STAT0023 Workshop 5: Simulation in R

This week's workshop explores some applications of simulation using computer-generated pseudo-random numbers — sometimes called 'Monte Carlo simulation'. We'll use the built-in random number routines from R, to see how simulation can be used to:

- Study properties of complex systems, that are hard to obtain analytically.

- Check the validity of large-sample approximations that are often used in statistics.

- Evaluate intractable integrals.

We will also look at some slightly more advanced techniques for sampling from 'non-standard' distributions. An introduction to the area is given in the lecture slides / handouts, available from the 'Course overview and useful materials' tab on the course Moodle page.

1   Setting up

For this workshop, in addition to the lecture slides / handouts you will need the following, all of which can be downloaded from the 'Week 5' tab of the Moodle page:

- These instructions.

- The R scripts `StockPrice.r`, `MCint.r` and `BetaSim.r`. You should save these to the appropriate place on your `N:` drive, as you created it during Workshop 1. Refer to Section 1 of the notes for Workshop 1 if necessary — and, as always, remember to make sure that your web browser doesn't change the filenames when downloading them.

Having downloaded the files, start up RStudio and use the S̲ession menu to set the working directory appropriately.

2   Pseudo-random number generation in R

We saw in the lecture that computers can't produce truly 'random' numbers: if we want to simulate realisations of random variables therefore, we need to use a 'pseudo-random number generator' (PRNG). We also saw that the starting-point for all PRNGs is to try and produce a sequence that, for practical purposes, is indistinguishable from a sequence of independent, identically distributed $U(0,1)$ random variables — because uniform random variables can be transformed into random variables from any other distribution. For example, to generate a Bernoulli random variable X with success probability p, start by generating $U \sim U(0,1)$ and then set $X = 1$ if $U \leqslant p$, $X = 0$ otherwise. To generate a random variable $X \sim Exp(\lambda)$, start by generating $U \sim U(0,1)$ and then set $X = -\lambda^{-1} \log U$. There are more examples on the slide 'Today $U(0,1)$, tomorrow the Universe ...' from the lecture.

We also saw in the lecture that for all PRNGs, the next value in sequence is a function of the most recent value(s). This means that (a) almost all PRNGs will repeat themselves after a finite 'period' (b) successive values are not independent. The challenge in devising a good PRNG is to find an algorithm for which the period is very long, and for which the joint distributions of sequences of successive values look like the joint distributions of independent uniform random variables. This is not easy, and the problem wasn't solved to modern standards until around the mid-1990s. As a result, unfortunately it isn't possible to give you a 'quick and simple' explanation of how the default PRNG in R works: suffice it to say that it has a period of $2^{19937} - 1 \approx 10^{6000}$, and joint distributions of up to 623 consecutive values are indistinguishable from a 623-dimensional distribution of independent uniforms — see `help(Random)`. This should be adequate for our purposes![1] This generator is one of the very few that passes all of the 'Diehard' battery of tests (see https://en.wikipedia.org/wiki/Diehard_tests).

The fact that PRNGs repeat themselves has a beneficial side-effect: if you perform a simulation using pseudo-random numbers, and you know where you started in the PRNG sequence, then you can always go back and repeat the simulation exactly later on. This is helpful for debugging code (it's hard to trace problems if your code produces different results every time you run it!). PRNGs are initialised by setting a **seed**, which is used to calculate a starting point in the sequence: if you use the same seed every time, you'll always get the same sequence of pseudo-random numbers. If you don't explicitly set a seed, many software packages (including R) will initialise their PRNG with a value taken from the system clock at the point when the PRNG is first called during a session: in this case you'll get a different set of pseudo-random numbers every time. In R, the command for setting the seed is `set.seed()`, with a single integer as an argument — for example `set.seed(2000)`.

3 Simulating complex systems: a Monte Carlo example from finance

One application of simulation is to explore the properties of systems that are too complex to study analytically. We'll use the 'stock price' example from the lecture to illustrate this.

Monte Carlo methods are used extensively in mathematical finance. A very simple model for stock prices is the geometric Brownian Motion model. In this model, the stock price S_t at time t is represented as

$$S_t = \exp\left[\left(\mu - \frac{1}{2}\sigma^2\right)t + \sigma B_t\right] ,$$

$$\text{where} \qquad B_t = \sum_{j=1}^{t} \varepsilon_j \qquad \text{with} \qquad \varepsilon_j \sim \mathcal{N}(0,1)$$

and μ and σ are parameters and the $\{\varepsilon_t\}$ are mutually independent. Notice that $B_t \sim N(0,t)$, but that successive values of B_t are correlated because they are constructed from a common sequence of underlying variables $\varepsilon_1, \varepsilon_2, \ldots$.

---

[1]However, as we enter the 'Big Data' era, people are starting to explore systems that have thousands of dimensions, and this may test the limits of what is possible with the current generation of PRNGs.

Assume that we buy a quantity of the stock at time $t = 0$ and then wait a random time $T \sim \Gamma(2,3)$ before selling. The stock price at time of sale is therefore S_T (notice that the subscript itself is a random variable).

