M5MS04 - Computational Statistics; Coursework 3 - Autumn 2018

Question 1

Q1 (a)

Let F be the cumulative density function

$$F(x) = \begin{cases} 0 \text{ for } x < 0 \\ (1 - e^{-x})/(1 - e^{-3}) \text{ for } x \in [0, 3] \\ 1 \text{ for } x > 3 \end{cases} \tag{1}$$

To sample X according to this CDF using the inversion method, we generate a sample $U \sim \text{Uniform}(0, 1)$ then set $X = F^{-1}(U)$, where F^{-1} is the inverse CDF for F,

$$F^{-1}(u) = -\log(1 - u(1 - e^{-3})) \text{ for } u \in [0, 1] \tag{2}$$

This method was implemented using the following code:

```
rm(list=ls())
set.seed(2304897)

F.for <- function(x) {
            if( x >= 0 && x <= 3) return((1 - exp(-x))/(1 - exp(-3)))
            if( x < 0 ) return(0)
            if (x > 3) return(1)
}

F.inv <- function(u) return( -log(1 - u*(1 - exp(-3))) )

U1 <- runif(10^2)
X1 <- F.inv(U1)
U2 <- runif(10^4)
X2 <- F.inv(U2)
```

The empirical cumulative distribution function (ecdf) for samples X_1 and X_2 are plotted against the cdf $F(x)$ in Figure 1. The smaller sample size leaves some doubt as to the similarity of the ecdf and the cdf, however including the larger sample size removes this, since its ecdf is in close agreement with $F(x)$.

Q1 (b)

The Monte Carlo estimates of $\text{E}[X]$ and $\text{E}[\sin(X)]$ are

$$\text{E}[X] \approx \frac{1}{n} \sum_{i=1}^{N} X_i \tag{3}$$

$$\text{E}[\sin X] \approx \frac{1}{n} \sum_{i=1}^{N} \sin(X_i) \tag{4}$$

where X_i are i.i.d. samples from generated by a sampler for F, in this case the inverse sampler of Q1 (a). To check the results of these approximations using numerical integration it was necessary to derive the density function $F'(x)$,

