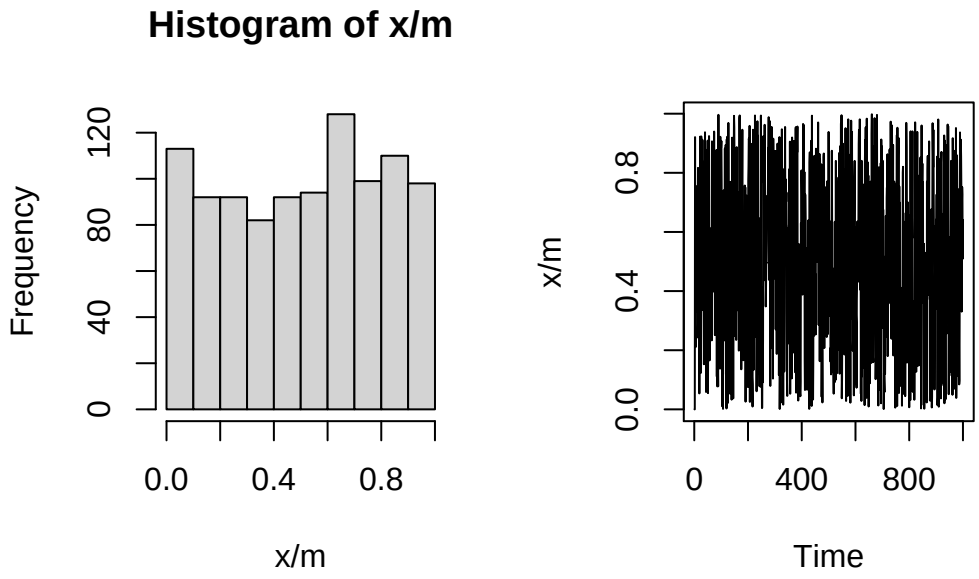
MTH210a: Lab 1 Solutions

1. **Read through the code in `PRNG.R` that attempts to implement the multiplicative congruential method to generate pseudo-random numbers. (Some values are missing in those.) Try various values of m, a, and the seed x_0, and assess the performance of the generator.**

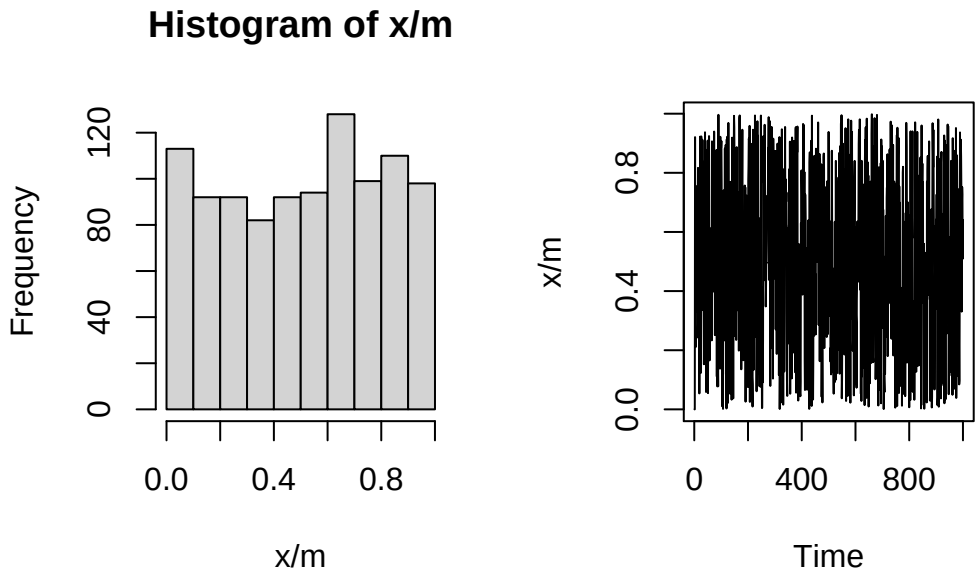
   We use the recommend choices of a and m and I make a choice of x_0.

```
##########################################
## Pseudo-random number generation
## using multiple congruential method
##########################################
m <- 2^31 - 1  # choose a value you want
a <-  7^5 # choose a value you want
x <- numeric(length = 1e3)
x[1] <- 7  #x0 -- choose
for(i in 2:1e3)
{
  x[i] <- (a * x[i-1]) %% m
}

# We will visualize using histograms (for testing uniformity)
# and trace plots (for checking independence)
par(mfrow = c(1,2))
hist(x/m) # looks close to uniformly distributed
plot.ts(x/m) # look like it's jumping around too
```



