Lab 10 — Statistical Simulation

1 Introduction

Simulation, in the statistical sense, means using computer-generated *pseudo-random numbers* to mimic a real event or set of data, or to generate observations from a given distribution or realisations of a stochastic process. This is a whole research area in its own right. In one workshop we are only able to scratch the surface of this subject. For further reading, three reference books are listed at the end.

First we look at how to generate random variables with a uniform distribution, then how to transform uniform random variables into ones with other distributions. Finally you will work through two examples, using simulation.

2 Generating Uniform Random Numbers

In contrast to random numbers produced by a physical device (such as rolling a die or drawing a number in the National Lottery) computer-generated random numbers are properly called *pseudo*-random because they are produced by a deterministic function. Such a function is called a random number generator (RNG) and, if it is well chosen, the sequence of numbers it produces will appear to be unpredictable and mutually independent. Most RNGs are designed to produce numbers that are uniformly distributed between 0 and 1. Some work better than others.

A simple but (potentially) very effective random number generator is a *congruential generator*. Given a, b, m and y_0, a sequence of N pseudo random numbers can be obtained from the recursion

$$y_n = ay_{n-1} + b \pmod{m} \quad \text{for } n = 1, \ldots, N.$$

In other words, y_n is the remainder when you divide $ay_{n-1} + b$ by m. Usually y_n is then divided by m to give a number between 0 and 1, which (hopefully) can be regarded as an observation from the Uniform$[0, 1]$ distribution. The value y_0 is called the "seed" as it is used as a starting point of the sequence. The idea works because if the values a, b and m are well chosen, the sequence of successive remainders $y_1, y_2, \ldots$ has little discernible structure.

Exercise: A trivial congruential generator is

$$y_n \equiv 5y_{n-1} + 1 \pmod{16}$$

If we start with the value $y_0 = 9$ we get $y_1 \equiv 5 \times 9 + 1 \pmod{16} = 14$. In R this can be done using the operator `%%` as follows:

```
> (5*9 + 1) %% 16
[1] 14
```

The next number in the sequence is $y_2 \equiv 5 \times 14 + 1 \pmod{16} = 7$, and so on.

Create the following function, making sure you understand all the commands, and carry out the following exercises:

```
trivial.cg <- function(n=1,y=9) {
 out <- rep(NA,n)
 for(i in 1:n) {
  y <- (5*y+1) %% 16
  out[i] <- y/16
 }
 out
}
```

1. Call `trivial.cg` to return one random number with a seed of 9. Check that you get the right answer.
2. Call `trivial.cg` to return one random number with a seed of 14.
3. Call `trivial.cg` to return two random numbers with a seed of 9. What do you get as the second number?
4. Call `trivial.cg` to return twenty random numbers with a seed of 9.

Notice that the last four numbers are exactly the same as the first four. This congruential generator has a *period* of 16 — and if you think about it, any generator with $m = 16$ is guaranteed to repeat itself after at most 16 iterations. Most RNGs produce sequences that repeat in this way, but good generators have a period much much greater than 16 (for example 7×10^{12}).

2.1 Different RNGs in R

To obtain n Uniform$[0, 1]$ random numbers in R, use the function

```
> runif(n)
```

Several types of RNG are available in R, all of which are more advanced than a congruential generator, but rely on similar principles.

The default method is the "Mersenne-Twister" method. This has a period of $2^{19937} - 1$, which is about 10^{6000}. If you ask R to calculate this number it returns `Inf`! To put it in context: on a modern computer you might be able to produce 20 million uniform random numbers per second (try `system.time(runif(2e7))` to find out exactly how long it takes). Now there are 86400 seconds in a day and 365 days in a year; thus you could generate about 6.3×10^{14} uniform random numbers per year. At this rate it would still take more than 10^{5985} years for this RNG to repeat itself — by which time the solar system will long since have ceased to exist, which is arguably a greater problem! If for some reason you want to specify another RNG, use the function `RNGkind` with the name of the RNG you want to use. For example

```
> RNGkind("Wichmann-Hill")
```

The names of all available RNGs can be found using

```
> help(RNGkind)
```

Usually there is no need to change the default setting. You can also force `runif` to use your own RNG, but you would only want to do this if you think you have a better RNG than any of those in R.

