SS2864B Assignment 4

Ali Al-Musawi

13/03/2020

Question 1

(a)

```r
directpoly = function(x, coef) {
  if (length(coef) < 2) stop("input coef must have length at least 2")
  result = numeric(length(x))
  # compute the vector c(P(x_1), P(x_2), ..., P(x_m))
  # where P(x_i) = (c_1) + (c_2)*(x_i) + (c_3)*(x_i)^2 + ... + (c_n)*(x_i)^(n-1)
  for (i in 1:length(coef)) {
    result = result + coef[i]*(x^(i-1))
  }
  return(result)
}

# test the function
directpoly(1:3, c(2, 17, -3))
```

```
## [1] 16 24 26
```

(b)

```r
hornerpoly = function(x, coef) {
  if (length(coef) < 2) stop("input coef must have length at least 2")
  n = length(coef)
  result = rep(coef[n], n)
  # compute the vector c(P(x_1), P(x_2), ..., P(x_m))
  # using horner's method
  for (i in (n-1):1) {
    result = result*x + coef[i]
  }
  return(result)
}

# test the function
hornerpoly(1:3, c(2, 17, -3))
```

```
## [1] 16 24 26
```

(c)

```r
# Test the running time of algorithms in parts a and b
system.time(directpoly(x=seq(-10,10, length=5000000), c(1,-2,2,3,4,6,7,8)))
```

```
##    user  system elapsed 
##    2.24    0.19    2.70
```

```
system.time(hornerpoly(x=seq(-10,10, length=5000000), c(1,-2,2,3,4,6,7,8)))
```

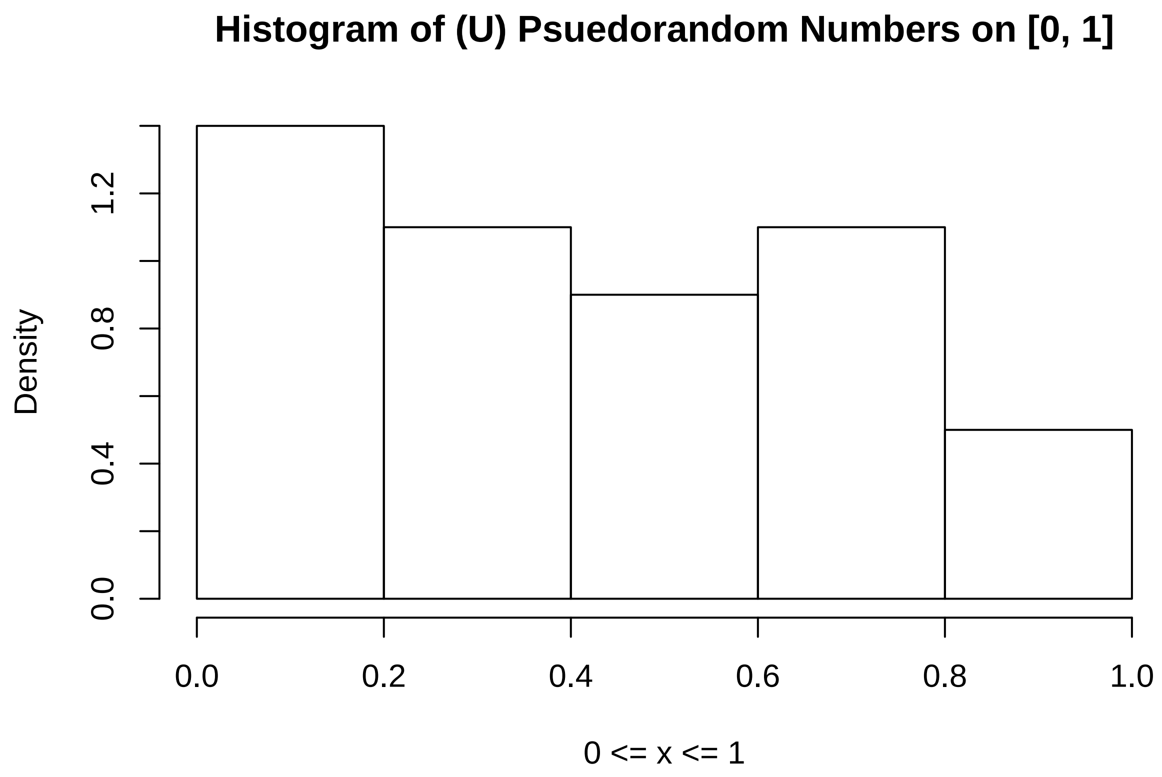
```
##    user  system elapsed 
##    0.12    0.07    0.20
```

Based on the output, Horner's method is much faster. According to the output produced on my computer, Horner's method is ~20 times faster.

