Statistical Computation and Simulation

João Neto

May 2014

===

Notes from course given by professor Maria Isabel Fraga Alves.

Refs:

- Slides of the course.
- Rizzo - Statistical Computing with R (http://www.crcpress.com/product/isbn/9781584885450) (2007)
- Secondini - Introduction to Markov chain and Monte Carlo simulation (http://www.ircphonet.it/staff/members/secondini/)
- Robert & Casella - Introducing Monte Carlo Methods with R (http://www.springer.com/statistics/computational+statistics/book/978-1-4419-1575-7) (2010)

Inverse Transformation

This is a method to generate random samples of distributions.

It is based on the following result called *Probability Integral Transformation*:

$$X \sim F_X \Rightarrow U = F_X(X) \sim \mathcal{U}(0,1)$$

We define the inverse transformation of F_X as

$$F_X^{-1}(u) = \inf\{x : F_X(x) = u\}, 0 < u < 1$$

With $U \sim \mathcal{U}(0,1)$,

$$P(F_X^{-1}(U) \leq x) = P(\inf\{t : F_X(t) = U\} \leq x) = P(U \leq F_X(x)) = F_U(F_X(x)) = F_X(x)$$

meaning that F_X^{-1} has the same distribution as X!

So, the method needs the inverse function $F_X^{-1}(u)$ to work.

For each value required the method follows:

1. Generate a random u from $\mathcal{U}(0,1)$
2. Return $F_X^{-1}(u)$

Here's the algorithm in R:

```r
# return n samples based on the inverse of the target cdf
inv.transform <- function(inv.f, n) {

# Non-vectorized version (for explanation purposes only)
#
#   result.sample <- rep(NA,n)
#
#   for (i in 1:n) {
#     u <- runif(1,0,1)              # step 1
#     result.sample[i] <- inv.f(u)   # step 2
#   }
#
#   result.sample

  # Vectorized version
  inv.f(runif(n,0,1))
}
```

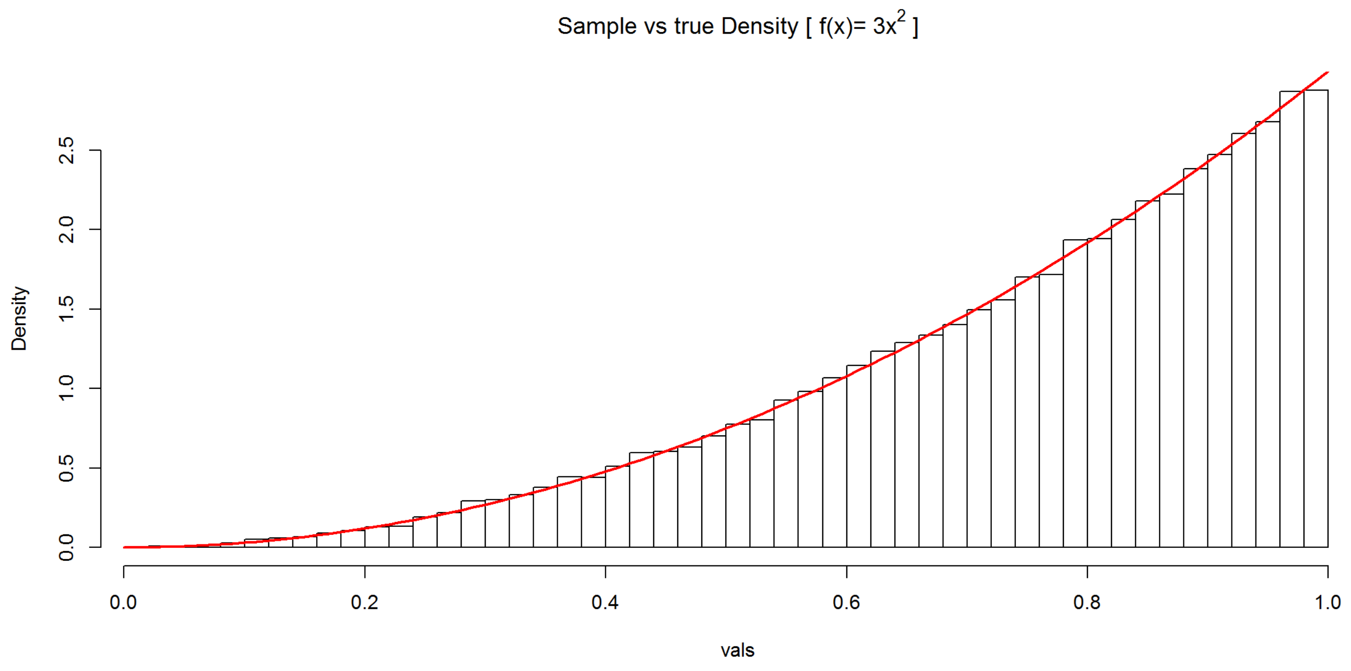
Eg: used this method to simulate a random variable which pdf is $f_X(x) = 3x^2, 0 < x < 1$

The cdf is given by

$$F_X(x) = \int_0^x f_x(t)dt = x^3$$

so, the inverse is $F_X^{-1}(u) = u^{1/3}$

```r
inv.f <- function(u) u^(1/3)

vals <- inv.transform(inv.f, 5e4)

# Checking if it went well
hist(vals, breaks=50, freq=FALSE, main=expression("Sample vs true Density [ f(x)=" ~3*x^2~"]"))
curve(3*x^2, 0, 1, col="red", lwd=2, add=T)
```



Let's change the previous problem a bit: let's assume the same shape but for $-1 < x < 2$.

We would get:

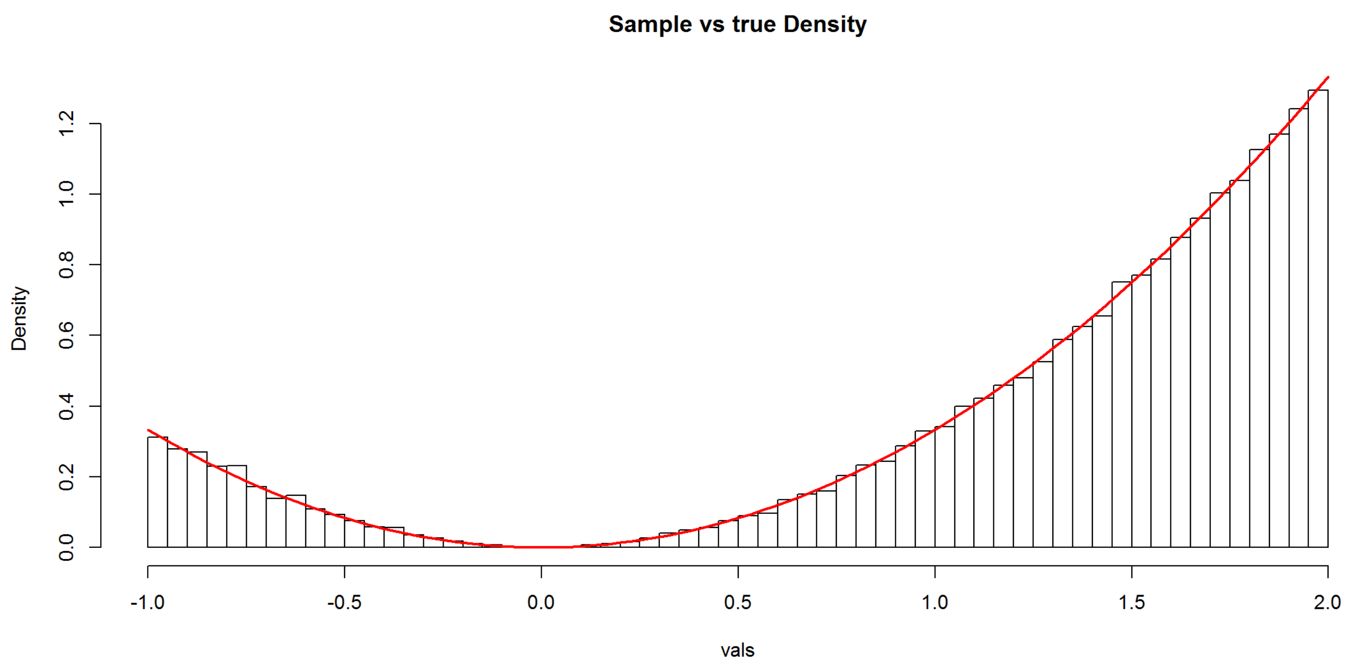
$$f_X(x) = \frac{1}{3}x^2$$

$$F_X(x) = \int_{-1}^{x} f_X(t)dt = \frac{x^3+1}{9}$$

$$F_X^{-1}(u) = (9u-1)^{1/3}$$

So we need a new inverse function in R:

```
# The next R code is like this due to the fact that cubic squares of negative numbers also have c
omplex roots
# But basically this is
#   inv.f <- function(u) (9*u-1)^(1/3)
inv.f <- function(u) ifelse((9*u-1)>=0,  (9*u-1)^(1/3),  -(1-9*u)^(1/3))

vals <- inv.transform(inv.f, 5e4)

# Checking if it went well
hist(vals, breaks=50, freq=FALSE, main="Sample vs true Density")
curve((1/3)*x^2, -1, 2, col="red", lwd=2, add=T)
```

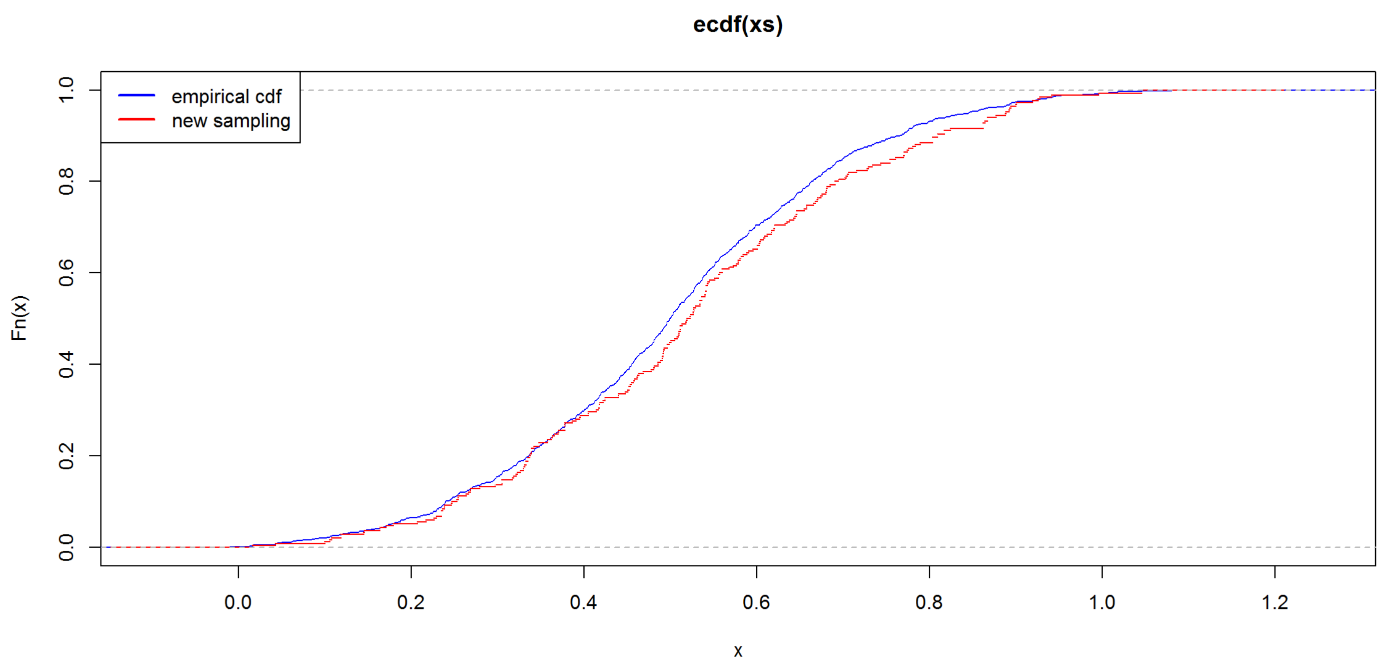


The next R code shows a way to generate an empirical cdf from some data (using the `distr` package (http://cran.r-project.org/web/packages/distr/distr.pdf)), and how to use it to generate new random samples.

```r
n <- 1e3
xs <- rnorm(n,.5,.2)    # a random sample

library(distr)
```

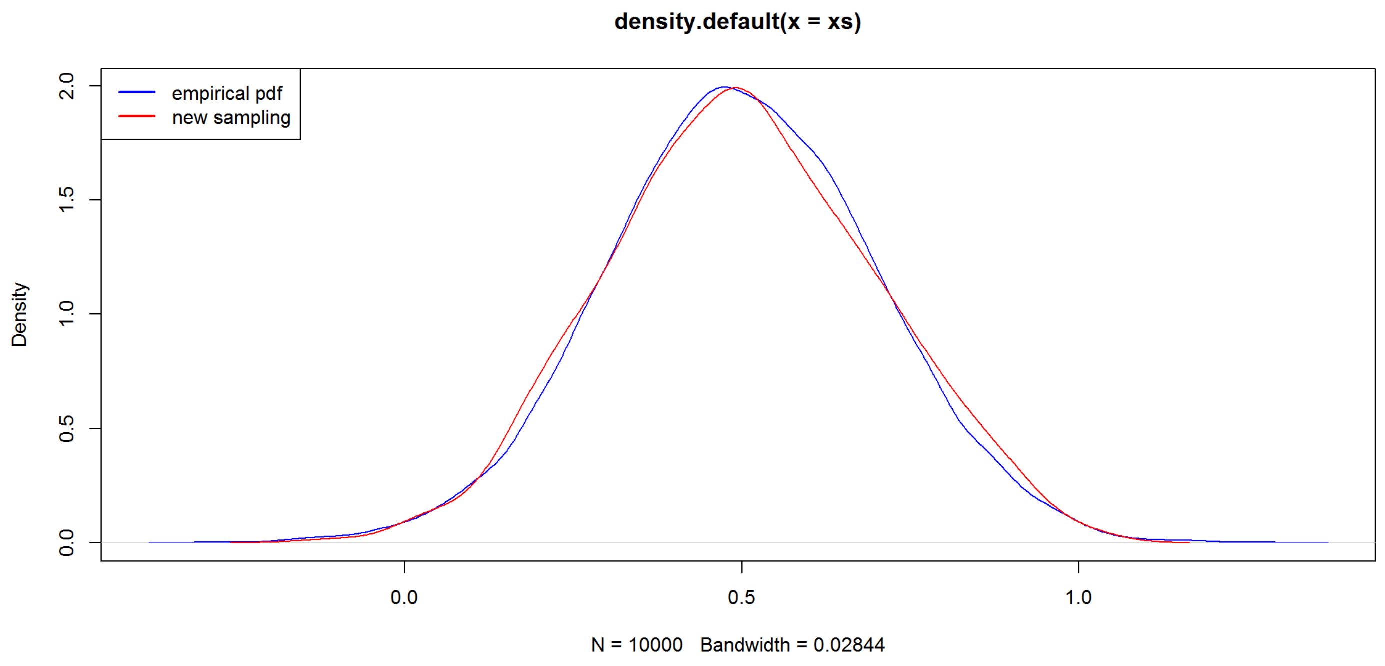
```
## Loading required package: startupmsg
## Utilities for start-up messages (version 0.9)
## For more information see ?"startupmsg", NEWS("startupmsg")
##
## Loading required package: sfsmisc
## Loading required package: SweaveListingUtils
## Utilities for Sweave together with TeX listings package (version 0.6.2)
## NOTE: Support for this package will stop soon.
## Package 'knitr' is providing the same functionality in a better way.
## Some functions from package 'base' are intentionally masked ---see SweaveListingMASK().
## Note that global options are controlled by SweaveListingoptions() ---c.f. ?"SweaveListingoptio
ns".
## For more information see ?"SweaveListingUtils", NEWS("SweaveListingUtils")
## There is a vignette to this package; try vignette("ExampleSweaveListingUtils").
##
##
## Attaching package: 'SweaveListingUtils'
##
## The following objects are masked from 'package:base':
##
##     library, require
##
## Object oriented implementation of distributions (version 2.5.3)
## Attention: Arithmetics on distribution objects are understood as operations on corresponding r
andom variables (r.v.s); see distrARITH().
## Some functions from package 'stats' are intentionally masked ---see distrMASK().
## Note that global options are controlled by distroptions() ---c.f. ?"distroptions".
## For more information see ?"distr", NEWS("distr"), as well as
##   http://distr.r-forge.r-project.org/
## Package "distrDoc" provides a vignette to this package as well as to several extension package
s; try vignette("distr").
##
##
## Attaching package: 'distr'
##
## The following objects are masked from 'package:stats':
##
##     df, qqplot, sd
```

```
emp.cdf <- DiscreteDistribution(xs)  # fit an empirical cdf
new.xs <- emp.cdf@r(250)             # generate new samples

# plot original sample vs new sample
plot(ecdf(xs), col="blue")
plot(ecdf(new.xs), col="red", pch=".", add=T)
legend("topleft",c("empirical cdf","new sampling"), col=c("blue","red"), lwd=2)
```



Another alternative is using the density function which uses (by default gaussian) kernels to estimate the density of the sample:

```
n<-1e4
xs <- rnorm(n,.5,.2)

d      <- density(xs)
new_xs <- sample(d$x, replace=TRUE, prob=d$y)

# plot original sample vs new sample
plot(density(xs), col="blue")
lines(density(new_xs), col="red")
legend("topleft",c("empirical pdf","new sampling"), col=c("blue","red"), lwd=2)
```



Acceptance-Rejection Method

Problem: Generate $X \sim f$ from an arbitray pdf f (especially when it's hard to sample from f).

