STA 3431 Assignment #2

Yihan Duan

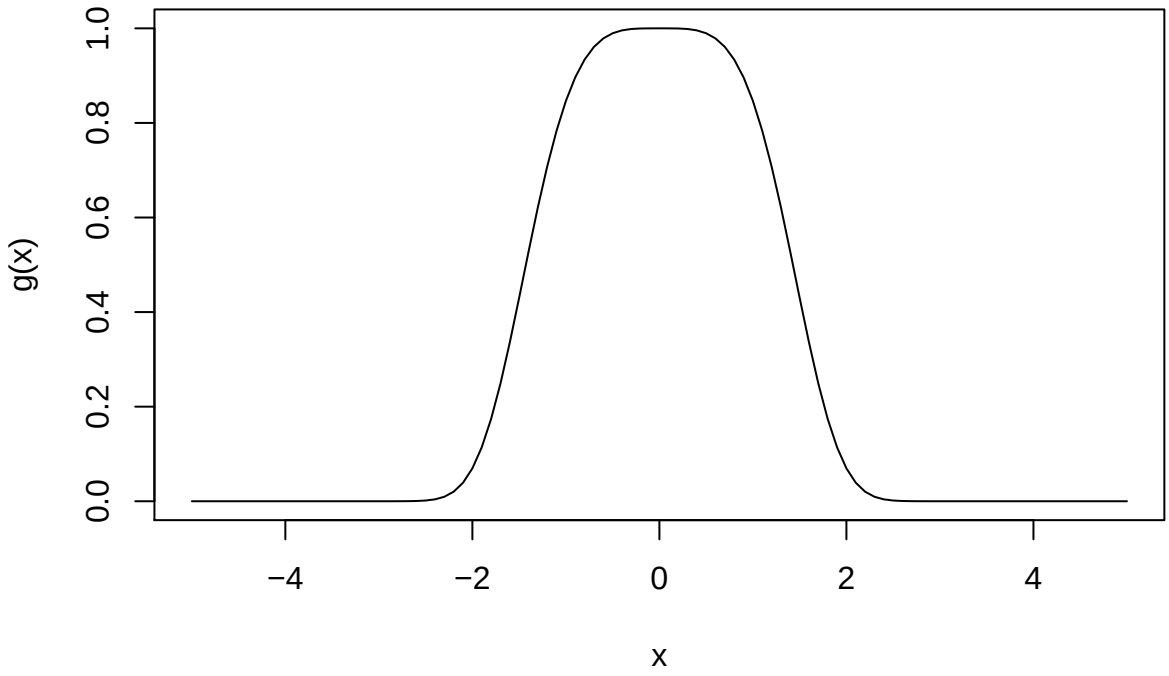
29/10/2021

Name: Yihan Duan Student #: 1003118547 Department: CS Program: MScAC Year: first year master E-mail: yihan.duan@mail.utoronto.ca

Question 1

First let's take a look at the target density function. Because $-\frac{-x^4}{6} \leq 0$ on R, and e^x is increasing on R^-, we know the target density function should be larger around 0 and should get smaller further away from 0. Let's verify that.

```
g = function(x) {
  return(exp(-x^4/6))
}

curve(g, from=-5, to=5)
```



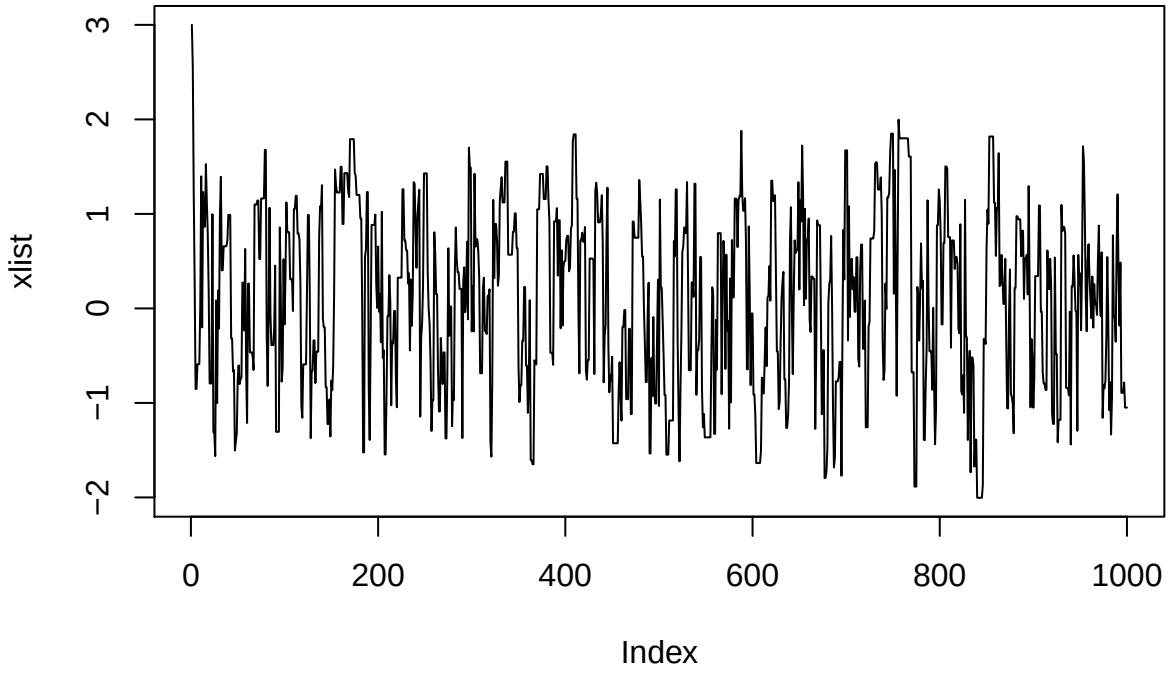
As we can see, the value for g(x) drops drastically around -3 and 3. As Metropolis works better if the initial value covers the "important" parts of the state space, we choose X_0 using a uniform distribution $U(-3, 3)$.