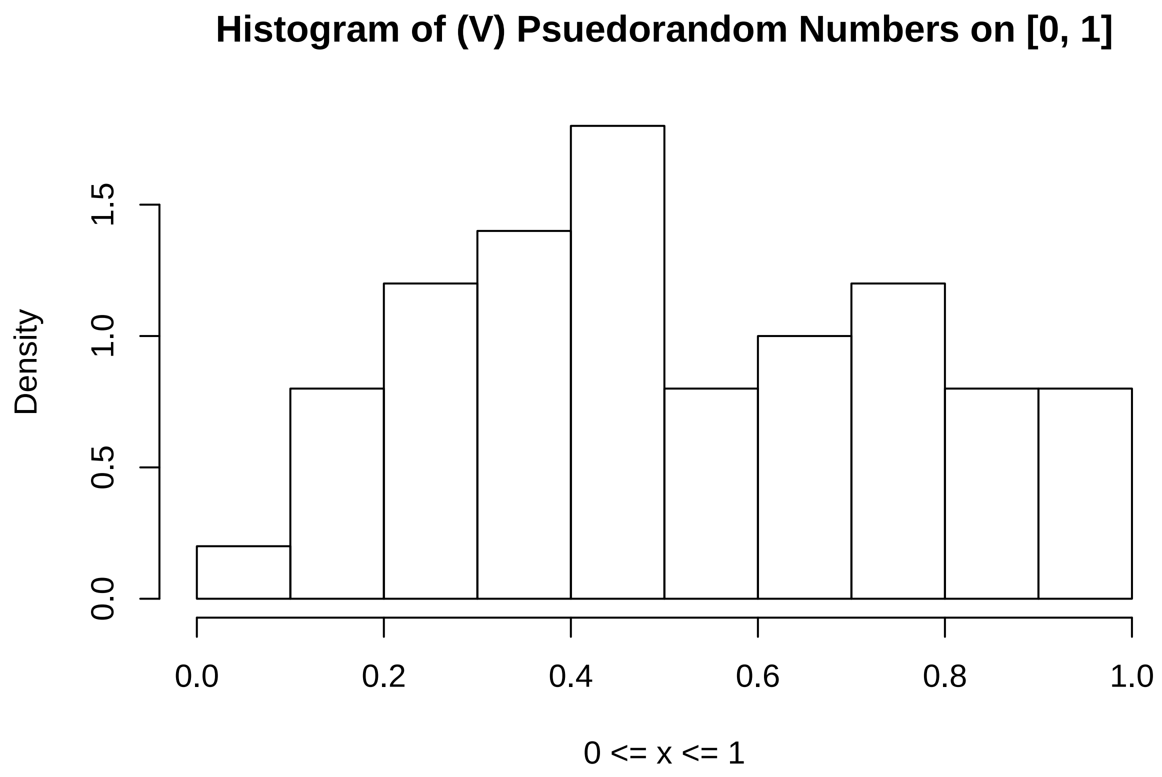
Question 2

(a)

