Computer Practical 1 Outline Solutions
Simulation Basics to Monte Carlo Methods

1. *Transformation Methods*

Recall the Box-Muller method which transforms pairs of uniformly-distributed random variables to obtain a pair of independent standard normal random variates. If

$$U_1, U_2 \overset{\text{iid}}{\sim} \mathsf{U}[0,1]$$

and

$$X_1 = \sqrt{-2\log(U_1)} \cdot \cos(2\pi U_2)$$
$$X_2 = \sqrt{-2\log(U_1)} \cdot \sin(2\pi U_2)$$

then $X_1, X_2 \overset{\text{iid}}{\sim} \mathsf{N}(0,1)$.

(a) Write a function which takes as arguments two vectors $(\mathbf{U}_1, \mathbf{U}_2)$ and returns the two vectors $(\mathbf{X}_1, \mathbf{X}_2)$ obtained by applying the Box-Muller transform elementwise.

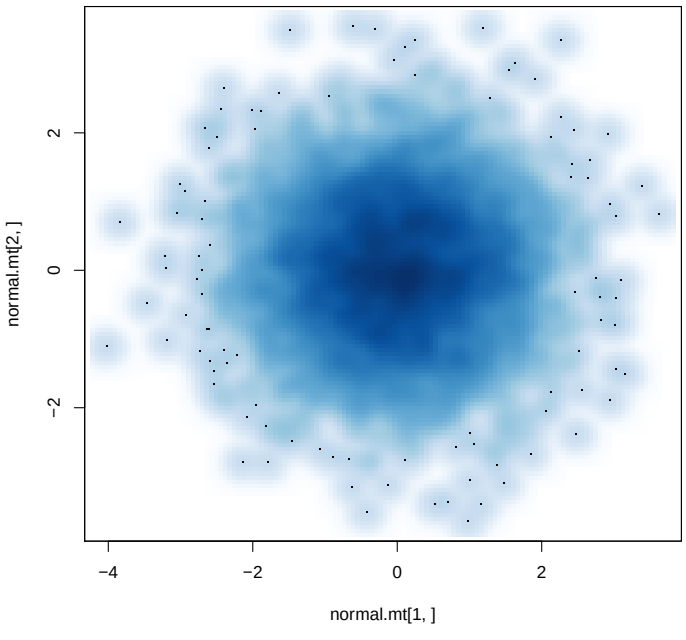
```
box.muller <- function(U1,U2) {
    X1 <- sqrt(-2*log(U1)) * cos(2*pi*U2)
    X2 <- sqrt(-2*log(U1)) * sin(2*pi*U2)
    return(rbind(X1,X2))
}
```

(b) The R function `runif` provides access to a PRNG. The type of PRNG can be identified using the `RNGkind` command; by default it will be a Mersenne-Twister (`http://en.wikipedia.org/wiki/Mersenne_twister`). Generate 10,000 $\mathsf{U}[0,1]$ random variables using this function, and transform this vector to two vectors each of 5,000 normal random variates.

```
unif.mt <-runif(10000)
normal.mt <-box.muller(unif.mt[seq(1,10000,2)],unif.mt[seq(2,10000,2)])
```

(c) Check that the result from (b) is plausibly distributed as pairs of independent, standard normal random variables, by creating a scatter plot of your data.

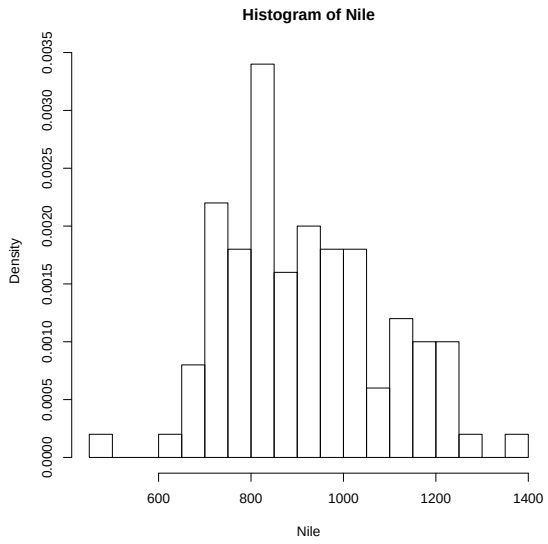
```
smoothScatter(normal.mt[1,],normal.mt[2,])
```



2. *The Bootstrap:* This question can be answered in two ways. The more direct (and perhaps more informative, if you have the time to do so and the inclination to implement a bootstrap algorithm from scratch) is to use the `sample` function to obtain bootstrap replicates, and to compute the required confidence intervals by direct means (hint: set `replace=TRUE`). More pragmatically, the `boot` library provides a function `boot` to obtain bootstrap samples and another, `boot.ci`, to provide various bootstrap confidence intervals.

