Computer Practical 2 Outline Solutions
Monte Carlo Methods

1. 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 it's inverse.

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's natural to try the change of variable $z = 1 - u^a$ which leads to:

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

$F_X(x)$ 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)
}
```

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

But $1 - U \overset{\mathcal{D}}{=} U$ and so we can simplify the final line to

```
x <- (1-u**(1/b))**(1/a)
```

iii. 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 $a = b = 2$. (You may wish to run it a few times and take the average.)

There's some variability in runtime, ranging from about 25 ms to 31 ms, with an average over ten runs of 27ms using my aged laptop.

(b) Rejection Sampling

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 it's 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:

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

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 $a = b = 2$.

The simple implementation above takes around 25s to produce 100,000 samples; around 1,000 times as long as the inversion sampler... but this is largely due to inefficient implementation: R is very slow when handling loops.
Entering:

```
library('compiler')
enableJIT(3)
```

and repeating the timing experiment reduces the time to 22s. `? compiler` will give some explanation of this.
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 between 0.4 and 0.65 seconds to run on my laptop, and so it takes between approximately 15 and 25 times as long as the inversion sampler to produce the same sample:

```
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 $100000M = 153960$ which seems eminently plausible (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 targetting this distribution.

This implementation took between 11 and 13 ms on my laptop; 12 ms on average over 10 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 targetting this distribution.

This takes a surprisingly consistent 13 ms on my latop.

(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 more efficient 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 efficient implementations, the rejection algorithm does a little better than we might expect, suggesting that the ancillary calculations done in this case are a little cheaper than those required for inversion sampling (looking at the expression involved, this seems plausible).
In contrast, the in efficient 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.

ii. To compare the importance sampling estimators with other algorithms it's 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.

The algorithms which use iid samples form 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^{-6}	Cost /ms	Var × Cost
Transformation	0.5334	5.9	2.7	15.9
Rejection (2)	0.5334	5.5	5.4	29.7
Importance	0.5327	11.8	1.2	14.2

In this case, somewhat surprisingly, the highest variance scheme is also the most efficient in computational terms: because it is so fast, it's also the one which produces the lowest variance per CPU cycle and may be the preferred estimator in many circumstances.

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

First obtain it's variance as you did for the other algorithms.

I obtained 3.9×10^{-6} which is comfortably the lowest amongst all the estimators and remains so after factoring in computational cost.

Now estimate it's bias: what's 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; the expectation agrees to at least four significant figures with those obtained by the other algorithms. Consequently, the variance and MSE are essentially the same.

2. Bayesian Inference via Monte Carlo. 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,3,5)
    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)
```

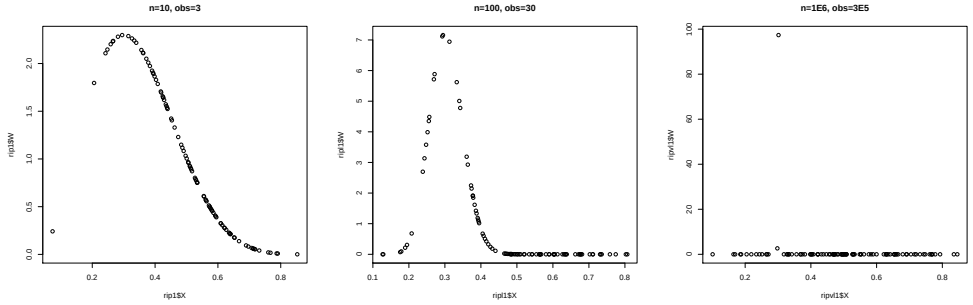
```
> mean(rip1$X*rip1$W)
[1] 0.4136343
> mean(rip2$X*rip2$W)
[1] 0.3959585
> mean(rip3$X*rip3$W)
[1] 0.3895696
> mean(rip1$X**2 * rip1$W) - mean(rip1$X * rip1$W)^2
[1] 0.01130371
> mean(rip2$X**2 * rip2$W) - mean(rip2$X * rip2$W)^2
[1] 0.01271694
> mean(rip3$X**2 * rip3$W) - mean(rip3$X * rip3$W)^2
[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 would your algorithm behave if n was much larger?

The performance for the mean here looks fairly consistent, although the variance estimates seem less convincing, so perhaps it might work even 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!
This 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 centre 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 sample size, but that strategy doesn't generalise very well.