We must find $Y \sim g$ under the only restriction that $\forall_{f(x)>0} : f(x) < c.\, g(x), c > 1$.

Instead of sampling from $f(x)$ which might be difficult, we use $c.\, g(x)$ to sample instead.

For each value required the method follows:

1. Generate a random y from g
2. Generate a random u from $U(0,1)$
3. **If** $u < f(y)/(c.\, g(y))$ **then** return y **else** reject y and goto 1.

The algorihtm in R:

```r
# generate n samples from f using rejection sampling with g (rg samples from g)
accept.reject <- function(f, c, g, rg, n) {
  n.accepts     <- 0
  result.sample <- rep(NA, n)

  while (n.accepts < n) {
    y <- rg(1)              # step 1
    u <- runif(1,0,1)       # step 2
    if (u < f(y)/(c*g(y))) { # step 3 (accept)
      n.accepts <- n.accepts+1
      result.sample[n.accepts] = y
    }
  }

  result.sample
}
```

From step 3,

$$P(accept|Y) = P(U < \frac{f(Y)}{cg(Y)}|Y) = \frac{f(Y)}{cg(Y)}$$

The total probability of acceptance is

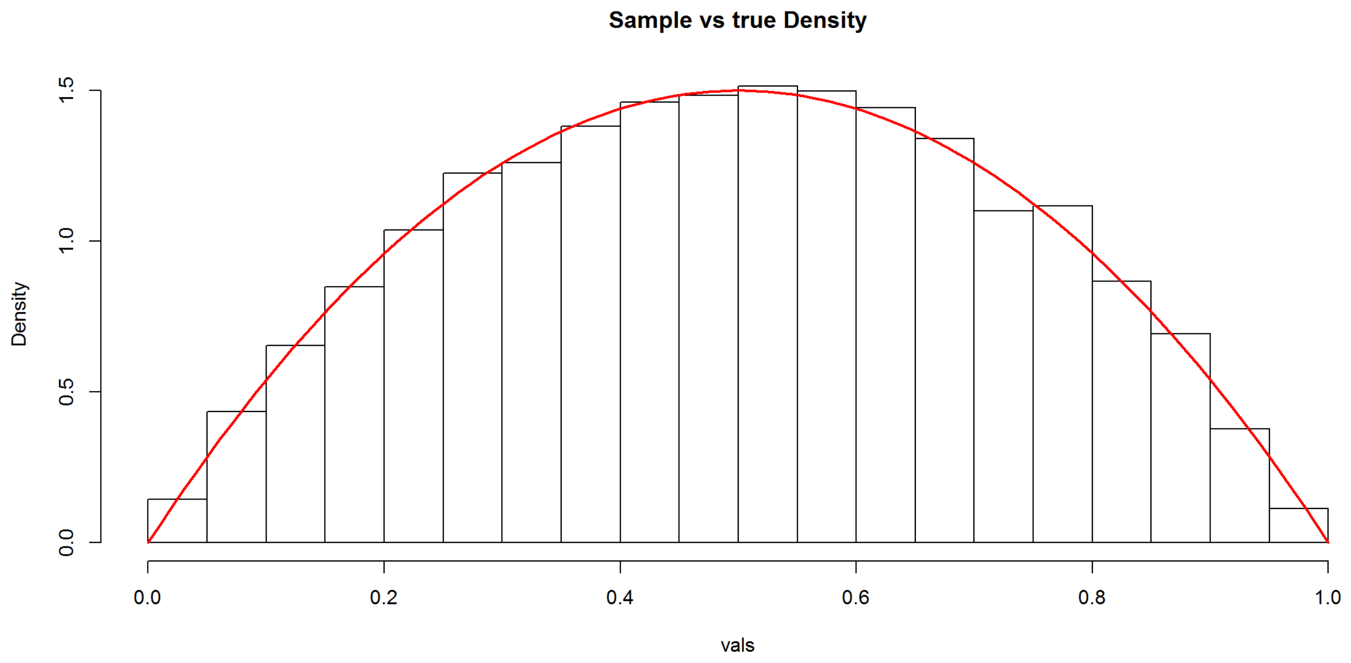
$$P(accept) = \sum_y P(accept|Y=y)P(Y=y) = \sum_y \frac{f(y)}{cg(y)} g(y) = \frac{1}{c}$$

The number of rejections until acceptance has the geometric distribution with mean c. On average each sample of X requires c iterations.

So, for this method to be efficient, Y should be easy to sample, and c should be as small as possible.

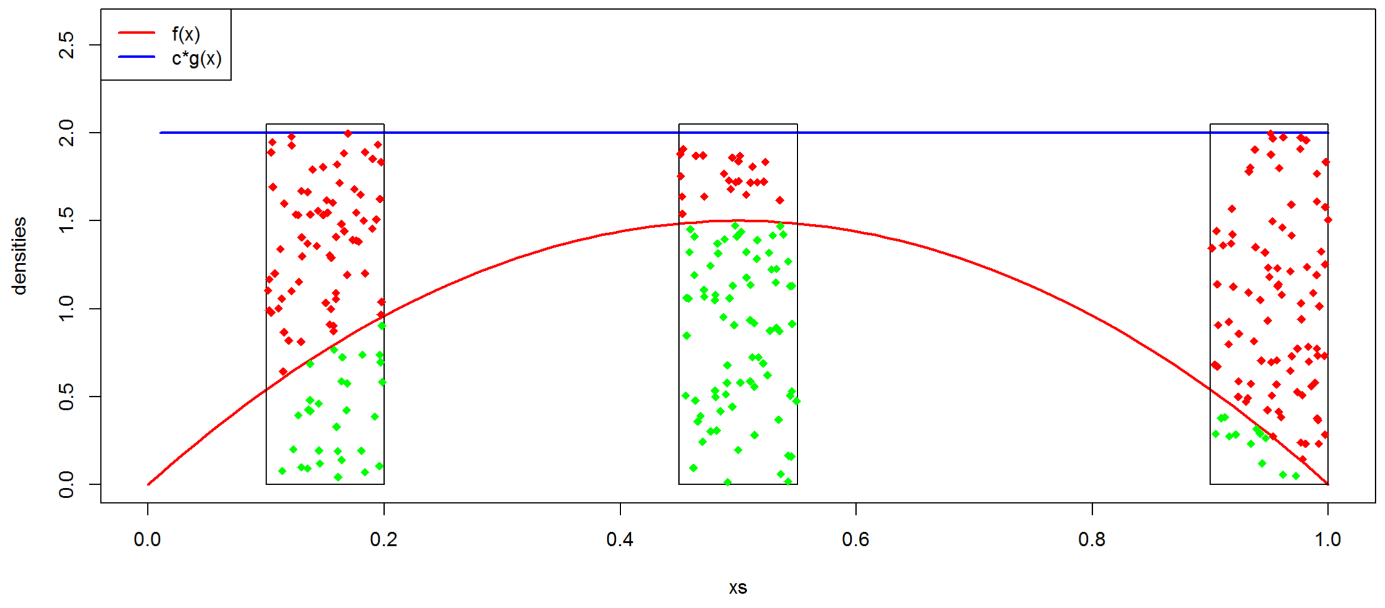
Eg, generate samples from distribution Beta(2,2), where we use the uniform has g, since $f(x) < 2 \times g(x)$:

```r
f  <- function(x) 6*x*(1-x)     # pdf of Beta(2,2), maximum density is 1.5
g  <- function(x) x/x           # g(x) = 1 but in vectorized version
rg <- function(n) runif(n,0,1)  # uniform, in this case
c  <- 2                         # c=2 since f(x) <= 2 g(x)

vals <- accept.reject(f, c, g, rg, 10000)

# Checking if it went well
hist(vals, breaks=30, freq=FALSE, main="Sample vs true Density")
xs <- seq(0, 1, len=100)
lines(xs, dbeta(xs,2,2), col="red", lwd=2)
```



Let's visualize the method accepting (green dots) or rejecting (red dots) at some specified segments :

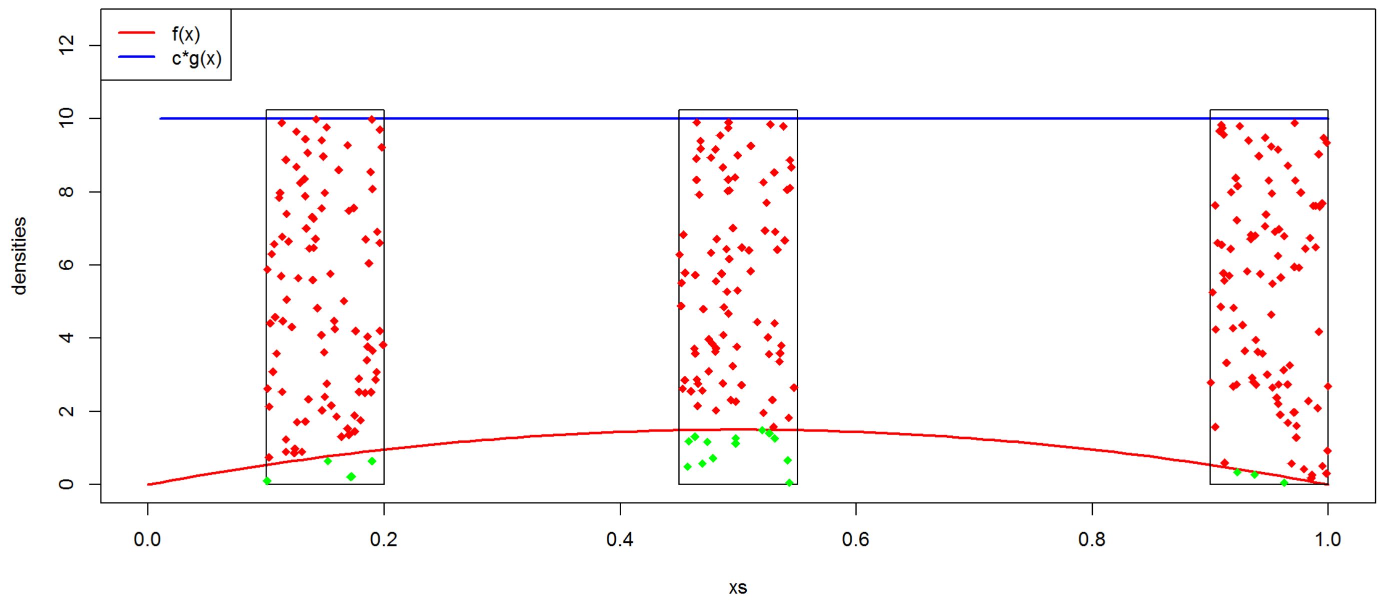
```r
xs <- seq(0, 1, len=100)
plot(xs, dbeta(xs,2,2), ylim=c(0,c*1.3), type="l", col="red", lwd=2, ylab="densities")
lines(xs, c*g(xs), type="l", col="blue", lwd=2)
legend("topleft",c("f(x)","c*g(x)"), col=c("red","blue"), lwd=2)

draw.segment <- function(begin.segment, end.segment) {
  segments(c(begin.segment,end.segment,end.segment,begin.segment), c(0,0,c*1.025,c*1.025),
           c(end.segment,end.segment,begin.segment,begin.segment), c(0,c*1.025,c*1.025,0))
  n.pts <- 100
  us <- runif(n.pts, 0, 1)
  ys <- begin.segment + rg(n.pts)*(end.segment-begin.segment)
  accepted <- us < f(ys)/(c*g(ys))
  points(ys, c*us, col=ifelse(accepted,"green","red"), pch=18)
}

draw.segment(0.10, 0.20)
draw.segment(0.45, 0.55)
draw.segment(0.90, 1.00)
```



The higher the density of f the more points are accepted, as one would expect.

However if $cg(x) >> f(x)$ we will have lots of rejections, which will decrease the quality of the simulation. Let's see what would happen, for the same amount of points, if $c = 10$.

```
c <- 10

xs <- seq(0, 1, len=100)
plot(xs, dbeta(xs,2,2), ylim=c(0,c*1.25), type="l", col="red", lwd=2, ylab="densities")
lines(xs, c*g(xs), type="l", col="blue", lwd=2)
legend("topleft",c("f(x)","c*g(x)"), col=c("red","blue"), lwd=2)

draw.segment(0.10, 0.20)
draw.segment(0.45, 0.55)
draw.segment(0.90, 1.00)
```



This number of points is not enough to get an estimate with the quality of the previous eg.

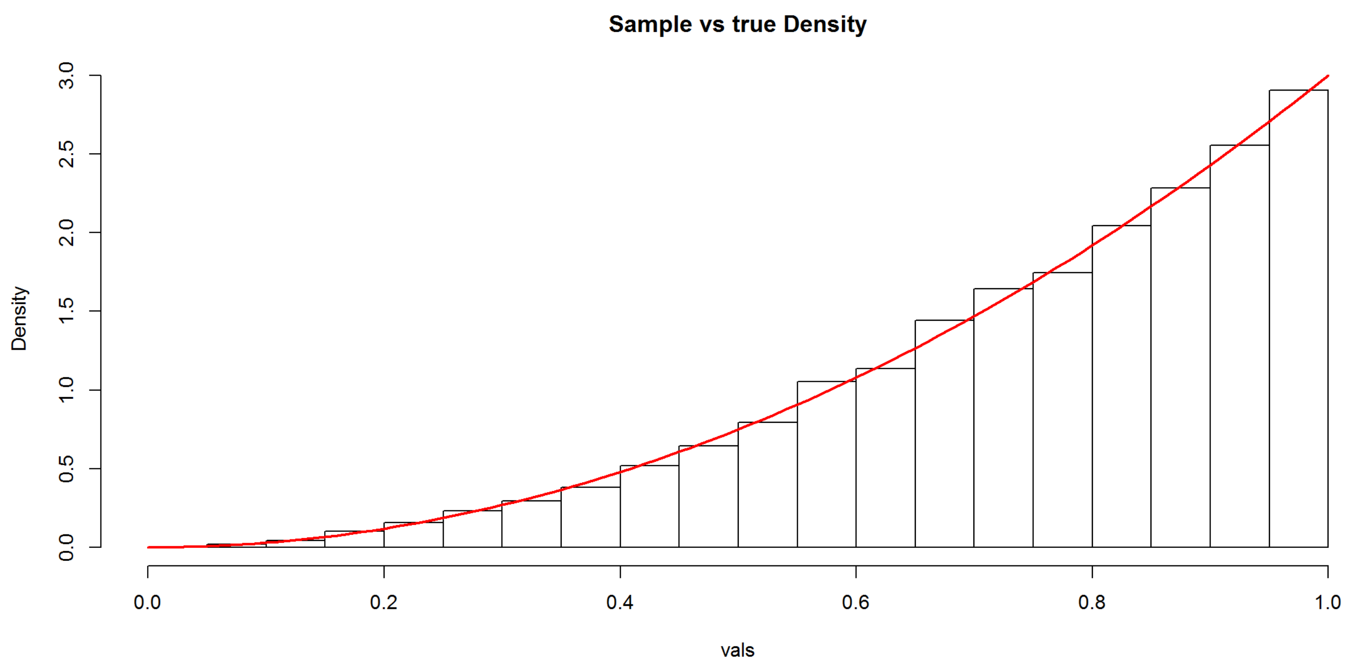
There is a R package (http://artax.karlin.mff.cuni.cz/r-help/library/ars/html/ars.html), `ars` which performs an optimized algorithm named *Adaptative Rejection Sampling*. There's a restriction that the original pdf must be log-concave (http://en.wikipedia.org/wiki/Logarithmically_concave_function).

Let's try with the initial eg for the pdf $f_X(x) = 3x^2$.

The function needs the log of the pdf, in this case, $log(3x^2)$ and its first derivative (http://www.wolframalpha.com/input/?i=d%2Fdx+log%283*x%5E2%29):

$$\frac{d}{dx} log(3x^2) = \frac{2}{x}$$

```
library(ars)

log.f <- function(x) log(3*x^2) # Log(f(x))
log.f.dx <- function(x) 2/x     # d/dx Log(f(x))

vals <- ars(1e4,                # how many points are required
            log.f, log.f.dx,    # the needed functions
            lb=TRUE, ub=TRUE,   # there are a lower and upper bounds for the pdf
            xlb=0, xub=1,       # and which are those bounds
            x=c(.1,.5,.9))      # some initial points inside the pdf

# Checking if it went well
hist(vals, breaks=30, freq=FALSE, main="Sample vs true Density")
xs <- seq(0, 1, len=100)
curve(3*x^2, 0, 1, col="red", lwd=2, add=T)
```



Monte Carlo Integration

Monte Carlo integration is an estimation of the true integration based on random sampling and in the Strong Law of Large Numbers (http://mathworld.wolfram.com/StrongLawofLargeNumbers.html).