$$F'(x) = f(x) = \frac{1}{1 - e^{-3}} e^{-x} \tag{5}$$

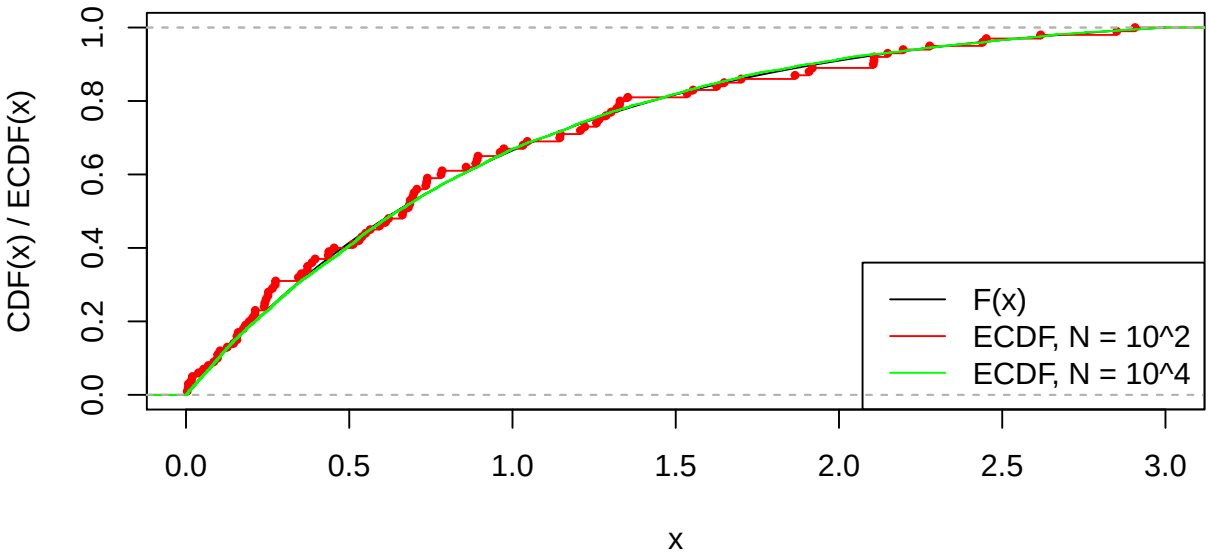

Figure 1: Q1 (a). ecdf versus cdf for sample sizes $N_1 = 10^2$ and $N_2 = 10^4$. We can see that the ecdf of samples from the inverse sampler are in good agreement with the cdf $F(x)$. The graph of $F(x)$ is difficult to make out because it is obscured by the ecdf for N_2.

for $x \in [0, 3]$ and returning zero density otherwise.

The code block below computes the Monte Carlo estimates using sample size $N = 10^4$, then calculates the associated asymptotic confidence intervals. It then evaluates $\mathrm{E}[X]$ and $\mathrm{E}[\sin X]$ using numerical integration over $f(x)$, before finally printing all results for comparison.

```
# Monte Carlo estimates
set.seed(39074)

N <- 10^4
X.MC1 <- F.inv(runif(N))
EX.MCmean    <- mean(X.MC1)
EX.MCconfint <- EX.MCmean + sqrt(var(X.MC1)/N)*qnorm(p=c(0.025, 0.975))

X.MC2 <- F.inv(runif(N))
EsinX.MCmean    <- mean(sin(X.MC2))
EsinX.MCconfint <- EsinX.MCmean + sqrt(var(sin(X.MC2))/N)*qnorm(p=c(0.025, 0.975))

# Numerical integration
f <- function(x) { # Density function F'(x)
  if(x < 0 || x > 3) return(0*x)
  if(x >= 0 && x <= 3) return(exp(-x)/(1- exp(-3)))
}
print(integrate(f, lower=-Inf, upper=Inf)$value) # Check it's a density function

## [1] 1

EX.NI <- integrate(function(x) return(x*f(x)), lower=0, upper=3)
EsinX.NI <- integrate(function(x) return(sin(x)*f(x)), lower=0, upper=3)
```

```r
# Compare results
print(EX.NI)
```

```
## 0.8428129 with absolute error < 9.4e-15
```

```r
print(EX.MCmean)
```

```
## [1] 0.8561415
```

```r
print(EX.MCconfint)
```

```
## [1] 0.8421681 0.8701148
```

```r
print(EsinX.NI)
```

```
## 0.5484365 with absolute error < 6.1e-15
```

```r
print(EsinX.MCmean)
```

```
## [1] 0.5489562
```

```r
print(EsinX.MCconfint)
```

```
## [1] 0.5427113 0.5552011
```

In both cases, numerical integration and Monte Carlo integration agree with one another. The confidence intervals for the Monte Carlo approximations contain the expected values obtained by numerical integration, which is encouraging. In practice, numerical integration would be preferable since it provides the exact expected value.

Q1 (c)

For a 95% confidence interval of length $l = 10^{-6}$ on the Monte Carlo estimate of $\mathrm{E}[\sin X]$, a sample size of approximately $10^{12.2}$ is required. This can be seen by noting that the length of the asymptotic confidence interval for the sample mean of $\sin X$ is

$$l = 2\sqrt{\frac{\mathrm{Var}(\sin X)}{N}} z_{1-\alpha} \tag{6}$$

where $\alpha = 0.025$ in this case. Rearranging and introducing the relevant inequality, we see that we need

$$N \geq \frac{4\,\mathrm{Var}(\sin X) z_{1-\alpha}^2}{L^2} \tag{7}$$

if $l \leq L$, with L strictly positive. Since the variance of $\sin X$ is not known, we substitute it for the sample variance. The code below determines the value of N, N_2, for $L = 10^{-6}$, along with checking that the calculation has been performed correctly by inverting the confidence interval length expression for $N_1 = 10^4$.

```r
l1 <- 2*sqrt(var(sin(X.MC2))/N)*qnorm(p=0.975)# Verify with the current length
l2 <- 10^-6
N1 <- var(sin(X.MC2))/(l1/(2*qnorm(1-0.025)))^2
N2 <- var(sin(X.MC2))/(l2/(2*qnorm(1-0.025)))^2
print(N1) # Agrees with N = 10^4
```

```
## [1] 10000
```

```r
print(N2) # 1.576 x 10^12, rounded to log(N2, base=10) = 10^12.2
```

```
## [1] 1.559943e+12
```

```r
print(log(N2, base=10))
```

```
## [1] 12.19311
```