If you were an investor considering whether to buy this stock, you might be interested in knowing something about the distribution of the sale price S_T. If you remember your STAT0005 material (or equivalent), you could use the iterated expectation and variance laws, together with things like your knowledge of the lognormal distribution and the moment generating function of the gamma distribution, to calculate the expected value and variance of S_T: $\mathbb{E}(S_T) = \mathbb{E}_T \left[\mathbb{E}_{S_T|T} (S_T|T) \right]$, for example. It's not entirely straightforward, though — and, of course, it only gives you the mean and variance. What if you want to know the probability that S_T is less than some threshold, or any other quantity that requires knowledge of the full distribution of S_T? Simulation is easier: we will just simulate a large sample from the distribution of S_T, and then look at the properties of the sample. Providing we run enough simulations, these sample properties will be very close to the corresponding properties of the **actual** distribution of S_T.

In the lecture, you saw some code for simulating the sale prices in this situation and for estimating the expected sale price when $\mu = 0.5$ and $\sigma^2 = 0.01$. The simulation code is provided to you in the script `StockPrice.r` that you downloaded earlier. Open this script in RStudio, and run each line one at a time. After defining the number of simulations required (`nsims <- 10000`), the script sets the values of μ and σ^2 and then simulates a sample of sale times from the relevant gamma distribution (`T <- rgamma(nsims,2,3)`). Conditional on these sale times, the next command (`rnorm(nsims,mean=0,sd=sqrt(T))`) samples corresponding values of B_T: from the model specification above, the conditional distribution of B_T given T is $N(0,T)$.[2] Notice that the `rnorm()` command allows you to supply a **vector** of standard deviations (here, `sd=sqrt(T)` is a vector, because you just generated T as a vector in the previous line): thus you can generate your sample of B_T values in a single command (no loops!). Finally, `SalePrice <- exp( (mu-0.5*sigsq)*T+(sqrt(sigsq)*Bt) )` calculates the actual sale prices corresponding to your sampled values of B_T, according to the model equation given above.

Having done that, here are some other things to try (you'll have to write the code for these yourself: just edit the script that you've downloaded):

1. Calculate the sample mean of your `SalePrice` vector: this is your estimate of the expected sale price. Compare the result with the person sitting next to you: do you get the same answer?

2. Calculate the sample variance of your `SalePrice` vector, and also the approximate probability that the price is less than 2 units (this is just the proportion of simulated values less than 2: `mean(SalePrice<2)` will do the trick, but don't go any further until you've worked out **why** this works!).

3. Produce a plot showing the estimated probability density function of the sale price. You can find the command to plot an estimated density function **somewhere** in this week's

---

[2]Note that you don't need to simulate the εs, you only need the value of B_T itself — the only reason for mentioning the εs above was in case you used these notes as your only source of information about geometric Brownian motion ☺

lecture slides . . .

4. Suppose now that you want to see how the expected price varies with μ. You could run the existing code many times, each time changing the value of `mu` — but this would be very tedious. As an alternative, you could write an R **function** to do it. So cut and paste your existing code to make a function out of it:

```
MeanPrice <- function(mu,sigsq,nsims) {
  T <- rgamma(nsims,2,3)
  Bt <- rnorm(nsims,mean=0,sd=sqrt(T))
  SalePrice <- exp( (mu-0.5*sigsq)*T+(sqrt(sigsq)*Bt) )
  mean(SalePrice)
}
```

Now you can go `MeanPrice(0.5,0.01,10000)` to reproduce your earlier estimate of the expected sale price. Do you get exactly the same answer?

To see how the expected price varies with μ, the obvious approach now is to make a grid of values for μ and to loop over this grid, calling your `MeanPrice()` function each time and storing the result to build up a vector of expected price estimates. But you know all about loops in R (i.e. avoid them if at all possible). Here's the smart alternative:

```
> mu.grid <- seq(0.01,1,0.01)
> PriceVec <-
+   mapply(MeanPrice, mu=mu.grid, MoreArgs=list(sigsq=0.01,nsims=1000))
```

This `mapply` command computes the function `MeanPrice` separately for all values of `mu` in the vector `mu.grid`, with the `sigsq` and `nsims` arguments fixed each time (`nsims` is set to 1000 here so that it doesn't take too long, and also to illustrate another point in the last sentence of this paragraph). The result is a vector of estimated prices, one for each element of `mu.grid`. Make a line graph showing how the expected sale price varies with μ. Do you think this gives an accurate picture of the form of the relationship between μ and $\mathbb{E}(S_T)$?

5. Finally for this example: add an extra command `set.seed(2000)` (or any other number of your choice) at the start of your `MeanPrice()` function i.e. **before** the line `T <- rgamma(nsims,2,3)`. Rerun the function definition so as to redefine the function to R; and then rerun the `mapply()` command and replot your line graph. How does it differ from the previous one?[3] Why is this?

This example illustrates the power of simulation. However, as discussed in the lecture, you can usually obtain more insights into a problem by solving it analytically where possible. Second-year students: if you are interested in this kind of financial application, consider taking the course STAT0013 **Stochastic Methods in Finance 1** in your final year.