The estimator of

$$\theta = \int_a^b g(x)\,dx$$

is computed as follows:

1. Generate $X_1, X_2, \ldots, X_n$ iid from Unif(a,b)
2. Compute $\overline{g(X)} = \frac{1}{n} g(X_i)$
3. The estimation $\hat{\theta} = (b-a)\overline{g(X)}$

$\hat{\theta}$ is itself a random variable, which by the Strong Law of Large Numbers: $\hat{\theta} \to \theta$ as $n \to \infty$

in R:

```
# pre-condition: a < b
MC.simple.est <- function(g, a, b, n=1e4) {
  xi <- runif(n,a,b)      # step 1
  g.mean <- mean(g(xi))   # step 2
  (b-a)*g.mean            # step 3
}
```

The reason why this works:

$$\begin{array}{rcll}\int_a^b g(x)\,dx & = & \int_{\mathbb{R}} g(x)\mathbf{1}_{[a,b]}(x)\,dx & \mathbf{1}_{[a,b]} = 1 \text{ if } x \in [a,b],\ 0 \text{ otherwise} \\ & = & (b-a)\int_{\mathbb{R}} g(x) f_U(x)\,dx & f_U(x) = \frac{1}{b-a}\,\mathbf{1}_{[a,b]}(x), U \sim \text{Unif}(a,b) \\ & = & (b-a)\,E[g(U)] & E[g(X)] = \int g(x)f(x)\,dx, X \sim f \end{array}$$

Eg: estimate

$$\theta = \int_2^4 e^{-x}\,dx = e^{-2} - e^{-4} = 0.1170$$

```
g <- function(x) exp(-x)

MC.simple.est(g, 2, 4)
```

```
## [1] 0.1161926
```

More generally, with pdf f over support A, to estimate the integral

$$\theta = \int_A g(x)f(x)\,dx$$

generate a random sample $x_1, \ldots, x_n$ from the pdf $f(x)$ and compute the mean of the sequence $g(x_i)$, ie, $\hat\theta = \frac{1}{n}\sum_i g(x_i)$.

With probability 1,

$$\lim_{n\to\infty} E[\hat\theta] = \theta$$

The variance of this estimator is:

$$var(\hat\theta) = var\Big(\frac{1}{n}\sum_i g(X_i)\Big) = \frac{1}{n^2}\sum_i var(g(X_i)) = \frac{var(g(X))}{n}$$

Importance Sampling

There are two problems with the previous method:

- It does not apply to unbounded intervals
- It performs poorly if the pdf is not very uniform, namely at distribution tails

Eg: $X \sim N(0,1)$, estimate $P(X > 4.5)$

```
# True value:
pnorm(4.5, lower.tail=FALSE)  # theta (could also be computed by 1-pnorm(4.5))
```

```
## [1] 3.397673e-06
```

```
# MC estimation:
n <- 1e4
indicators <- rnorm(n)>4.5
sum(indicators)/n             # hat.theta
```

```
## [1] 0
```

Simulating directly, there will be a positive hit only every 300k iterations!

One way to solve this is to consider other more appropriate densities.

This leads to a general method called *importance sampling*.

Instead of evaluate $E_f[g(X)] = \int_A g(x)f(x)\, dx$ this method includes a candidate or auxiliar density $h(x)$,

$$E_f[g(X)] = \int_A g(x)\, \frac{f(x)}{h(x)}\, h(x)\, dx = E_h\Big[\frac{g(X)f(X)}{h(X)}\Big]$$

So,

$$E_f[g(X)] \approx \frac{1}{n} \sum_{i=1}^{n}\, g(y_i)\frac{f(y_i)}{h(y_i)}$$

where $y_1, y_2, \ldots, y_n$ are generated by pdf h.

The candidate h should be chosen as to satisfy as much as possible: $h(x) \approx |g(x)|f(x)$.

The restrictions are:

- The variance of gf/h must be finite (the tails of h must be higher than those of f)
- The support of h must include the support of f

This is the same idea of the accept-reject method.

The importance-sampling function in R:

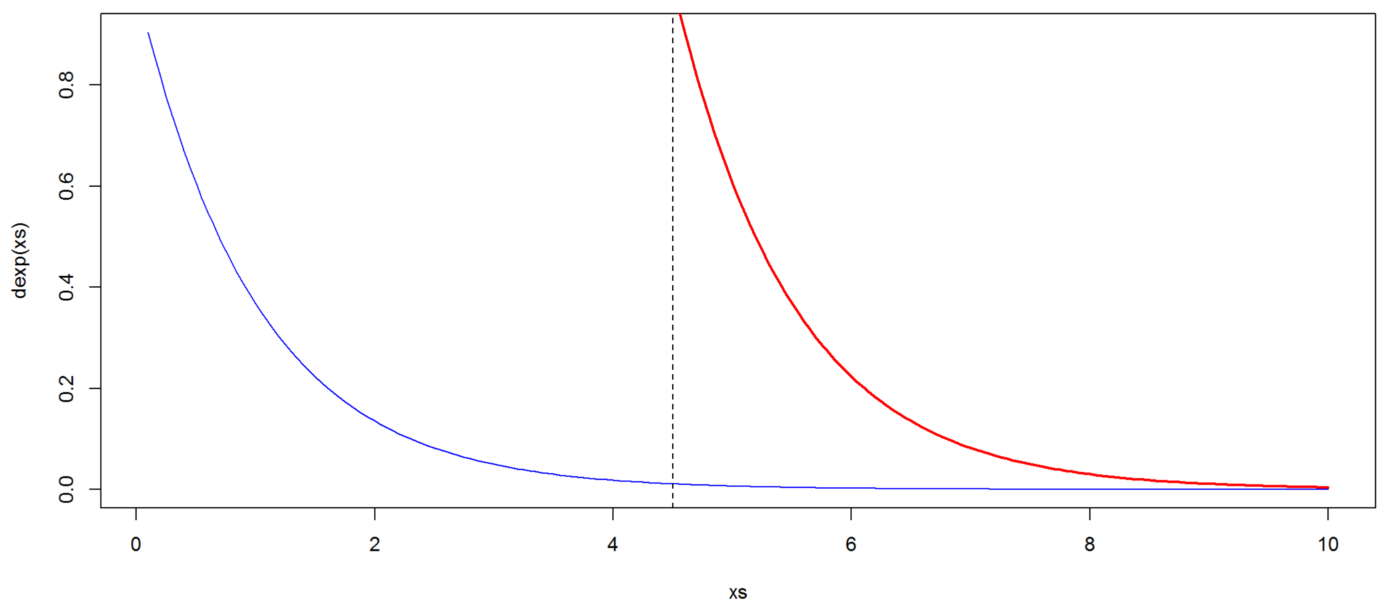
```
# rh generates samples from candidate pdf
i.sampling <- function(f, g, h, rh, n=1e4) {
  ys <- rh(n)
  mean(g(ys)*f(ys)/h(ys))
}
```

Let's use the previous eg of estimating $P(X > 4.5)$.

Our target function will be the exponential pdf truncated at 4.5:

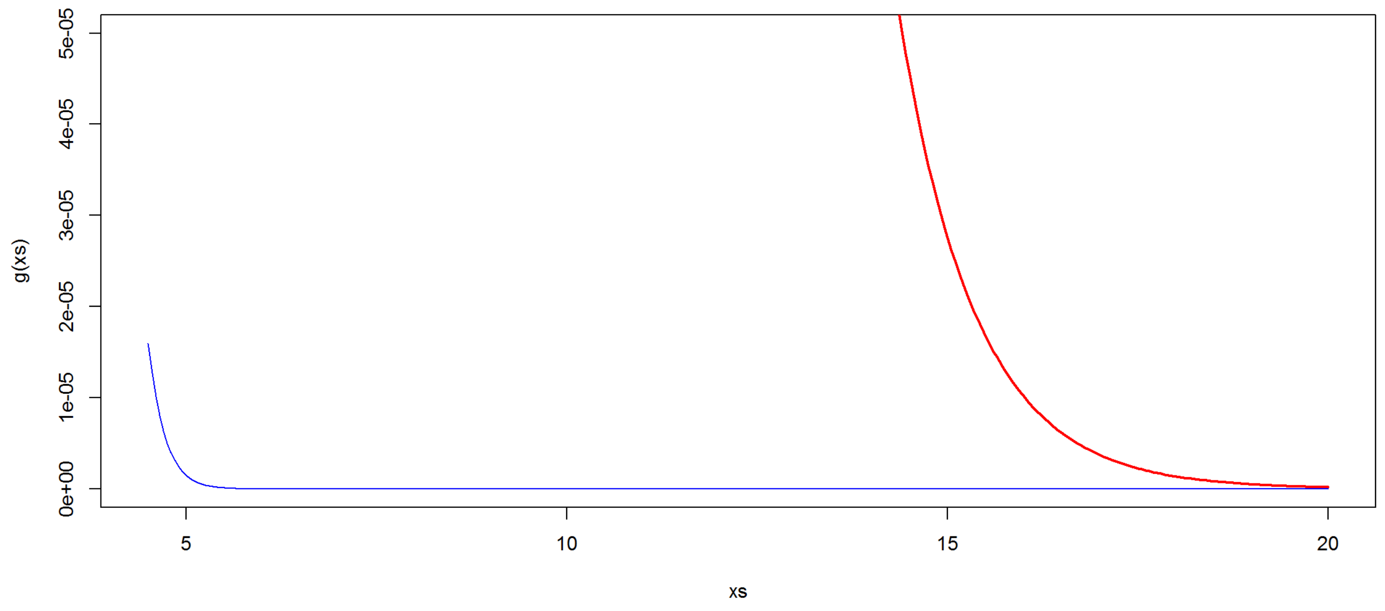
$$h(x) = \frac{e^{-x}}{\int_{4.5}^{\infty} e^{-x}\, dx} = e^{-(x-4.5)}$$

```
xs <- seq(0.1,10,by=0.05)
plot(xs,dexp(xs),col="blue", type="l")   # the exponential pdf
lines(xs,exp(-(xs-4.5)),col="red",lwd=2) # the truncated pdf
abline(v=4.5,lty=2)
```



Here's a plot of the target pdf g (in blue) and the candidate pdf h (in red):

```r
g <- dnorm
h <- function(x) exp(-(x-4.5))

xs <- seq(4.5,20,by=0.05)
plot(xs,g(xs),col="blue", type="l", ylim=c(0,0.5e-4))
lines(xs,h(xs),col="red",lwd=2)
```



```r
# True value:
pnorm(4.5, lower.tail=FALSE)  # theta (could also be computed by 1-pnorm(4.5))
```

```
## [1] 3.397673e-06
```

```
# do the Importance Sampling
f  <- function(x) x/x          # uniform pdf
rh <- function(n) rexp(n)+4.5 # rexp() shifted to 4.5
i.sampling(f,g,h,rh)
```

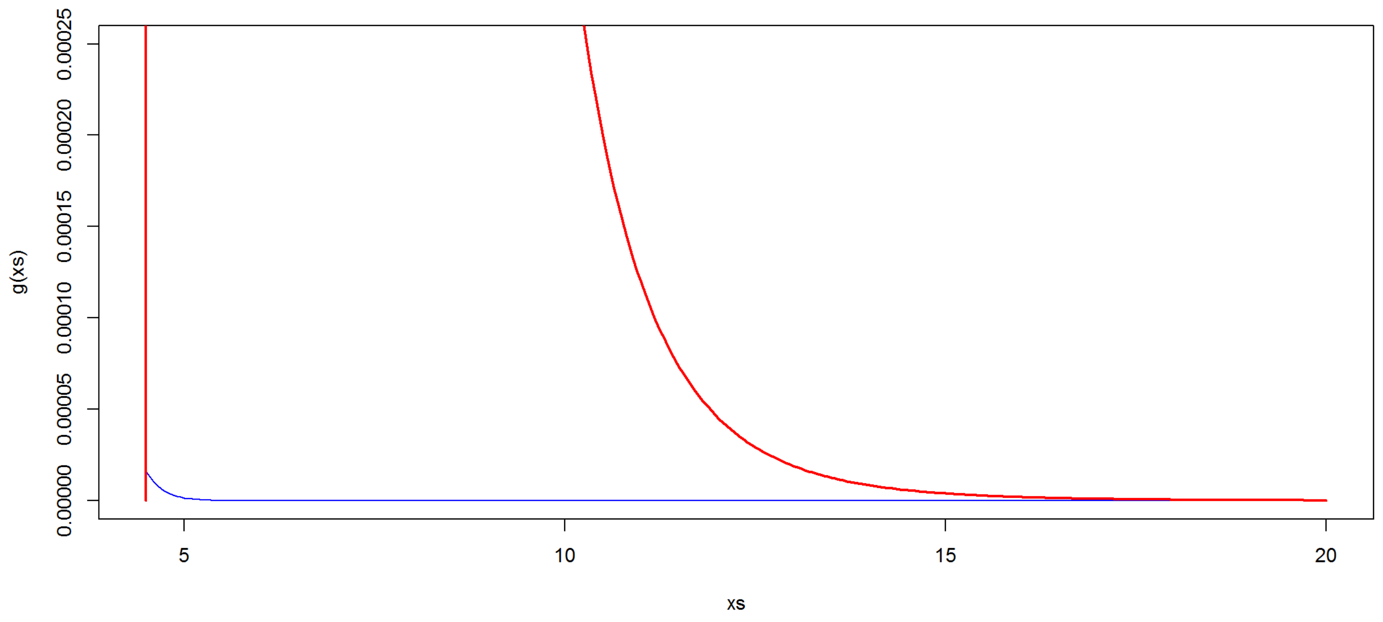
```
## [1] 3.407674e-06
```

Another sampling choosing a pareto distribution for h:

```
library(VGAM)
```

```
## Loading required package: stats4
## Loading required package: splines
##
## Attaching package: 'VGAM'
##
## The following object is masked from 'package:distr':
##
##     Max
```

```
h <- function(x) {
  dpareto(x, scale=4.5, shape=10)
}

xs <- seq(4.5,20,by=0.05)
plot(xs,g(xs),col="blue", type="l", ylim=c(0,0.25e-3))
lines(xs,h(xs),col="red",lwd=2)
```



```r
rh <- function(n) rpareto(n, 4.5, 10)

i.sampling(f,g,h,rh)
```

```
## [1] 3.384464e-06
```

MC in Inference

In statistical inference there is a sample $x_1, \ldots, x_n$ taken from a certain probabilistic model.

An important task is to **estimate** a statistic $\theta(x_1, \ldots, x_n)$, usually for estimation of a model parameter. If we don't know the distribution of θ we can study it using Monte Carlo methods.

Say, to estimate the mean $E[\theta]$:

1. For each $i = 1 \ldots N$:
    1. Generate a sample $x_1, \ldots, x_n$
    2. Compute $\theta^{(i)} = \theta(x_1, \ldots, x_n)$
2. Compute $E[\theta] = \frac{1}{N} \sum_i \theta^{(i)}$

Eg: given two standard normal iid random vars X_1, X_2 estimate $E[|X_1 - X_2|]$

The analytical solution is found by:

$$\begin{array}{rcll} \int\int |x_1 - x_2| f(x_1, x_2) dx_1 dx_2 & = & \int\int |x_1 - x_2| f(x_1) f(x_2) dx_1 dx_2 & X_1 \perp X_2 \\ & = & \int_{-\infty}^{+\infty} \int_{-\infty}^{\infty} |x_1 - x_2| \frac{1}{\sqrt{2\pi}} e^{-x_1^2/2} \frac{1}{\sqrt{2\pi}} e^{-x_2^2/2} dx_1 dx_2 & X_1, X_2 \sim \mathcal{N}(0,1) \\ & = & \int_{-\infty}^{+\infty} \int_{-\infty}^{\infty} |x_1 - x_2| \frac{1}{2\pi} e^{-(x_1^2 + x_2^2)/2} dx_1 dx_2 & \\ & = & \frac{2}{\sqrt{\pi}} & \\ & \approx & 1.12838 & \end{array}$$

note: Mathematica formula to compute this integral:

```
Integrate[(Abs[x - y]*Exp[(-x^2 - y^2)/2])/(2*Pi), {x, -Infinity, Infinity}, {y, -Infinity, Infin
ity}]
```

In R:

```r
n <- 1e5
X1 <- rnorm(n,0,1)
X2 <- rnorm(n,0,1)

f <- function(x1,x2) abs(x1-x2)

thetas <- f(X1,X2)
mean(thetas)
```

```
## [1] 1.127228
```

The standard error of a mean $\overline{X}$ of a sample size n is $\sqrt{var(X)/n}$. When the distribution is unknown we replace it by the empirical distribution of the sample $x_1, \ldots, x_n$, which becomes

$$\widehat{var(X)} = \frac{1}{n} \sum_{i=1}^{n} (x_i - \overline{x})^2$$

And the standard error:

$$\widehat{se(\overline{x})} = \sqrt{\frac{1}{n^2} \sum_{i=1}^{n} (x_i - \overline{x})^2} = \frac{1}{n} \sqrt{\sum_{i=1}^{n} (x_i - \overline{x})^2}$$

From the previous eg:

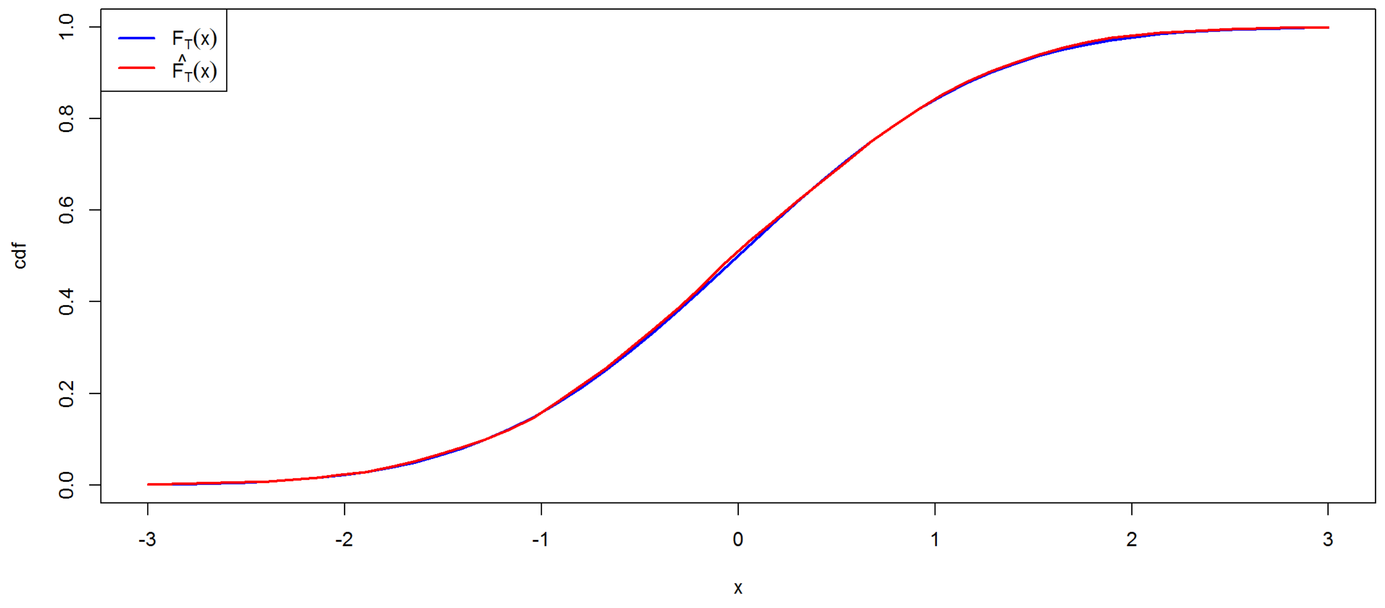
```
# standard error = sqrt( var(|X1-X2|)/n )
sqrt(var(thetas)/n)
```

```
## [1] 0.002694323
```

To estimate the cdf of θ,

$$F_T(x) = P(T \leq x) \approx \widehat{F_T}(x) = \frac{1}{n} \sum_{i=1}^{n} \mathbf{1}_{\theta^{(i)} \leq x}$$

```
n <- 1e4
thetas <- rnorm(n,0,1)    # the sample

bins <- 50; xs <- seq(-3,3,len=bins)      # how many divisions of the x-axis we use

cdf.hat <- rep(NA,length(xs))
for (i in 1:bins) {
  cdf.hat[i] <- sum(thetas < xs[i]) / n   # compute the cdf estimation
}

# plot real cdf (in blue) vs estimated cdf (in red)
plot(xs,pnorm(xs,0,1), type="l", col="blue", lwd=2, xlab="x", ylab="cdf")
lines(xs,cdf.hat, col="red", lwd=2)
legend("topleft",c(expression(F[T](x)),expression(hat(F[T])(x))), col=c("blue","red"), lwd=2)
```



Markov Chain Monte Carlo (MCMC) Integration

When we want to estimate $E[g(\theta)]$ we can find the sample mean

$$\bar{g} = \frac{1}{m}\sum_{i=1}^{m} g(x_i)$$

where the sample $x_1, \dots, x_m$ is sampled from an appropriate density.

If $x_1, \dots, x_m$ are independent then by the laws of large numbers, the mean converges in probability to $E[g(\theta)]$. This can be done by regular Monte Carlo integration.

However it can be difficult to implement a method to generate iid observations. But even if the observations are dependent, a MC integration can still be applied if the generated (dependent) observations have a joint density roughly the same of the joint density of a random, iid sample. To achieve this it is used Markov Chains (http://en.wikipedia.org/wiki/Markov_chain), which provides the sampler that generates the dependent observations from the target distribution.

The Metropolis-Hastings algorithm is a MCMC method that tries to achieve this task. The main idea is to generate a Markov Chain $X_t | t = 0, 1, \dots$ such that its stationary distribution is the target distribution. The algorithm, given X_t , knows how to compute X_{t+1}. To do that it must be able to generate a candidate point Y from a proposal distribution $g(\cdot|X_t)$ which (probably) depends on the previous state. This point Y may or may not be accepted. If it is accepted, then $X_{t+1} = Y$, otherwise the chain remains in the same place $X_{t+1} = X_t$.

The proposal distribution g must be chosen so that the generated chain will converge to a stationary distribution, in this case the target distribution f. The proposal distribution is the way we generate possible good points for the target distribution. If the proposal is not well chosen, the algorithm will produce lots of rejections and the time to converge to the target distribution might take more time than we have available.

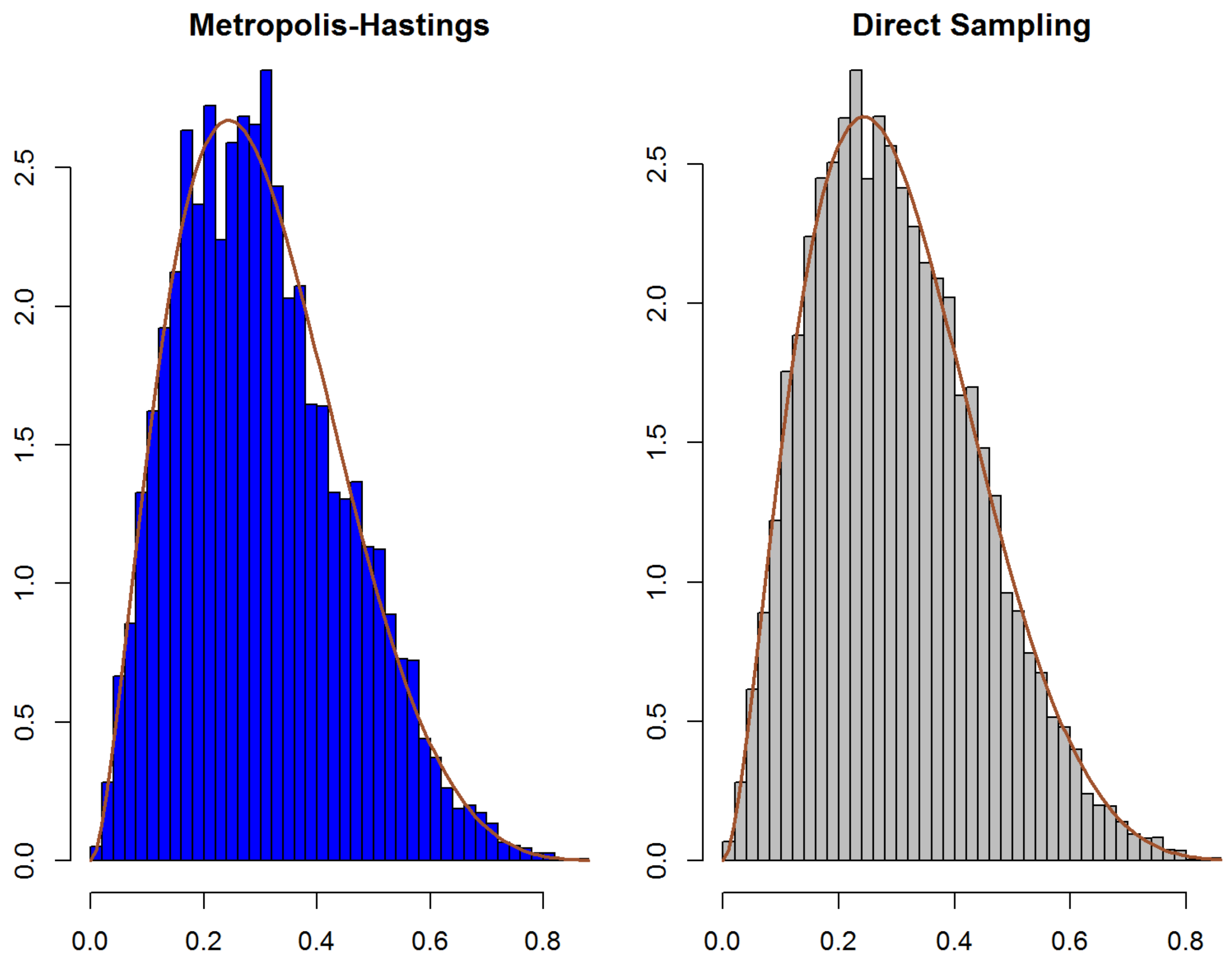
If the generation of candidates does not depend on the current region of the chain, the proposal distribution can be independent of x_t, and the algorithm will accept the new candidate y if $f(y)/g(y) \geq f(x_t)/g(x_t)$

Here's the R code (algorithm details can be found at Rizzo's book, pag.248):

```
metropolis.hastings <- function(f,  # the target distribution
                                g,  # the proposal distribution
                                rg, # a sample from the proposal distribution
                                x0, # initial value for chain, in R it is x[1]
                                chain.size=1e5,  # chain size
                                burn.perc=0.1) { # burn in percentage

  x <- c(x0, rep(NA,chain.size-1))  # initialize chain

  for(i in 2:chain.size)   {
    y <- rg(x[i-1])                 # generate Y from g(.|xt) using sampler rg
    alpha <- min(1, f(y)*g(x[i-1],y)/(f(x[i-1])*g(y,x[i-1])))
    x[i] <- x[i-1] + (y-x[i-1])*(runif(1)<alpha)  # update step
  }

  # remove initial part of the chain before output result
  x[(burn.perc*chain.size) : chain.size]
}
```

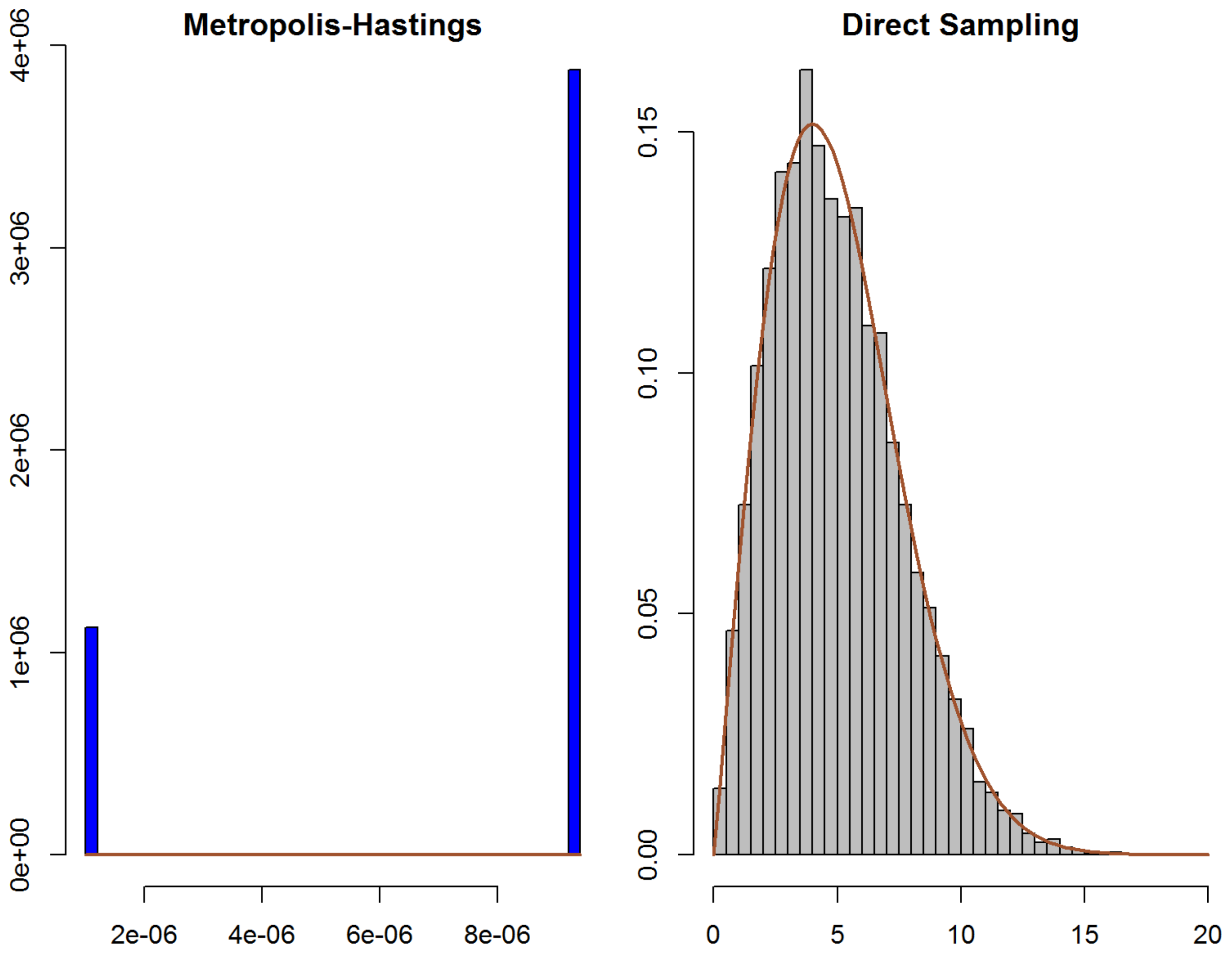
This first eg samples from an uniform distribution (the proposal distribution) to generate a sample from a Beta(2.7, 6.3) distribution:

```
a<-2.7; b<-6.3; size<-1e4

f  <- function(x)   dbeta(x,a,b)
rg <- function(x)   runif(1,0,1)
g  <- function(x,y) 1 # i.e., dunif(x,0,1)

X <- metropolis.hastings(f,g,rg,x0=runif(1,0,1),chain.size=size)

par(mfrow=c(1,2),mar=c(2,2,1,1))
hist(X,breaks=50,col="blue",main="Metropolis-Hastings",freq=FALSE)
curve(dbeta(x,a,b),col="sienna",lwd=2,add=TRUE)
hist(rbeta(size,a,b),breaks=50,col="grey",main="Direct Sampling",freq=FALSE)
curve(dbeta(x,a,b),col="sienna",lwd=2,add=TRUE)
```



This second eg samples from a chi-squared distribution to generate a sample from the Rayleigh distribution (http://en.wikipedia.org/wiki/Rayleigh_distribution) (assume parameter $\sigma = 4$):

```
library(VGAM)

sigma <- 4
f  <- function(x)   drayleigh(x,sigma) # the target distribution
rg <- function(x)   rchisq(1,df=x)     # a sample from proposal g(.|x)
g  <- function(x,y) dchisq(x,df=y)     # the pdf at g(x|y)

X <- metropolis.hastings(f,g,rg,x0=rchisq(1,df=1),chain.size=size)

par(mfrow=c(1,2),mar=c(2,2,1,1))
hist(X,breaks=50,col="blue",main="Metropolis-Hastings",freq=FALSE)
curve(drayleigh(x,sigma),col="sienna",lwd=2,add=TRUE)
hist(rrayleigh(size,sigma),breaks=50,col="grey",main="Direct Sampling",freq=FALSE)
curve(drayleigh(x,sigma),col="sienna",lwd=2,add=TRUE)
```

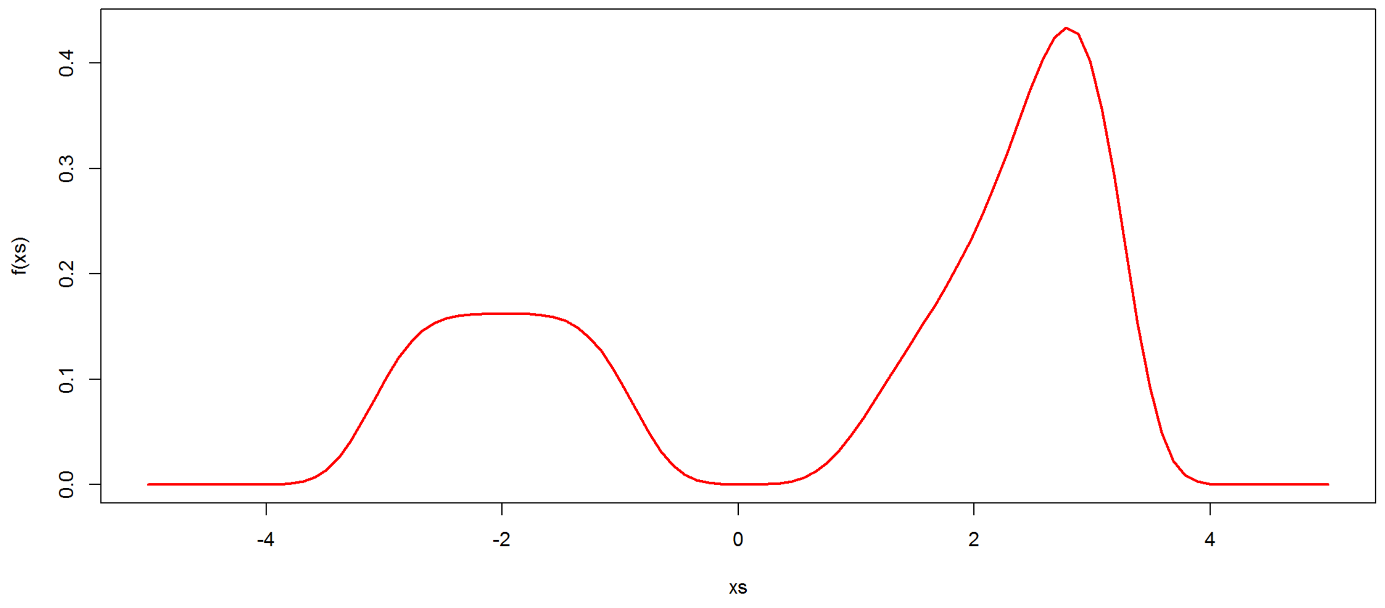


In this next eg we wish to compute the expected value of

$$f(x) = c * \left(exp(-\frac{(x-2)^4 - 2x}{2}) + 5exp(-\frac{(x+2)^4}{2}) \right), x \in \mathcal{R}$$