The final section of this question asked us to estimate how long the inverse sampler would take to run with $N = 10^{12.2}$ on the computer I'm working on. Assuming that the runtime scaled linearly in N, the algorithm would take a little under two days to run on this computer. This would not be practical. The algorithm could be sped up by generating samples across multiple cores simultaneoulsy (i.e. by parallelizing).

As can be seen in the code block below, I estimated the runtime to be approximately 48 hours. I obtained this estimate by timing the algorithm with $N = 10^6$, then scaling that time by $10^{6.2}$ (since $10^{6.2}$ times the runtime for $N = 10^6$ is the runtime for $N = 10^{12.2}$, assuming that the algorithm's execution time scales linearly with N).

```
set.seed(23874)
N <- 10^6
t_1e6  <- system.time(F.inv(runif(N))) ['elapsed']
t_1e12 <- 10^(log(N2, base=10) - 6)*t_1e6
print(as.numeric(t_1e6))          # seconds for N = 10^6
```

```
## [1] 0.11
```

```
print(as.numeric(t_1e12))         # seconds for N approx 10^12.2
```

```
## [1] 171593.7
```

```
print(as.numeric(t_1e12)/(60^2)) # hours for N approx 10^12.2
```

```
## [1] 47.66493
```

```
# N <- 1e12.2
# X <- F.inv(runif(N))
# I dare you.
```

Question 2

Q2 (a)

With $X \sim \mathcal{N}(0,1)$, $Y = X + 20Z$, and $Z \sim t_3$, we can expand $\mathrm{E}[XY]$ into

$$\mathrm{E}[XY] = \mathrm{E}[X(X + 20Z)] \tag{8}$$
$$= \mathrm{E}[X^2] + 20\,\mathrm{E}[XZ] \tag{9}$$

The latter term in this equation is equal to zero. X and Z are independent, so $\mathrm{Cov}(X,Z) = 0$, and $\mathrm{E}\,X = \mathrm{E}\,Z = 0$, allowing us to conclude that $\mathrm{Cov}(X,Z) = \mathrm{E}[XZ] = 0$. Meanwhile, $\mathrm{E}[X^2] = \mathrm{Var}(X) = 1$. Consequently we can conclude that

$$\mathrm{E}[XY] = 1 \tag{10}$$

Q2 (b)

$\mathrm{E}[XY]$ can be estimated via Monte Carlo integration as

$$\mathrm{E}[XY] \approx \frac{1}{n}\sum_{i=1}^{n} X_i Y_i = \frac{1}{n}\sum_{i=1}^{n} X_i(X_i + 20Z_i) \tag{11}$$

where X_i are independent samples from $\mathcal{N}(0,1)$ and Z_i are samples from t_3.

```
set.seed(75903)
N  <- 10^5
X1 <- rnorm(N)
Z  <- rt(N, df=3)
Y  <- X1 + 20*Z
mean(X1*Y) # Monte Carlo estimate
```

```
## [1] 1.05568
```

```
mean(X1*Y) + sqrt(var(X1*Y)/N)*qnorm(p=c(0.025, 0.975)) # Conf. int
```

```
## [1] 0.8410488 1.2703107
```

While the estimate for this specific run is close to the actual expected value, the estimate's confidence intervals reveal that the Monte Carlo estimate is uncertain even for this relatively large sample size.