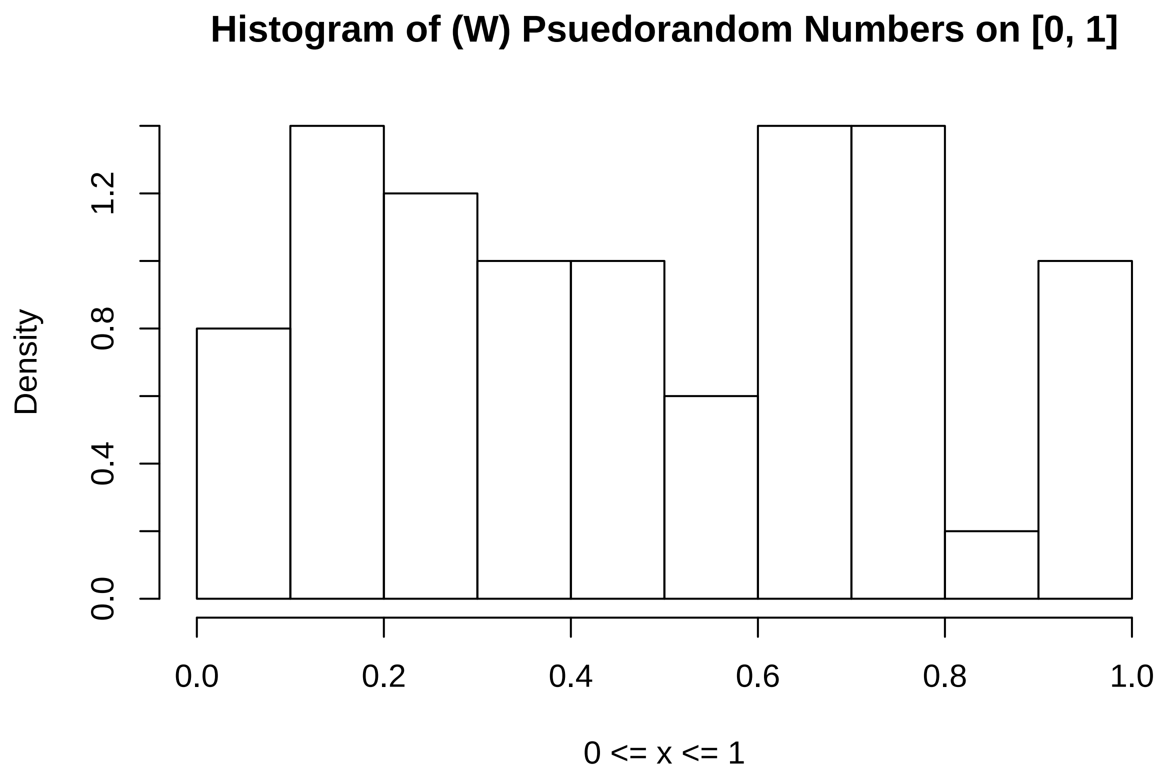
```
my.unif = function(n, a, c = 0, m, x0) {
  x = numeric(n)
  # set the initial seed
  x[1] = (a*x0 + c) %% m
  # compute the sequence of pseudorandom numbers
  for(i in 2:n) {
    x[i] = (a*x[i-1] + c) %% m
  }
  return(x/m)
}

# test the function
U = my.unif(50, 172, 13, 30307, 17218)
hist(U, main = 'Histogram of (U) Psuedorandom Numbers on [0, 1]', xlab = '0 <= x <= 1', freq = F)
```



```r
V = my.unif(50, 171, 51, 32767, 2020)
hist(V, main = 'Histogram of (V) Psuedorandom Numbers on [0, 1]', xlab = '0 <= x <= 1',freq = F)
```



```
W = runif(50)
hist(W, main = 'Histogram of (W) Psuedorandom Numbers on [0, 1]', xlab = '0 <= x <= 1',freq = F)
```



Indeed, by comparing our pseudo-random variables with the R generated uniform random variable, we see that the output is close to uniform for the second case, while the first one looks like has a lower probability as x increases to 1.0

Question 3

```r
n = 1000
set.seed(2020)
U = runif(n)
```

(a-b)

```r
# Compute summary statistics of elements of U using R functions
Rmean = mean(U)
names(Rmean) = "Sample Mean"
Rvar = var(U)
names(Rvar) = "Sample Variance"
Rsd = sd(U)
names(Rsd) = "Sample Standard Deviation"

# Compute distribution moments using population parameters
Tmean = 0.5*(0+1)
names(Tmean) = "Theoretical Mean"
Tvar = 0.5*(1-0)^2
names(Tvar) = "Theoretical Variance"
Tsd = sqrt(Tvar)
```

```r
names(Rsd) = "Theoretical Standard Deviation"

# Display the results
Rmean
```

```
## Sample Mean
##    0.492167
```

```r
Tmean
```

```
## Theoretical Mean
##              0.5
```

```r
Rvar
```

```
## Sample Variance
##      0.08345052
```

```r
Tvar
```

```
## Theoretical Variance
##                  0.5
```

```r
Rsd
```

```
## Theoretical Standard Deviation
##                       0.288878
```

```r
Tsd
```

```
## Theoretical Variance
##            0.7071068
```

```r
# Compute absolute difference
difference = abs(c(Rmean - Tmean, Rvar - Tvar, Rsd - Tsd))
names(difference) = c("Error in Mean", "Error in Variance", "Error in Standard Deviation")

# Display the difference
difference
```

```
##               Error in Mean           Error in Variance
##                 0.007832988                 0.416549484
## Error in Standard Deviation
##                 0.418228752
```