   (a) The `Nile` dataset shows the annual flow rate of the Nile river. Use a histogram or other visualisation to briefly explore this data.

```
hist(Nile,n=20,freq=FALSE)
```



There's apparently some *skewness* in the data. We'll consider it as a time series later...

   (b) What are the mean and median annual flow of the Nile over the period recorded in this data set?

```
mean(Nile)
```
919.35
```
median(Nile)
```
893.5
Further evidence of positive skewness.

   (c) Treating the data as a simple random sample, appeal to asymptotic normality to construct a confidence interval for the mean annual flow of the Nile.

The lazy solution:
```
sigma.hat <-sqrt(var(Nile)/length(Nile))
c(qnorm(0.025),qnorm(0.975))*sigma.hat + mean(Nile)
```
Leading to: [886.182, 952.518]

   (d) Using the `boot::boot` function to obtain the sample and `boot::boot.ci` to obtain confidence intervals from that sample, or otherwise, obtain a *bootstrap percentile interval* for both the mean and median of the Nile's annual flow. For the median you may also wish to obtain the interval obtained by an optimistic appeal to asymptotic normality combined with a bootstrap estimate of the variance (`boot::boot.ci` will provide this).

   Note the following: `boot::boot` does the actual bootstrap resampling. It needs a function of two arguments to compute the statistic for each bootstrap sample: the first contains the *original* data and the second the index of the values included in a particular bootstrap resampling.

Directly:
Generate the bootstrap samples:
```
nb <- c()
for(i in 1:10000) {nb[i] <-mean(sample(Nile,length(Nile),replace=TRUE))}
```
Compute the appropriate interval:
```
quantile(nb,c(0.025,0.975))
```

Using `boot`:
Load the relevant library:

```
library(boot)
```

Construct an appropriate evaluation function:

```
f.mean <- function (x,i) { mean(x[i]) }
```

Obtain the bootstrap resamples:

```
boot.mean <-boot::boot(Nile,f.mean,9999,stype='i')
```

Obtain the bootstrap percentile confidence intervals:

```
bpi.mean <-boot::boot.ci(boot.mean,type='perc')
```

On this occasion leading to: (886.7, 952.7)

Moving on to the median, we repeat the same strategy:
Either:

```
nb2 <-c()
for(i in 1:10000) {nb2[i] <- median(sample(Nile,length(Nile),replace=TRUE))}
quantile(nb2,c(0.025,0.975))
```

or:

```
f.median <- function (x,i) {median(x[i])}
boot.median <-boot::boot(Nile,f.median,9999,stype='i')
bpi.median <-boot::boot.ci(boot.median,type='perc')
```

on this occasion leading to: (845, 940),

```
boot::boot.ci(boot.median,type='norm')
```

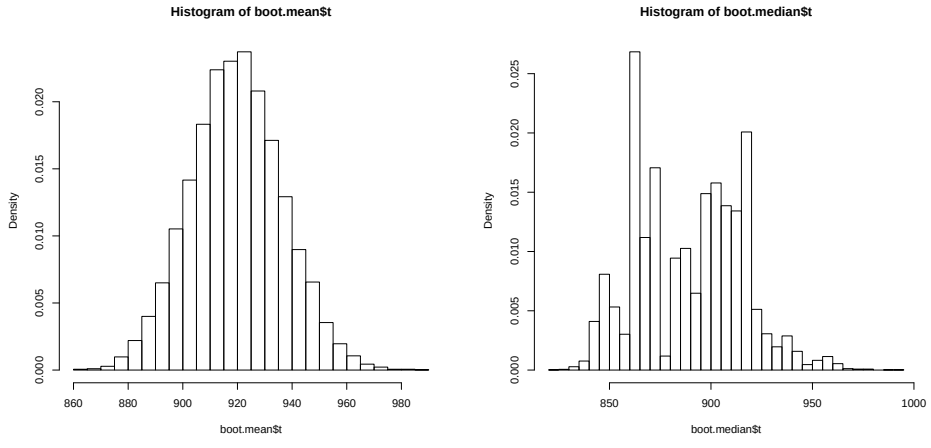
while the normal approximation leads to (846.2, 947.9).