Q2 (c)

Rao-Blackwellized Monte Carlo integration reduces the variance of the Monte Carlo estimate for $\mathrm{E}[XY]$ by exploiting the Rao-Blackwell theorem. By averaging over the conditional expected value of either X or Z, we can obtain a lower-variance estimate of $\mathrm{E}[XY]$. Here, we can consider one of two conditional expectations:

$$\mathrm{E}[X^2 + 20XZ] = \mathrm{E}[g(X,Z)] \approx \frac{1}{n}\sum_{i=1}^{n} \mathrm{E}[g(X,Z)|Z = Z_i] \tag{12}$$
$$\approx \frac{1}{n}\sum_{i=1}^{n} \mathrm{E}[g(X,Z)|X = X_i] \tag{13}$$

In the first case we would sample Z_i and in the second we would sample X_i. The former estimator is exactly accurate, since

$$\mathrm{E}[g(X,Z)|Z = z_i] = \mathrm{E}[X^2] + 20z_i\,\mathrm{E}[X] = 1 \text{ for all } Z_i \tag{14}$$

The latter, on the other hand, has non-zero variance:

$$\mathrm{E}[g(X,Z)|X = x_i] = x_i^2 + 20x_i\,\mathrm{E}[Z_i] = x_i^2 \tag{15}$$

Presumably we're supposed to condition on X_i, so this is what I chose to do. Code implementing my Rao-Blackwellized Monte Carlo integrator is provided below.

```
# Using notation from the course notes
set.seed(2348) # It seems this is necessary in every code block
N   <- 10^5
phi <- function(X) return(X^2)
X2   <- rnorm(N)
mean(phi(X2)) # Rao-Blackwellized estimate
```

```
## [1] 1.003022
```

```
mean(phi(X2)) + sqrt(var(phi(X2))/N)*qnorm(p=c(0.025, 0.975))
```

```
## [1] 0.9941517 1.0118916
```

These confidence intervals highlight that Rao-Blackwellization yields a massive decrease in the variance of our Monte Carlo estimate. This decrease does not come at the expense of additional compute effort, but does come at the expense of providing domain-specific knowledge (i.e. the conditional expected value expression).

Q2 (d)

```
# Sample standard deviations of either estimate
MC_SE <- sqrt(var(X1*Y)/N) # Monte-Carlo integration
RBMC_SE <- sqrt(var(phi(X2))/N) # Rao-Blackwellized Monte Carlo integration
MC_SE/RBMC_SE # Ratio w/o adjusting for compute
```

```
## [1] 24.19748
```

The Rao-Blackwellized Monte Carlo estimate has a substantially lower variance than the standard Monte Carlo Estimate. This is because the RB estimate does not contain the term $1/n \sum_{i=1}^{N} 20 X_i Z_i$, which would act to increase variance. In terms of computational effort required, the Rao-Blackwellized estimator is preferable. It samples N times, whereas standard Monte Carlo samples $2N$ times - N times from t_3 and N times from $\mathcal{N}(0, 1)$. We might expect that the RBMC algorithm takes around half the time of standard MC. Assuming that these are the only sources of compute time in either algorithm, then the algorithms would take a similar time to run if the RBMC algorithm used twice as many samples as the standard Monte Carlo algorithm. Doubling the sample size would scale the standard error of the RBMC estimate by a factor of $\frac{1}{\sqrt{2}}$. This would mean that the ratio of standard errors would become

```
MC_SE/(RBMC_SE/sqrt(2)) # Ratio after adjusting for compute
```

```
## [1] 34.22041
```

Is the Monte Carlo (MC) algorithm half as fast as the Rao-Blackwellised Monte Carlo (RBMC) algorithm? Let's find out.

```
# Check the timings
set.seed(30854)
N <- 10^7
MC.ftime <- function(x) {rnorm(N); rt(N, df=3)}
RBMC.ftime <- function(x) {rnorm(N)}

MC.time <- system.time(MC.ftime(1))['elapsed']
RBMC.time <- system.time(RBMC.ftime(1))['elapsed']
as.numeric(MC.time/RBMC.time)
```

```
## [1] 3.622951
```

The reality appears to suggest that MC is almost four times slower than RBMC. Speculating, this may be because `rnorm` is more heavily optimised than `rt`. The ratio of standard errors given equal compute time is therefore something closer to 46 in favour of RBMC.

```
as.numeric(MC_SE/(RBMC_SE/sqrt(MC.time/RBMC.time)))
```

```
## [1] 46.05761
```