First let's plot it, and we see it's bimodal:

```
#  c is 1/30.8636 necessary to make it a density, but we didn't need to know it
f <- function(x) (exp(-((x-2)^4-2*x)/2) + 5*exp(-(x+2)^4/2)) / 30.8636

xs <- seq(-5,5,len=100)
plot(xs,f(xs),type="l",col="red",lwd=2)
```



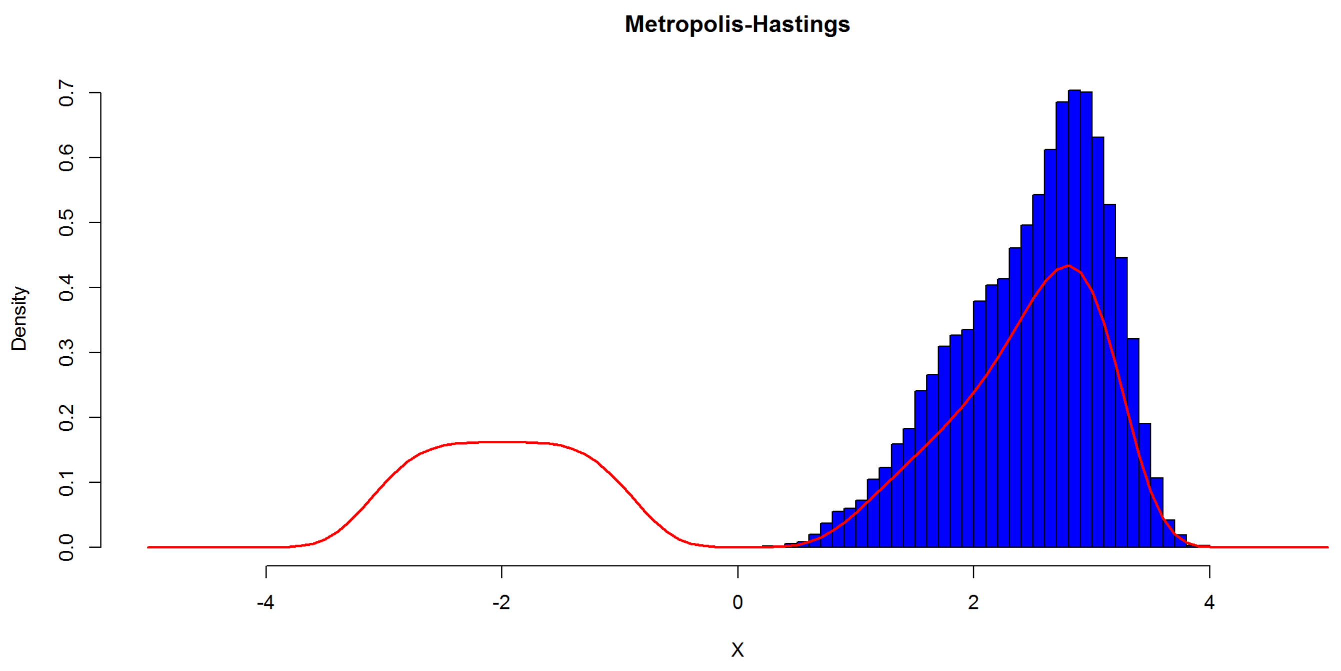
To find $E_f[X]$ we'll use the Metropolis-Hastings algorithm. In this case, the candidate function after x_t will be $q(y|x_t) \sim \mathcal{N}(x_t, \sigma^2)$. In our first test we choose $\sigma = 0.1$:

```
g  <- function(x, y) dnorm(x,y,0.1)
rg <- function(x)    rnorm(1,x,0.1)

set.seed(101)
X <- metropolis.hastings(f,g,rg,x0=1,chain.size=5e4)
mean(X) # the answer?
```

```
## [1] 2.482991
```

This value seems wrong. Let's compare the histogram of chain X with the true density:

```
hist(X,breaks=50,col="blue",xlim=c(-5,5),main="Metropolis-Hastings",freq=FALSE)
curve(f(x),col="red",lwd=2,add=TRUE)
```

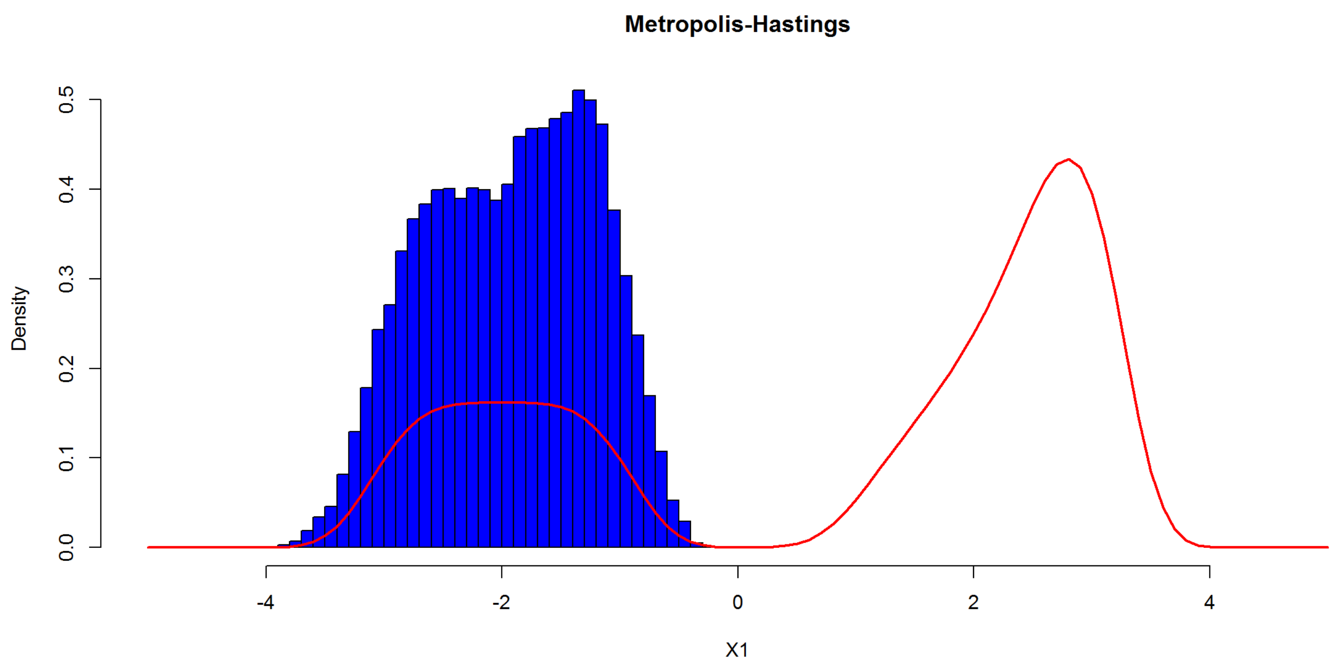


What happened? Since the candidate function is a normal with a very short σ the potential candidates that it produces are very close to the last x_t which means the algorithm is unable to cross 0 to the left side. We can check what happens if we start at a negative x_0:

```
set.seed(101)
X1 <- metropolis.hastings(f,g,rg,x0=-2,chain.size=5e4)
mean(X1)
```

```
## [1] -1.928099
```

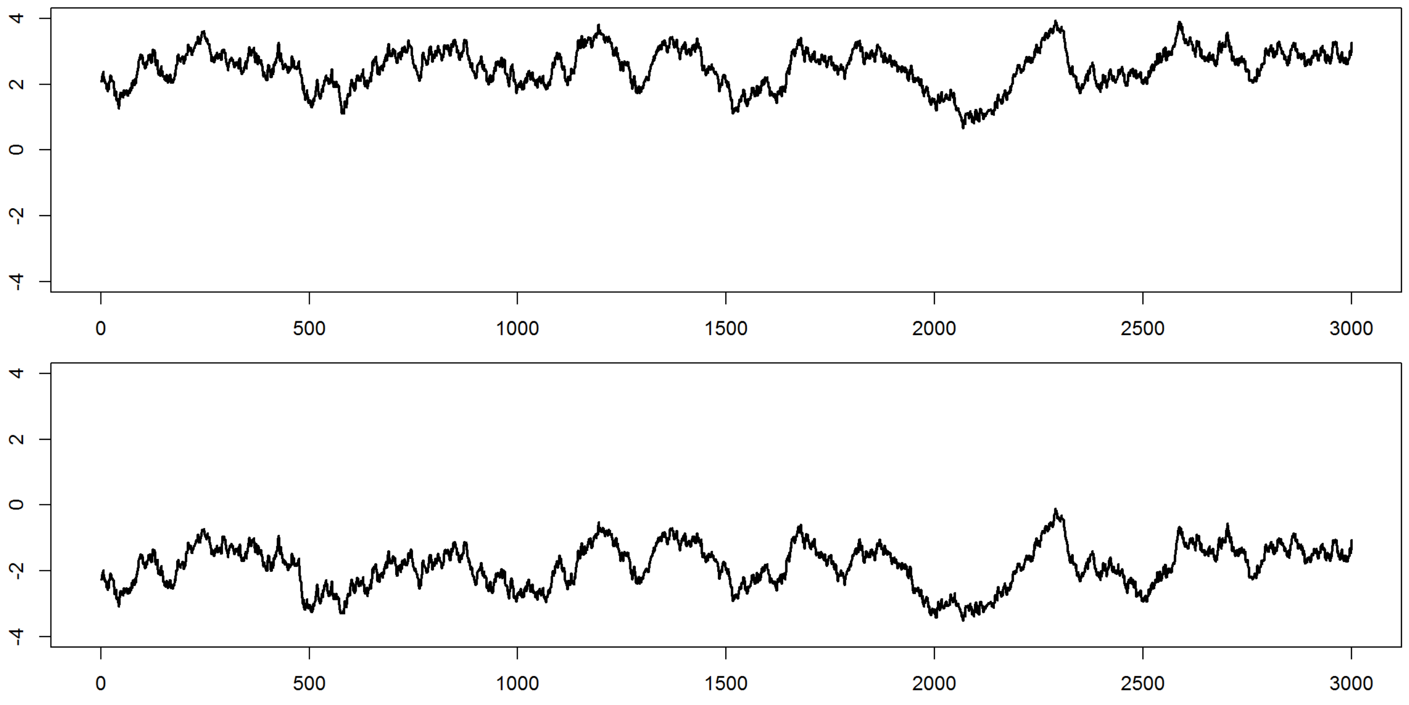
```
hist(X1,breaks=50,col="blue",xlim=c(-5,5),main="Metropolis-Hastings",freq=FALSE)
curve(f(x),col="red",lwd=2,add=TRUE)
```



Precisely what was expected, now the chain is unable to cross to the right side.

Let’s visualize a bit of both previous markov chains and see how they are unable to jump to the other side of the bimodal density:

```
par(mfrow=c(2,1),mar=c(2,2,1,1))
plot(1:3000,X[1:3000], lwd=2,type="l",ylim=c(-4,4))
plot(1:3000,X1[1:3000], lwd=2,type="l",ylim=c(-4,4))
```



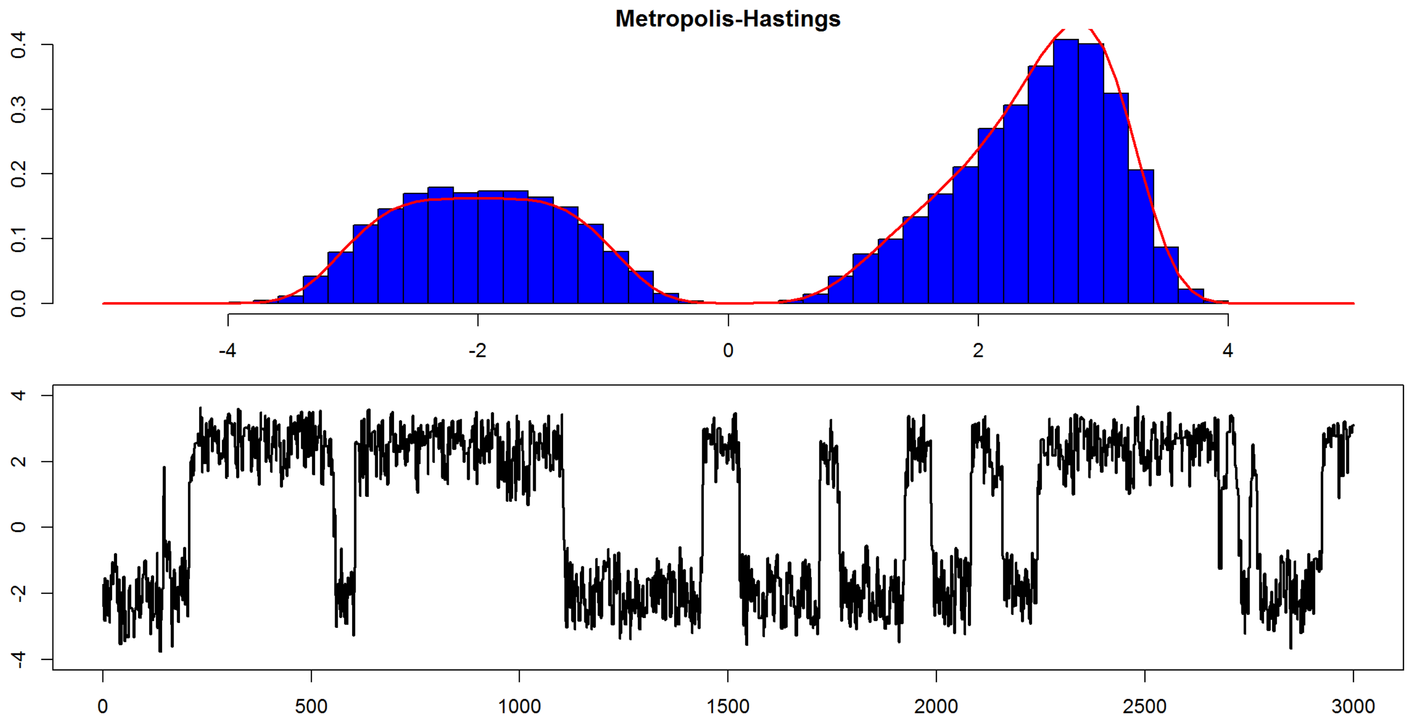
So let’s try a higher sigma, say $\sigma = 1$:

```
g  <- function(x, y) dnorm(x,y,1)
rg <- function(x)    rnorm(1,x,1)

set.seed(101)
X2 <- metropolis.hastings(f,g,rg,x0=runif(1,-4,4),chain.size=5e4)
mean(X2) # the answer
```

```
## [1] 0.797984
```

It seems a more sensible answer. Let's check the histogram and the initial part of the chain:

```
par(mfrow=c(2,1),mar=c(2,2,1,1))
hist(X2,breaks=50,col="blue",xlim=c(-5,5),main="Metropolis-Hastings",freq=FALSE)
curve(f(x),col="red",lwd=2,add=TRUE)
plot(1:3000,X2[1:3000], lwd=2,type="l",ylim=c(-4,4))
```



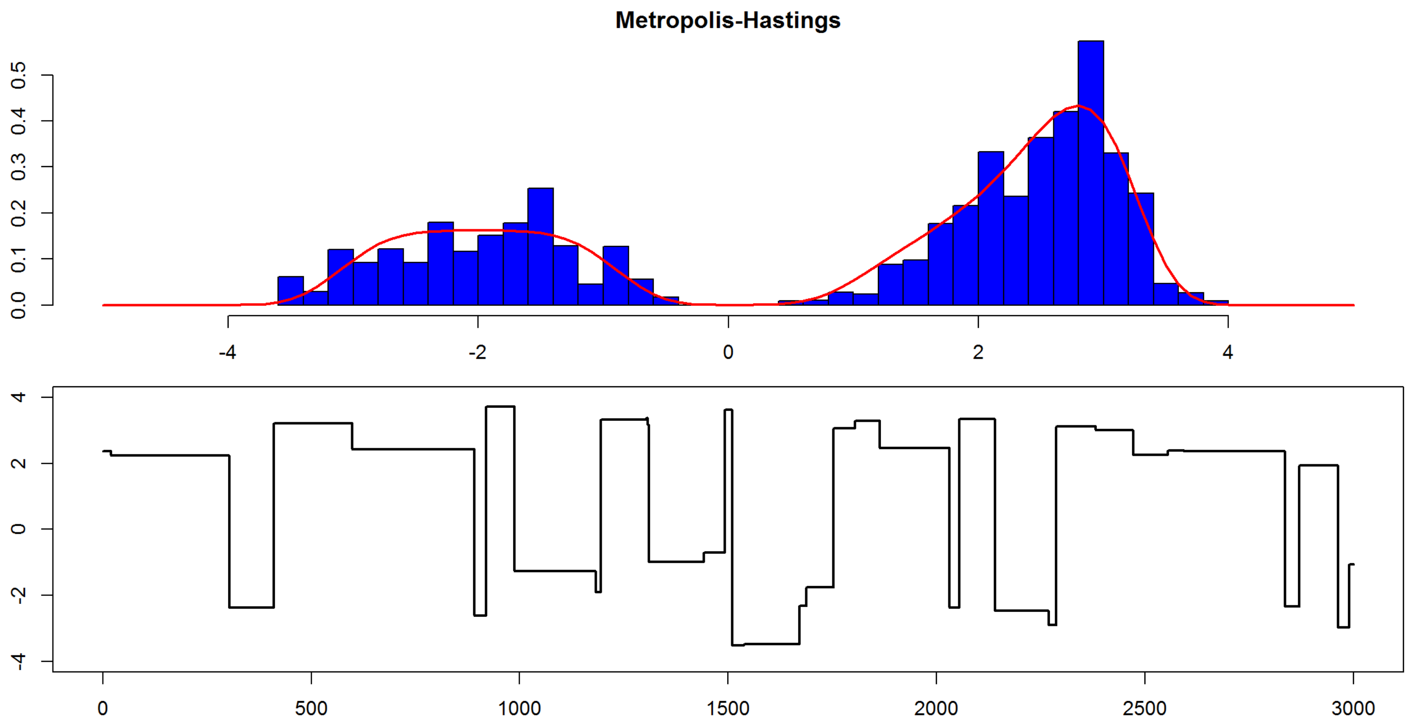
Now the candidate function is able to make longer jumps, and both parts of the density are visited, providing a good estimate of the true density.

