,\infty)}(r)\right).$$

From the the above expression we can see that R and Θ are independent random variables (since the joint density is the multiplication of a density depending only on R and a density depending only on Θ, and this is the definition of independence). The marginal density of Θ is

$$f_\Theta(\theta) = \int_0^\infty f_{R,\Theta}(r,\theta)\,\mathrm{d}r = \frac{1}{2\pi}\mathbb{1}_{[0,2\pi)}(\theta),$$

hence Θ is uniformly distributed in $[0,2\pi)$. This implies that we can easily simulate an observation of Θ by simulating $U_1 \sim \mathrm{Unif}[0,1)$ and multiplying U_1 by 2π. Analogously, the marginal density of R is

$$f_R(r) = \int_0^{2\pi} f_{R,\Theta}(r,\theta)\,\mathrm{d}\theta = re^{-\frac{r^2}{2}}\mathbb{1}_{[0,\infty)}(r).$$

Simulating an observation of R with this density can be used using the inverse-function method with a little extra trick. By the integration by substitution formula (which is the one-dimensional case of the change of variable we just discussed), the density function of $S := R^2$ is $f_S(s) = e^{-s/2}$. So R^2 is distributed as an exponential variable with parameter $1/2$. He have already seen how to simulate an observation of an exponential using a uniform random variable. To be precise, we can simulate R^2 by simulating $U_2 \sim \mathrm{Unif}[0,1)$ and setting $R^2 = -2\log(1-U_2)$, or even simpler, $R^2 = -2\ln(U_2)$, as U_2 and $1-U_2$ have exactly the same distribution. As a conclusion of this discussion, we can simulate a sample of $(R,\Theta$ by

simulating $U_1, U_2 \sim \text{Unif}[0,1)$ and setting

$$\begin{cases} \Theta = 2\pi U_1 \\ R = \sqrt{-2\ln(U_2)}. \end{cases}$$

Finally, since $X = R\cos(\Theta)$ and $Y = R\sin(\Theta)$, we can recover a simulation of (X, Y) by setting

$$\begin{cases} X = \sqrt{-2\ln(U_2)}\cos(2\pi U_1) \\ Y = \sqrt{-2\ln(U_2)}\sin(2\pi U_1). \end{cases}$$

Observe that this method will always produce two instead of only one observation of a standard normal random variable, furthermore, this two observations are assumed to be independent.

Exercise 2.3 Make a programme in R to simulate a sample of a normal distributions with mean μ and variance σ^2 using the Box-Müller method.

Chapter 3

Monte-Carlo integration

Index

Bibliography

[Eck87] R. Eckhardt. “Stan Ulam, John Von Neumann and the Monte Carlo Method”. In: *Los Alamos Science Special Issue* (1987), pp. 131–136.

[HD62] T. E. Hull and A. R. Dobell. “Random Number Generators”. In: *SIAM Rev* 4.3 (1962), pp. 230–254.

[Leh43] D. Lehmer. “Mathematical methods in large-scale computing units”. In: *Proceedings of a Second Symposium on Large-Scale Digital Calculating Machinery* (1943), pp. 141–146.