(c)

```r
# The empirical probability
Rp = mean(U < 0.6)
names(Rp) = "The empirical probability"
# The theoretical probability
Tp = (0.6 - 0)/(1 - 0)
names(Tp) = "The theoretical probability"

# Display the results
Rp
```

```
## The empirical probability
##                     0.622
```

```
Tp
```

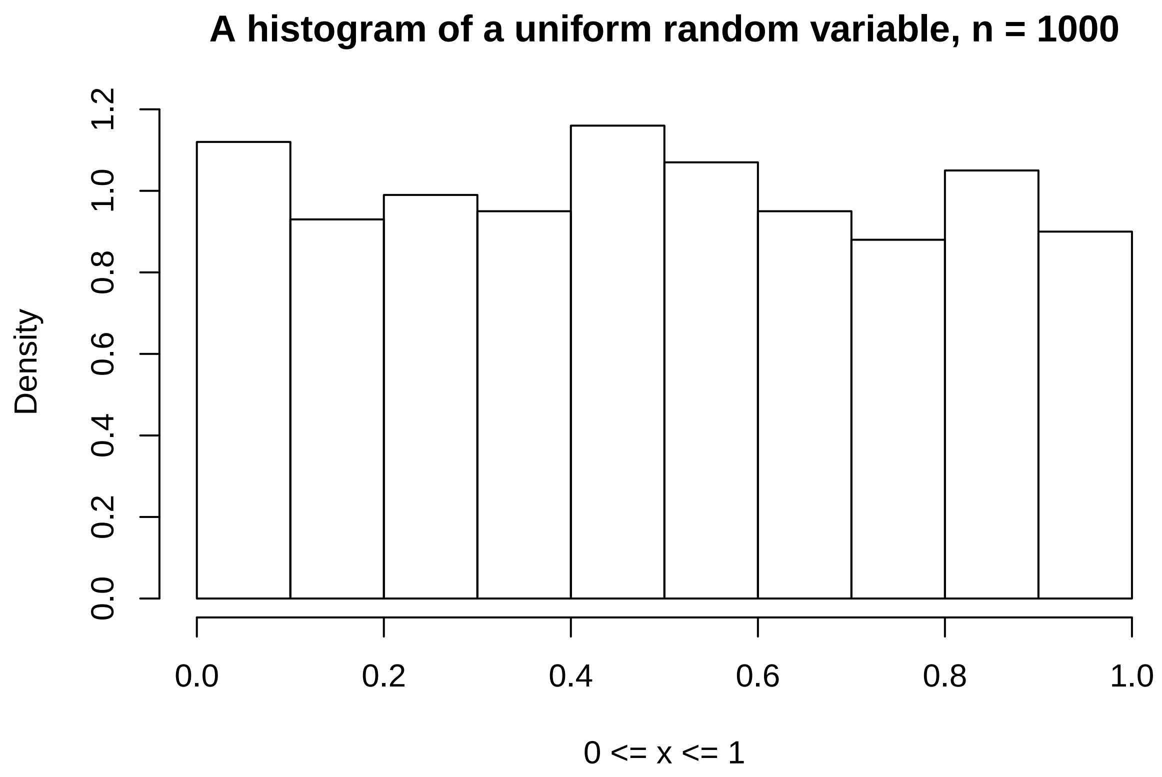
```
## The theoretical probability
##                         0.6
```

```
# Compute and Display the difference
d = Rp - Tp
names(d) = "Difference between empirical and theoretical probability"
d
```

```
## Difference between empirical and theoretical probability
##                                                     0.022
```

(d)

```
hist(U, xlab = '0 <= x <= 1', main = 'A histogram of a uniform random variable, n = 1000', freq = F)
```



Based on the output, the bars seem to be flattening around 1.0, which is very close to theoretical uniform distribution on [0, 1].