2. **Change the code in `PRNG.R` to implement the mixed congruential method. Is there much visible difference between these methods?**

```
##########################################
## Pseudo-random number generation
## using mixed congruential method
##########################################
m <- 2^31 - 1  # choose a value you want
a <-  7^5 # choose a value you want
c <- 13
x <- numeric(length = 1e3)
x[1] <- 7  #x0 -- choose
for(i in 2:1e3)
{
  x[i] <- (a * x[i-1]) %% m
}

# We will visualize using histograms (for testing uniformity)
# and trace plots (for checking independence)
par(mfrow = c(1,2))
hist(x/m) # looks close to uniformly distributed
plot.ts(x/m) # look like it's jumping around too
```



3. **The file `BinomAR.R` contains partial code for the implementation of the Accept-Reject method for a Binomial(n, p) problem (as discussed in class). The function `draw_binom()` is left incomplete at some place. Complete the function and run the line**

```
draw_binom(n = 10, p = .25)
```

If written correctly, the above line will return

- **an** $X \sim \text{Binom}(10, .25)$
- **the number of times the algorithm looped**

Change the values of n **and** p **and observe what happens. What happens when** n **is very large?**

```
##########################################
## Accept Reject algorithm to draw from
## Binomial(n,p)
##########################################
# setting the seed makes it so that the same sets of
# random variables are realized.
set.seed(1)

# Function draws one value from Binom(n,p)
# n = number of trials
# p = probability of success
draw_binom <- function(n, p)
{
  accept <- 0 # Will track the acceptance
  try <- 0 # Will track the number of proposals
```

```
  # upper bound calculated in the notes
  x <- 0:n
  all_c <- choose(n,x) * (1-p)^(n - 2*x) * p^(x-1)  # from notes
  c <-  max(all_c) + .00001 # what is the value of c ?

  while(accept == 0)
  {
    try <- try + 1

    U <- runif(1)
    prop <- rgeom(1, prob = p) #draw proposal

    ratio <-  dbinom(prop, size = n, prob= p)/(c* dgeom(prop, p))# calculate the ratio
    if(U < ratio)
    {
      accept <- 1
      rtn <- prop
    }
  }
  return(c(rtn, try))
}
draw_binom(n = 10, p = .25)
```

```
[1] 4 1
```

```
###
# If we want X1, ..., Xn ~ Binom(n.p)
# we need to call the function multiple times

# sample size
N <- 1e3
samp <- numeric(N)
n.try <- numeric(N)
for(t in 1:N)
{
   # I use as a dummy variable often
   foo <- draw_binom(n = 10, p = .25)
   samp[t] <- foo[1]
   n.try[t] <- foo[2]
}
mean(samp) #should be n*p = 2.5
```

```
[1] 2.51
```

```
mean(n.try)
```

```
[1] 2.308
```

In the above code `samp` contains 1000 iid draws from Binom$(10, p = .25)$ and `n.try` contains the number of loops required for acceptance in each of the 1000 draws.

Now, I will change values of n and p to see the behavior. Notice that the mean of the draws of `samp` will change, since the target distribution has changed. The number of loops will also change, since the bound c has also changed. We see a similar thing happen for when I change n.

```
# sample size
N <- 1e3   # reducing this
samp <- numeric(N)
n.try <- numeric(N)
for(t in 1:N)
{
   # I use as a dummy variable often
   foo <- draw_binom(n = 10, p = .50)
   samp[t] <- foo[1]
   n.try[t] <- foo[2]
}
mean(samp) #should be n*p = 50
```

```
[1] 4.937
```

```
mean(n.try)
```

```
[1] 30.262
```

```
# sample size
N <- 1e3   # reducing this
samp <- numeric(N)
n.try <- numeric(N)
for(t in 1:N)
{
   # I use as a dummy variable often
   foo <- draw_binom(n = 100, p = .25)
   samp[t] <- foo[1]
   n.try[t] <- foo[2]
}
mean(samp) #should be n*p = 25
```

```
[1] 25.038
```

```
mean(n.try) # when n is large the number of loops is too large!
```

```
[1] 1028.832
```

