# STAT40400 - Assignment 4

Jamie Kennedy - 17372983

11/22/2020

**Exercise 1**

($a$) The first thing to do is set the seed

```
set.seed(1234)
```

In the lab we are given code to perform Metropolis Hastings algorithm

```
metropolis.hastings <- function(f, # the target distribution
                                g, # the proposal distribution
                                random.g, # a sample from the proposal distribution
                                x0, # initial value for chain, in R it is x[1]
                                sigma, # proposal st. dev.
                                chain.size=1e5){ # chain size
  x <- c(x0, rep(NA,chain.size-1))
  for(i in 2:chain.size) {
    y <- random.g(x[i-1],sigma)
    alpha <- min(1, f(y)*g(y,x[i-1],sigma)/(f(x[i-1])*g(x[i-1],y,sigma)))
    if( runif(1)<alpha){
      x[i] <- y      # accept
    }
    else{
      x[i] <- x[i-1]    # reject
    }
  }
  x
}
```

To modify the code in the lab to monitor the rate at which the proposed values are accepted we can create another chain which takes the value 1 when a proposed value is accepted and takes the value 0 when a proposed value is rejected. Then the proportion of times this new chain takes the value 1 compared to the total length of the chain will be the acceptance rate. We do this with the following function

```
metropolis.hastings.acc <- function(f, # the target distribution
                                g, # the proposal distribution
                                random.g, # a sample from the proposal distribution
                                x0, # initial value for chain, in R it is x[1]
                                sigma, # proposal st. dev.
                                chain.size=1e5){ # chain size
  x <- c(x0, rep(NA,chain.size-1))
  new <- rep(NA,chain.size)
  for(i in 2:chain.size) {
    y <- random.g(x[i-1],sigma)
    alpha <- min(1, f(y)*g(y,x[i-1],sigma)/(f(x[i-1])*g(x[i-1],y,sigma)))
    if( runif(1)<alpha){
      x[i] <- y      # accept
      new[i] <- 1
    }
    else{
      x[i] <- x[i-1]    # reject
      new[i] <- 0
    }
  }
  length(which(new==1))/length(new)
}
```

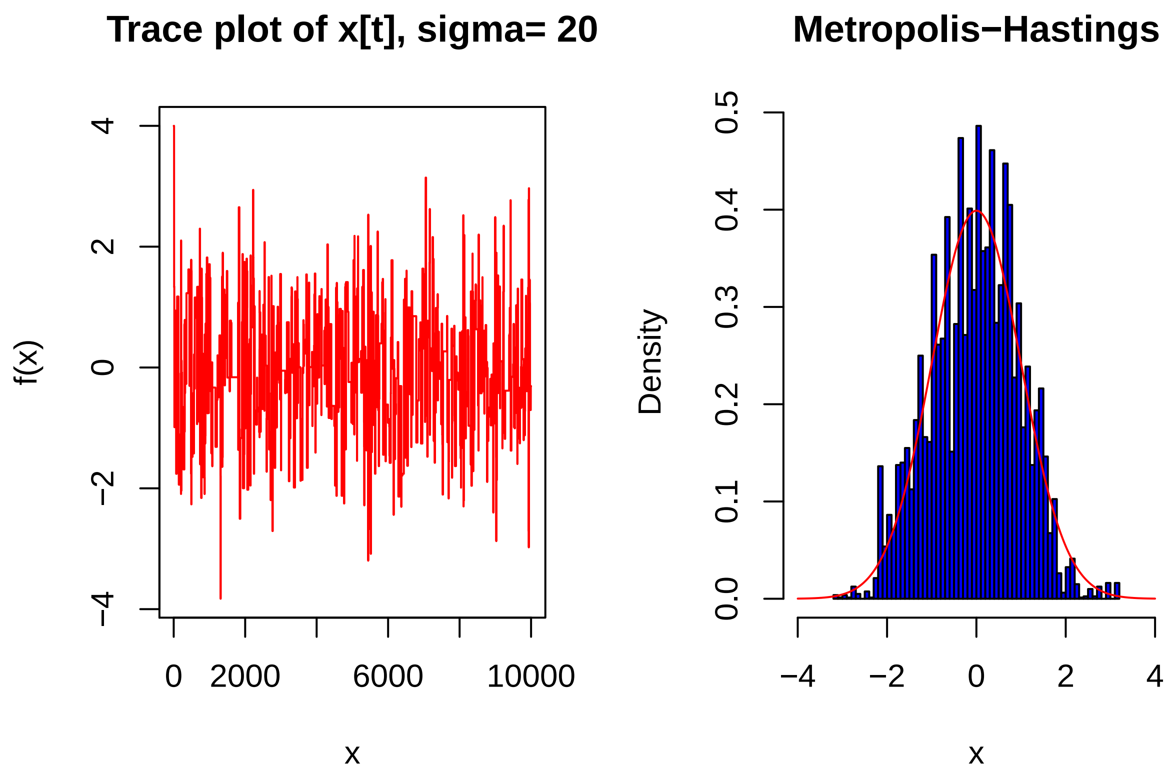
We can use this function to tune the proposal variance because it tells us the rate of acceptance of the proposal distribution and we know that there is a good value for variance when the rate of acceptance of the proposal distribution is somewhere between 20% and 40%.
Therefore, by looking at the acceptance rate for each of the three scenarios in the lab we can see which scenario has the better variance.
The first case we look at is when $\sigma = 20$

```
sigma=20
f <- function(x) dnorm(x,0,1)
random.g <- function(x,sigma) rnorm(1,x,sigma)
g <- function(x,y,sigma) dnorm(y,x,sigma)
x0 <- 4
chain.size <- 1e4
rate_20 <-  metropolis.hastings.acc(f,g,random.g,x0,sigma,chain.size)
rate_20
```

```
## [1] 0.064
```