---

[3] If it doesn't look very obviously different from the previous one, you haven't redefined the function to R.

4 Checking large-sample approximations

You have all heard of the Central Limit Theorem, and you have probably also encountered situations in your statistics courses where the lecturer has said something like 'such-and-such a result is exact when the data are normally distributed, but it works for other distributions as well providing the sample size is large enough'. Such statements are based on **asymptotic** distributional results, relating to the limiting distribution of some quantity as the sample size tends to infinity. Of course, in practice we never have infinite samples: to what extent, therefore, are we justified in using these asymptotic results in any practical situation? More concisely: how large is 'large enough'?

Simulation can help to answer this question. To illustrate, let's consider the problem of calculating a confidence interval for a population mean. We'll do it for an independent random sample $X_1, \ldots, X_n$, drawn from a lognormal distribution with parameters $\mu = 0$ and $\sigma^2 = 9$ (**remember**: if $X \sim \text{Lognormal}(\mu, \sigma^2)$ then $\log X \sim N(\mu, \sigma^2)$).

To calculate a 95% confidence interval for the mean of a distribution on the basis of an independent random sample $X_1, \ldots, X_n$, the standard large-sample formula is $\overline{X} \pm t_{0.025} s/\sqrt{n}$ where $\overline{X}$ is the sample mean, s is the sample variance and $t_{0.025}$ is the upper 2.5% point of a t distribution with $n - 1$ degrees of freedom. When the distribution of the $\{X_i\}$ is not normal, this formula is justified by the Central Limit Theorem 'when n is large enough'. But: how accurate is it for our lognormal example, in samples of size, say, $n = 30$? One way to find out is to simulate a large number of samples of size 30 from our lognormal distribution, calculate a 95% confidence interval from each using the standard formula, and then count how often the **true** mean of the distribution falls inside the confidence interval. The mean of $\text{Lognormal}(\mu, \sigma^2)$ is $\exp(\mu + \sigma^2/2)$, which is $\exp(4.5)$ for our example.

Here are your instructions:

1. Open a new R script pane in RStudio. You can type your commands into this pane, run them, correct your mistakes and then save your work when you've finished.

2. Produce a plot of the density of $\text{Lognormal}(0, 9)$:

   ```
   x.grid <- seq(0.1,10,0.1)
   fx <- dlnorm(x.grid, meanlog=0, sdlog=3)
   plot(x.grid,fx,type="l")
   ```

   You might want to add some other arguments to the `plot()` command, to create a more professional result. The main point, however, is that the density is highly skewed.

3. Simulate 100 samples of size 30, and calculate 95% confidence intervals for the true mean for each sample:

   ```
   n <- 30
   nsims <- 100
   X <- matrix(rlnorm(n*nsims,meanlog=0,sdlog=3),nrow=nsims)
   CI95 <- apply(X, MARGIN=1, FUN=function(x) { t.test(x)$conf.int })
   ```

The last two commands above need some explaining. The X matrix has `nsims` rows and `n` columns, and it is filled with pseudo-random numbers from the required lognormal distribution (the `rlnorm()` command produces these). The intention is that each row represents a different sample of n values. Next, the `apply` command is used to calculate a standard 95% confidence interval for each row (`MARGIN=1`) of the matrix. The syntax here can be interpreted as 'treat each row in turn as though it was a vector x, and calculate the result of the expression `t.test(x)$conf.int` for that row'. By default, the `t.test()` command produces a `list` result, one element of which is called `conf.int` and contains the lower and upper 95% limits for the population mean: `t.test(x)$conf.int` just extracts these values from the test result. So, if you look at your `CI95` object now, you'll see that it is a matrix with 2 rows and 100 columns: each column contains the confidence interval for the corresponding sample. To find out how often the interval contains the true mean, which is $\exp(4.5)$ as noted earlier:

```
TrueMean <- exp(4.5)
Hits <- (CI95[1,] < TrueMean) & (CI95[2,] > TrueMean)
mean(Hits)
```

The `Hits` object is a logical vector which is `TRUE` if the true mean is simultaneously above the lower confidence limit and below the upper one; and `FALSE` otherwise. For what proportion of your simulations does the true mean lie within the interval?

4. The result you've just produced is subject to simulation error, because you only did 100 simulations. If you want to be reasonably sure that your simulations estimate the **coverage** (i.e. the proportion of confidence intervals containing the true mean) to an accuracy of two decimal places, you should run about 10 000 simulations.[4] So: repeat the exercise, but now using `nsims=10000`. How close are you to a coverage of 0.95?

5. Your results above were based on samples of size 30. Does the coverage improve if you increase the sample size to 100? 1000? 10 000? It's going to get slow if you go much higher than this . . .

6. To try and understand what's going on, you might want to do things like plot the distribution of the simulated sample means, or to compute their variances. To calculate a vector of sample means, one for each simulation, you could go `SampleMeans <- apply(X, MARGIN=1, FUN=mean)` (compare with the `apply()` command that was used to calculate the confidence intervals above) or, more simply, `SampleMeans <- rowMeans(X)`. The rest of the detective work is up to you!