Question 3

This question requested that we implement an importance sampler for a distribtion with density $f(x, y)$. The specified sampling distribution was $\mathcal{N}(0, 1)$. Talking in terms of one-variable expected values for the sake of clarity, the theory underlying importance sampling exploits the fact that

$$E[\phi(X)] = E[\phi(Y)\frac{h(Y)}{g(Y)}] \tag{16}$$

where $X \sim h$, $Y \sim g$, $\phi(y)h(y) = 0 \implies g(y) = 0$, and Y and X have the same support. g's distribution is referred to as the sampling distribution and h is the target distribution. In this case, the target distribution is f and the sampling distribution g is $\mathcal{N}(0, I_2)$, where I_2 is the rank-2 identity matrix. The function ϕ is:

$$\phi(x, y) = |x^3 + y^3 + 3xy^2 + 3x^2y| \tag{17}$$

My importance sampler is implemented as follows. It samples $K = (K_1, K_2)$ from $\mathcal{N}(0, I_2)$, then computes the joint density of the draw according to f and according to g. In the code, the latter density is computed as a product `prod` of two univariate normal densities. The weight ('importance') for the drawn K is then computed as $w = f(K_1, K_2)/g(K_1, K_2)$. The weighted value of the $\phi(K_1, K_2)$, $w\phi(K_1, K_2)$ is then returned (NB in the code `phi1` refers to ϕ).

To verify that the importance sampler was functioning correctly, I tested its output under an alternative function, $\phi_2(x, y) = e^{-(x^2+y^2)}$, against the output of a numerical integration (courtesy of the package `rmutil`). As we will see, ϕ is pathological.

```
rm(list=ls())
set.seed(896134)

f <- function(x, y){
  return( (sqrt(2)/pi^1.5)*exp(-0.5*(x - y)^2)/( 1 + (x + y)^2) )
}

# install.packages('rmutil')
library(rmutil)

##
## Attaching package:  'rmutil'

## The following object is masked from 'package:stats':
##
##    nobs

## The following objects are masked from 'package:base':
##
##    as.data.frame, units

int2(f, a = c(-Inf, -Inf), b = c(Inf, Inf)) # verify f is correct

## [1] 1

phi1 <- function(x, y) return( abs(x^3 + y^3 + 3*x^2*y + 3*x*y^2) )
phi2 <- function(x, y) return( exp(-(x^2 + y^2)) ) # to verify the sampler

f.Isample <- function(phi, g=dnorm, r=rnorm, param=c(mean=0, sd=1)){
  K   <- r(n=2, param) # Sample from sampling distribution
  g_K <- prod(g(K, param))
  f_K <- f(K[1], K[2])
  w   <- f_K/g_K # Compute importance
  return(w*phi(K[1], K[2])) # Return weighted function value
}
```

To evaluate the importance sampler, I ran it five times under both ϕ_1 and ϕ_2.

```
N <- 10^4

# int2(function(x, y) return(phi1(x, y)*f(x, y)),
#      a = c(-Inf, -Inf), b = c(Inf, Inf)) # Does not terminate (!)
for(i in 1:5){ # but not for phi1
  print(mean(replicate(N, f.Isample(phi=phi1))))
}
```

```
## [1] 8.90481
## [1] 12.0016
## [1] 21.2252
## [1] 9.161853
## [1] 7.19912
```

```
int2(function(x, y) return(phi2(x, y)*f(x, y))) # E[phi_2(X, Y)]
```

```
## [1] 0.3699276
```

```
for(i in 1:5){ # Estimates are stable for phi2
  print(mean(replicate(N, f.Isample(phi=phi2))))
}
```

```
## [1] 0.3746972
## [1] 0.3766691
## [1] 0.3644088
## [1] 0.354489
## [1] 0.373941
```

The second set of results reveals that the sampler run under ϕ_2 generates estimates that are stable and in reasonably good agreement with the expected value as obtained by numerical integration. The first set of results, on the other hand, highlight that for ϕ_1 (the function specified in the question) the importance sampler's estimates are unstable. The reason for this can be seen in Figure , where $\phi(x, y)f(x, y)$ increases as we move along the axis $y = x$ away from the origin. This, and analytical considerations (which are not included here due to time pressures) suggests that this expectation may either be infinite or be too large to represent directly on a machine. The general implementation of the importance sampler above (which, as its arguments reveal, accepts alternative sampling distributions) was run with a variety of alternative sampling distributions (Cauchy and the t-distribution), but continued to give unstable answers.