This shows us that when $\sigma = 20$ the rate of acceptance is 0.064 which is a bit higher than our optimal range. We can also look at some plots to help us visualize the effect of the variance.

```
# run metropolis hastings for this scenario
x<-metropolis.hastings(f,g,random.g,x0,sigma,chain.size)

# create plots
par(mfrow=c(1,2))
plot(x, xlab="x", ylab="f(x)", main=paste("Trace plot of x[t], sigma=", sigma),col="red", type="l")
xmin = min(x[(0.2*chain.size) : chain.size])
xmax = max(x[(0.2*chain.size) : chain.size])
xs.lower = min(-4,xmin)
xs.upper = max(4,xmax)
xs <- seq(xs.lower,xs.upper,len=1e3)
hist(x[(0.2*chain.size) : chain.size],50, xlim=c(xs.lower,xs.upper),col="blue",xlab="x", main="Metropoli
lines(xs,f(xs),col="red", lwd=1)
```



It is clear from these plots that the chain is not mixed well. Also, the histogram of sampled values is not representative of the target N(0,1) density. This confirms that $\sigma = 20$ is not optimal which we previously saw when its acceptance rate was not between 20% and 40%.

The next scenario we look at is when $\sigma = 0.2$

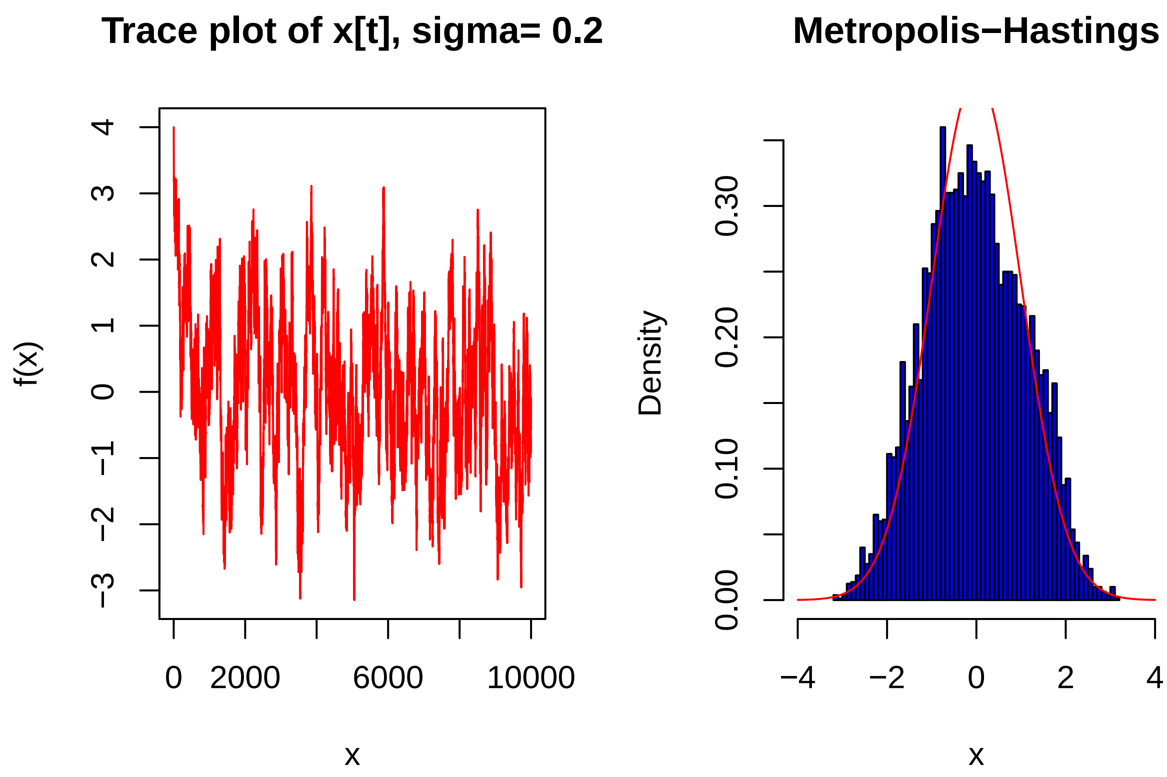
```
sigma=0.2
f <- function(x) dnorm(x,0,1)
random.g <- function(x,sigma) rnorm(1,x,sigma)
g <- function(x,y,sigma) dnorm(y,x,sigma)
x0 <- 4
chain.size <- 1e4
rate_0.2 <-  metropolis.hastings.acc(f,g,random.g,x0,sigma,chain.size)
rate_0.2
```

```
## [1] 0.9342
```