This example is deliberately extreme, but it does show the use of simulation in helping you to understand the accuracy of approximations. If it makes you doubt everything you ever learned, run the exercise again for samples of size 30, but this time from a Poisson distribution with mean 2.

---

[4]**Exercise:** prove this!

5   Monte Carlo integration

In the lecture, we saw that simulation can be used to estimate the value of intractable integrals $\int f(x)dx$ if $f(x)$ can be written as $f(x) = \phi(x)g(x)$, where $g(x)$ is a probability density function from which we can simulate. The trick is to notice that such integrals can be written as

$$\int f(x)dx = \int \phi(x)g(x)dx = \mathbb{E}_X\left[\phi\left(X\right)\right] = \theta \text{ , say,}$$

where X is a random variable distributed with pdf $g(\cdot)$. It follows that θ can be estimated by simulating a large sample $X_1, \ldots, X_n$ from the density $g(\cdot)$ and then setting $\widehat{\theta} = n^{-1}\sum_{i=1}^{n}\phi\left(x_i\right)$. As seen in the lecture, this approach can be applied to high-dimensional integrals and, in high dimensions, can estimate θ to a specified level of accuracy with a much lower computational cost than quadrature methods.

The accuracy of a Monte Carlo integral estimate can be assessed by calculating its standard error, which can then be used to compute a confidence interval for the true value of θ if necessary (always assuming that the standard formula for calculating a confidence interval is itself accurate — and we learned in the previous section that this isn't always the case!). Given two alternative Monte Carlo algorithms for estimating the same integral therefore, we will generally prefer the one that yields the smallest standard error for the same amount of computational effort. Sometimes we can use clever tricks to try and make the standard error as small as possible.

5.1   Importance sampling

You may have spotted an apparent difficulty with the basic idea of Monte Carlo integration: it's limited to integrands that can be written in the form $\phi(x)g(x)$. In practice though, this isn't restrictive: if we want to integrate an arbitrary function $f(x)$ then, for any density $g(x)$ that is non-zero throughout the range of integration, we can always write $f(x) = \left[f(x)/g(x)\right]g(x)$. If we then define $\phi(x) = f(x)/g(x)$, the integrand $f(x)$ can be written as $\phi(x)g(x)$ which is in the correct form for Monte Carlo integration.

This raises another question though: apart from the restriction that $g(x)$ must be non-zero over the range of integration, we can choose **any** pdf $g(x)$ if we want to use Monte Carlo integration. How do we know which one to choose? One consideration, obviously, is that we should choose something from which we can simulate fairly easily. Other than that, it seems sensible to choose a density $g(x)$ for which $\phi(x) = f(x)/g(x)$ is approximately constant, because in this case the simulated quantities $\phi(x_1), \ldots, \phi(x_n)$ will all be similar to each other and the standard error of $\widehat{\theta} = n^{-1}\sum_{i=1}^{n}\phi\left(x_i\right)$ will be small. How to ensure that $f(x)/g(x)$ is approximately constant? Obviously, choose $g(x)$ to be roughly proportional to $f(x)$! (but note that this won't be possible if $f(x)$ changes sign).

This idea forms the basis of the technique called **importance sampling**. In most textbooks, it is presented slightly differently from the discussion above, involving **two** probability density functions $g_1(x)$ and $g_2(x)$. There are reasons for this, but for present purposes it seems unnecessarily confusing.

5.2 Antithetic variates

Suppose we have two equally good Monte Carlo estimators of θ: $\widehat{\theta}_1$ and $\widehat{\theta}_2$, say. By 'equally good', we mean that they have the same variance, V_θ say. It can be shown (exercise) that if the correlation between $\widehat{\theta}_1$ and $\widehat{\theta}_2$ is large and negative then $\text{Var}\left[\frac{1}{2}(\widehat{\theta}_1 + \widehat{\theta}_2)\right] \ll V_\theta$. One way to achieve this is to base $\widehat{\theta}_2$ on simulated data that are negatively correlated with the data on which $\widehat{\theta}_1$ is based. This is the basis for the **antithetic variates** approach to variance reduction in Monte Carlo simulation. Obviously, in general it's twice as much work (or thereabouts) to compute two estimators rather than one; so the variance reduction needs to be more than a factor of 2 for this to be worthwhile (otherwise, you could just work with $\widehat{\theta}_1$ and double the number of simulations). This is possible however, as the following trivial example demonstrates.

Example: Consider the integral $\theta = \int_0^1 x\,dx$. This can be written as $\theta = \int_0^1 x \cdot 1\,dx = \mathbb{E}(U)$ where $U(0,1)$. A very simple Monte Carlo estimator of θ is $\widehat{\theta}_1 = U_1$, where U_1 is a single simulated value from the $U(0,1)$ distribution (this is the sample mean from a sample of size 1). We have $\text{Var}\left(\widehat{\theta}_1\right) = 1/12$. If we double the sample size and calculate the sample mean of **two** independent $U(0,1)$ quantities, the variance of the result is $1/24$.