(e) Are there any interesting qualitative differences between the various confidence intervals? How does this relate to the data?

```
print(c(mean(Nile)-bpi.mean$percent[4],bpi.mean$percent[5]-mean(Nile)))
# [1] 32.71 33.29
print(c(median(Nile)-bpi.median$percent[4],bpi.median$percent[5]-median(Nile)))
# [1] 48.5 46.5
```

Interesting is a relative term and you might identify other things, but one notable observation is the slight asymmetry of the BPIs for both the mean and median. This reflects the fact that even the mean of a small sample of non-normal random variables is not normally distributed. The BPIs for the median are also wider, suggesting that the bootstrap distribution for the median is more variable. Closer inspection reveals substantial differences between the two bootstrap distributions:

```
hist(boot.mean$t,breaks=30,freq=FALSE)
hist(boot.median$t,breaks=30,freq=FALSE)
```



(f) Are your findings stable? If you repeat the bootstrap sampling do you recover similar behaviour?

This depends on the sample size you use; in the example above we chose $B = 9999$. For this choice there is a reasonable degree of stability, as shown by for example repeating the entire procedure several times and inspecting the variability of our estimates:

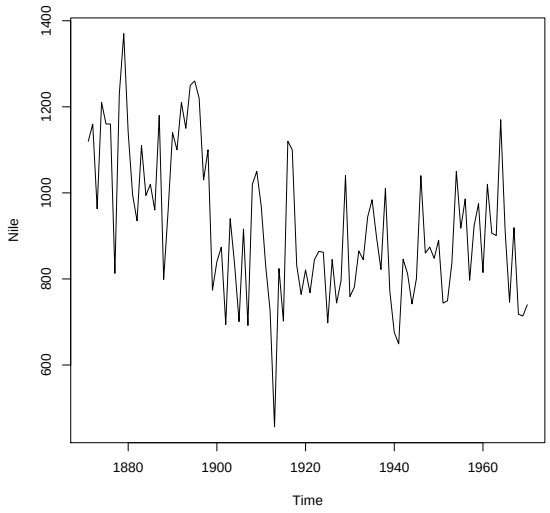
```
bpi.mean.replicates <-matrix(nrow=100,ncol=2)
bpi.median.replicates <-matrix(nrow=100,ncol=2)
for (N in 1:100){
boot.mean <-boot::boot(Nile,f.mean,9999,stype='i')
bpi.mean.replicates[N,] <-boot.ci(boot.mean,type='perc')$percent[4:5]
boot.median <-boot::boot(Nile,f.median,9999,stype='i')
bpi.median.replicates[N,] <-boot.ci(boot.median,type='perc')$percent[4:5]
}
```

The replicates seem to be reasonably stable. This is a convenient point to emphasize that one should *always* attempt to assess the sensitivity of computational procedures to their design parameters and other factors which may influence their performance, before we rely on their output.

(g) Are there any reasons to doubt the accuracy of these confidence intervals?

Absolutely. The computational method is irrelevant—the data is a time series and plotting it shows a clear departure from independence, so any method predicated on this assumption is untrustworthy.

`ts.plot(Nile)`:



3. *Convergence of Sample Approximations*

(a) The `stats::ecdf` and `stats::plot.ecdf` functions compute and plot empirical distribution functions from a provided sample.

(i) Show plots of the empirical distribution function of samples of a variety of sizes ranging from 10 to 10,000 from a $\mathsf{U}[0,1]$ distribution. Add to your plots the distribution function of the $\mathsf{U}[0,1]$ distribution.

The following code produces the type of plots required, which illustrate the Glivenko-Cantelli Theorem.

```
ru  <- runif(10000)
F10  <- ecdf(ru[1:10])
F50  <- ecdf(ru[1:50])
F500 <- ecdf(ru[1:500])
F3000 <- ecdf(ru[1:3000])
F10000 <- ecdf(ru)

seq <- seq(0,1,1/1000)
plot.ecdf(F10,main='Sample␣Size␣n=10')
lines(seq,punif(seq),col='red')

plot.ecdf(F50,main='Sample␣Size␣n=50')
```

```
lines(seq,punif(seq),col='red')

plot.ecdf(F500,main='Sample Size n=500')
lines(seq,punif(seq),col='red')

plot.ecdf(F3000,main='Sample Size n=3000')
lines(seq,punif(seq),col='red')

plot.ecdf(F10000,main='Sample Size n=10000')
lines(seq,punif(seq),col='red')
```

(ii) Repeat part *(i)* with a standard normal distribution.

Slight modifications of the above code suffice...