This shows us that when $\sigma = 0.2$ the rate of acceptance is 0.9342 which again is higher than our optimal range. We can also look at some plots to help us visualize the effect of the variance.

```
# run metropolis hastings for this scenario
x<-metropolis.hastings(f,g,random.g,x0,sigma,chain.size)

# create plots
par(mfrow=c(1,2))
plot(x, xlab="x", ylab="f(x)", main=paste("Trace plot of x[t], sigma=", sigma),col="red", type="l")
xmin = min(x[(0.2*chain.size) : chain.size])
xmax = max(x[(0.2*chain.size) : chain.size])
```

```
xs.lower = min(-4,xmin)
xs.upper = max(4,xmax)
xs <- seq(xs.lower,xs.upper,len=1e3)
hist(x[(0.2*chain.size) : chain.size],50, xlim=c(xs.lower,xs.upper),col="blue",xlab="x", main="Metropoli
lines(xs,f(xs),col="red", lwd=1)
```



Here the chain hasn't converged - the trace plot shows that the Markov chain hasn't reached stationarity and the histogram show a mis-match with the target density. This confirms that $\sigma = 0.2$ is not optimal which we previously saw when its acceptance rate was not between 20% and 40%.
The final scenario we look at is when $\sigma = 3$

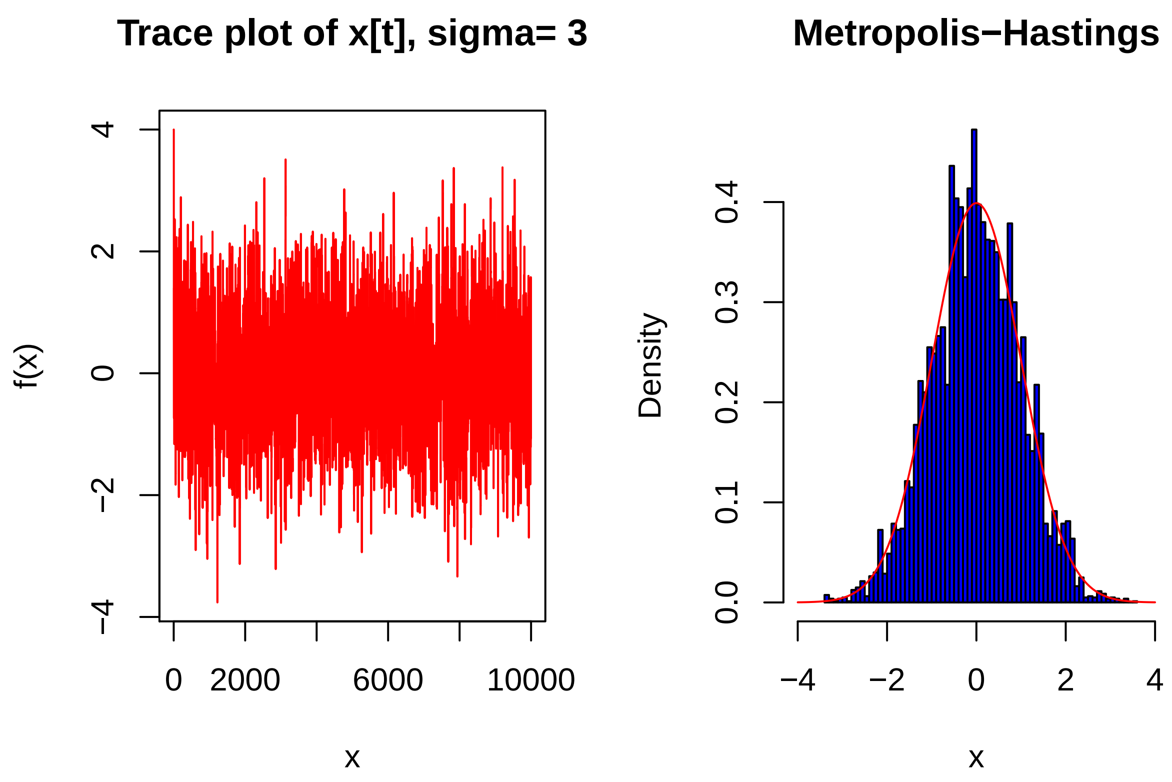
```
sigma= 3
f <- function(x) dnorm(x,0,1)
random.g <- function(x,sigma) rnorm(1,x,sigma)
g <- function(x,y,sigma) dnorm(y,x,sigma)
x0 <- 4
chain.size <- 1e4
rate_3 <-  metropolis.hastings.acc(f,g,random.g,x0,sigma,chain.size)
rate_3
```

```
## [1] 0.3791
```