2.2 Setting the seed

In the first example the seed y_0 was chosen to be 9. The same sequence of numbers was obtained every time the same seed was used. Similarly the RNG called by `runif` needs a seed. Of course `runif` does not use the same seed every time you call it, otherwise you would get the same sequence of random numbers every time you call it — which you usually do not want (except sometimes, see below).

Each time `runif` is called, it accesses an object in your workspace called `.Random.seed`. It uses this to compute the required pseudo-random numbers, and then overwrites `.Random.seed` ready for the next time you happen to use `runif`.

Change the default RNG to be another type called "Marsaglia-Multicarry":

```
> RNGkind("Marsaglia-Multicarry")
```

Then type in

```
> .Random.seed
[1]       401 247803058  64738443
```

taking care to use the correct upper and lower case letters. Your seed will be different to the one here. The first number, 401, encodes which RNG is being used, the last two integers are the seed.

The seed for the Marsaglia-Multicarry method uses 2 integers; the seed for the default Mersenne-Twister method uses 625 integers.

Save the current seed using

```
> seed <- .Random.seed
```

Now get 5 uniform observations and look at the random seed again.

```
> runif(5)
[1] 0.51476200 0.08528351 0.05742169 0.76459165 0.06405747
> .Random.seed
[1]        401 1852444774  328143383
```

Notice the seed has changed. Reset the seed to be what it was before using

```
> .Random.seed <- seed
> runif(5)
[1] 0.51476200 0.08528351 0.05742169 0.76459165 0.06405747
```

The command `runif` returned exactly the same sequence of numbers as before, as the same seed was used.

You should **never** make up a new `.Random.seed`, because arbitrary values do not necessarily result in a good RNG. In the above example `.Random.seed` was simply replaced by an earlier seed, so we know it was acceptable. There is, however, a function in R which allows one to "choose" a seed, guarantees good behaviour of the resulting RNG and is easy to use:

```
> set.seed(25)
> .Random.seed
[1]        401  539310548 -611118395
> runif(5)
[1] 0.09414048 0.07374299 0.79480559 0.99556102 0.29460221
> set.seed(25)
> .Random.seed
[1]        401  539310548 -611118395
> runif(5)
[1] 0.09414048 0.07374299 0.79480559 0.99556102 0.29460221
```

The argument for `set.seed` is any integer of your choice.

What is the initial value of the seed? `.Random.seed` is in your workspace because you have already used `runif` (or `rnorm`, `rexp` etc.) in G3 workshops. If you start R with a new workspace, what value does the seed take then? The object `.Random.seed` is only created when it is first required — it is initiated using the integers stored in the computer's internal clock at that time.

Why would we want to set the seed? Usually we want to produce a different sequence of random numbers each time, so we accept the current value of `.Random.seed`. But sometimes it is very useful to ensure that the *same* set of random numbers will be generated. Suppose that in your MSc project you run a simulation study and write up the results. Later your supervisor says, "That's all very well, but what are the standard errors for your simulated estimates?" If you don't know the seed then you need to do a completely new simulation and thus change all of your results, in order to compute the standard errors. If you know the seed, you can simulate the same data and compute the standard errors, requiring minimal changes to your write up.

Other reasons for setting the seed include comparing different statistical routines or different estimators using the same simulated data; and allowing other people to replicate your results.

2.3 Assessing the properties of a RNG

There are many ways to assess the distributional properties of a RNG. The main requirements are that the pseudo-random numbers produced behave like independent observations from the required distribution, in this case Uniform$[0, 1]$. A simple test for uniformity is to construct a frequency table for intervals in $[0, 1]$, and do a chi-squared goodness of fit test. Many other tests are available, for example a Kolmogorov-Smirnov goodness-of-fit test (command `ks.test` in R). The choice will depend on what departure from uniformity is most relevant.

There are also many tests for departure from independence. One approach is to plot consecutive pairs of simulated numbers, $(U_1, U_2), (U_3, U_4), \ldots$. For a poor congruential RNG these points fall on a lattice.