(iii) Repeat part *(i)* with a Cauchy distribution.

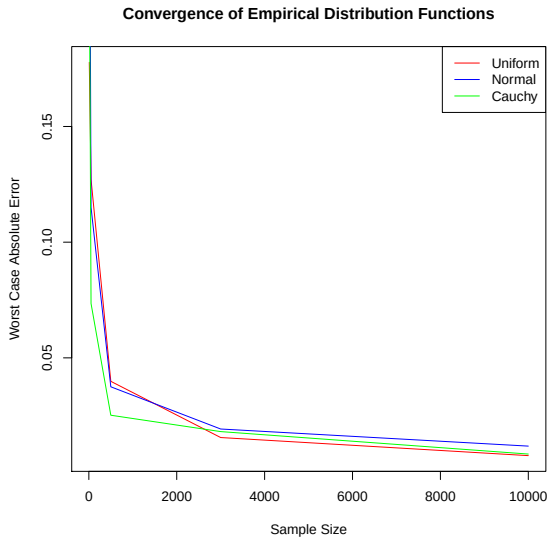
Slight modifications of the above code suffice...

(b) For each of the three distributions considered in the previous part, determine $\sup_x |\hat{F}_n(x) - F(x)|$ for each n considered (for simplicity, consider only the sup over the sampled values of x) and plot these quantities against n. Do you notice anything interesting?

Having implemented the 3 slight variants of the code for part *(a) (i)* this is reasonably straightforward:

```
eu <- c(10, max(abs(F10(ru[1:10]) - punif(ru[1:10]))))
eu <- rbind(eu, c(50, max(abs(F50(ru[1:50]) - punif(ru[1:50])))))
eu <- rbind(eu, c(500, max(abs(F500(ru[1:500]) - punif(ru[1:500])))))
eu <- rbind(eu, c(3000, max(abs(F3000(ru[1:3000]) - punif(ru[1:3000])))))
eu <- rbind(eu, c(10000, max(abs(F10000(ru[1:10000]) - punif(ru[1:10000])))))

en <- c(10, max(abs(Fn10(rn[1:10]) - pnorm(rn[1:10]))))
en <- rbind(en, c(50, max(abs(Fn50(rn[1:50]) - pnorm(rn[1:50])))))
en <- rbind(en, c(500, max(abs(Fn500(rn[1:500]) - pnorm(rn[1:500])))))
en <- rbind(en, c(3000, max(abs(Fn3000(rn[1:3000]) - pnorm(rn[1:3000])))))
en <- rbind(en, c(10000, max(abs(Fn10000(rn[1:10000]) - pnorm(rn[1:10000])))))

ec <- c(10, max(abs(Fc10(rc[1:10]) - pcauchy(rc[1:10]))))
ec <- rbind(ec, c(50, max(abs(Fc50(rc[1:50]) - pcauchy(rc[1:50])))))
ec <- rbind(ec, c(500, max(abs(Fc500(rc[1:500]) - pcauchy(rc[1:500])))))
ec <- rbind(ec, c(3000, max(abs(Fc3000(rc[1:3000]) - pcauchy(rc[1:3000])))))
ec <- rbind(ec, c(10000, max(abs(Fc10000(rc[1:10000]) - pcauchy(rc[1:10000])))))

plot(eu,type='l',col='red',main='Convergence of Empirical Distribution Functions'
    ,xlab='Sample Size', ylab='Worst Case Absolute Error')
lines(en,col='blue')
lines(ec,col='green')
legend('topright',c('Uniform','Normal','Cauchy'), lty=1, col=c('red','blue','
    green'))
```

And leads to the following graph:



The striking feature of which is that performance is just as good for the Cauchy distribution as for the uniform or the normal; indeed, as the distribution function maps all of these things onto $[0, 1]$ this is well behaved even for distributions which often misbehave. Whether the resulting distributional approximation is good enough for particular tasks, of course, depends on what those tasks are.

4. The *Kumaraswamy distribution*[1] with parameters $a > 0$ and $b > 0$ describes a random variable with range $[0, 1]$. It has density:

$$f_X(x; a, b) = abx^{a-1}(1 - x^a)^{b-1} \text{ for } x \in [0, 1].$$

(a) Transformation Method

(i) Compute the distribution function and its inverse.

[1] Kumaraswamy, P. (1980). "A generalized probability density function for double-bounded random processes". Journal of Hydrology 46 (1-2): 79–88.