This shows us that when $\sigma = 3$ the rate of acceptance is 0.3791 which is in our optimal range of 20% to 40%. We can also look at some plots to help us visualize the effect of the variance.

```
# run metropolis hastings for this scenario
x<-metropolis.hastings(f,g,random.g,x0,sigma,chain.size)
```

```
# create plots
par(mfrow=c(1,2))
plot(x, xlab="x", ylab="f(x)", main=paste("Trace plot of x[t], sigma=", sigma),col="red", type="l")
xmin = min(x[(0.2*chain.size) : chain.size])
xmax = max(x[(0.2*chain.size) : chain.size])
xs.lower = min(-4,xmin)
xs.upper = max(4,xmax)
xs <- seq(xs.lower,xs.upper,len=1e3)
hist(x[(0.2*chain.size) : chain.size],50, xlim=c(xs.lower,xs.upper),col="blue",xlab="x", main="Metropoli
lines(xs,f(xs),col="red", lwd=1)
```



Here the chain has converged - the trace plot shows that the Markov chain appears to have reached stationarity and the histogram of sampled values agrees with the target N(0, 1) density. This confirms that $\sigma = 3$ is a suitable proposal variance which confirms what we found when we saw its acceptance rate was in the range vetween 20% and 40%. In summary, we can use the rate at which proposed points are accepted to tune the proposal variance. This is often referred to as proposal scaling. It has been an issue of extensive study for many years. The usual approach is to monitor the rate at which proposed points are accepted and to select a value of $\sigma$ so that it leads to an acceptance rate of around 20% to 40%. For our three scenarios we see:
1. The proposal distribution with $\sigma = 0.2$ generates candidates very similar to the current value that are typically accepted, as a consequence means it takes a long time to move through the parameter space.
2. The proposal with $\sigma = 3$ is much closer to the target distribution, so that more sensible candidates are generated and subsequently accepted.
3. While the proposal with $\sigma = 20$, often generates candidate observations too far in the tail of the target and these are subsequently rejected.

$(b)$ For the first scenario when $\sigma = 20$ we can estimate the probability that $X > 2$ where $X \sim N(0, 1)$ with the following code

```r
sigma=20
f <- function(x) dnorm(x,0,1)
random.g <- function(x,sigma) rnorm(1,x,sigma)
g <- function(x,y,sigma) dnorm(y,x,sigma)
x0 <- 4
chain.size <- 1e4
x <- metropolis.hastings(f,g,random.g,x0,sigma,chain.size)
prob_20<-length(which(x>2))/length(x)
prob_20
```

```
## [1] 0.0166
```

This returns a probability of 0.0166 which is lower than the true value of 0.02275 so when the proposal distribution has $\sigma = 20$ the sample we generate contains too many values less than 2.
For the second scenario when $\sigma = 0.2$ we can estimate the probability that $X > 2$ where $X \sim N(0,1)$ with the following code

```r
sigma=0.2
f <- function(x) dnorm(x,0,1)
random.g <- function(x,sigma) rnorm(1,x,sigma)
g <- function(x,y,sigma) dnorm(y,x,sigma)
x0 <- 4
chain.size <- 1e4
x <- metropolis.hastings(f,g,random.g,x0,sigma,chain.size)
prob_0.2<-length(which(x>2))/length(x)
prob_0.2
```

```
## [1] 0.029
```

This returns a probability of 0.029 which is higher than the true value of 0.02275 so when the proposal distribution has $\sigma = 0.2$ the sample we generate contains too many values greater than 2.
Finally, when $\sigma = 3$ we can estimate the probability that $X > 2$ where $X \sim N(0,1)$ with the following code

```r
sigma=3
f <- function(x) dnorm(x,0,1)
random.g <- function(x,sigma) rnorm(1,x,sigma)
g <- function(x,y,sigma) dnorm(y,x,sigma)
x0 <- 4
chain.size <- 1e4
x <- metropolis.hastings(f,g,random.g,x0,sigma,chain.size)
prob_3<-length(which(x>2))/length(x)
prob_3
```