Rather than generate an additional **independent** $U(0,1)$ random variable however, consider setting $\widehat{\theta}_2 = V_1$ where $V_1 = 1 - U_1$. Clearly, V_1 also has a $U(0,1)$ distribution, but it is perfectly negatively correlated with U_1. The antithetic variates estimator $\frac{1}{2}\left(\widehat{\theta}_1 + \widehat{\theta}_2\right)$ is then

$$\frac{1}{2}\left(\widehat{\theta}_1 + \widehat{\theta}_2\right) = \frac{1}{2}\left(U_1 + V_1\right) = \frac{1}{2}\left(U_1 + 1 - U_1\right) = 1/2 \ ,$$

which is a constant and therefore has variance zero. And is equal to $\int_0^1 x\,dx = [x^2/2]_0^1$ ... from a random sample of two simulated quantities, we have produced the exact value of θ with no error! The magic ingredient was the negative correlation between $\widehat{\theta}_1$ and $\widehat{\theta}_2$. ∎

This example **is** a trivial one, obviously. However, many applications of antithetic variate methodology can be considered as an extension of it. Thus: simulations are often based a sample $u_1, \ldots, u_n$ from a $U(0,1)$ distribution. Basing $\widehat{\theta}_1$ on u_i and $\widehat{\theta}_2$ on $1 - u_i$ may achieve some reduction in variance compared to using $\widehat{\theta}_1$ alone. Unfortunately, this trick will not always work. A sufficient condition for variance reduction is that θ can be written (perhaps via a change of variables from x to u in the integration) in the form $\theta = \int_0^1 \psi(u)\,du$ where $\psi(u)$ is monotonic on the range $(0, 1)$.

5.3 Example

In this section we'll compare some different Monte Carlo integration approaches, to evaluate the integral

$$\int_a^\infty \frac{1}{\pi(1 + x^2)}\,dx = \int_a^\infty f(x)\,dx = \theta \ , \text{ say,}$$

where $a > 0$ is some constant. In fact, this is $P(X > a)$ where X has a standard Cauchy distribution (i.e. a t distribution with 1 degree of freedom). You may spot that the integral

can be done analytically, but it serves to illustrate some different approaches to Monte Carlo integration. We consider five different approaches below. Be aware that notation like 'X' and '$g(x)$' represents different things in each approach — although $f(x)$ and θ represent the same things throughout.

1: 'Hit-or-miss' MC. Notice that θ can be written as

$$\theta = \int_{-\infty}^{\infty} I(x > a)\frac{1}{\pi(1 + x^2)}dx ,$$

where $I(x > a) = 1$ is an **indicator function** taking the value 1 if $x > a$ and zero otherwise. The function $f(x) = \dfrac{1}{\pi(1 + x^2)}, -\infty < x < \infty$ is the density of a standard Cauchy. To estimate θ therefore, we could just sample $x_1, \ldots, x_n$ from a standard Cauchy distribution and define

$$\widehat{\theta}_1 = \frac{1}{n}\sum_{i=1}^{n} I(x_i > a).$$

Actually, exactly the same technique was used at one point in the 'Monte Carlo finance' example from Section 3. Can you find it?

2: 'Hit-or-miss' MC using symmetry. The required integral is the area under the upper tail of the standard Cauchy density. But by symmetry, this is equal to the area under the lower tail. Instead of evaluating the proportion of simulations in the upper tail therefore, we could evaluate the proportion in **both** tails and then halve the result.

This corresponds to taking $\phi(x) = \frac{1}{2}I(|x| > a)$ and $f(x) = \dfrac{1}{\pi(1 + x^2)}, -\infty < x < \infty$, so

$$\theta = \frac{1}{2}\int_{-\infty}^{\infty} I(|x| > a)\frac{1}{\pi(1 + x^2)}dx.$$

The associated MC estimator is obtained by sampling $x_1, \ldots, x_n$ from a standard Cauchy distribution and defining

$$\widehat{\theta}_2 = \frac{1}{2n}\sum_{i=1}^{n} I(|x_i| > a).$$

Can you figure out which of $(\widehat{\theta}_1)$ and $\widehat{\theta}_2$ has the smaller variance? (**Hint**: what are the distributions of $I(x_i > a)$ and $I(|x_i| > a)$, and what are the variances of these distributions?)

3. Another use of symmetry. An alternative way to calculate the sum of the two tail areas is to subtract the central area from 1 and write

$$\theta = \frac{1}{2}\left(1 - 2\int_0^a \frac{1}{\pi(1 + x^2)}\right)dx = \frac{1}{2}\left(1 - 2\int_0^a \frac{a}{\pi(1 + x^2)}\frac{1}{a}dx\right).$$

This integral can be written as $\mathbb{E}\left(\dfrac{a}{\pi(1+X^2)}\right)$ where X has density $g(x) = 1/a$ for $0 < x < a$ — in other words, where $X \sim U(0,a)$. So to evaluate the original integral, we can sample $u_1, \ldots, u_n$ from a $U(0,a)$ distribution and define

$$\widehat{\theta}_3 = \frac{1}{2}\left(1 - \frac{2}{n}\sum_{i=1}^{n}\frac{a}{\pi(1+u_i^2)}\right).$$