The cumulative distribution function is defined as:

$$F_X(x) = \int_{-\infty}^{x} f_X(u)du.$$

The density is zero outside the unit interval, so

$$F_X(x) = \begin{cases} 0 & x < 0 \\ \int_0^x f_X(u)du & x \in [0,1] \\ 1 & x > 1. \end{cases}$$

and for $x \in [0,1]$ we obtain:

$$F_X(x) = \int_0^x abu^{a-1}(1-u^a)^{b-1}du.$$

Noting that $-au^{a-1}$ is the derivative of $1-u^a$, it is natural to try the change of variable $z = 1-u^a$ which leads to:

$$F_X(x) = -\int_1^{1-x^a} abu^{a-1}z^{b-1}\frac{dz}{au^{a-1}} = b\int_{1-x^a}^{1} z^{b-1}dz = b\left[z^b/b\right]_{1-x^a}^{1} = 1-(1-x^a)^b.$$

$F_X(x)$ is a strictly increasing function on $[0,1]$ and so an inverse exists; noting that $u = F_X(x) \Leftrightarrow F_X^{-1}(u) = x$ we have that:

$$\begin{aligned} u &= 1-(1-x^a)^b, \\ 1-x^a &= (1-u)^{1/b}, \\ \left(1-(1-u)^{1/b}\right)^{1/a} &= x \qquad \Rightarrow F_X^{-1}(u) = \left(1-(1-u)^{1/b}\right)^{1/a}. \end{aligned}$$

(ii) Implement a function which uses the inverse transform method to return a specified number of samples from this distribution with specified a and b parameters, using `runif` as a source of uniform random variables.

A Direct solution would be to write simply:

```
rkumaraswamy <- function(n=1,a=1,b=1) {
    u <- runif(n)
    x <- (1-(1-u)^(1/b))^(1/a)
}
```

But $1-U \stackrel{\mathcal{D}}{=} U$ and so we can simplify the final line to
`x <- (1-u^(1/b))^(1/a)`

(iii) Implement a function which uses the inverse transform method to return a specified number of samples from this distribution with specified a and b parameters, using `runif` as a source of uniform random variables.

(iv) Use the `system.time`[2] function to determine how long your implementation takes to obtain a sample of size 100,000 from the Kumaraswamy distribution with $a = b = 2$. (You may wish to run it a few times and take the average.)

```
system.time(rkumaraswamy(100000,2,2))
```
There's some variability in runtime, ranging from about 6 ms to 9 ms, with an average over 100 runs of 6.5 ms using a desktop Mac.
Note: The `rprof` package can be extremely helpful when optimizing R code.

(b) Rejection Sampling

[2] `system.time` has poor rounding properties on Windows machines. A cheap alternative in Windows is to use `Sys.time` instead, which just records the current date and time. You can take the difference of the return value of this function at the beginning and the end of the period to be timed. The correspondence between the two timing methods is not fantastic. A better alternative is to use the `microbenchmark` package.

(i) Implement a function which uses rejection sampling to return a specified number of samples from this distribution with specified a and b parameters, using `runif` as its sole source of randomness (use a $\mathsf{U}[0,1]$ proposal distribution).

We need to identify a constant M such that

$$\sup_{x\in[0,1]} f(x)/g(x) \leq M,$$

with $g(x) = \mathbb{I}_{[0,1]}(x)$.
Differentiating the ratio of densities with respect to x we find:

$$\begin{aligned}\frac{\partial}{\partial x}\frac{f(x)}{g(x)} &= \frac{\partial}{\partial x}\frac{abx^{a-1}(1-x^a)^{b-1}}{1} \\ &= ab\left[(a-1)x^{a-2}(1-x^a)^{b-1} + x^{a-1}(b-1)(1-x^a)^{b-2}(-ax^{a-1})\right] \\ &= ab\left[x^{a-2}(1-x^a)^{b-2}\left\{(a-1)(1-x^a) - a(b-1)x^a\right\},\right]\end{aligned}$$

which equals zero at $x^\star$, where $x^\star$ is the x that satisfies:

$$[(a-1)+a(b-1)]x^a = a-1 \qquad\qquad \Rightarrow x^\star = \left(\frac{a-1}{ab-1}\right)^{1/a},$$

or at the boundaries $x = 0, 1$. As this is the only stationary point in the interior of the interval and the density is positive here, zero at 0 and 1 and continuous, it must be a maximum.
Having identified M, it's straightforward to implement the rejection sampler, although we might want to do something more efficient if we were to do this seriously — R is not good at loops.

```
dkumaraswamy <- function(x,a,b) {
    a*b * (x ^ (a-1)) * (1 - x^a) ^ (b-1)
}

rrejectk <- function(n=1,a=1,b=1) {
    naccept <- 0
    x <- c()
    M <- dkumaraswamy(((a-1)/(a*b-1))^(1/a), a, b) + 1E-8
    while(naccept < n) {
        x[naccept+1] <- runif(1)
        u <- runif(1)
        if(u < dkumaraswamy(x[naccept+1],a,b) / M) { naccept <- naccept + 1}
    }
    return(x)
}
```