Sigma = 1

First lets try $\sigma = 1$. Let's choose $X_0 = 3$ and run it for 1000 iterations to see decide on burn-in.

```
# define h here
h = function(y) { return(y^2) }

# parameters
N = 1000
X = 3
sigma = 1
xlist = rep(0, N)
hlist = rep(0, N)

for (i in 1:N) {
  Y = X + sigma * rnorm(1)
  U = runif(1)
  alpha = g(Y) / g(X)
  if (U < alpha) {
    X = Y
  }
  xlist[i] = X
  hlist[i] = h(X)
```

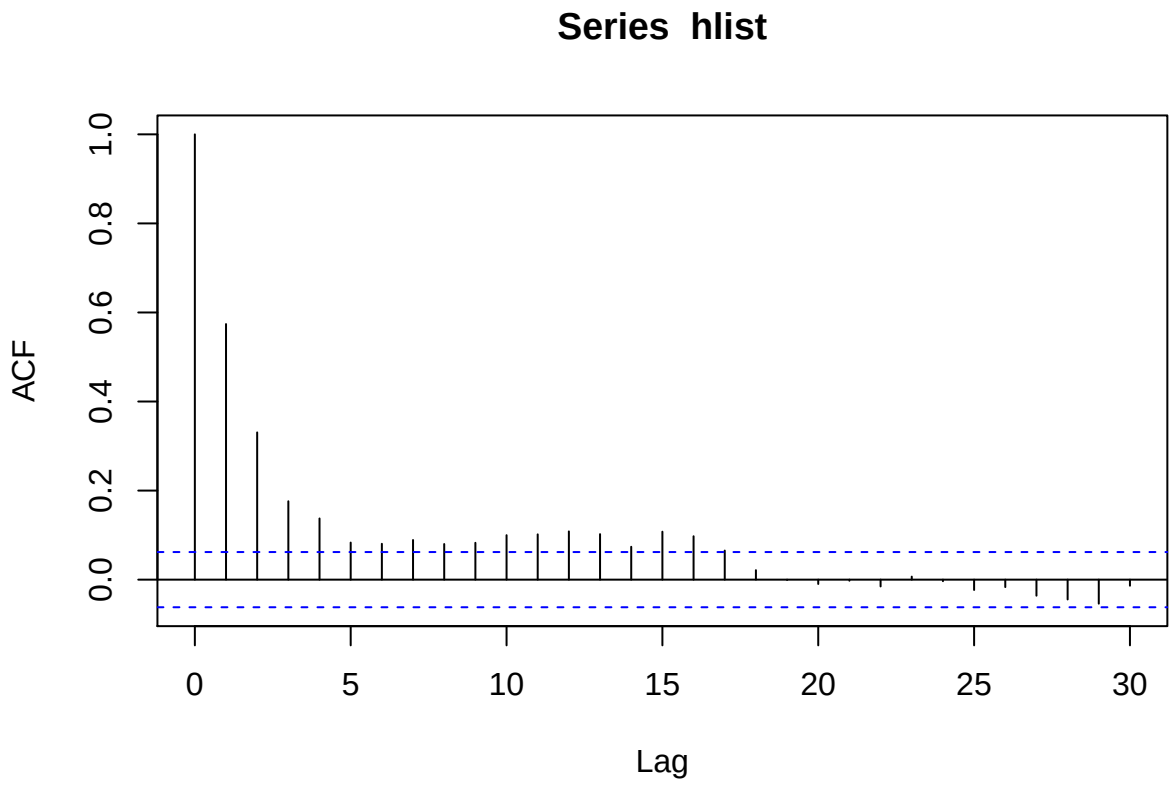
```
}

plot(xlist, type='l')
```



It seems like not much burn in is needed before it becomes stable. Let's set the number of burn-in's to be 1000 just to be safe.

