Computer Practical 1 Outline Solutions
Simulation Basics

1. *Pseudorandom Numbers & Transformation Methods*

(a) RANDU was an extremely popular PRNG for many years (it was widely used on IBM and DEC mainframes, for example). It's a linear congruential generator with recursion:

$$Z_j = 65539 Z_{j-1} \mod 2^{31} \qquad X_j = Z_j / 2^{31}$$

⋆ i. Iterate the recursion twice to write Z_j as a function of Z_{j-1} *and* Z_{j-2}. Is there a clear issue here?

$$\begin{aligned} Z_j =& 65539 Z_{j-1} \mod 2^{31} \\ =& (65536+3) Zj - 1 \mod 2^{31} \\ =& (65536+3)((65536+3) Z_{j-2} \mod 2^{31}) \mod 2^{31} \\ =& ((2^{16}+3)(2^{16}+3) Z_{j-2} \mod 2^{31} \\ =& (2^{32} + 6 \times 2^{16} + 9) Z_{j-2} \mod 2^{31} \\ =& (6 \times 2^{16} + 9) Z_{j-2} \mod 2^{31} \\ =& (6 \times Z_{j-1} - 9 Z_{j-2}) \mod 2^{31} \end{aligned}$$

where the penultimate line follows by noting that $2^{32} z \mod 2^{31} = 0$ for any integer z.

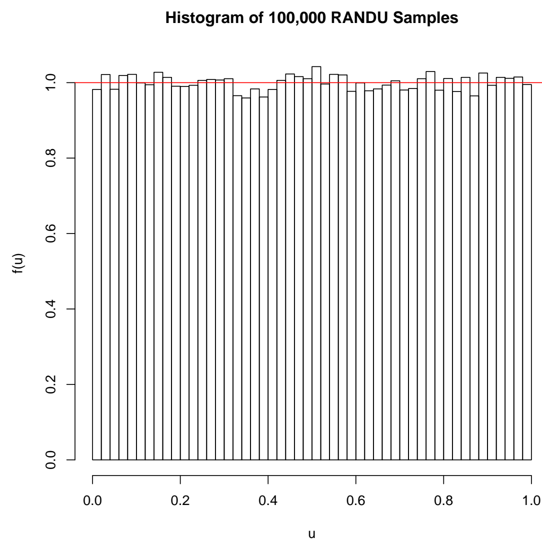
ii. The following code is one possible implementation of the PRNG. Using this or your own code obtain a long sequence of random numbers from this generator.

```
randu.map <- function(z,a=2**16 + 3, c=0, M=2**31) {
    (a * z + c) %% M
}
randu <- function (n,z0, a=(2**16) + 3, c=0, M=2**31) {
    z <- vector('numeric', n)
    z[1] <- randu.map(z0,a,c,M)
    for(i in seq(from=2,to=n,by=1)) z[i] <- randu.map(z[i-1],a,c,M)
    return(z / M)
}

unif <- randu(n=100000,z0=123142)
```

Plot a histogram of a reasonably long sequence obtained from this generator. Is there any clear departure from uniformity?

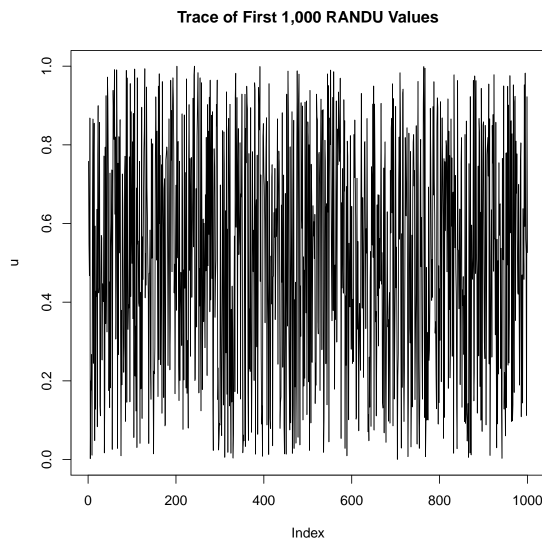
```
unif <-randu(n=100000,z0=123142)
hist(unif, main='Histogram␣of␣100k␣RANDU␣Samples', prob=TRUE, n=50, xlab='u', ylab='f(u)')
abline(1,0,col='red')
```



There doesn't appear to be any clear evidence here of a departure from uniformity.

iii. Plot a time series showing the evolution of the sequence over a thousand or so iterations. Is there any clear pattern?