(ii) Use the `system.time` function to determine how long your implementation takes to obtain a sample of size 100,000 from the Kumaraswamy distribution with $a = b = 2$.

```
system.time(rrejectk(100000,2,2))
```

The simple implementation above takes around 0.66s to produce 100,000 samples; around 100 times longer than the inversion sampler. . . but this is largely due to inefficient implementation: R is very slow when handling loops.
In practice, a better strategy would be to 'guess' how many proposals will be required and to operate vectorially, generating another block of proposals if it turns out that we don't have enough. The following solution uses a recursive approach combined with this strategy and takes around 0.03 s on my desktop, and so it takes around 5 times longer than the inversion sampler:

```
rrejectk2 <- function(n=1, a=1, b=1) {
    M <- dkumaraswamy(((a-1)/(a*b-1))^(1/a), a, b) + 1E-8
    x <- runif(n*ceiling(1.1*M))
    u <- runif(n*ceiling(1.1*M))
    xa <- x[u < (dkumaraswamy(x,a,b) / M)]
    if(length(xa) > n) {
        return(xa[1:n])
    }
    return(c(xa,rrjectk2(n-length(xa),a,b)))
}
```

(iii) Modify your function to record how many proposals are required to obtain this sample; how do the empirical results compare with the theoretical acceptance rate?

```
rrejectk.count <- function(n=1,a=1,b=1) {
    naccept <- 0
    npropose <- n
    x <- c()
    M <- dkumaraswamy(((a-1)/(a*b-1))^(1/a), a, b) + 1E-8
    while(naccept < n) {
        x[naccept+1] <- runif(1)
        u <- runif(1)
        if(u < dkumaraswamy(x[naccept+1],a,b) / M)
            { naccept <- naccept + 1}
        else
            { npropose <- npropose + 1}
    }
    return(list(X=x,np = npropose))
}
```

Such a modification is straightforward. With one run I found that I needed to make 153,901 proposals in order to obtain 100,000 accepted samples.
Theoretically, one requires a NB $(100,000, 1/M)$ number of samples (we're looking for 100,000 successes in independent Bernoulli trials with success probability $1/M$), which has mean $100000M = 153960$. This is consistent with our observation (checking the variance of the negative binomial distribution confirms this).

(c) Importance Sampling

(i) Implement a function which uses a uniform proposal to return a weighted sample (i.e. both the sampled values and the associated importance weights) of size 100,000 which is properly weighted to target the Kumaraswamy distribution of parameters $a = b = 2$. Use the normalising constants which you know in this case.

```
rimportk <- function(n=1,a=1,b=1) {
    x <- runif(n)
    w <- dkumaraswamy(x, a, b)
    return(list(x=x,w=w))
}
```

(ii) Use the `system.time` function to determine how long your implementation takes to obtain a sample of size 100,000 targeting this distribution.

```
system.time(rimportk(100000,2,2))
```
This implementation took between 6 ms and 12 ms on my desktop; mean 6.6ms over 100 runs.

(iii) Produce a variant of your function which returns the self-normalised version of your weighted sample (this is easy; just divide the importance weights by their mean value after producing the original weighted sample).

```
rimportk2 <- function(n=1,a=1,b=1) {
    x <- runif(n)
    w <- dkumaraswamy(x, a, b)
    return(list(x=x,w=w/mean(w)))
}
```

(iv) Use the `system.time` function to determine how long your implementation takes to obtain a sample of size 100,000 targeting this distribution.

```
system.time(rimportk2(100000,2,2))
```
Over 100 runs this code required 6–10 ms; mean 6.9ms.

(d) Comparison

(i) The *inverse transformation* and *rejection* algorithms both produce iid samples from the target distribution and so the only thing which distinguishes them is the time it takes to run the two algorithms. Which is preferable?

The inverse transformation method is better than the efficient version of the rejection sampler and *much* better than the inefficient implementation.

How many uniform random variables do the two algorithms require to produce the samples (this is, of course, a random quantity with rejection sampling, but the average value is a good point of comparison) and how does this relate to their relative computational costs?