4. **Transforming the range of integration.** Going back to the original integral, a transformation of variables $(y = 1/x)$ shows that

$$\int_a^\infty \frac{1}{\pi(1+x^2)}dx = \int_0^{1/a}\frac{1}{\pi(1+x^2)}dx = \int_0^{1/a}\frac{1}{a\pi(1+x^2)}\,a\,dx = \mathbb{E}\left[\frac{1}{a\pi(1+X^2)}\right]$$

where now we take $X \sim U(0,1/a)$ with density $g(x) = a, 0 < x < 1/a$. Yet another MC option, therefore, is to sample $u_1, \ldots, u_n$ from a $U(0,1/a)$ distribution and define

$$\widehat{\theta}_4 = \frac{1}{n}\sum_{i=1}^{n}\frac{1}{a\pi(1+u_i^2)}.$$

5. **Importance sampling.** Since the integrand is roughly proportional to $1/x^2$ for most of its range, the choice $g(x) = a/x^2, x > a$ may lead to a reasonable importance sampling estimator. We therefore write

$$\theta = \int_a^\infty \phi(x)\,f(x)dx = \int_a^\infty \frac{x^2}{a\pi(1+x^2)}\frac{a}{x^2}dx.$$

We can sample $v_1, \ldots, v_n$ from $g(x)$ by inversion of its distribution function, $G(x) = 1 - a/x, x > a$ (see lecture slides for details of the inversion method), to give $v_i = a/u_i$, where $u_i \sim U(0,1)$. Then define

$$\widehat{\theta}_5 = \frac{1}{n}\sum_{i=1}^{n}\frac{v_i^2}{a\pi(1+v_i^2)}.$$

In principle, all of these approaches can be combined with antithetic variates to try and reduce the variance of the final estimator. In approach 4 for example, note that if $U \sim U(0,1/a)$ then $V = 1/a - U$ is also distributed as $U(0,1/a)$. Thus, an extra estimator of θ could be based on $v_i = 1/a - u_i, i = 1, \ldots, n$ and we define

$$\widehat{\theta}_{4a} = \frac{1}{2}\left\{\frac{1}{n}\sum_{i=1}^{n}\frac{1}{a\pi(1+u_i^2)} + \frac{1}{n}\sum_{i=1}^{n}\frac{1}{a\pi(1+v_i^2)}\right\}.$$

If we want to evaluate the usefulness of the antithetic variates approach, this is cheating slightly because the computational cost of two sums is greater than the cost of a single sum in approach 4. If you worry about the immorality of such a comparison, you are welcome to amend the code below!

Finally ...   at the start of the workshop you downloaded the file `MCint.r`. Open this now, and click the ⟦ Source ▾ ⟧ button to run it. You will see the `Cauchy.tail.MC` object in your workspace. This is a function that implements approaches (1) to (5) above, and attempts to improve on these using antithetic variates.

Don't worry too much if you can't make sense of the code in `MCint.r`. Some parts are not entirely obvious: it has been written in such a way as to **guarantee** that the same uniform random number sequence is used for all of the approaches, which is a bit tricky. The reason for doing this is to ensure that any differences between them are due to genuine differences between the methods, rather than to differences between the random numbers. You're not going to be quizzed on this code in detail! You may be quizzed on your understanding of Monte Carlo integration though, so you should take the exercise below seriously.

After sourcing the script, try the following:

```
n.MC <- 50          # number of MC samples ("n" in workshop notes)
nsims <- 100        # Repeat the integration n.sims times, to
                    # get an idea of variability in performance
a <- 2              # a in P(X>a), where X is standard Cauchy

MC.res <- MC.res.anti <- matrix(NA,nrow=nsims,ncol=5) # to store results
for (i in 1:nsims){
 temp <- Cauchy.tail.MC(a,n.MC)
 MC.res[i,] <- temp$basic
 MC.res.anti[i,] <- temp$anti
}

pcauchy(a,lower.tail=F) # true value of integral
summary(MC.res)         # approximately the correct answers?
summary(MC.res.anti)

round(apply(MC.res,MARGIN=2,FUN=sd),5)      # SDs for basic MC integration
round(apply(MC.res.anti,MARGIN=2,FUN=sd),5) # SDs when using antithetic variates

# some methods are equivalent (they give exactly the same results if
# the same initial uniform variates are used) ...
pairs(cbind(MC.res,MC.res.anti))
```

Is there a clear winner?