Question 4

(a)

Note for this question, we need to find the quantile function of $f(x)$ in order to use it to generate samples from its associated distribution. Here, we show our steps:

$$F(x) = \int_0^x \frac{y}{a^2} e^{-\frac{y^2}{2a^2}} dy$$

Let $u = -\frac{y^2}{2a^2}$:

$$F(x) = \int_0^{-\frac{x^2}{2a^2}} -e^u du$$

$$\therefore F(x) = 1 - e^{-\frac{x^2}{2a^2}}$$

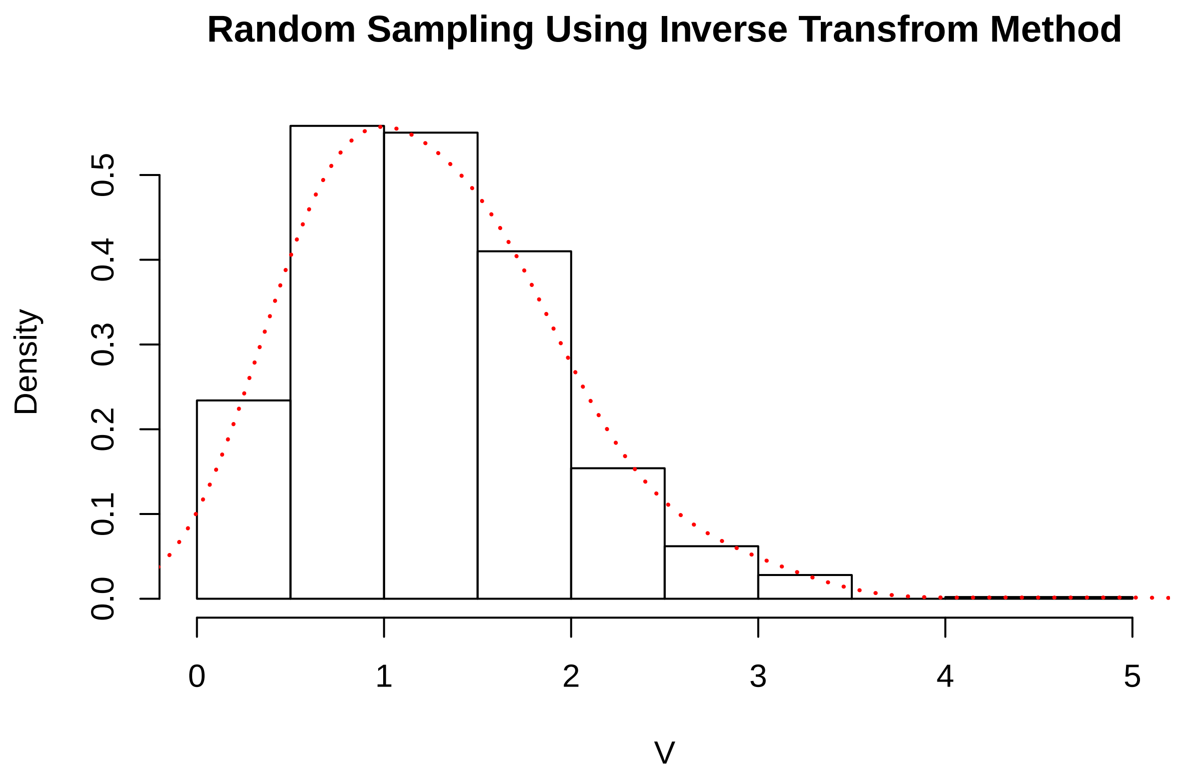
$$p = 1 - e^{-\frac{x^2}{2a^2}} \Rightarrow \ln(1-p) = -\frac{x^2}{2a^2} \Rightarrow x = \sqrt{-2a^2\ln(1-p)} = Q(p)$$

Note that the square root argument is 0 or positive. Note, that $F(x)$ is not one-to-one on $\mathbb{R}$, but we used the fact that $x \geq 0$.

```
# Defining the inverse cdf as computed above
my.quantile = function(a, p) {
  return(sqrt(-2*a^2*log(1-p)))
}

# Defining inverse transform sampling function
inverse.sampling.f = function(n, a = 1) {
  U = runif(n)
  return(my.quantile(a, U))
}

# Testing the inverse transform sampling function
t = inverse.sampling.f(20)
t
```

```
##  [1] 2.08503762 1.08588493 1.16247215 0.50214397 2.07190391 1.15217797
##  [7] 1.25067820 0.97538133 2.36061828 0.69893589 0.64907776 0.87427798
## [13] 0.90658390 2.06952392 0.08816076 0.61904354 0.32456353 0.63357690
## [19] 1.92441995 0.77417924
```

```
V = inverse.sampling.f(1000)
hist(V, main = 'Random Sampling Using Inverse Transfrom Method', probability = T)
lines(density(V, adjust=2), lty="dotted", col=2, lwd=2)
```



We have a right-skewed distribution whose values range on $[0, 4]$. We have smoothened out the density curve.