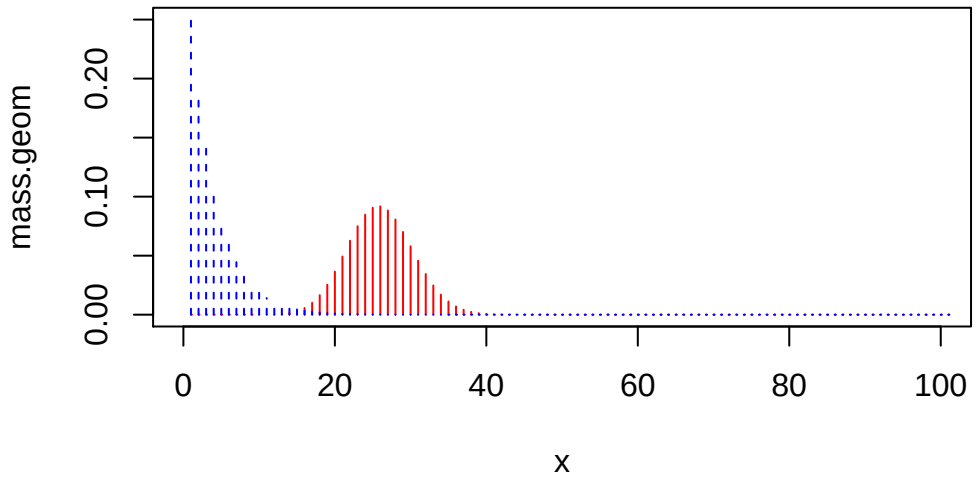
4. **Go over the rest of the code in `BinomAR.R` and observe what happens when the simulation is repeated many times. Further, observe how the performance of the upper bound c changes when different Geometric proposals are used. All of this is implemented in the code.**

```
##########################################
## A closer look at Binomial and Geometric
##########################################
# Turns out, this choice of Binomial and Geometric
# can work, but not always. In the code below,
# increase n to see what happens

p <- .25
n <- 100    # for situation this doesn't work well
x <- 0:(n)
mass.geom <- dgeom(x, p)
mass.bin <- dbinom(x, size = n, prob = p)

all_c <- choose(n,x) * (1-p)^(n - 2*x) * p^(x-1)
(c <- max(all_c))
```

```
[1] 1028.497
```

The value of c is 1028.5. This value is reasonable. This is apparent from the pmf plot below, where in red is the Binomial target, and in blue we have the geometric proposal mass functions.

```
plot(x, mass.geom, pch = 16, col = "red", type= "n")
points(mass.bin, pch = 16, col = "red", type= "h")
points(mass.geom, pch = 16, col = "blue", type = "h", lty = 2)
```

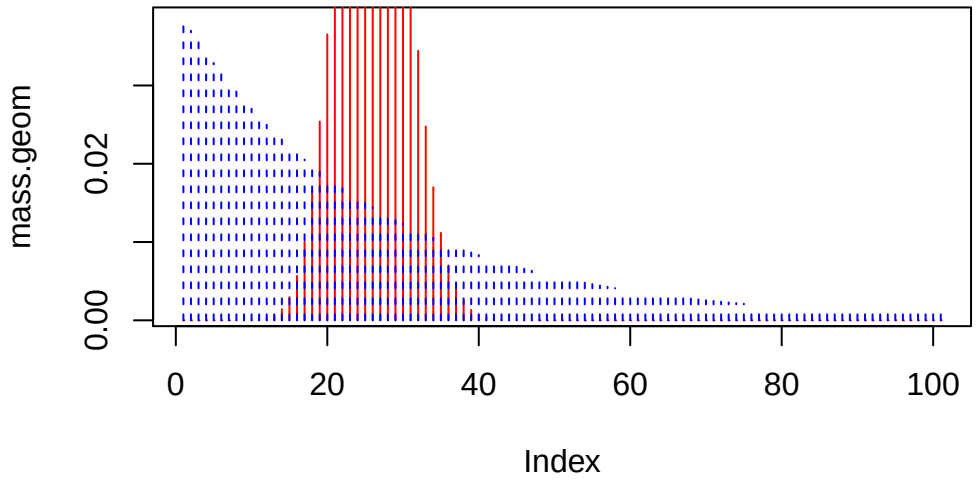


Now, we will change the proposal so that the mean of the geometric matches the mean of the proposal. You can see the value of c is much smaller now, and this is reinforced by the proposal/target comparison plot.

```
# Matching the means:
# choosing p* for rgeom so that np = (1-p*)/p*
p.star <- 1/(n*p + 1)
mass.geom <- dgeom(x, p.star)
mass.bin <- dbinom(x, size = n, prob = p)

all_c <- choose(n,x) * (1-p.star)^(n - 2*x) * p.star^(x-1)
(c <- max(all_c))
```

```
[1] 6.045532
```

```
plot(mass.geom, pch = 16, col = "red", type= "n")
points(mass.bin, pch = 16, col = "red", type= "h")
points(mass.geom, pch = 16, col = "blue", type = "h", lty = 2)
```



5. **Implement Problem 8 from Section 3.4 in the notes in R.**

Simulate from the following "truncated Poisson distribution'' with pmf:

$$\Pr(X = i) = \frac{e^{-\lambda}\lambda^i/i!}{\sum_{j=0}^{m} e^{-\lambda}\lambda^j/j!} \quad i = 0, 1, 2, \dots, m\,.$$

Implement in R with $m = 20$ and $\lambda = 20$.