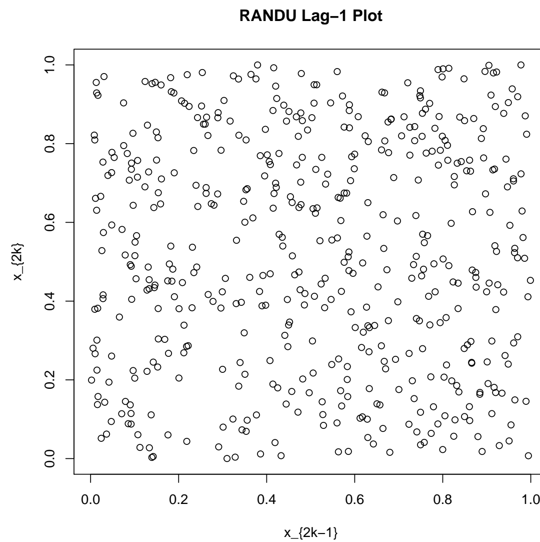
```
ts.plot(unif[1:1000],xlab='Index',ylab='u',main='Trace␣of␣First␣1,000␣RANDU␣Values')
```



Again, I don't think any pattern is obvious.

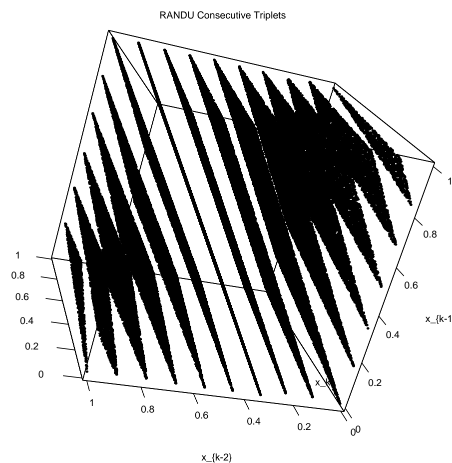
iv. Plot the even items in the sequence against the odd items; is there any clear pattern between successive items in the sequence?

```
plot(unif[seq(1,1000,2)],unif[seq(2,1000,2)],xlab='x[2k-1]',ylab='x[2k]',main='RANDU Lag-1 Plot')
```



v. Investigate what you see if you treat the triples in the `randu` data set as coordinates in $\mathbb{R}^3$. If the `rgl` package is available then you might find its `plot3d` function useful. Do you see anything interesting?

```
 library('rgl')
rgl::plot3d(unif[1:99998],unif[2:99999],unif[3:100000], xlab='x[k-2]',ylab='x[k-1]',zlab='x[k]',main='RANDU Consecutive Triplets')
```



A little manipulation demonstrates that these points are, indeed, concentrated on a small number of planes. If we were to use these points as though uniformly distributed in the unit cube then we could unknowingly make serious errors.

(b) 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

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

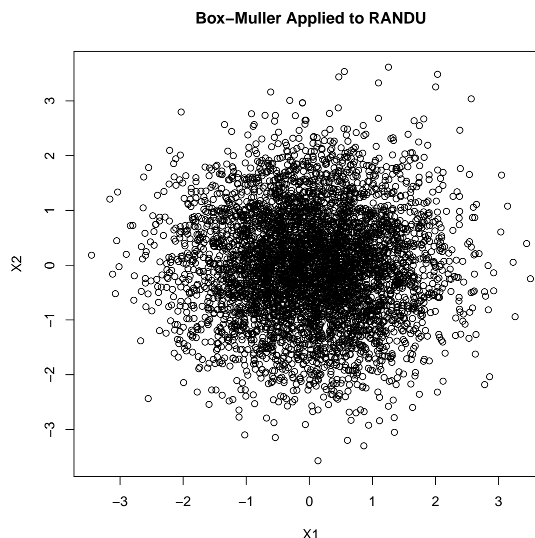
then $X_1, X_2 \overset{\text{iid}}{\sim} \mathsf{N}(0,1)$.

i. Write a function which takes as arguments two vectors and returns the two vectors 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)
    rbind(X1,X2)
}
```

ii. Apply this function with the first vector corresponding to those numbers generated at odd indices and the second to those generated at even indices in your sequence; you should find 10,000 or so samples sufficient. Plot the two "independent" normal variables obtained by doing this, one against the other. What do you see?