An exagerated value of sigma will have another type of disadvantage: most candidates will be so far the interesting area that they will be simply rejected which will result in a poor estimate:

```
g  <- function(x, y) dnorm(x,y,100) # sigma = 100 (!)
rg <- function(x)    rnorm(1,x,100)

set.seed(101)
X3 <- metropolis.hastings(f,g,rg,x0=runif(1,-4,4),chain.size=5e4)
mean(X3)
```

```
## [1] 0.9172019
```

```
par(mfrow=c(2,1),mar=c(2,2,1,1))
hist(X3,breaks=50,col="blue",xlim=c(-5,5),main="Metropolis-Hastings",freq=FALSE)
curve(f(x),col="red",lwd=2,add=TRUE)
plot(1:3000,X3[1:3000], lwd=2,type="l",ylim=c(-4,4))
```



The plateau's above show repeated rejections, making the chain stay at the last change of x_t.

Maximum Likelihood Estimation

Maximum Likelihood Estimation (MLE) is a method of estimating the parameters of a statistical model.

Assume we have a model $f(x|\theta)$ with unknown parameter θ and a random sample $\underset{\sim}{X} = (X_1, X_2, \ldots, X_n)$ drawn from that model. What is the most probable value for θ?

LME says that we should maximize the likelihood function $L(\theta|\underset{\sim}{x})$:

$$\hat{\theta} = \arg\max_{\theta} f(\underset{\sim}{x}|\theta) = \arg\max_{\theta} \prod_{i=1}^{n} f(x_i|\theta)$$

To ease the computations, usually we compute instead the log-likelihood $L^*(\theta|\underset{\sim}{x})$:

$$\hat{\theta} = \arg\max_{\theta} \log f(\underset{\sim}{x}|\theta) = \arg\max_{\theta} \sum_{i=1}^{n} \log f(x_i|\theta)$$

If there is no analytical solution we need to find this maximum with computational methods.

If θ is a scalar we can obtain the maximum by solving

$$\frac{d}{d\theta} \log L^*(\theta|\underset{\sim}{x}) = 0$$

Note that if the likelihood function has lots of local maxima, then this task is far from trivial. If it is a concave function, then it's possible to get the θ that maximizes it.

Example 1-D parameter: consider a random sample $\underset{\sim}{X} = (X_1, X_2, \ldots, X_n)$ taken from the exponential pdf

$$f(x|\theta) = \theta e^{-\theta x}, x > 0, \theta > 0$$

The log-likelihood L^* is

$$L^*(\theta|\underset{\sim}{x}) = \sum_{i=1}^{n} \log f(x_i|\theta) = \sum_{i=1}^{n} \log \theta e^{-\theta x} = n \log \theta - \theta \sum_{i=1}^{n} x_i$$

Analytically,

$$\frac{d}{d\theta} \log L^*(\theta|\underset{\sim}{x}) = 0$$

becomes

$$\frac{n}{\theta} - \sum_{i=1}^{n} x_i = 0 \iff \theta = \frac{n}{\sum_{i=1}^{n} x_i} \iff \hat{\Theta} = \frac{1}{\overline{X}}$$

In R:

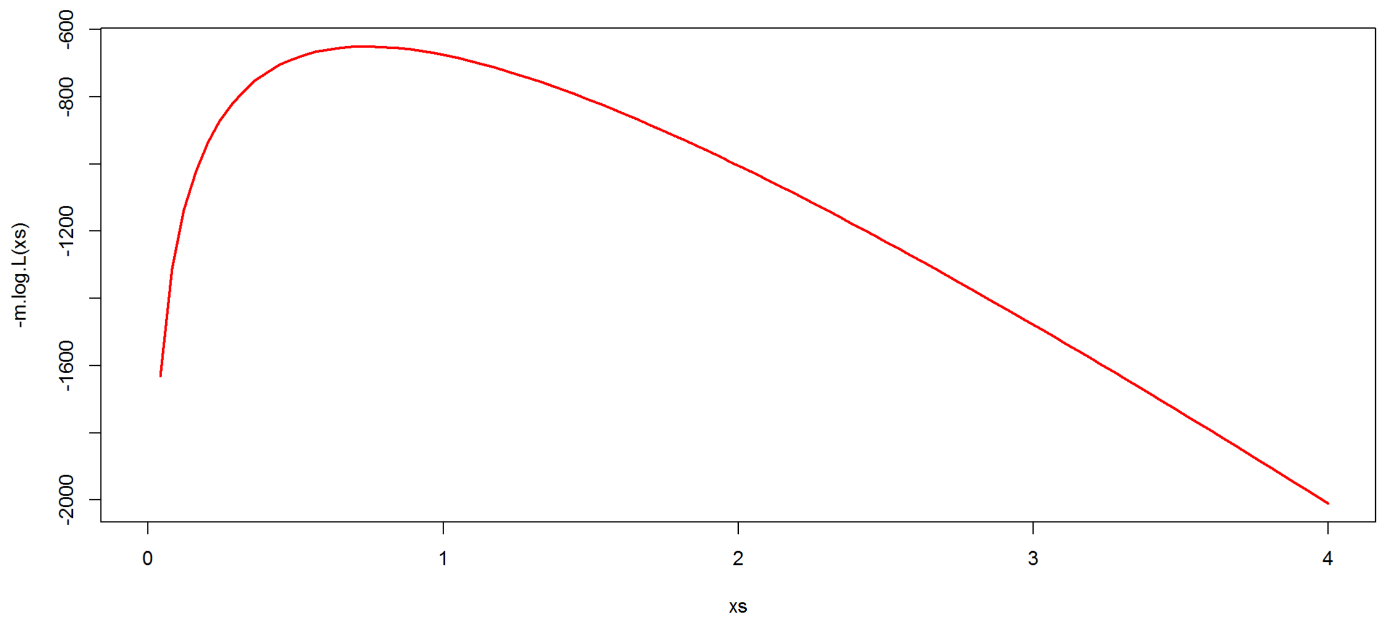
```
set.seed(121)
theta <- 0.75 # value unknown

x <- rexp(500, theta)   # create a random sample

theta.hat <- 1/mean(x)  # analitical solution for MLE
theta.hat
```

```
## [1] 0.7398035
```

But let's assume the analitycal solution was unavailable.

- Method 1: function `stats4::mle` receives the $-L^*$ function and a starting point, and tries to optimize θ:

```
library(stats4)

# -log likelihood for exponential pdf given sample x
get.minus.log.L <- function(x) {  # closure: we wrap random sample x into the log.likelihood
  function(theta) {
    - (length(x)*log(theta) - theta * sum(x))
  }
}

m.log.L <- get.minus.log.L(x)

xs <- seq(0,4,len=100); plot(xs, -m.log.L(xs), type="l", col="red", lwd=2)
```



```
result <- mle(m.log.L, start=list(theta=1))
```

```
## Warning in log(theta): NaNs produced
```

```
## Warning in log(theta): NaNs produced
```

```
## Warning in log(theta): NaNs produced
```

```
## Warning in log(theta): NaNs produced
```

```
summary(result)
```

```
## Maximum likelihood estimation
##
## Call:
## mle(minuslogl = m.log.L, start = list(theta = 1))
##
## Coefficients:
##       Estimate Std. Error
## theta 0.739804 0.03308498
##
## -2 log L: 1301.371
```

- Method 2: function `optimize` to perform one dimensional optimization which needs an interval to search the optimum.

```
optimize(m.log.L, lower=0, upper=4, maximum=FALSE) # we find the minimum because we are reusing t
he previous minus.log.L
```

```
## $minimum
## [1] 0.7397983
##
## $objective
## [1] 650.6853
```

- Method 3: function `uniroot` that computes the one dimensional root of a function. In this case, the root we wish to know is the value that answers $\frac{d}{d\theta}\log L^*(\theta|\underset{\sim}{x}) = 0$

```
# d/dtheta log likelihood for exponential pdf given sample x
get.d.log.L <- function(x) {
  function(theta) {
    length(x)/theta - sum(x)
  }
}

d.log.L <- get.d.log.L(x)

uniroot(d.log.L, lower=0, upper=4)
```

```
## $root
## [1] 0.7398048
##
## $f.root
## [1] -0.00116161
##
## $iter
## [1] 8
##
## $init.it
## [1] NA
##
## $estim.prec
## [1] 0.0001074219
```

All methods return the same value 0.7398035.

Example 2-D parameter: consider a random sample $\underset{\sim}{X} = (X_1, X_2, \ldots, X_n)$ taken from the gamma pdf

$$f(x|\theta) = f(x|\alpha,\lambda) = \frac{\lambda^\alpha}{\Gamma(\alpha)} x^{\alpha-1} e^{-\lambda x}$$

The log-likelihood L^* is

$$L^*(\alpha,\lambda|\underset{\sim}{x}) = n\alpha\log\lambda - n\log\Gamma(\alpha) + (\alpha-1)\sum_{i=1}^{n}\log x_i - \lambda\sum_{i=1}^{n} x_i$$

In R:

```r
set.seed(121)
alpha  <- 5
lambda <- 2 # unknown values

x <- rgamma(500, shape=alpha, rate=lambda)   # create a random sample

# log likelihood for gamma pdf given sample x
get.log.L <- function(x) {
  n         <- length(x)
  sum.x     <- sum(x)
  sum.log.x <- sum(log(x))

  function(theta) {
    alpha  <- theta[1]
    lambda <- theta[2]
    - (n*alpha*log(lambda) - n*log(gamma(alpha)) + (alpha-1)*sum.log.x - lambda*sum.x)
  }
}
```

The function `optim` is a general purpose optimization method (check its help file) that receives the function to be minimized, and a vector of initial values for the parameters to start the search

```r
optim(par=c(1,1), fn=get.log.L(x))  # check $par for the MLE parameters
```

```
## $par
## [1] 4.945030 1.970068
##
## $value
## [1] 734.6287
##
## $counts
## function gradient
##       75       NA
##
## $convergence
## [1] 0
##
## $message
## NULL
```

Herein we used $(1,1)$ as the initial searching point but it’s not obvious where we should begin. One way to find a good initial estimate is to use the method of moments (at least when we know the moments). In this case, we know that the mean and variance of the gamma are:

$$E[X|\alpha,\lambda] = \frac{\alpha}{\lambda} \approx \overline{x}$$

$$var(X|\alpha,\lambda) = \frac{\alpha}{\lambda^2} \approx s^2$$

We use the sample mean and the sample variance to plug into the previous system of equations and solve it to get an estimate of the parameters:

```
mean(x)
```

```
## [1] 2.510211
```

```
var(x)
```

```
## [1] 1.251881
```

In this case, the initial values for the parameters would be quite good (since we have a big sample), namely $\lambda \approx 2$ and $\alpha \approx 5$.

A second method is by using a 'do it yourself' gradient descent method.

The iterative method of Newton-Raphson (http://en.wikipedia.org/wiki/Newton's_method) used to find the roots of a function, says that we can approximate the real value, given an initial value θ_0 by the following first order of the Taylor expansion:

$$\theta_1 = \theta_0 - \frac{h(\theta_0)}{h'(\theta_0)}$$

This method should be iterated

$$\theta_{i+1} = \theta_i - \frac{h(\theta_i)}{h'(\theta_i)}$$

until a sufficient accurate value is produced.

In our example of finding the MLE of the gamma function, we find the two derivates and equal them to zero (the place where the maximum can be found).

As a reminder, the log-likelihood is:

$$L^*(\alpha, \lambda | \underset{\sim}{x}) = n\alpha \log \lambda - n \log \Gamma(\alpha) + (\alpha - 1) \sum_{i=1}^{n} \log x_i - \lambda \sum_{i=1}^{n} x_i$$

The derivates are:

$$\frac{\delta}{\delta \lambda} L^* = \frac{n\alpha}{\lambda} - \sum_{i=0}^{n} x_i$$

$$\frac{\delta}{\delta \alpha} L^* = n \log(\lambda) - n\psi(\alpha) + \sum_{i=0}^{n} \log(x_i)$$

where $\psi(x) = \frac{\Gamma'(x)}{\Gamma(x)}$ is the `digamma` function (note: its derivate is the `trigamma` function).

The next R function computes the Newton-Raphson iterative method, by first updating the α and then λ already using the most current value for α.

```
# Newton-Raphson gradient descent for gamma MLE
# x       -- the random sample
# theta.0 -- the initial search point
# epsilon -- the convergence objective
gamma.mle.NR <- function(x, theta.0, epsilon=1e-6) {

  n          <- length(x)
  sum.x      <- sum(x)
  sum.log.x <- sum(log(x))

  alpha.i   <- theta.0[1]
  lambda.i  <- theta.0[2]

  repeat {

    # alpha update
    num <- n*log(lambda.i) - n*digamma(alpha.i) + sum.log.x
    den <- -n*trigamma(alpha.i)
    alpha.next <- alpha.i - num / den

    # lambda update
    num <- n*alpha.next / lambda.i - sum.x
    den <- -n^2*alpha.next / lambda.i^2
    lambda.next <- lambda.i - num / den

    # did it converge?
    if (abs(alpha.next - alpha.i)   > epsilon ||
        abs(lambda.next - lambda.i) > epsilon ) {
          alpha.i  <- alpha.next
          lambda.i <- lambda.next
        }
    else
      break
  }

  c(alpha.next, lambda.next)
}

gamma.mle.NR(x, c(1,1))
```

```
## [1] 4.939481 1.967535
```

Exercises

Using the Accept Reject Method

We want to generate samples from pdf $f_X(x) = 20x(1-x)^3, 0 < x < 1.$

Since the pdf is concentrated in a [0,1] interval, let's use $g(x) = 1, 0 < x < 1$

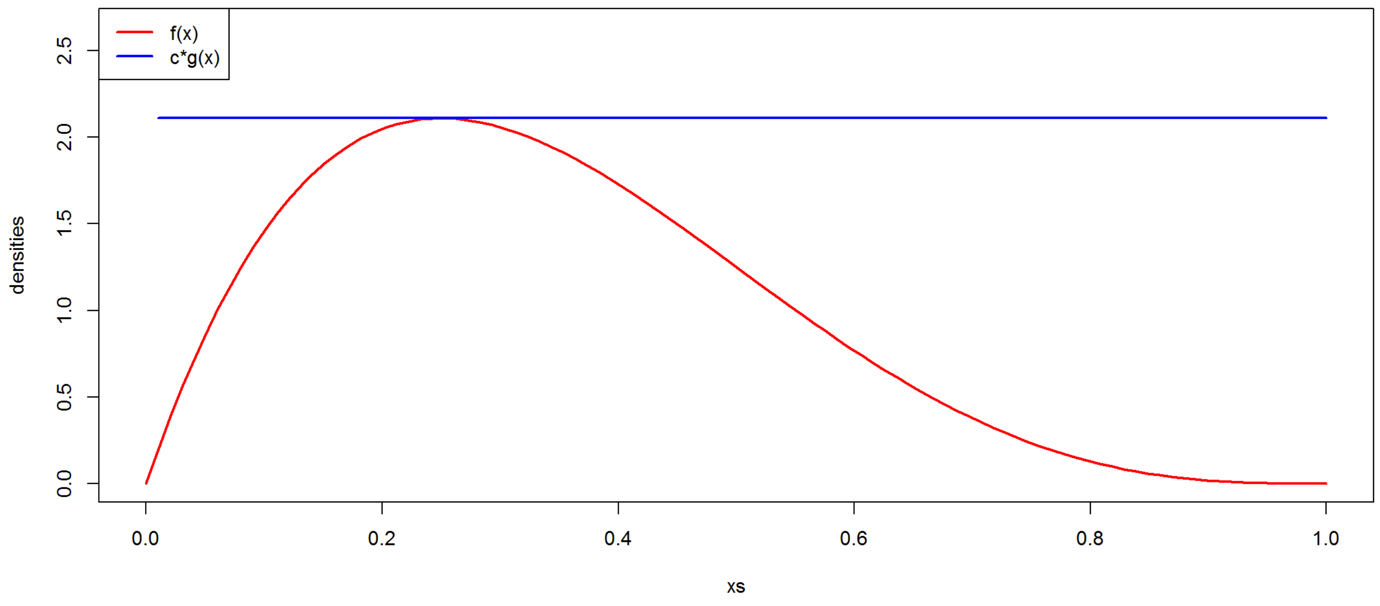
Now we need to compute a proper value for c such that $f(x) \leq c.\, g(x)$. Let's use Calculus to compute the maximum of $f(x)$ in order to get a close value for c.

$$\frac{d}{dx} 20x(1-x)^3 = 0 \iff x = \frac{1}{4}$$

(there is also another solution at $x = 1$ but it is a minimum)

