Computer Practical 2
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.

      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.

      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.)

   (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).

      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 $a = b = 2$.

      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?

   (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.

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

      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).

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

   (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? 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?

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

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 10,000 (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 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 of each run and compute the sample variance of the collection of estimates you obtain.

How do the algorithms compare?

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.

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.

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.

(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.

(c) How would your algorithm behave if n was much larger?

(d) How might you address this problem?