We need two random variables per trial and around 100,000 M trials to obtain 100,000 acceptances by rejection; but just one random variable per acceptance with the inversion sampling algorithm, which suggests a factor of 3.08 between the two methods.
This is of the same order of magnitude as the difference in timing between the inversion sampler and the efficient implementation of the rejection sampler.
In contrast, the inefficient rejection sampler is orders of magnitude slower. This illustrates the need to be careful with "computational overheads" and to be aware of limitations and inefficiencies of any particular platform when implementing these algorithms.

To compare the importance sampling estimators with other algorithms it is necessary to have some idea of how well they work. To this end, use all four algorithms to estimate the expectation of X when $X \sim f_X(\cdot; a, b)$ using samples of size 100,000. (You might want to make the sample smaller if your implementation takes too long to produce a result with this sample size).

The algorithms which use iid samples from the target and the simple importance sampling scheme are unbiased and so we can use their variance as a measure of how well they perform. Noting that their variance scales with the reciprocal of sample size, an appropriate figure of merit is the product of the estimator variance and computational cost (cheaper schemes could be run for longer to reduce their variance without requiring any further computing).

We're interested here not in the variance of the target distribution—which we could easily estimate from a single sample—but in the *estimator variance*: a characterisation of the variability between repeated runs of our algorithms. This Monte Carlo variance tells us how much variability we introduce into the estimate by using Monte Carlo instead of the exact population mean. To characterise it, run each of your algorithms a large number of times, 100, say, obtain an estimate from each run and compute the sample variance of the collection of estimates you obtain.

How do the algorithms compare?

Algorithm	**Mean**	**Var** / 10^{-7}	**Cost** /ms	**Var** × **Cost**
Transformation	0.5334	5.2	7.1	36.7
Rejection (2)	0.5333	4.2	20.1	85.9
Importance	0.5332	11.4	6.5	74.7

In this case the transformation scheme still wins out. It seems that, on my Mac at least, there is a fixed overhead of around 6 ms for each scheme below which even the most otherwise computationally efficient schemes cannot go. On other machines where this does not occur, you might get a different result in which the tradeoff of higher variance of importance sampling in return for better computational savings is worth making.

(ii) The self-normalized importance sampling scheme is biased and this further complicates the comparison.

First obtain its variance as you did for the other algorithms.

I obtained 3.3×10^{-7} which is comfortably the lowest amongst all the estimators and remains so even after factoring in computational cost.

Now estimate its bias: what is the average difference between the estimates you obtained with this algorithm and the average result obtained with one of the unbiased schemes? The mean squared error can be expressed as the sum of variance and bias squared. Perform a comparison of the algorithms which considers MSE as well as computational cost.

The simplest approach is to note that the bias of this estimate is tiny; I get its expectation to be 0.5333, which agrees to four significant figures with those obtained by the other algorithms. Consequently, the variance and MSE are essentially the same.

5. One very common use of Monte Carlo methods is to perform Bayesian inference.

Consider a scenario in which you wish to estimate an unknown probability given n realisations of a Bernoulli random variable of success probability p. You can view your likelihood as being $\mathsf{Bin}(n, p)$. The traditional way to proceed is to impose a Beta prior on p and to exploit conjugacy. Consider instead a setting in which you wish to use a Kumaraswamy distribution with $a = 3$ and $b = 1$ as a prior distribution (perhaps you're dealing with a problem related to hydrology).

(a) Develop a Monte Carlo algorithm which allows you to compute expectations with respect to the posterior distribution.

The simplest option is to sample from the prior using the code developed in question 1 and the importance weight using the likelihood and self-normalised importance sampling to compute expectations with respect to the posterior.

```
ripost <- function(sample.size, n, obs) {
    x <- rkumaraswamy(sample.size,3,1)
    w <- dbinom(obs,n,x)
    w <- w / mean(w)
    list(X=x,W=w)
}
```

(b) Use your algorithm to compare the prior mean and variance of p with its posterior mean and variance in two settings: if $n = 10$ and you observe 3 successes and if $n = 100$ and you observe 30 successes.