```
plot(t(box.muller(unif[seq(1,10000,2)],unif[seq(2,10000,2)])),main='Box-Muller␣Applied␣to␣
RANDU')
```



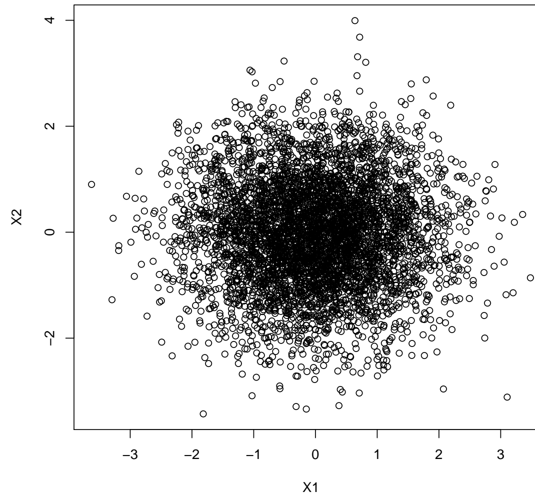
Even knowing that there *is* a clear lack of independence between these points it's difficult to *see* any problem here although it would be unwise to rely on these "independent normal" samples.

iii. 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 U[0, 1] random variables using this function, convert this to two vectors of 5,000 elements in the same way as you did with the RANDU values.

```
unif.mt <-runif(10000)
unif.mt.odd <-unif.mt[seq(1,10000,2)]
unif.mt.even <-unif.mt[seq(2,10000,2)]
```

iv. Apply your Box Muller transformation to the same elements of this array as you did with the `randu` dataset. How do the results compare?

```
plot(t(box.muller(unif.mt.odd,unif.mteven)))
```



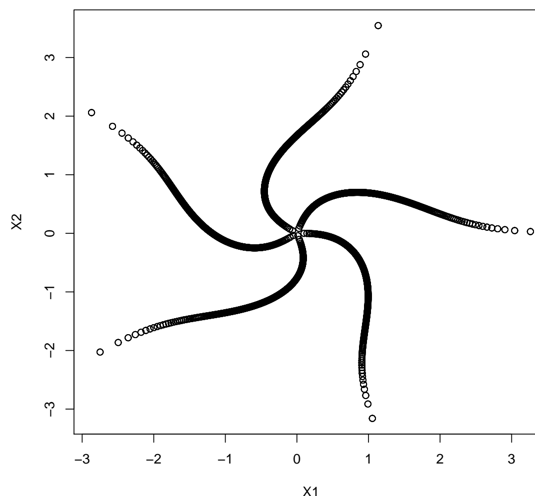
We can perhaps convince ourselves that this sample looks a little more normally distributed and a little more isotropic than the previous case, but without some formal analysis it's pretty difficult to claim that a difference is clearly visible here.

⋆ v. Try exploring other values of the coefficients with the linear congruential generator, e.g., try the sequence arising from the recursion

$$Z_j = 1229 Z_{j-1} + 1 \mod 2048 \qquad X_j = Z_j/2048$$

and see what happens when you push the resulting values through the Box Muller transformation.

```
unif.bad <-randu(10000,1371,a=1229,c=1,M=2**11)
plot(t(box.muller(unif.bad[seq(1,10000,2)],unif.bad[seq(2,10000,2)])))
```



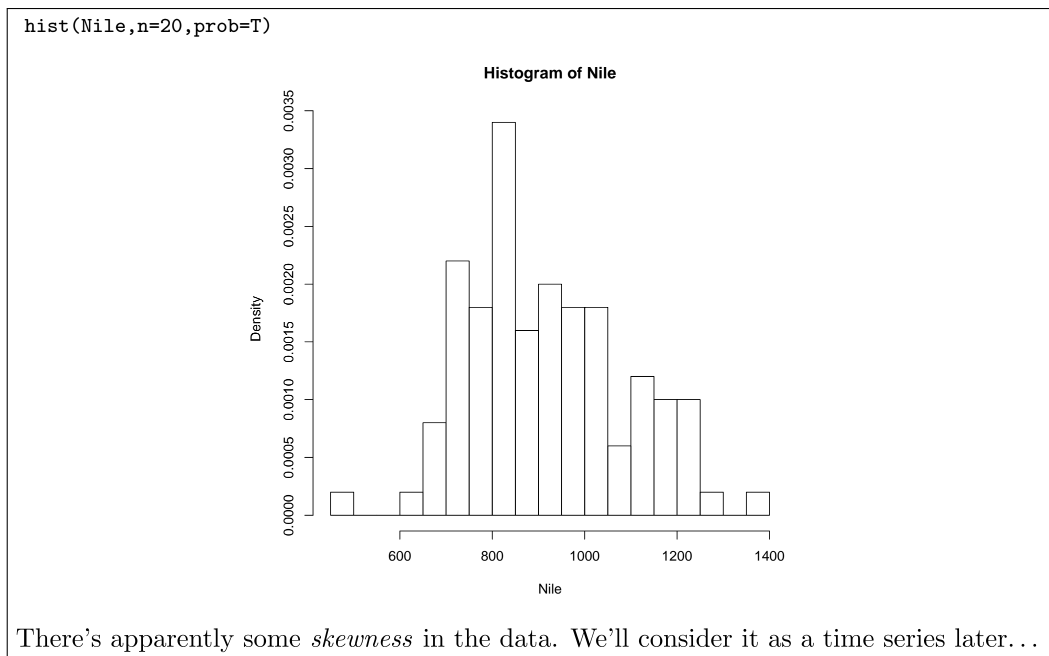
This time it's easy to spot that something is amiss. With other parameter values you'll see different behaviour; the performance of linear congruential generators various from terrible to mediocre. You shouldn't use any of them, but some are certainly better than others. . .

This question really serves to illustrate the importance of a good PRNG and the difficulty of spotting the failings of bad ones. You have access to good PRNGs and R has good default settings so you shouldn't need to worry about this for most purposes. If you implement statistical software using other platforms, particularly parallel ones, then this *is* something which it's important to keep in mind.

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. More pragmatically, the `boot` library provides a function `boot` to obtain bootstrap samples and another, `boot.ci` which will provide various bootstrap confidence intervals.

   (a) The `Nile` dataset shows the rivers. Use a histogram or other visualisation to briefly explore this data.