Example: Create a new function very similar to `trivial.cg` called `test.cg` based on the congruential generator $y_{i+1} = 1229 * y_i + 1 \pmod{2048}$. Then obtain 1000 pseudo-random numbers using

```
> ruout <- test.cg(1000)
```

and plot the even observations against the odd observations.

```
> plot(ruout[(1:500)*2-1],ruout[(1:500)*2],pch=".")
```

Notice that all the plotted points lie on 5 lines. This is a lattice of points. The implication is that if we know the value of an observation U_i with i odd, then the value of U_{i+1} will be one of five values. The assumption that the observations are independent now looks very tenuous.

Now change the RNG back to the default Mersenne-Twister method:

```
> RNGkind("default")
```

and obtain a similar plot using this RNG. Notice that there is no discernible lattice, indicating that it is a much better RNG.

The examples given here are just simple illustrations of ways in which sequences can appear "non-random". In practice, there are many other tests of randomness that a good RNG should pass. There is a standard set of tests called the "Diehard battery", which is often used to check the performance of new RNGs. The Mersenne-Twister generator passes all these; few other RNGs do. The Wikipedia article at `https://en.wikipedia.org/wiki/Diehard_tests` summarises the tests.

3 Random numbers from Other Distributions

In general, to simulate from other distributions, one or more uniform random numbers are generated, and then transformed to give the correct distributional properties. Transforming

a random value from Uniform$[0,1]$ to one from a Uniform$[a,b]$ distribution is trivial and is left as an exercise.

The following examples illustrate simple methods for obtaining random values from common distributions (though these are not always the most efficient methods).

3.1 Bernoulli

To simulate a random value X from a Bernoulli distribution with parameter p, generate a uniform random number U, and compute X using

$$X = \begin{cases} 1 & \text{if } U \leq p \\ 0 & \text{if } U > p \end{cases}$$

3.2 Binomial

A Binomial(n,p) random variable is the sum of n independent Bernoulli(p) random variables. Generate $X_1, \ldots, X_n$ using the Bernoulli(p) method above, and then compute $Y = \sum_{i=1}^{n} X_i$ as the required value from binomial distribution.

Note that if n is large, this will require a large number of uniform random numbers for each binomial value, which will be slow. In that case other methods may be preferable, such as the next one below.

3.3 An arbitrary discrete random variable

Suppose X has a known probability mass function $P(X = r) = p_r$ for $r = 0, 1, 2, \ldots$. Generate U from Uniform$[0,1]$ and then compute X using

$$X = \begin{cases} 0 & \text{if } U \leqslant p_0 \\ 1 & \text{if } p_0 < U \leqslant p_0 + p_1 \\ 2 & \text{if } p_0 + p_1 < U \leqslant p_0 + p_1 + p_2 \\ & \ldots \\ k & \text{if } \sum_{i=0}^{k-1} p_i < U \leqslant \sum_{i=0}^{k} p_i \\ & \ldots \end{cases}$$

3.4 Exponential with parameter λ

Generate U from Uniform$[0,1]$ and then compute $X = -\frac{1}{\lambda}\log(U)$. This works because

$$P(X \leq x) = P\Big(-\frac{1}{\lambda}\log(U) \leq x\Big) = P(U \geq e^{-\lambda x}) = 1 - e^{-\lambda x}$$

which is the distribution function of the Exponential(λ) distribution. The last step above follows because if $U \sim$ Uniform$[0,1]$ then $P(U \geq u) = 1 - u$ for all $u \in [0,1]$.

3.5 Standard normal distribution

Central limit method

If $U_1, U_2, \ldots, U_n$ are independent Uniform$[0,1]$ random variables then they have mean $1/2$ and variance $1/12$; thus the distribution of

$$\left(\sum_{i=1}^{n} U_i - \frac{n}{2}\right) / \sqrt{\frac{n}{12}}$$

tends to Normal(0,1) as $n \to \infty$. A convenient value for n is 12. Then we get

$$Z = \sum_{i=1}^{12} U_i - 6 \;\sim\; N(0,1) \quad \text{approx.}$$

Note that the tails of this distribution are not very accurate, in particular Z cannot exceed ± 6. This is a problem for large scale simulations or where the tail behaviour is important (such as extreme value simulation).

The Box-Müller method

The previous method requires twelve (or in general n) uniform random numbers for each normal random value generated. The Box-Müller method is an exact transformation method that uses an exponential and a uniform to generate two independent standard normal values.