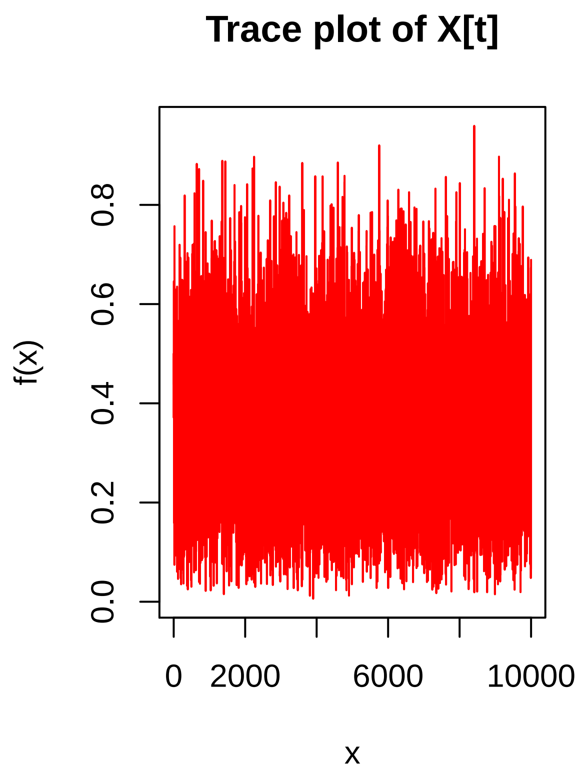
```
## [1] 0.0226
```

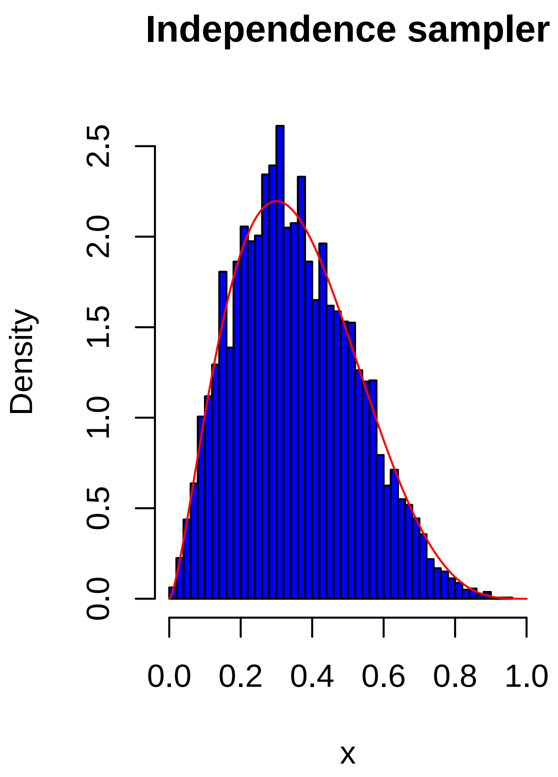
This returns a probability of 0.0226 which is quite close to the true value of 0.02275 so when the proposal distribution has $\sigma = 3$ the sample we generate contains a suitable number of values greater than 2. Thus, we can see that $\sigma = 3$ gives the closest estimate of $P(X > 2)$ to the true value.

**Exercise 2**
An independence sampler occurs when $q(\theta, \phi) = f(\phi)$ i.e. the candidate observation is drawn independently of the current state of the chain. The acceptance probabilty can be written as $\alpha(\theta, \phi) = min(1, \frac{w(\phi)}{w(\theta)})$ where $w(\theta) = \frac{\pi(\theta)}{f(\theta)}$. In this case the proposal distribution is a uniform distribution and the target distribution is the $beta(2.5, 4.5)$ distribution which we know lives on the interval $(0, 1)$ so we use the $uniform(0, 1)$ distribution as the proposal distribution. We can sample from the $beta(2.5, 4.5)$ distribution using an independence

sampler with a uniform proposal distribution using the following code

```r
# first set the seed
set.seed(555)

mh.independence  <- function(f, g, random.g, x0, chain.size=1e4){

  x <- c(x0, rep(NA,chain.size-1))

  for(i in 2:chain.size) {
    y <- random.g(x[i-1])
    alpha <- min(1, f(y)*g(y)/(f(x[i-1])*g(x[i-1])))
    if( runif(1)<alpha){
      x[i] <- y      # accept
    }
    else{
      x[i] <- x[i-1]    # reject
    }
  }
  x
}

f <- function(x) dbeta(x,2.5,4.5)
random.g <- function(x) runif(1)
g <- function(x) dunif(x)
x0 <- 0.5
chain.size <- 1e4
x<-mh.independence(f,g,random.g,x0,chain.size)

par(mfrow=c(1,2))
plot(x, xlab="x", ylab="f(x)", main=paste("Trace plot of X[t]"),col="red", type="l")
xmin = min(x[(0.2*chain.size) : chain.size])
xmax = max(x[(0.2*chain.size) : chain.size])
xs.lower = min(0,xmin)
xs.upper = max(1,xmax)
xs <- seq(xs.lower,xs.upper,len=1e3)
hist(x[(0.2*chain.size) : chain.size],50, xlim=c(xs.lower,xs.upper),col="blue",xlab="x", main="Independe
lines(xs,f(xs),col="red", lwd=1)
```





Here the trace plot shows that the Markov chain appears to have reached stationarity and the histogram of sampled values agrees with the target $beta(2.5, 4.5)$ density.