```
hist(Nile,n=20,prob=T)
```



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

   (b) What's the mean and median length of a river in this category?

```
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))` Estimate variance.

`c(qnorm(0.025),qnorm(0.975))*sigma.hat + mean(Nile)` Compute interval.

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 which takes 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.

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:

```
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:

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

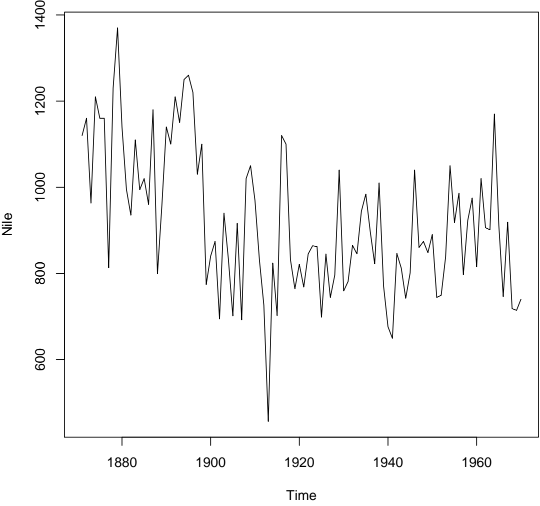
Interesting is a relative term and you might identify other things, but something which I find interesting is that the BPIs are not symmetric around the mean or median and this reflects the fact that even the mean of a small sample of non-normal random variables is not normally distributed.

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

This depends on the sample size you use, there is a reasonable degree of stability with the sample size used here.

(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 it 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.

```
\lstinline{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=3␣000')
lines(seq,punif(seq),col='red')

plot.ecdf(F10000,main='Sample␣Size␣n=10␣000')
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 (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:

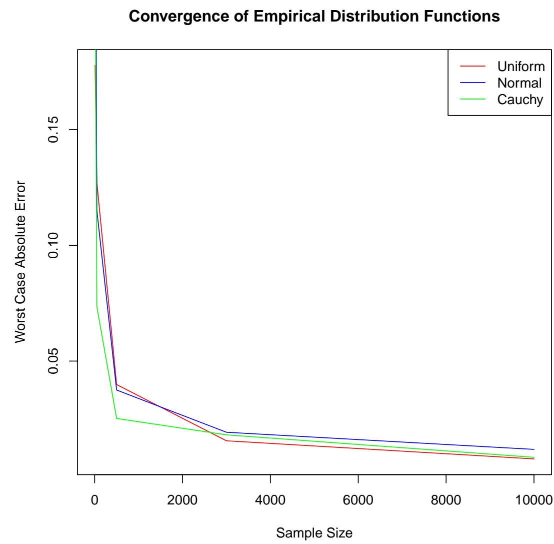
```
eu <- c(10, max(abs(F10(ru[1:10]) - punif(ru[1:10]))))
eu <- rbind(eu, c(50, max(abs(F50(ru[1:50]) - punif(ru[1:50])))))
eu <- rbind(eu, c(500, max(abs(F500(ru[1:500]) - punif(ru[1:500])))))
eu <- rbind(eu, c(3000, max(abs(F3000(ru[1:3000]) - punif(ru[1:3000])))))
eu <- rbind(eu, c(10000, max(abs(F10000(ru[1:10000])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

$\star$ Indicates questions which can easily be done later if you're short of time.