Let $R \sim$ Exponential$(\frac{1}{2})$ and $W \sim$ Uniform$[0, 2\pi]$, and define X and Y by

$$X = \sqrt{R}\cos W \quad \text{and} \quad Y = \sqrt{R}\sin W$$

Then it can be shown that X and Y are independent and have standard normal distributions. Since we know how to generate uniforms and exponentials we can use these to compute standard normal random variables.

3.6 Inversion method

Suppose that X is a continuous random variable with known cumulative distribution function $F(x)$. If $F(x)$ is strictly increasing then the inverse function $F^{-1}(u)$ is unique. If an analytical expression for $F^{-1}(u)$ can be found then simulating from F is easy, using the inversion method.

Generate U from Uniform$[0,1]$ and then set $X = F^{-1}(U)$. This works because:

$$P(X \le x) = P\Big(F^{-1}(U) \leqslant x\Big) = P\Big(U \leqslant F(x)\Big) = \int_0^{F(x)} 1 \, du = F(x)$$

as required. This method is the continuous analogue of the method Section 3.3 for a discrete distribution. Also, the method described above for the exponential distribution is a special case of this.

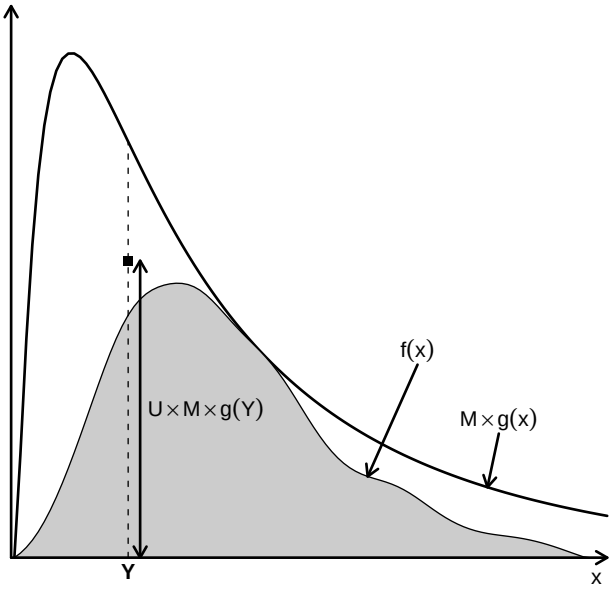

Figure 1: Illustration of the rejection method to sample from a density f (shaded area). A random variate Y is drawn from a density g (proportional to upper curve), and this value is retained with probability $f(Y)/Mg(Y)$ i.e. if $U \leq f(Y)/Mg(Y)$ where $U \sim U(0,1)$. In this illustration, $U > f(Y)/Mg(Y)$ 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.

3.7 Rejection method

Suppose we want to simulate X from a distribution with pdf f, and that we know how to simulate a random variable Y from a distribution with pdf g and the same support as f. If there is a constant M such that $f(x) \leq Mg(x)$ for all x, then we can use the rejection method. 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. Assume that the support for both f and g is $\mathbb{R}$. The following mathematical exercises should help you to understand the method in more detail:

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

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

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

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

where Y has density g. Let E denote the event $[Y \leq x] \cap [U < f(Y)/Mg(Y)]$. 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.

In practice, the algorithm will be slow if lots of Ys get rejected before one is accepted; since the probability of acceptance is $1/M$, we can see that ideally M should be as 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.

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

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

Let $Y \sim \text{Uniform}[0,1]$, so that $g(x) = 1$ 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

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

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

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

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 $X = Y$.

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

3.8 Exercises

Use `runif()` and the above information to simulate a sample of 100 observations from each of the following distributions. Use plots, tables and summary statistics to verify that your samples appear to come from the correct distribution.

1. Uniform$[-3, 10]$.

2. Bernoulli(0.6).
3. Binomial(15,0.6).
4. Exponential($\frac{1}{2}$).
5. Normal(2,5) using both the central limit and Box-Müller methods.
6. A distribution with $F(x) = (1 + e^{-x})^{-1}, -\infty < x < \infty$ using the inversion method.
7. Beta(2,3.5) using the rejection method.