Now let's see what's the best choice for 'lag.max' in 'acf' function.

```
acf(hlist)
```



Looks like our 'acf' becomes very small at around 10. Let's try to automate this process.

```
get_lag_max <- function(hlist) {
  N = length(hlist)
  acfs = acf(hlist, plot=FALSE, lag.max=N)$acf
  for (i in 1:N) {
    if (acfs[i] < 0.01) {
      return(i)
    }
  }
  return(N)
}

get_lag_max(hlist)
```

```
## [1] 20
```

Now lets start a real run.

```
set.seed(1234)

run_Metropolis <- function(M, B, sigma, print_result=TRUE) {
  X = runif(n=1, min=-3, max=3)  # overdispersed starting distribution
  xlist = rep(0,M)  # for keeping track of chain values
  hlist = rep(0,M)  # for keeping track of h function values
```

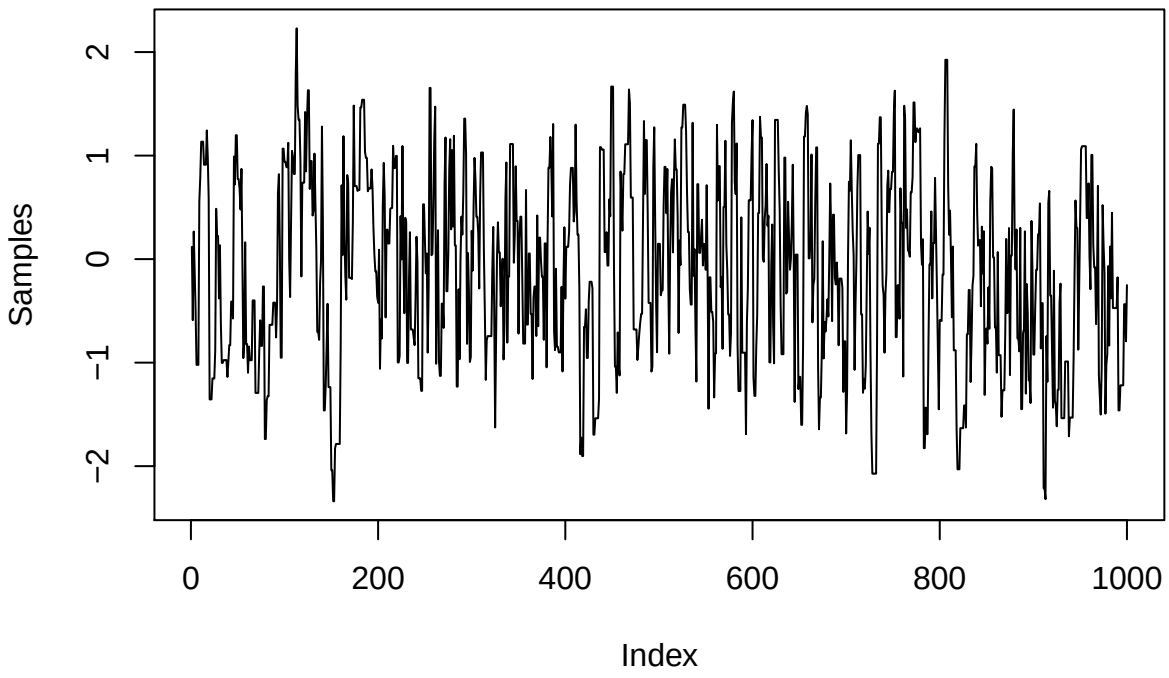
```
  numaccept = 0

  for (i in 1:M) {
    Y = X + sigma * rnorm(1)  # proposal value
    U = runif(1)  # for accept/reject
    alpha = g(Y) / g(X)  # for accept/reject
    if (U < alpha) {
      X = Y  # accept proposal
      numaccept = numaccept + 1
    }
    xlist[i] = X
    hlist[i] = h(X)
  }

  u = mean(hlist[(B+1):M])

  if (print_result) {
    cat("ran Metropolis algorithm for", M, "iterations, with burn-in", B, "\n")
    cat("acceptance rate =", numaccept/M, "\n")
    cat("mean of h is about", u, "\n")
    se1 =  sd(hlist[(B+1):M]) / sqrt(M-B)
    cat("iid standard error would be about", se1, "\n")

    varfact <- function(xxx) { 2 * sum(acf(xxx, plot=FALSE, lag.max=get_lag_max(hlist))$acf) - 1 }
    thevarfact = varfact(hlist[(B+1):M])
    se = se1 * sqrt( thevarfact )
    cat("varfact = ", thevarfact, "\n")
    cat("true standard error is about", se, "\n")
    cat("approximate 95% confidence interval is (", u - 1.96 * se, ",",
        u + 1.96 * se, ")\n\n")
  }

  return(list(xlist, hlist, u))
}
result = run_Metropolis(11000,1000,1)
```

```
## ran Metropolis algorithm for 11000 iterations, with burn-in 1000
## acceptance rate = 0.7284545
## mean of h is about 0.8227934
## iid standard error would be about 0.008913107
## varfact =  4.042915
## true standard error is about 0.01792159
## approximate 95% confidence interval is ( 0.7876671 , 0.8579197 )
```