(b)

To do this question, we need to implement the rejection-sampling algorithm, so an appropriate choice of M is desired. We will maximize $f(x)$ for this purpose.

$$f'(x) = -\frac{1}{a^4}(x^2 - a^2)e^{-\frac{x^2}{2a^2}}$$

$$f'(x) = 0 \Leftrightarrow x^2 = a^2 \Rightarrow x = a$$

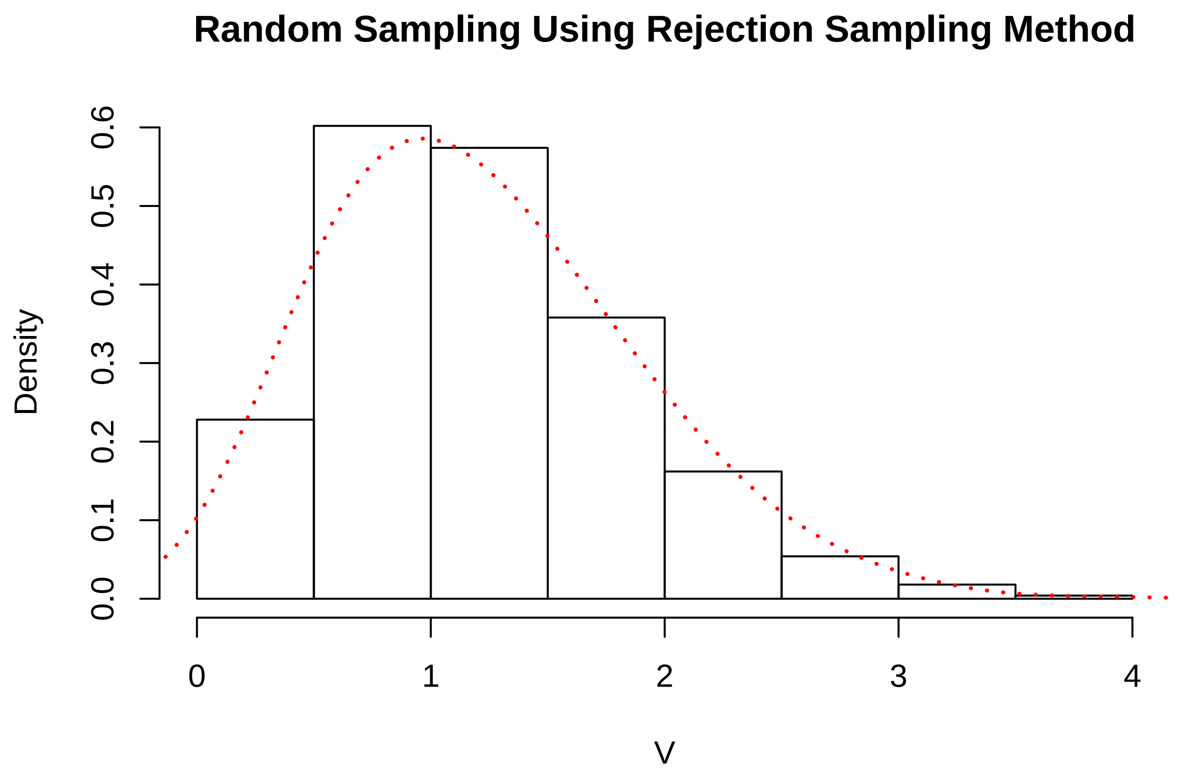
$$\therefore \max(f(x)) = f(a) = \frac{1}{a}e^{-\frac{1}{2}}$$