Note: these are exercises, to help you understand some of the algorithms for generating random numbers from different distributions. In practice we would use the standard R functions. 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.

4 Uses of Simulation

The idea of simulating random variables may at first seem like an odd thing to do but it has many statistical uses, such as:

- to test or investigate the behaviour of other R functions, e.g., `> hist(rnorm(100,3,2))`;
- to assess statistical methods, we can simulate data from a known distribution and see how the methods perform;
- to design experiments and to randomise clinical trials;
- to study probability models or stochastic processes that can be expressed in terms of conditional distributions but that are difficult to solve analytically;
- to implement numerical methods such as Monte Carlo integration of a function (see below) or simulated annealing to optimise a function;
- to obtain robust estimates by using bootstrap or other re-sampling methods;
- to sample from a Bayesian posterior distribution using MCMC methods, as seen in STAT0031

4.1 Monte-Carlo integration

You have already met the trapezium rule and Simpson's rule for computing an integral, in Lab 6. An alternative approach uses simulation (often called Monte-Carlo) methods to *estimate* an integral (as if it were an unknown parameter).

Suppose that the desired integral can be expressed as

$$\Theta = \int_a^b \phi(x) f(x) dx = E[\phi(X)]\,,$$

where f is a probability density function. A random sample $X_1, X_2, \ldots, X_n$ is simulated from f. Let $Y_1 = \phi(X_1), \ldots, Y_n = \phi(X_n)$. Then an estimate of the integral is

$$\hat{\Theta} = \frac{1}{n} \sum_{i=1}^{n} \phi(X_i) = \bar{Y}.$$

That is, our estimator is the sample mean of the Y_is. If we generated another random sample we would get a different estimate. The variance of the distribution of estimates that we might get (i.e., the variance of the estimator) is

$$\mathrm{var}(\hat{\Theta}) = \frac{1}{n^2} \sum_{i=1}^{n} \mathrm{var}(\phi(X_i)) = \frac{\mathrm{var}(Y)}{n},$$

where $Y = \phi(X)$. Usually we would choose n to be large enough so that our estimate of Θ is sufficiently precise, i.e., so that its standard error is sufficiently small. An estimate of the standard error is

$$\mathrm{se}(\hat{\Theta}) \approx \frac{S_Y}{\sqrt{n}} \quad \text{where} \quad S_Y^2 = \frac{1}{n-1} \sum_{i=1}^{n} (\phi(X_i) - \hat{\Theta})^2\,.$$

Example: suppose we wish to integrate $\log(\log(x))$ for x between 2 and 10. That is, compute

$$\Theta = \int_2^{10} \log(\log(x)) dx\,.$$

We can express this in the above form by putting $\phi(x) = 8\log(\log(x))$ and $f(x) = \frac{1}{8}$ for $2 \le x \le 10$, so that $X \sim \mathrm{Uniform}[2, 10]$.

So we generate $X_1, X_2 \ldots, X_n$ from a Uniform$[2, 10]$ distribution and estimate the integral as $\hat{\Theta} = \frac{8}{n} \sum_{i=1}^{n} \log(\log(X_i))$. In R, $\hat{\Theta}$ and its approximate standard error (e.g., for $n = 1000$) can be obtained using

```
> n <- 1000; ruout <- runif(n,min=2,max=10)
> phi.x <- 8*log(log(ruout))
> mean(phi.x)
[1] 3.984976
> sd(phi.x)/sqrt(n)
[1] 0.07529205
```

Comments: the above integral can be easily calculated using other methods, such as the trapezium rule. But Monte-Carlo integration can be applied with very little difficulty to high-dimensional and more complicated integrals where other methods become intractable. If, for example, f is a function over k dimensions, it is easy to generate a k-dimensional uniform random sample (take a vector of k univariate uniforms), and hence the integral can be estimated in exactly the same way as above. On the other hand, methods like the trapezium rule rapidly become infeasible if k is large (if you evaluate f at, say, 25 points in each dimension, you end up needing 25^k function evaluations to compute your integral!).

Some methods (e.g., the trapezium method) cannot integrate between infinite limits. You can have infinite limits with Monte-Carlo integration, provided the range of the randomly generated numbers have the same limits (see exercise below). Also there are several clever tricks that can be used to reduce the variance of $\hat{\Theta}$. For details on this consult the books in the bibliography.