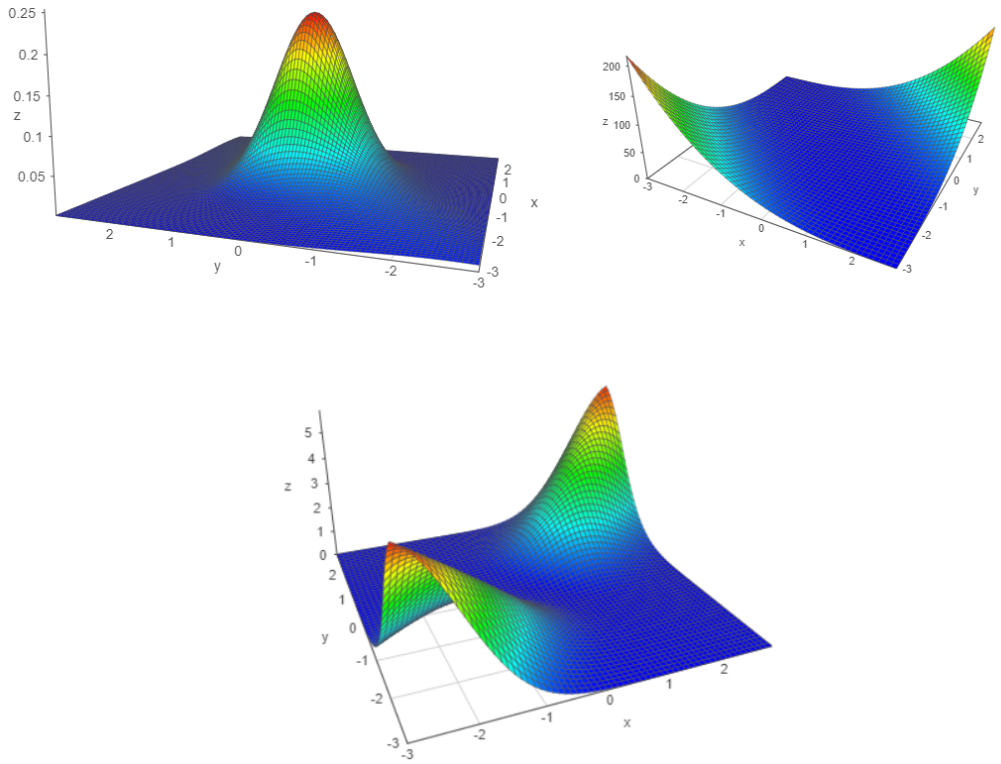

Figure 2: (Left) Joint density function $f(x, y)$. (Right) $\phi(x, y)$. (Center) Product of these functions, $f(x, y)\phi(x, y)$.

Question 4

In this question we were asked to evaluate a pseudo-random number generator (PRNG) developed by our lecturer, Professor Gandy, by inspecting a vector x provided in a data file. If the generator performed as intended, then each entry of the vector x would be an i.i.d. draw from Uniform(0, 1).

Initially, it looks as though the generator is performing as intended: the ecdf of the sample matches that of Uniform$(0, 1)$ quite closely, as can be seen in Figure 3. There is a minor deviation visible.

Run a KS test to assess whether this deviation is statistically significant.

```
ks.test(x, punif)
```

```
##
##  One-sample Kolmogorov-Smirnov test
##
## data:  x
## D = 0.02524, p-value = 0.1564
## alternative hypothesis: two-sided
```

The p-value is sufficiently large that we cannot reject the null hypothesis that the sample was drawn from Uniform$(0, 1)$.