The prior mean and variance can be easily estimated by simple Monte carlo.

```
> rks <- rkumaraswamy(100000,3,1)
> mean(rks)
[1] 0.7500746
> var(rks)
[1] 0.03748851
```

We can estimate the posterior mean and variance using our importance sampling algorithm. Doing this a few times with the two parameter values specified and a fairly small sample we find:

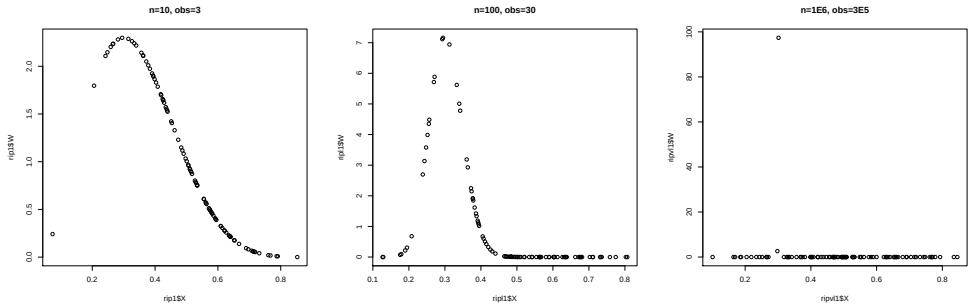
```
> rip1 <- ripost(100,10,3)
> rip2 <- ripost(100,10,3)
> rip3 <- ripost(100,10,3)
> mean(rip1$X*rip1$W)
[1] 0.4136343
> mean(rip2$X*rip2$W)
[1] 0.3959585
> mean(rip3$X*rip3$W)
[1] 0.3895696
> mean(rip1$X^2 * rip1$W) - mean(rip1$X * rip1$W)^2
[1] 0.01130371
> mean(rip2$X^2 * rip2$W) - mean(rip2$X * rip2$W)^2
[1] 0.01271694
> mean(rip3$X^2 * rip3$W) - mean(rip3$X * rip3$W)^2
[1] 0.01162405

> ripl1 <- ripost(100,100,30)
> ripl2 <- ripost(100,100,30)
> ripl3 <- ripost(100,100,30)
> mean(ripl1$X*ripl1$W)
[1] 0.3129667
> mean(ripl2$X*ripl2$W)
[1] 0.3261331
> mean(ripl3$X*ripl3$W)
[1] 0.3109552
> mean(ripl1$X^2 * ripl1$W) - mean(ripl1$X * ripl1$W)^2
[1] 0.002847201
> mean(ripl2$X^2 * ripl2$W) - mean(ripl2$X * ripl2$W)^2
[1] 0.001571596
> mean(ripl3$X^2 * ripl3$W) - mean(ripl3$X * ripl3$W)^2
[1] 0.00211944
```

(c) How does your algorithm behave if n is much larger (and we observe $3n/10$ successes)?

The performance for the mean here looks fairly consistent, although the variance estimates seem less convincing—that is, they are more variable across the three runs. Perhaps it might stabilize for very large n?

```
> ripvl <- ripost(100,10^6,3*10^5)
> ripvl1 <- ripost(100,10^6,3*10^5)
> ripvl2 <- ripost(100,10^6,3*10^5)
> ripvl3 <- ripost(100,10^6,3*10^5)
> mean(ripvl1$X*ripvl1$W)
[1] 0.3014544
> mean(ripvl2$X*ripvl2$W)
[1] 0.3033601
> mean(ripvl3$X*ripvl3$W)
[1] 0.2977458
> mean(ripvl1$X^2 * ripvl1$W) - mean(ripvl1$X * ripvl1$W)^2
[1] 3.188226e-07
> mean(ripvl2$X^2 * ripvl2$W) - mean(ripvl2$X * ripvl2$W)^2
[1] 5.134781e-16
> mean(ripvl3$X^2 * ripvl3$W) - mean(ripvl3$X * ripvl3$W)^2
[1] -1.387779e-17
```

Indeed,the posterior mean seems to be estimated reasonably consistently, but the variance estimate is terrible — it varies by 10 orders of magnitude between these three estimates!
The disparity in performance between mean and variance estimates is actually an artefact of the compact support of the parameter. There is always a sample *fairly* close to the mode of the posterior and so estimates of the mean don't look bad, but in fact, the distribution is being very poorly approximated here. Look at the weighted samples for the three sample sizes considered:



With the largest number of observations, a single sample contributes almost all of the mass to the estimate and this can lead to very poorly behaving estimators.

(d) How might you address this problem?

There are many things we could do, but the most natural might be to try to find a proposal distribution that's more like the posterior than the prior; in one dimension it's not too difficult to do this as we could plot a function equal to the product of prior and likelihood and then try to fit a distribution to it.
In this case the problem is simple enough we could probably get around it by increasing the Monte Carlo sample size, but that strategy does't generalise very well.