If you don't learn anything else as a result of this exercise, you should take away the message that Monte Carlo integration can be a bit of an art form: it requires patience, creativity and good calculus skills.
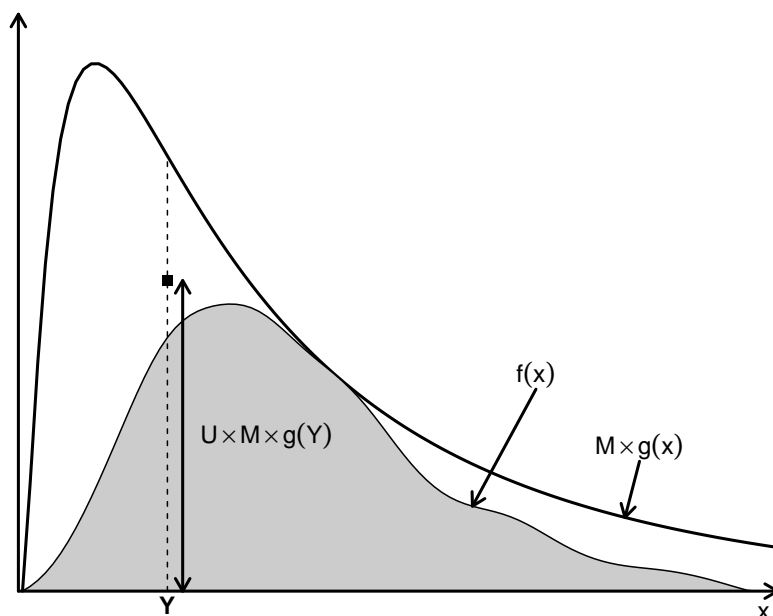
Figure 1: Illustration of the rejection method to sample from a density f (shaded area). A random variate Y is drawn from a density g (proportional to upper curve), and this value is retained with probability $f(Y)/[Mg(Y)]$ i.e. if $U \leqslant f(Y)/[Mg(Y)]$ where $U \sim U(0,1)$. In this illustration, $U > f(Y)/[Mg(Y)]$ so the candidate value is rejected. Candidate values are rejected most often in regions where g is large relative to f, because g generates too many values in these regions.

6   Rejection Sampling

Suppose we want to simulate X from a distribution with pdf f, and that none of the existing approaches (e.g. the inversion method) is feasible. The **rejection method** is designed for use when we know how to simulate a random variable Y from a distribution with pdf g and the same support as f, **providing** there is a constant M such that $f(x) \leqslant Mg(x)$ for all x.[5] The algorithm is as follows:

1. Generate Y from g and U from Uniform$[0,1]$.

2. If $U < f(Y)/[Mg(Y)]$, set $X = Y$.

3. Otherwise reject Y (and U) and go back to step 1.

Figure 1 provides some insight into why the rejection method works. For those that want a formal proof (which is not required for the course), Box 1 leads you through the required steps.

In practice, the algorithm will be slow if lots of Ys get rejected before one is accepted. It can be shown (Box 1) that the probability of acceptance is $1/M$. Ideally therefore, M should be as

---

[5]Note: the notation $f(\cdot)$ and $g(\cdot)$ has changed from that used in Section 5 — there's no connection between them.

Box 1: why rejection sampling works (optional)

The proof that rejection sampling works is beyond the scope of this course. However, some of you will want to know. If this is you, the following mathematical exercises should help you to understand it in more detail:

1. Show that if $f(x) \leqslant Mg(x)$ for all x, we must have $M \geqslant 1$. (**Hint:** remember that f and g are densities).

2. Noting that

$$P\left(U < f(Y)/[Mg(Y)]\right) = \int_{y\in\mathbb{R}} P\left(U < f(y)/\left[Mg(y)\right]\right) g(y)dy$$

   by the Law of Total Probability, show that the probability of accepting the generated Y in step 2 of the algorithm is $1/M$.

3. Now notice that the distribution function of the random variable X from the rejection algorithm is

$$
\begin{aligned}
F(x) = P(X \leqslant x) &= P\left(Y \leqslant x | U < f(Y)/\left[Mg(Y)\right]\right) \\
&= \frac{P\left(\{Y \leqslant x\} \cap \{U < f(Y)/\left[Mg(Y)\right]\}\right)}{P\left(U < f(Y)/\left[Mg(Y)\right]\right)},
\end{aligned}
$$

   where Y has density g. Let E denote the event $\{Y \leqslant x\} \cap \{U < f(Y)/\left[Mg(Y)\right]\}$. By writing $P(E) = \int_{y\in\mathbb{R}} P(E|Y = y)g(y)dy$ (or otherwise, if you can find an easier way!), show that X has density f and hence that the rejection algorithm works.

---

small as possible (if $g = f$, then trivially $M = 1$ which is the smallest possible value, and in this case all of the Ys get accepted!). This means that we should try to choose a g that is as close as possible to f in some sense.

A useful feature of the rejection algorithm is that we only need to know the 'target' density $f(\cdot)$ up to a constant of proportionality: if we can't (or don't want to) evaluate the normalising constant to ensure that $\int_{\mathbb{R}} f(x)dx = 1$, we don't have to! The reason is that the quantity M automatically scales the 'proposal' density $g(\cdot)$ to match the 'target' $f(\cdot)$ (see Figure 1). If you prefer a more mathematical explanation: in step 2 of the algorithm, the ratio $f(Y)/M$ doesn't depend on how $f(\cdot)$ is scaled, because if (for example) $f(\cdot)$ is doubled then M can also be doubled to compensate.