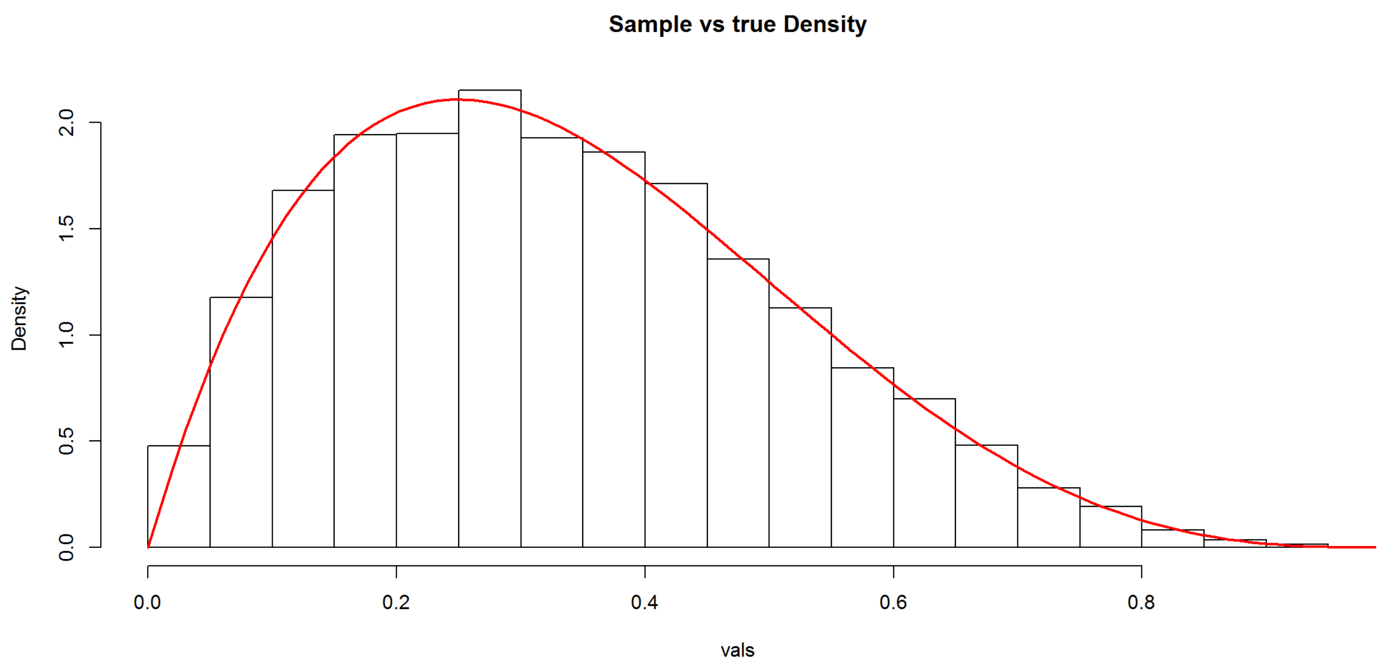
At $x = 1/4$, $f(x) = 135/64$ which becomes the value for c. Just to check how close we got $c.\ g(x)$:

```
f  <- function(x) 20*x*(1-x)^3  # in fact, this is the kernel of a Beta(2,4)
g  <- function(x) x/x
rg <- function(n) runif(n,0,1)
c  <- 135/64

xs <- seq(0, 1, len=100)
plot(xs, f(xs), ylim=c(0,c*1.25), type="l", col="red", lwd=2, ylab="densities")
lines(xs, c*g(xs), type="l", col="blue", lwd=2)
legend("topleft",c("f(x)","c*g(x)"), col=c("red","blue"), lwd=2)
```



Now the sample simulation:

```
vals <- accept.reject(f, c, g, rg, 10000)

# Checking if it went well
hist(vals, breaks=30, freq=FALSE, main="Sample vs true Density")
xs <- seq(0, 1, len=100)
lines(xs, f(xs), col="red", lwd=2)
```



Generate sample using inverse transformation

Given the following pdf:

$$f_X(x) = \frac{1}{20} e^{-(x+10)/20} e^{-e^{-(x+10)/20}}$$

create a function that generates samples from it.

First let's compute the cdf:

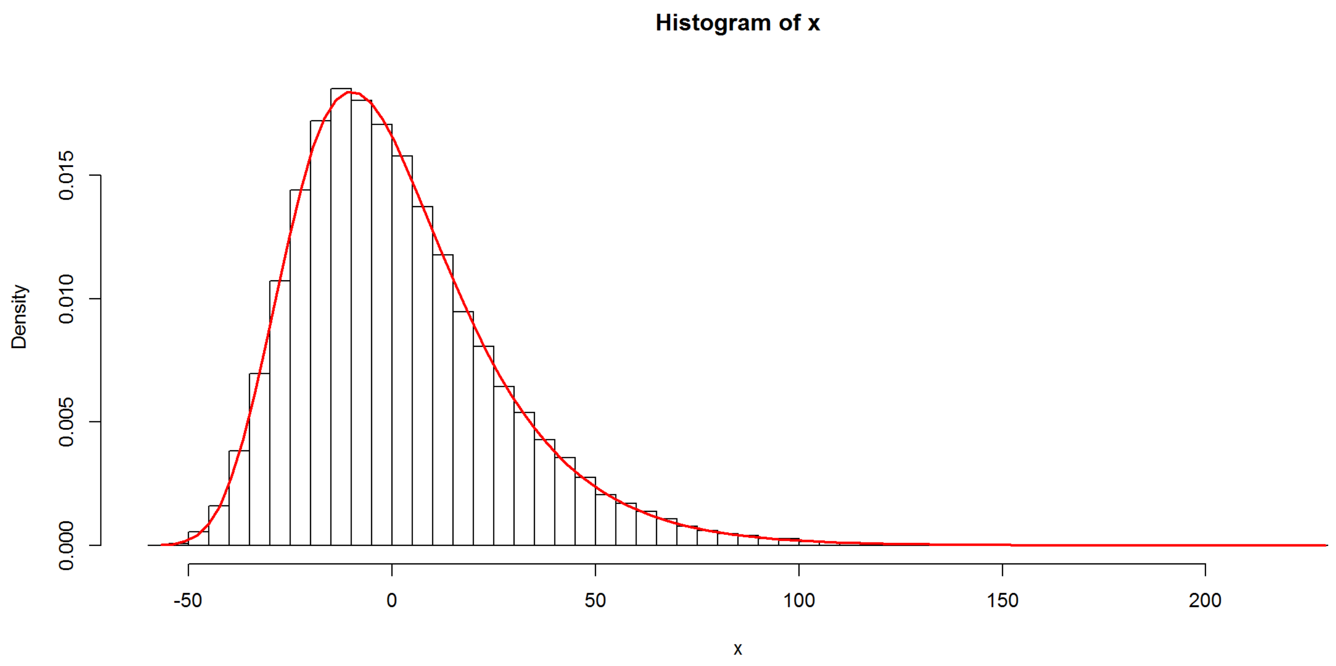
$$F_X(x) = \int_{-\infty}^{x} f_X(x)\, dx = e^{-e^{-\frac{x+10}{20}}}$$

Then it's inverse:

$$F_X^{-1}(u) = -\log(-\log(u)) \times 20 - 10$$

Now we just generate a sample from $U \sim \mathcal{U}(0,1)$ and feed it to the inverse cdf:

```r
inv.F <- function(u) {
  -log(-log(u))*20 - 10
}

U <- runif(1e5)
x <- inv.F(U)


f <- function(x) { # the original pdf
  x <- (x+10)/20
  exp(-x)*exp(-exp(-x))/20
}

hist(x, breaks=50, prob=TRUE)
curve(f(x),min(x),max(x), col="red", lwd=2, add=T)
```



Use of uniform NPAs to compute an integral

We wish to compute the value of

$$I_x = \int_0^x e^{-t^2/2} dt$$

by using $U \sim \mathcal{U}(0,1)$.

Step 1: change integral to appropriate variable, herein $u = t/x$:

$$\begin{aligned} I_x &= \int_{\phi^{-1}(0)}^{\phi^{-1}(x)} e^{-\phi(u)^2/2}\, \phi'(u)\, du \quad & t = \phi(u) = xu, \frac{dt}{du} = \phi'(u) = x \\ &= \int_0^1 \underbrace{e^{-(ux)^2}\, x}_{g(U)} \times 1\, du & f_U(x) = 1, U \sim \mathcal{U}(0,1) \\ &= E_U\left[g(U)\right] & \end{aligned}$$

Step 2: define g, and find the mean of g(uniform random sample):

```
sim.I <- function(x)  {  # this is a closure
  function(u) exp(-(x*u)^2/2) * x
}

g <- sim.I(10) # function used to simulate I_10
U <- runif(1e5)
Ts <- g(U)
mean(Ts)
```

```
## [1] 1.244219
```

We can also approximate I_{10} analytically to check this result:

$$\begin{array}{rcll} I_{10} & = & \int_0^{10} e^{-t^2/2}\,dt & \\ & \approx & \int_0^{\infty} e^{-t^2/2}\,dt & \text{right tail of the } \mathcal{N}(0,1) \text{ kernel} \\ & = & \frac{1}{2}\sqrt{2\pi} & \int_{-\infty}^{\infty} e^{-x^2/2}\,dx = \sqrt{2\pi} \end{array}$$

```
0.5 * sqrt(2*pi)
```

```
## [1] 1.253314
```

For $x = 2$ and with 10^7 samples from $U \sim \mathcal{U}(0,1)$ give a 95% confidence interval:

```
conf.interval <- function(alpha, data.sample) {
  data.size <- length(data.sample)
  data.mean <- mean(data.sample)
  data.sd   <- sd(data.sample)
  z      <- qnorm(1 - alpha/2)

  val <- z*data.sd/sqrt(data.size)
  c(data.mean - val, data.mean + val)
}

I2 <- sim.I(2)
Ts <- I2(runif(1e7))
conf.interval(0.05,Ts)
```

```
## [1] 1.195871 1.196587
```

```
mean(Ts)
```

```
## [1] 1.196229
```

The analytical solution of I_2 is to subtract 0.5 from $F_{\mathcal{N}}(2)$ (to get the area between zero and 2) and to multiply it by $\sqrt{2\pi}$ (the normalizing scalar of the normal pdf):

```
(pnorm(2) - 0.5)*sqrt(2*pi) # the true value
```

```
## [1] 1.196288
```

Use of Importance Sampling to compute an integral

We wish to estimate by simulation the value of

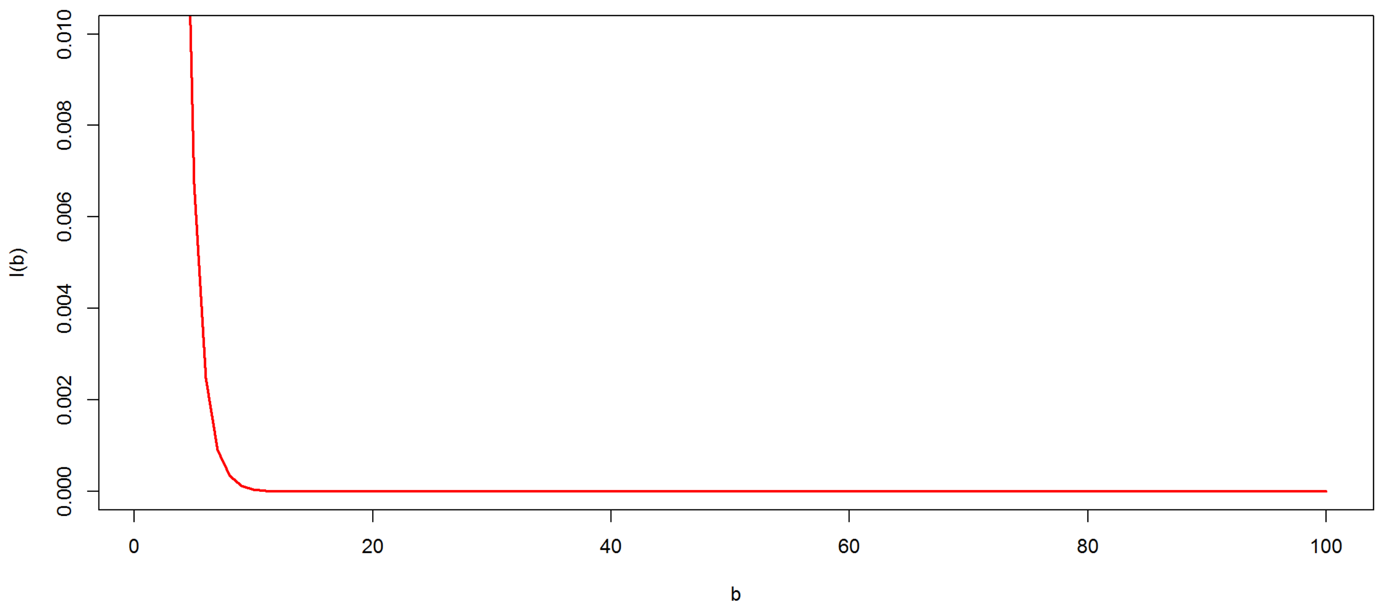
$$I(b) = \int_b^\infty e^{-x} e^{-e^{-x}} \, dx$$

Besides the unbounded limit, this density decreases very fast and R underflows quite easily.

The analytical solution is

$$1 - e^{-e^{-b}}$$

```
x<-1:100

# the target density
f <- function(x){exp(-x)*exp(-exp(-x))}

# the solution of integrate exp(-x)*exp(-exp(-x)) dx, x=b..Inf
solution <- function(b) 1-exp(-exp(-b))
plot(x,solution(x), type="l", col="red", ylim=c(0,0.01), lwd=2, xlab="b", ylab="I(b)")
```

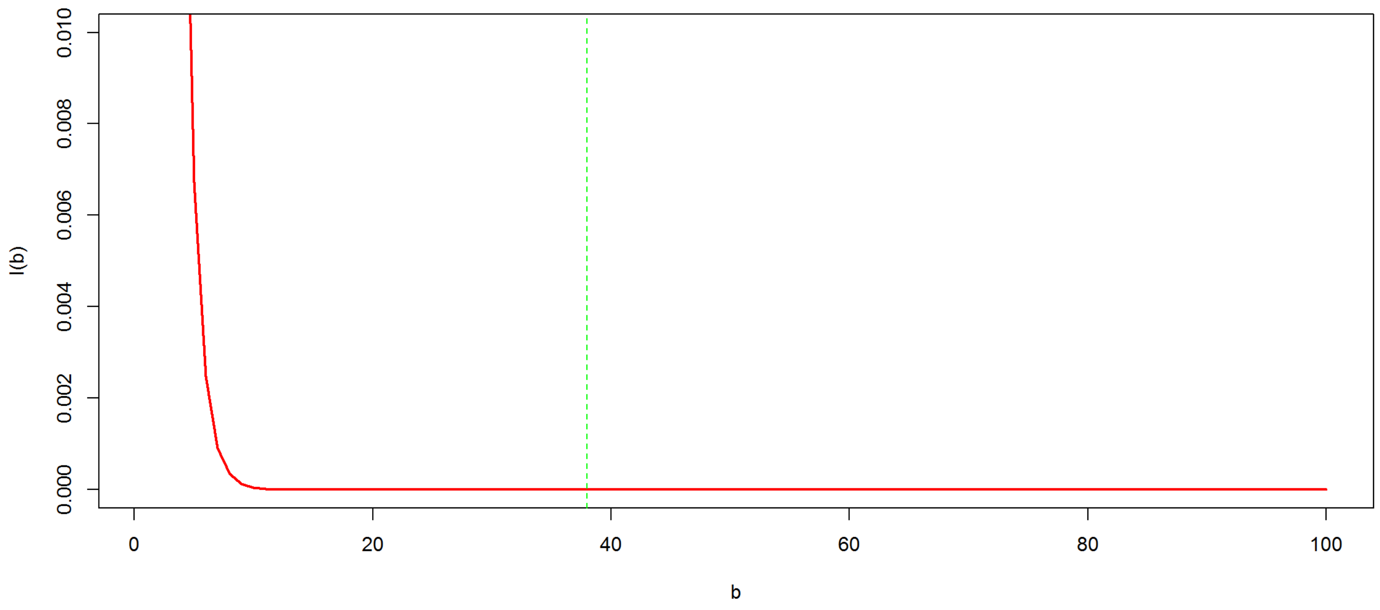


Let’s pick a value for b where R underflows:

```
b <- min(which(solution(x)==0)) # first b where R underflows, so let's use this as an eg
b
```

```
## [1] 38
```

```
plot(x,solution(x), type="l", col="red", ylim=c(0,0.01), lwd=2, xlab="b", ylab="I(b)")
abline(v=b,col="green",lty=2)
```

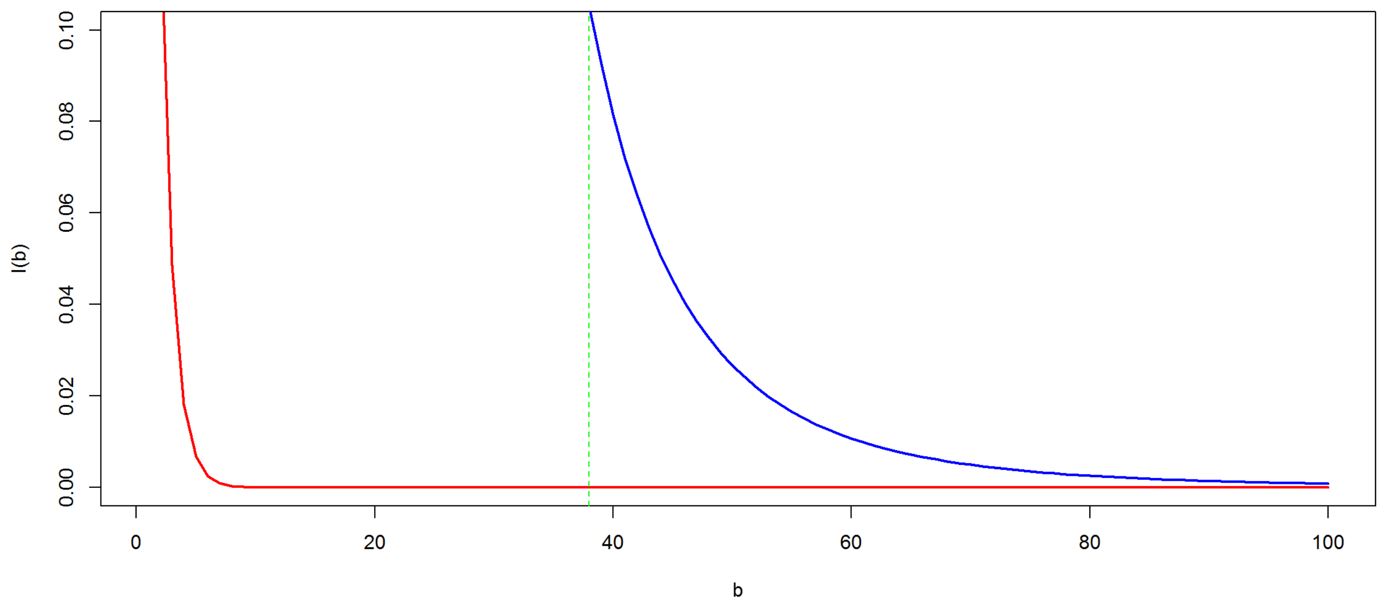


We'll use importance sampling, by first picking a candidate density with heavier tails. Our candidate density will be a pareto distribution

$$f(X|x_m, \alpha) = \frac{\alpha \; x_m^\alpha}{x^{\alpha+1}}$$

where the location x_m is b and the parameter α is, say, 4:

```
alpha <- 4
h <- function(x,xm,alpha) alpha*xm^alpha/x^(alpha+1)

plot(x,solution(x), type="l", col="red", ylim=c(0,0.1), lwd=2, xlab="b", ylab="I(b)")
abline(v=b,col="green",lty=2)
lines(x, h(x, xm=b, alpha=alpha), col="blue", lwd=2)
```



Notice how the candidate function (in blue) is above the target function (in red).