We can look at the acceptance rates of the independence sampler

```
mh.independence.acc  <- function(f, g, random.g, x0, chain.size=1e4){

  x <- c(x0, rep(NA,chain.size-1))
  new <- rep(NA,chain.size)

  for(i in 2:chain.size) {
    y <- random.g(x[i-1])
    alpha <- min(1, f(y)*g(y)/(f(x[i-1])*g(x[i-1])))
    if( runif(1)<alpha){
      x[i] <- y      # accept
      new[i] <- 1
    }
    else{
      x[i] <- x[i-1]    # reject
      new[i] <- 0
    }
  }
  length(which(new==1))/length(new)
}

f <- function(x) dbeta(x,2.5,4.5)
random.g <- function(x) runif(1)
```

```
g <- function(x) dunif(x)
x0 <- 0.5
chain.size <- 1e4
rate <- mh.independence.acc(f,g,random.g,x0,chain.size)
rate
```

```
## [1] 0.547
```

This value is a bit higher than the optimal region of (0.2,0.4) so the independence sampler slightly over-accepts from the proposal distribution.
We can also compare the mean and variance of our output to the actual mean and variance of the $beta(2.5, 4.5)$ distribution. The mean of our sample is

```
mu <- mean(x)
mu
```

```
## [1] 0.3543589
```

This value of 0.3543589 is close to the actual mean of $\frac{5}{14} = 0.3571428$. The variance of our sample is

```
var <- var(x)
var
```

```
## [1] 0.02794516
```

This value of 0.02794516 is close to the actual variance of $\frac{\alpha\beta}{(\alpha+\beta)^2(\alpha+\beta+1)} = \frac{(2.5)(4.5)}{(2.5+4.5)^2(2.5+4.5+1)}$ =0.028699.
In summary, the statistical properties of our sample are very similar to the statstical properties of the $beta(2.5, 4.5)$ distribution and we previously saw plots that showed us that the sample agrees with the target distribution. This would lead us to believe that our algorithm performs very well. However, we should be aware that the acceptance rate of our algorithm is a bit higher than we would like so there is room for improvement in our algorithm.

**Exercise 3**

$(a)$ Hand-written solution at end.

$(b)$ Hand-written solution at end.

$(c)$ To perform metropolis within gibbs sampling we must choose a proposal distribution for the metropolis update. It is common practise to use a normal proposal distribution for the metropolis update as this is a symmetric distribution so the expression for $\alpha$ becomes simplified. Thus, the first thing to do was find a suitable variance for the proposal distribution. Although not ideal, the method I used to do this was to run a metropolis algorithm with $f = \frac{1}{z!(18-z)!}$ and a normal proposal distribution for various values of variance. I then calculated the acceptance rate corresponding to each variance until I found a variance with acceptance rate close to the optimal acceptance rate of 0.234. It turned out that a variance of 10 was suitable as it has an acceptance rate of 0.2525. I used the following code to do this

```
# create function for metropolis algorithm
mh.symmetric <- function(f,random.g,x0,sigma,chain.size){
  x <- c(x0, rep(NA,chain.size-1))
  for(i in 2:chain.size) {
    y <- random.g(x[i-1],sigma)
    alpha <- min(1, f(y)/(f(x[i-1])))
    if( runif(1)<alpha){
      x[i] <- y      # accept
    }
    else{
      x[i] <- x[i-1]    # reject
    }
  }
  x
}

# create function to find acceptance rate
mh.symmetric.acc <- function(f,random.g,x0,sigma,chain.size=1e5){
  x <- c(x0, rep(NA,chain.size-1))
  new <- rep(NA,chain.size)
  for(i in 2:chain.size) {
    y <- random.g(x[i-1],sigma)
    alpha <- min(1, f(y)/(f(x[i-1])))
    if( runif(1)<alpha){
      x[i] <- y       # accept
      new[i] <- 1
    }
    else{
      x[i] <- x[i-1]    # reject
      new[i] <- 0
    }
  }
  length(which(new==1))/length(new)
}

# lets tune variance
sigma=20
f <- function(x) 1/(factorial(x)*factorial(18-x))
random.g <- function(x,sigma) rnorm(1,x,sigma)
x0 <- 1
chain.size <- 1e4
rate_20 <-  mh.symmetric.acc(f,random.g,x0,sigma,chain.size)
rate_20
```

```
## [1] 0.1374
```

```
sigma=5
rate_5 <-  mh.symmetric.acc(f,random.g,x0,sigma,chain.size)
rate_5
```

```
## [1] 0.4568
```

```
sigma=10
rate_10 <-  mh.symmetric.acc(f,random.g,x0,sigma,chain.size)
rate_10
```

```
## [1] 0.2535
```

```
# this is a good rate of acceptance
```

Next we define a function that will perform the metropolis update in our metropolis within gibbs algorithm.

```
# define f
f <- function(x) 1/(factorial(x)*factorial(18-x))
# create function
mh.symmetric.kernel <- function(f,previous){
    y <- rnorm(1,previous,10)  # proposed value uses previous value of lambda
    alpha <- min(1, f(y)/(f(previous)))
    if( runif(1)<alpha){
      x <- y      # accept
    }
    else{
      x <- previous   # reject
    }
  x
}
```