6.1 Example

Suppose we wish to simulate $X \sim \text{Beta}(\alpha, \beta)$ with $\alpha \geqslant 1$ and $\beta \geqslant 1$. The density of this distribution is

$$f(x) = x^{\alpha-1}(1-x)^{\beta-1}\Gamma(\alpha+\beta)/\left[\Gamma(\alpha)\Gamma(\beta)\right] \quad \text{for } x \in [0,1].$$

It looks more friendly if you write it as

$$f(x) = Kx^{\alpha-1}(1-x)^{\beta-1} \quad \text{for } x \in [0,1] \, ,$$

where K is a constant of proportionality.

To sample from this Beta distribution, we'll consider rejection sampling where the proposal distribution is $U(0,1)$ i.e. $g(x) = 1$ for $x \in [0,1]$.

You can check that $f(x)$ is maximised when $x = (\alpha-1)/(\alpha+\beta-2)$ and hence that the maximum value of $f(x)$ is

$$M = K\frac{(\alpha-1)^{\alpha-1}(\beta-1)^{\beta-1}}{(\alpha+\beta-2)^{(\alpha+\beta-2)}} \, ,$$

so that $f(x) \leqslant M = Mg(x)$ for x between 0 and 1 (remember that $g(x) = 1$ over this interval). To use the rejection method then, simulate Y from Uniform$[0,1]$ and also U from Uniform$[0,1]$. If

$$U \; < \; Y^{\alpha-1}(1-Y)^{\beta-1}\frac{(\alpha+\beta-2)^{(\alpha+\beta-2)}}{(\alpha-1)^{\alpha-1}(\beta-1)^{\beta-1}} \quad \left(\text{i.e., } \frac{f(Y)}{Mg(Y)}\right)$$

then we accept $X = Y$ as our value from the beta distribution. Otherwise we discard both Y and U and repeat the process until the above inequality is satisfied, when we accept $X = Y$. As noted above, the constant of proportionality K cancels so that we don't need to worry about its value.

Question: why are the restrictions $\alpha \geqslant 1$, $\beta \geqslant 1$ necessary in this example?

The script `BetaSim.r` defines a function called `BetaSim()`, which uses this rejection algorithm to sample from a beta distribution. In fact, it allows you to generate a **vector** of values, using the masking idea from Workshop 3 (remember the Mandelbrot example?) to keep track of which elements of the vector have been rejected. Open the script, source it and read through the function definition to check that you understand it. Then try the following:

1. Generate 1000 values from a Beta(2,1) distribution: `SimData <- BetaSim(1000,2,1)`, or something similar. The `SimData` object is a list containing two components: `x` (the simulated values) and `Ntries` (the numbers of trials taken to generate each element of `x`). As a check on the accuracy of the simulations, you might want to calculate the mean and variance of the simulated values: the mean and variance of a Beta(α, β) distribution are $\alpha/(\alpha+\beta)$ and $\alpha\beta/\left[(\alpha+\beta)^2(\alpha+\beta+1)\right]$, respectively.

2. Still working with the sample you generated in step 1, a more thorough check is to produce a quantile-quantile (Q-Q) plot. You have encountered these plots before, as a way of checking the assumption of normality in regression models. However, the basic idea — to plot the quantiles of a sample against those of an assumed distribution — can be used

for any distribution. R doesn't have a built-in command to produce Q-Q plots for non-normal distributions however, so we need to write our own code for it. Here it is (note that `ppoints(1000)` calculates a grid of 1000 evenly spaced values in the range $(0, 1)$):

```
> Q.theoretical <- qbeta(ppoints(1000), 2, 1)
> Q.simulated <- sort(SimData$x)
> plot(Q.theoretical, Q.simulated)
```

Don't be afraid to professionalise the plot!

3. Hopefully, those checks reassured you that the simulated values really **do** follow the required Beta distribution. We haven't looked at the efficiency of the rejection algorithm yet, though. To explore this, look at the `Ntries` component of your `SimData` object. Calculate the mean of this component: `mean(SimData$Ntries)`. Does this agree with the theoretical value? To figure out what the theoretical value should be, you'll need to ask yourself: what is the **distribution** of the number of trials required until a proposed value is accepted by the rejection algorithm? You should be able to answer this question using your knowledge from first-year statistics courses, coupled with a look back to page 12.

4. Try generating values from other Beta distributions as well, and check the accuracy and efficiency of the method. How does the mean number of trials change as α and β get larger? What is the distribution of the number of trials when $\alpha = \beta = 1$? Why is this?

Note: this is just an exercise, to demonstrate that the rejection algorithm works and to help you understand it. In practice, there are much more efficient ways of sampling from Beta distributions[6] — and, as with anything else, you should use built-in R functions where these are available. For example to simulate a value from a Beta distribution, use the function `rbeta()`. It should be more reliable and faster than anything we are likely to write — and speed matters if the function is called many times, which is very likely in the case of random number generators.

7   Moodle quiz

No new quiz this week, you've got your assessment to do.

---

[6]If you want to know some good ways of sampling from a variety of common and less common distributions, see
http://www.ucl.ac.uk/~ucakarc/work/software/randgen.txt. This is documentation for a set of pseudo-random number generation routines that were written in the mid-1990s, before the advent of R and at a time when it was really difficult to find freely-available code for simulating from a range of distributions. The algorithms there are not the absolute best that can be found today, but they're not bad.