Exercise: Use Monte-Carlo integration to compute the integral

$$\Theta = \int_0^\infty x^2 e^{-x} dx.$$

You need to consider what distribution f to sample from (hint: it must have support $[0, \infty)$). Generate a sample of size 1000 and obtain an estimate of Θ. How does this value compare to the theoretical answer? Repeat to get several different estimates of Θ. Are the results as reliable as you would expect? Find the approximate standard error of the estimate.

4.2 Simulation modelling

For complex probability models, it can be very difficult or impossible to obtain an analytic expression for a required distribution. For example, to model the traffic flow on a long section of motorway would be difficult analytically, due to the complexity of the problem. Simulation modelling may be used in this situation to predict the distribution of traffic flow, and to see how sensitive it is to the model assumptions and to changes in the traffic network (e.g. closing down one carriageway due to roadworks).

We will consider a much simpler model. A building's air conditioning system consists of a cooling unit and air pump, eight major air ducts arranged in the network shown in Figure 2, and a safety outlet at the end of the network. Each duct has an air filter, which, without maintenance, will become blocked after an *exponentially distributed* time. The mean time to blockage for the filter in each duct is: 1 year for ducts A, B and C, 2 years for ducts D and E, and 3 years for ducts F, G and H. Assume that the times to blockage for each filter are mutually independent.

It is not essential for air to flow through every duct. However, if the air is unable to flow (via any route) from the pump to the safety outlet then the air conditioning unit breaks down, necessitating costly repairs. At this point, all air filters are replaced at the same time. The task is to simulate the distribution of times until the air conditioning unit breaks down due to blocked air filters.

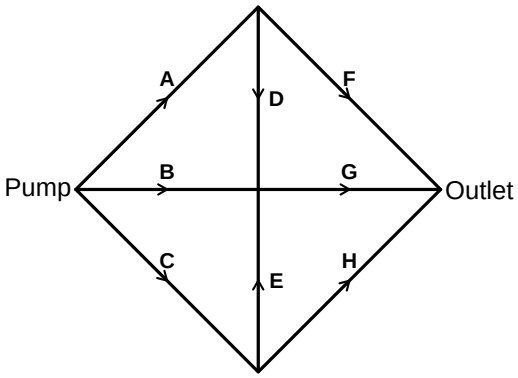

Figure 2: Diagram of airflow in an airconditioning system.

Exercise: Start by writing down the five paths that lead from the pump to the safety outlet, e.g. $\{A, F\}$. Now suppose we have $T_A, \ldots, T_H$, the time for each air filter to become blocked. Write down X_1 the time for the path $\{A, F\}$ to become blocked in terms of T_A and T_F. Do this for all 5 paths. Write down Y, the time until the whole system gets blocked, in terms of $X_1, \ldots, X_5$.

Now write an R function that simulates $T_A, \ldots, T_H$, `m` times (with `m` as a function argument) and returns a vector of m values of Y. Run your function to obtain a vector of 1000 simulated breakdown times. Plot a histogram of the output, and find the mean and variance of the simulated breakdown times. How often should the air conditioning unit be serviced to ensure that there is only a 1% chance of a breakdown between successive services?

Adapt your function to print out the proportion of times that the final unblocked path is $\{B, G\}$.

The distribution above can be found using analytical methods because all the distributions are exponential. However this approach can easily be adapted for any combination of distributions, provided pseudo-random numbers can be generated from them. Adapt your function again to simulate the breakdown times if the time to blockages have a Weibull distribution with shape parameter 2 and scale parameters: 1 for ducts A, B and C, 2 for ducts D and E, and 3 for ducts F, G and H.

Reference Books

- J. F. Monahan, *Numerical Methods of Statistics*. Cambridge, 2001.
- B. D. Ripley, *Stochastic Simulation*. Wiley, 1987.
- J. R. Thompson, *Simulation: A Modeler's Approach*. Wiley, 1999.

Finally

Now that you have finished the last workshop of the course: you have learned a lot of R commands, and hopefully discovered how to find out about other commands that you *haven't* learned. The course Moodle page contains a link to an "R reference card" by Tom Short, which is an 'at-a-glance' summary of commonly used commands. You may find this useful in the future.