Now we need to generate a random sample from the h pareto pdf. Random samples can be generated using inverse transform sampling (wiki ref (http://en.wikipedia.org/wiki/Pareto_distribution#Random_sample_generation)). In this case

$$X = \frac{x_m}{U^{1/\alpha}} \sim \text{Pareto}(x_m, \alpha)$$

with $U \sim \mathcal{U}(0,1)$

So:

```
set.seed(101)
U <- runif(1000)
X <- b/U^(1/alpha) # alternative: library(VGAM); X <- rpareto(N,b,alpha)

mean(f(X)/h(X,xm=b,alpha=alpha)) # importance sampling
```

```
## [1] 3.11027e-17
```

To check the result, notice that $1 - e^{-e^{-b}} \approx e^{-b}$:

```
exp(-b)
```

```
## [1] 3.139133e-17
```

Using variable change to compute far away tails

We wish to compute $P(X > 20)$ for $X \sim \mathcal{N}(0,1)$.

This tail is so far out that a direct simulation does not work.

We are going to solve this one with a change of variable:

$$\begin{aligned}
P(X>20) &= \int_{20}^{+\infty} f_X(x)dx & f_X(x) = \frac{1}{\sqrt{2\pi}}e^{-x^2/2} \\
&= \int_{\phi^{-1}(20)}^{\phi^{-1}(+\infty)} f_X(\phi(u))\,\phi'(u)\,du & x=\phi(U)=\frac{1}{U},\ \frac{dx}{du}=\phi'(U)=-\frac{1}{U^2}\Rightarrow dx=-\frac{1}{U^2}du \\
&= -\int_0^{1/20}\frac{1}{\sqrt{2\pi}}e^{-1/2u^2}\times\frac{-1}{u^2}\,du & \\
&= \int_0^{1/20}\underbrace{\frac{e^{-1/2u^2}}{20u^2\sqrt{2\pi}}}_{g(U)}\times 20\,du & f_{U(0,1/20)}=20
\end{aligned}$$

So the expression is the expected value $E[g(U)]$, where $U\sim\mathcal{U}(0,1/20)$

```
g <- function(u) exp(-1/(2*u^2))/(20*u^2*sqrt(2*pi))

n <- 1e5
U <- runif(n,0,1/20)
mean( g(U) )
```

```
## [1] 2.482248e-89
```

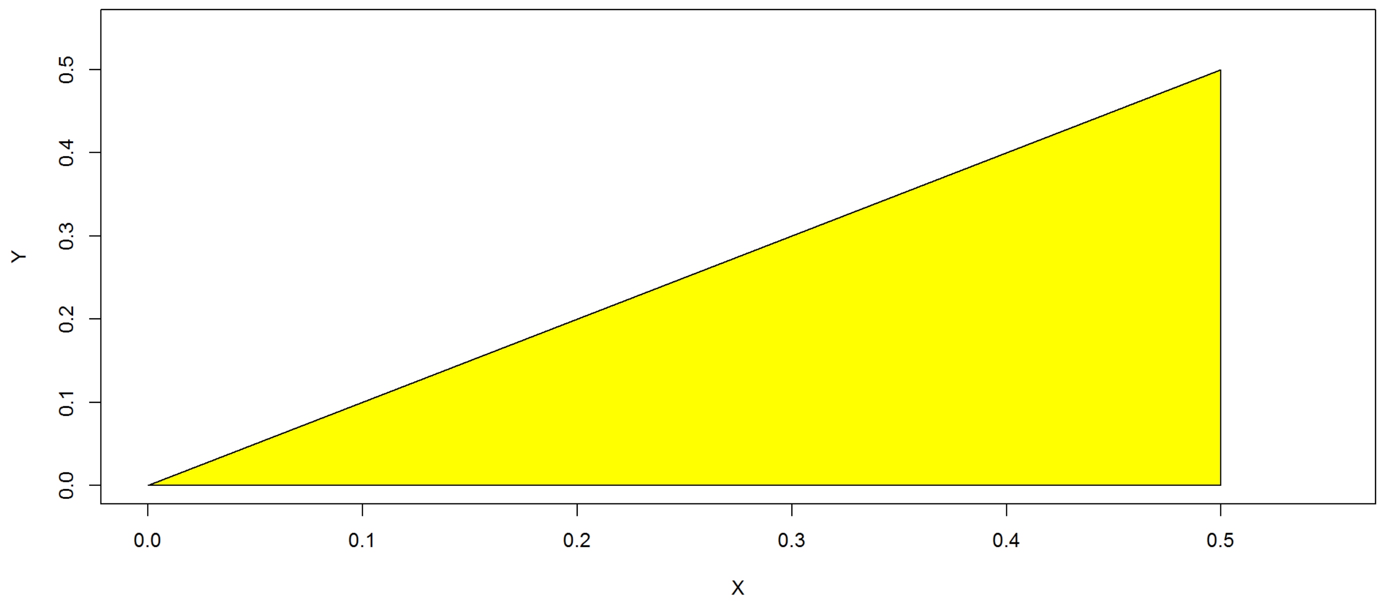
The true value:

```
pnorm(-20)
```

```
## [1] 2.753624e-89
```

Simulation of Expected Values of a Joint Distribution

Let the joint pdf of X and Y be uniform in the triangle area:



Say we wish to compute $E[XY]$. Since the joint pdf $f_{X,Y}(x,y)$ is uniform in the triangle which area is $1/8$, and that $\int\int f_{X,Y}(x,y) = 1$:

$$f_{X,Y}(x,y) = 8$$

Knowing the joint pdf we can compute the expected value:

$$E[XY] = \int_0^{0.5} \int_0^x xy \times f_{X,Y}(x,y)\; dy\; dx = \int_0^{0.5} \int_0^x 8xy\; dy\; dx = \frac{1}{16} = 0.0625$$

But let's assume this integral was too hard to solve. how to simulate its value?

The first attempt would be to generate a random value for X between 0 and 0.5 and then generate Y between 0 and x:

```
n <- 5e5
x <- runif(n, 0, 0.5)
y <- runif(n, 0, x)
mean(x*y)
```

```
## [1] 0.04171467
```

Soon we realise the simulation is not right…

What went wrong? The problem is that we cannot generate X this way.

Let's find the marginal pdf of X, $f_X(x)$:

$$f_X(x) = \int_0^x f_{X,Y}(x,y)\; dy = \int_0^x 8\; dy = 8x$$

That is, the marginal pdf is not constant as assumed in the previous simulation.

To obtain the correct simulation let's use Inverse Transformation. For that we need to generate numbers based on the inverse of the cdf of $f_X(x)$:

$$F_X(x) = \int_0^x f_X(t)\; dt = \int_0^x 8t\; dt = 4x^2$$

So the inverse function is:

$$F_X^{-1}(x) = \frac{1}{2}x^{1/2}$$

that we finally use in the correct simulation:

```
set.seed(321)
inv.cdf <- function(x) .5*x^(.5)  # função inversa da cdf

n <- 5e5
x <- inv.cdf(runif(n, 0, 1))      # X = F^{-1}(U)
y <- runif(n, 0, x)
mean(x*y)
```

```
## [1] 0.06256497
```

And voilá!

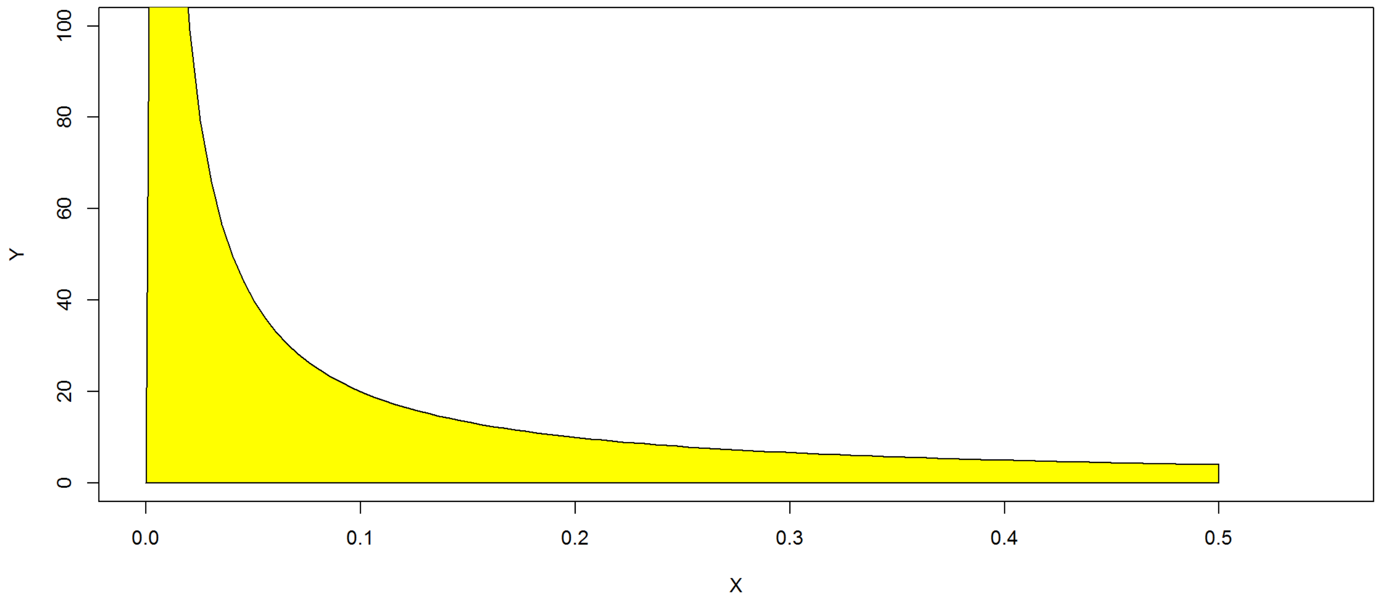
To conclude let's show what was the model simulated by our first attempt, where we obtain $E[XY] = 0.04162$.

$$X \sim U(0, 0.5)$$
$$Y|X \sim U(0, x)$$

The joint pdf would be:

$$f_{X,Y}(x,y) = f_X(x) f_{Y|X}(y|x) = 2 \times \frac{1}{x} = \frac{2}{x}$$

Here is the respective graphic:



We can now find the expected value of $X \times Y$:

$$E[XY] = \int_0^{0.5} \int_0^x xy \times f_{X,Y}(x,y)\, dy\, dx = \int_0^{0.5} \int_0^x xy \frac{2}{x}\, dy\, dx = \frac{1}{24} = 0.04166667$$

The first simulation computed the theoretical value of this second model!

Simulation of Expected Values of a Joint Distribution II

Program an algorithm that generates pairs from the joint pdf

$$f_{X,Y}(x,y) = e^{-x} e^{-y} e^{-e^{-y}}, y < x, x \in \mathcal{R}$$

We know that $f_{X,Y}(x,y) = f_{Y|X}(y|x) f_X(x)$. First let's compute the marginal pdf:

$$\begin{array}{rcll} f_X(x) & = & \int_{-\infty}^{x} f_{X,Y}(x,y)\,dy & \\ & = & e^{-x}\int_{-\infty}^{x} e^{-y}e^{-e^{-y}}\,dy & \\ & = & e^{-x}e^{-e^{-y}}\Big|_{-\infty}^{x} & \int_a^b \phi'(x)e^{\phi(x)}dx = e^{\phi(x)}\Big|_a^b \\ & = & e^{-x}e^{-e^{-x}} & \end{array}$$

The respective cdf is $F_X(x) = \int_{-\infty}^{x} e^{-t}e^{-e^{-t}}dt = e^{-e^{-x}}$.

And its inverse (we'll need it for the inverse transformation method) is $F^{-1}(u) = -\log(-\log(u))$

Now for the conditional pdf:

$$\begin{array}{rcl} f_{Y|X}(y|x) & = & \frac{f_{X,Y}(x,y)}{f_X(x)} \\ & = & \frac{e^{-x}e^{-y}e^{-e^{-y}}}{e^{-x}e^{-e^{-x}}} \\ & = & e^{-y}e^{e^{-x}}e^{-e^{-y}} \end{array}$$

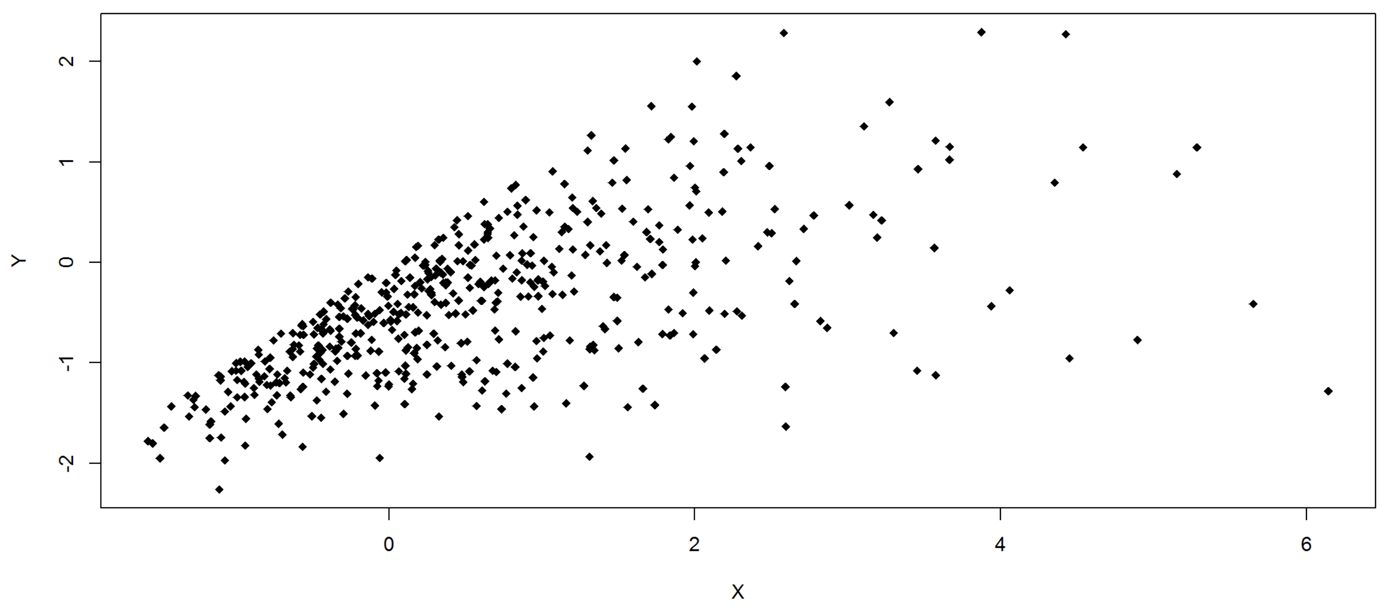
And the cdf:

$$F_{Y|X}(y|x) = \int_{-\infty}^{y} f_{Y|X}(y|x)\,dy = \ldots = e^{e^{-x}}e^{-e^{-y}}$$

having as inverse $F^{-1}(u|x) = -\log(-\log(e^{-e^{-x}} \times u))$

Finally, the R code:

```r
# inverse of F_X(x)
inv.f_X <- function(u) -log(-log(u))

# inverse of F_Y|X(y|x)
inv.f_Y_X <- function(u,x) -log(-log(exp(-exp(-x))*u))

n <- 500 # number of random pairs
X <- inv.f_X(runif(n,0,1))
Y <- inv.f_Y_X(runif(n,0,1),X)

plot(X,Y,pch=18)
```


Y
X
2
1
0
-1
-2
0
2
4
6