Now we have everything we need to define our function to carry out our metropolis within gibbs algorithm.

```
gibbs_gen <- function(n, y){
  x <-array(0,c(n+1,2))
  x[0,1] = 1
  x[0,2] = 1

  for(t in 1:(n+1)){
    x[t,1] = rgamma(1,51,11.27725)
    x[t,2] = mh.symmetric.kernel(f,x[t,1])
  }
  x
}
```

We can estimate the posterior distribution of $\lambda$ and posterior distribution of $Z$ using this funtion. The first column of the output represents the posterior of $\lambda$ and the second column represents the posterior of $Z$.

```
y= c(18,13,8,3,4,3,0,0,0,1)
output = gibbs_gen(1000,y)
```

We can look at the mean and variance of each posterior.

```
mean_lambda = mean(output[,1])
mean_lambda
```

```
## [1] 4.546558
```

```
mean_Z = mean(output[,2])
mean_Z
```

```
## [1] 6.123553
```

```
var_lambda = var(output[,1])
var_lambda
```

```
## [1] 0.3802382
```

```
var_Z = var(output[,2])
var_Z
```

```
## [1] 7.50334
```

We can see that our posterior for $\lambda$ has a mean of 4.54 and a variance of 0.38. This leads us to believe that our prior knowledge that we were 75% sure that the value of $\lambda$ was less than 5 was accurate. Our posterior for $Z$ has a mean of 6.12 and a variance of 7.5. This means that we think the number of employees out of the sample of 50 who had no absences during the previous 12 months is around 6 ,however, we cannot be too certain about this estimate as it has large variance. To get a better idea of where the values of $\lambda$ and $Z$ are likely to lie we can create confidence intervals.

```r
s1 = sqrt(var_lambda)
lower = mean_lambda - 1.96*(s1/sqrt(1000))
upper = mean_lambda + 1.96*(s1/sqrt(1000))
CI_lambda <- list("Lower bound:"=lower, "Upper bound:"=upper,
                "95% Confidence interval for lambda:",c(lower,upper))
print(CI_lambda)
```

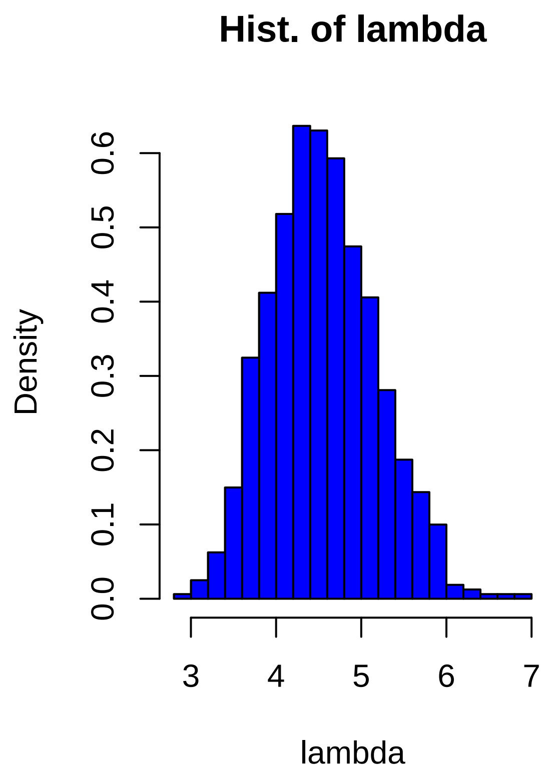
```
## $`Lower bound:`
## [1] 4.508338
##
## $`Upper bound:`
## [1] 4.584777
##
## [[3]]
## [1] "95% Confidence interval for lambda:"
##
## [[4]]
## [1] 4.508338 4.584777
```

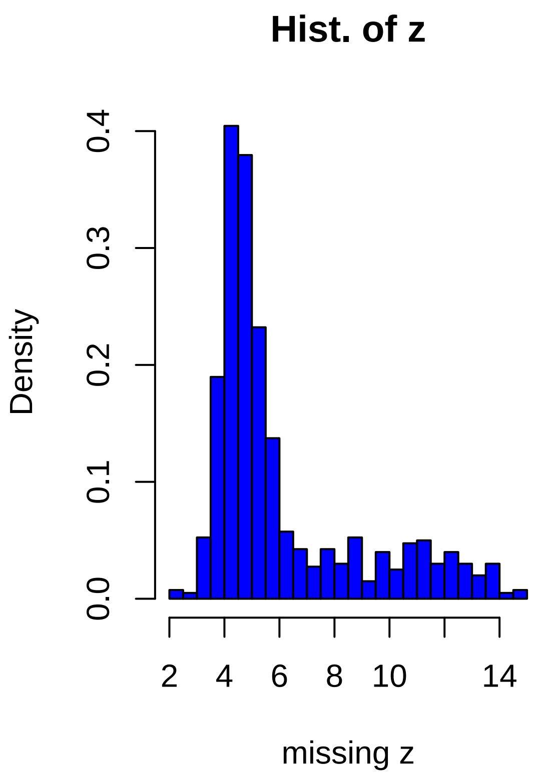
```r
s2 = sqrt(var_Z)
lower = mean_Z - 1.96*(s2/sqrt(1000))
upper = mean_Z + 1.96*(s2/sqrt(1000))
CI_Z <- list("Lower bound:"=lower, "Upper bound:"=upper,
                "95% Confidence interval for Z:",c(lower,upper))
print(CI_Z)
```

```
## $`Lower bound:`
## [1] 5.953774
##
## $`Upper bound:`
## [1] 6.293331
##
## [[3]]
## [1] "95% Confidence interval for Z:"
##
## [[4]]
## [1] 5.953774 6.293331
```