However, this value is problematic. This is because the probability of acceptance is $1/M$, yet $|e^{-\frac{1}{2}}| < 1$, so $1/M$ will exceed 1. Instead, we set $M = \lceil \max(f(a)) \rceil$.

```
# Define the given density function
my.density = function(x, a = 1) {
  return((x/a^2)*exp(-x^2/(2*a^2)))
}

# Define the rejection sampling function
rejection.sampling.f = function(n, a = 1) {
  # Initialize relevant variables
  x = double(n)
  M = ceiling((1/a)*exp(-0.5))
  init<-0
  # Repeat until the sample size is n
  repeat {
    # Generate observations from a proposal distribution
    t = runif(n-init, 0, 5)
    u = runif(n-init)
    # Test the accepted elements
    accept = (M*u <= my.density(t, a))
    m = sum(accept)
    if (m > 0)
      x[(init+1):(m+init)] = t[accept]
    if ((m+init) == n)
      break
    init = init + m
  }
  return(x)
}

# Test the rejection sampling function
t = rejection.sampling.f(20)
t
```

```
##  [1] 1.4341503 0.9003099 1.6410784 0.3683689 2.0588864 1.3814525 0.4940266
##  [8] 1.7348706 1.1260286 0.7437603 1.3063139 1.2225539 0.7559633 1.7664157
## [15] 2.2538846 1.4644757 1.8593040 0.8710939 2.6585598 0.8412694
```

```
V = rejection.sampling.f(1000)
hist(V, main = 'Random Sampling Using Rejection Sampling Method', probability = T)
lines(density(V, adjust=2), lty="dotted", col=2, lwd=2)
```



Based on the output generated, we get the exact same distribution as the inverse-transform method.

(c)

```
system.time(inverse.sampling.f(100000))
```

```
##    user  system elapsed 
##       0       0       0 
```

```
system.time(rejection.sampling.f(100000))
```

```
##    user  system elapsed 
##    0.06    0.00    0.06 
```

We see that the inverse transform method is 10 times faster. This is due to the fact that the rejection sampling method uses a repeat structure, while the inverse method is completely vectorized.

Question 5

```
n = 10000
u = runif(n)
```

Recall that if $U \sim \mathcal{U}(a,b)$, then $\mathbb{E}[U^2] = \frac{a^2+ab+b^2}{3}$. Let $\mu^{(2)} = \mathbb{E}[X^2]$. Then we need to estimate it using a CI. ### (a)

```
mu2.actual = 1/3
u2 = u^2
mu2.estimated = mean(u2)
```

```r
mu2.lower = mu2.estimated-1.96*sd(u)/sqrt(n)
mu2.upper = mu2.estimated+1.96*sd(u)/sqrt(n)
# compute the error
error = mu2.actual - mu2.estimated
names(error) = 'actual - estimated'
error
```

```
## actual - estimated
##       0.0008026875
```

```r
# display the ci
ci = list(lower = mu2.lower, second.moment = mu2.estimated, upper = mu2.upper)
ci
```

```
## $lower
## [1] 0.326888
##
## $second.moment
## [1] 0.3325306
##
## $upper
## [1] 0.3381733
```

Note that the error is very low, and the CI is narrow.

(b)

```r
v = (u^2 + (1 - u)^2)/2
mu2.estimated = mean(v)
mu2.lower = mu2.estimated-1.96*sd(u)/sqrt(n)
mu2.upper = mu2.estimated+1.96*sd(u)/sqrt(n)
# compute the error
error = mu2.actual - mu2.estimated
names(error) = 'actual - estimated'
error
```

```
## actual - estimated
##       0.0004603763
```

```r
# display the ci
ci = list(lower = mu2.lower, second.moment = mu2.estimated, upper = mu2.upper)
ci
```

```
## $lower
## [1] 0.3272303
##
## $second.moment
## [1] 0.332873
##
## $upper
## [1] 0.3385156
```

The error this time is smaller than the case above.

(c)

```
v = ((u/2)^2 + (1 - u/2)^2)/2
mu2.estimated = mean(v)
mu2.lower = mu2.estimated-1.96*sd(u)/sqrt(n)
mu2.upper = mu2.estimated+1.96*sd(u)/sqrt(n)
# compute the error
error = mu2.actual - mu2.estimated
names(error) = 'actual - estimated'
error
```

```
## actual - estimated
##       2.951627e-05
```

```
# display the ci
ci = list(lower = mu2.lower, second.moment = mu2.estimated, upper = mu2.upper)
ci
```

```
## $lower
## [1] 0.3276612
##
## $second.moment
## [1] 0.3333038
##
## $upper
## [1] 0.3389465
```

The error is much much lower than the last two cases. This is the most accurate estimate.