Inspecting the ecdf isn't sufficient to conclude that Professor Gandy's proposed PRNG is up to scratch. We need to verify that the samples are independent. If they were independent, then they would also be uncorrelated. This would imply that the sample autocorrelation as computed over x would be approximately 0. The sample autocorrelation values are, in reality:

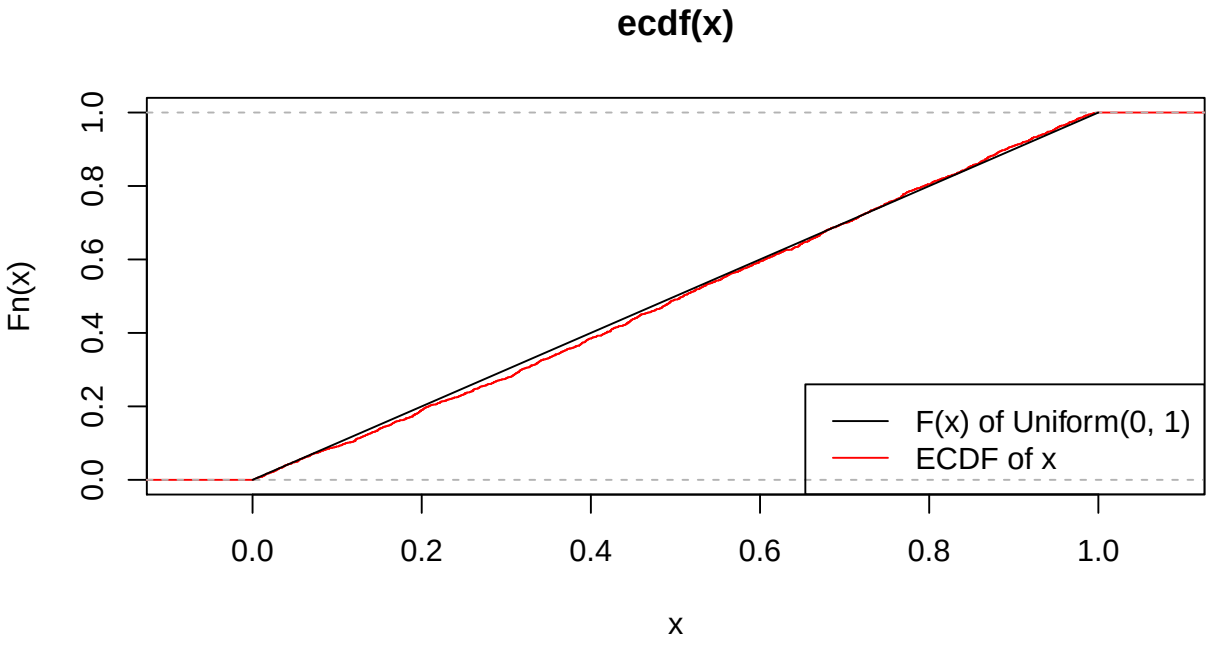

Figure 3: This is a figure caption.

```
##
## Autocorrelations of series 'x', by lag
##
##      0      1      2      3      4      5      6      7      8      9
##  1.000  0.245  0.023  0.021  0.015  0.010 -0.013 -0.019 -0.050 -0.027
##     10
## -0.016
```

It appears that there is non-zero autocorrelation for a lag value of $\tau = 1$ - adjacent values in x are positively correlated. This is clear from inspecting a plot of adjacent values, as is shown in Figure 4. The distribution has a pronounced peak in the middle, possibly indicating a wrapped Laplace distribution.

```
## Warning:  Removed 1 rows containing missing values (geom_point).
```

The PRNG proposed by the lecturer does not generate pseudo-random numbers since the values are autocorrelated. I think this data was probably generated by initializing $x_1 = \epsilon_1$ then letting $x_t = x_{t-1} + \epsilon_t$, with ϵ_t being a random deviate drawn according to a wrapped unimodal distribution on the unit interval, centred on 0 with a standard devation ≈ 0.35.