These confidence intervals show us tht we are 95% confident that $\lambda$ lies in the range [4.508,4.585] and $Z$ lies in the range [5.95,6.29].
We can visualize the posterior distributions of $\lambda$ and $Z$ with the following plots.

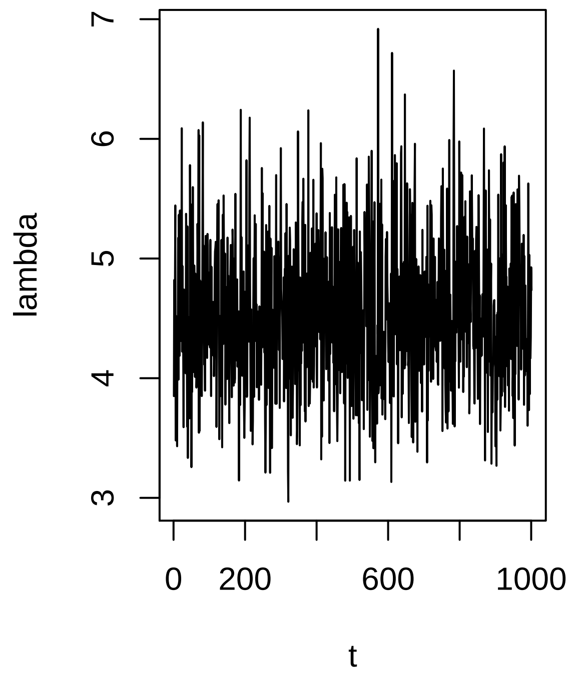
```r
par(mfrow=c(1,2))
hist(output[(0.2*1000):1000,1],20,col="blue",
     xlab="lambda", main="Hist. of lambda",freq=FALSE)
hist(output[(0.2*1000):1000,2],20,col="blue",
     xlab="missing z", main="Hist. of z",freq=FALSE)
```

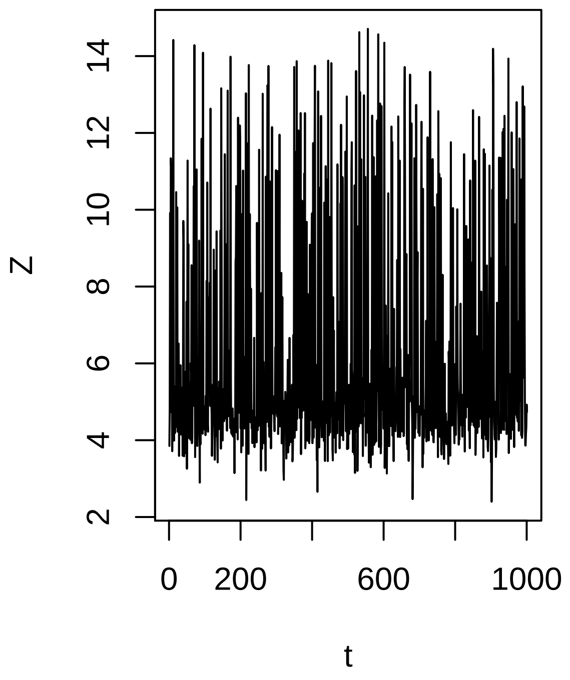




We can see from these plots that the value of $\lambda$ usually lies somewhere between 4 and 5 and if it doesnt then it has a similar chance to be above 5 as it does to be below 4. The value of $Z$ usually lies between 4 and 6. There is very little chance of it being less than this and obviously it can't be negative as we can't have a negative number of employees. Therefore, if the value of $Z$ is not between 4 and 6 then it is likely to be greater than 6.
We can look at trace plots of our chains.

```r
par(mfrow=c(1,2))
plot(1:length(output[,1]),output[,1],type='l',lty=1,xlab='t',ylab='lambda')
plot(1:length(output[,2]),output[,2],type='l',lty=1,xlab='t',ylab='Z')
```





It is clear that the posterior desnsity for $\lambda$ is centred around 4.5 and that the posterior density for $Z$ is roughly around 5.