To implement AR for this, we choose proposal Poisson distribution. To run code for this, we have to do some theory first. The proposal pmf is

$$q_i = \Pr(Y = i) = \frac{\lambda^i}{i!}e^{-\lambda} \qquad i = 0, 1, 2, \dots\,.$$

For this proposal, we first find c,

$$\sup_{i=0,1,\dots,m} \frac{\Pr(X = i)}{\Pr(Y = i)} = \sup_{i=0,1,\dots,m} \frac{e^{-\lambda}\lambda^i/i!}{\sum_{j=0}^{m} e^{-\lambda}\lambda^j/j!} \frac{i!}{\lambda^i e^{-\lambda}} \mathbb{I}(i \in \{0, 1, 2, \dots, m\}) \leq \frac{1}{\sum_{j=0}^{m} e^{-\lambda}\lambda^j/j!} = c\,.$$

Using this, we can find the ratio

$$\frac{p_i}{cq_i} = \mathbb{I}(i \in \{0, 1, 2, \dots, m\})\,.$$

Then in the A-R ratio, it is just an indicator, so we don't need to get a uniform coin. We will now write code. First let's find the true value of \$c\$. This is the CDF of Poisson at $x = 20$.

```
1/ppois(20, lambda = 20)
```

```
[1] 1.788613
```

```
trunc_pois <- function(m = 20, lam = 20)
{
  accept <- 0
  try <- 0
  while(!accept)
  {
    try <- try + 1
    prop <- rpois(1, lambda = lam)
    if(prop <= m)
    {
      accept <- 1
    }
  }
```

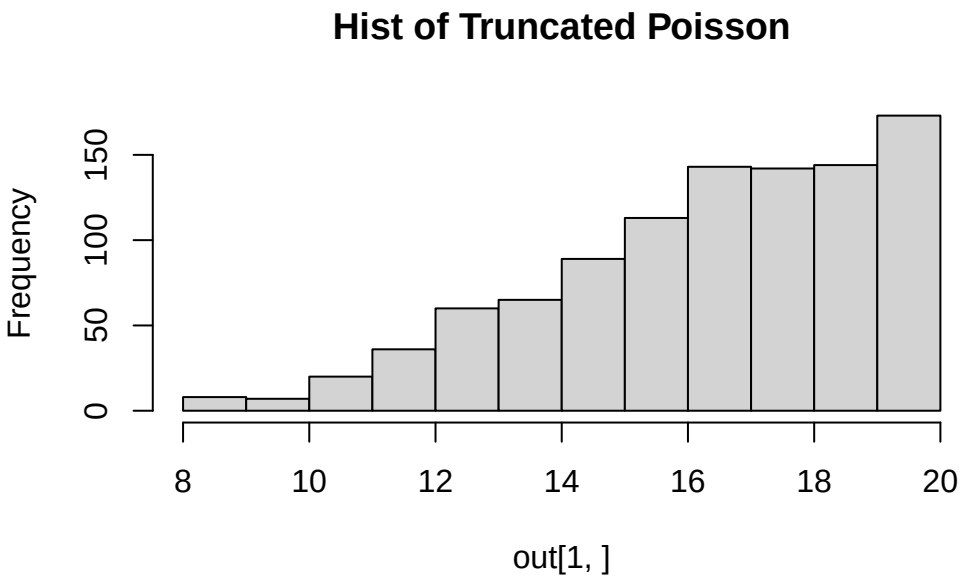
```
  return(c(prop, try))
}

N <- 1e3
out <- replicate(N, trunc_pois())
# out is 2 x 1000 matrix. First row are the samples
# second row are the number of loops

mean(out[1, ]) # mean of trunc pois
```

```
[1] 16.809
```

```
mean(out[2, ]) # mean of number loops, similar to c
```

```
[1] 1.799
```

```
hist(out[1, ], main = "Hist of Truncated Poisson")
```



6. **Consider a Geometric(p) target distribution. We are aware than a Poisson target distribution cannot be used as a valid proposal distribution for accept-reject. "Verify" this claim by writing code that calculates the bound c.**
   As it turns out, for the Geometric(p) distribution, it is extremely challenging to find a valid proposal distribution.

   We will "verify" this by looking at the value of c

```r
# Let's choose many values of x
x <- 0:500

all_c <- rgeom(x, p = .10)/rpois(x, lambda = 10)
max(all_c)
```

```
[1] 7.25
```

```r
# ok all_c is not large. But let's increase values of x
x <- 0:50000

all_c <- rgeom(x, p = .10)/rpois(x, lambda = 10)
max(all_c)
```

```
[1] Inf
```