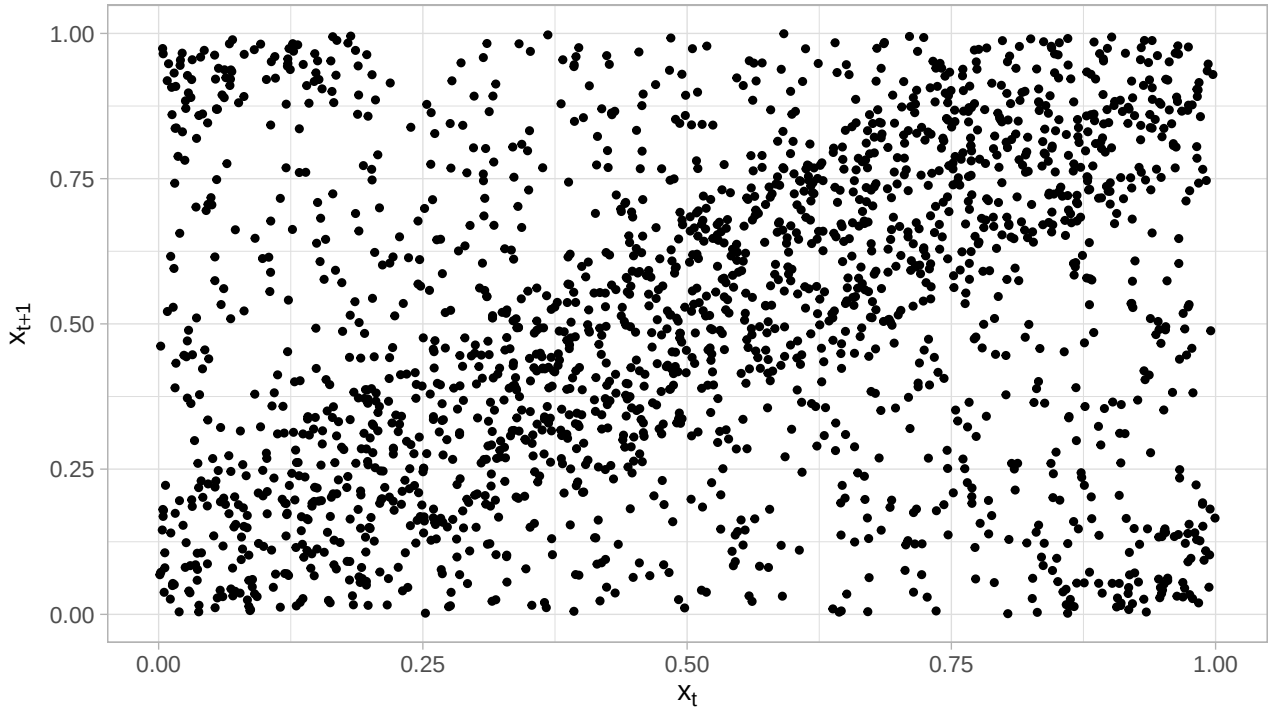

Figure 4: Plot of x_{t+1} versus x_t to reveal positive correlation between adjacent values of x.

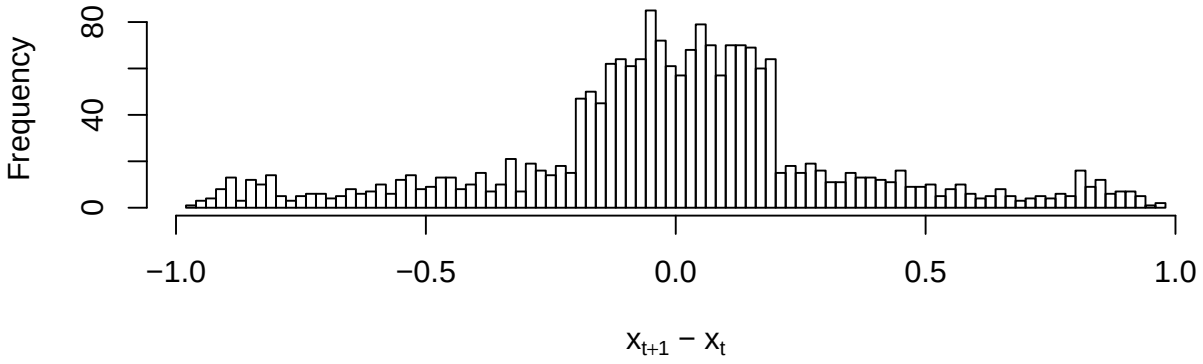

Figure 5: Histogram of x's lag-1 differences, $x_{t+1} - x_t$.