We can see that the acceptance rate is higher than the optimal but still quite good. The varfact and standard error is a little high and the confidence interval is wider then we would like. We can further reduce this by increasing the number of samples.

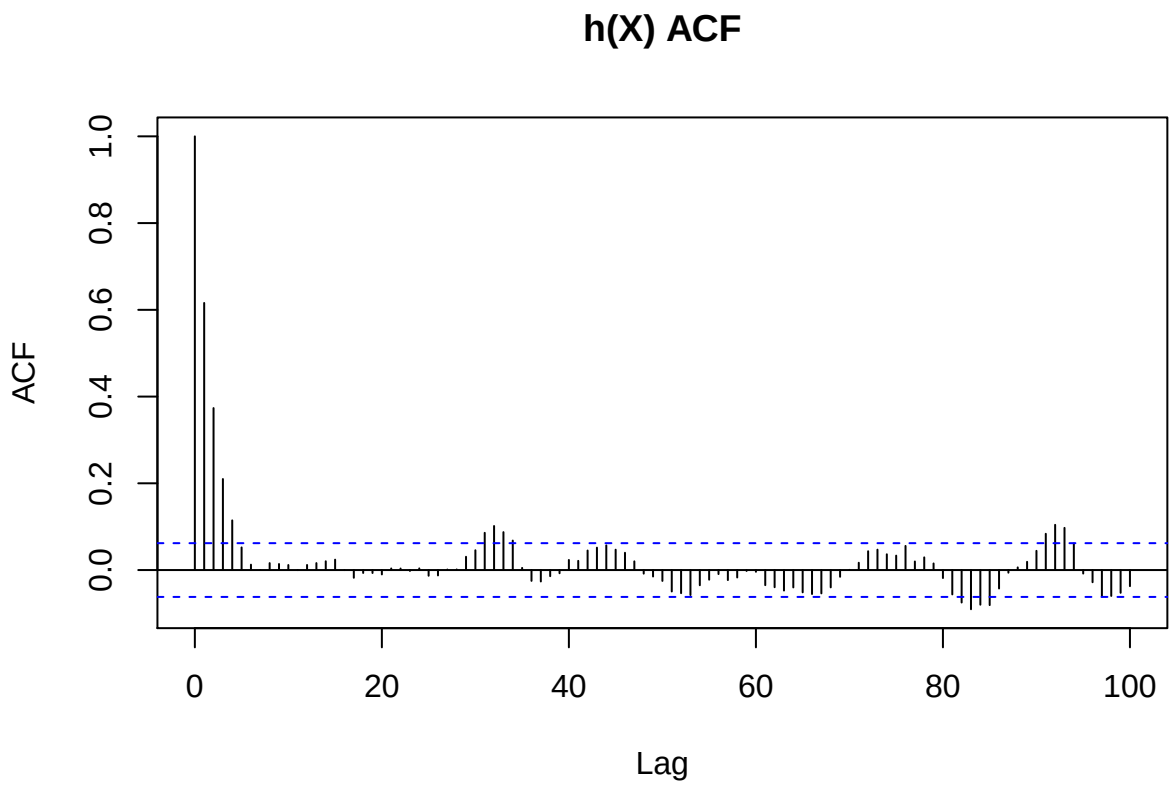
Let's see the mixing of the samples. We will choose the first 1000 samples to investigate.

```
plot(result[[1]][1001:2000], type='l', ylab='Samples', xlab='Index')
```



Again, plot the 'acf' function.

```
acf(result[[2]][1001:2000], plot=TRUE, lag.max=100, main = "h(X) ACF")
```



They seems to be mixing really well. The value doesn't plateau and covers most of the important area. The correlation between close estimates comes down to 0 really fast.

Now let's re-run the algorithm for a few times and see the accuracy.

```
set.seed(1234)
estimates = c(20)
for (i in 1:20) {
  estimates[i] = run_Metropolis(11000,1000,1,FALSE)[[3]]
}

cat("The estimates are:\n")
```

```
## The estimates are:
```

```
estimates
```

```
##  [1] 0.8227934 0.8175611 0.8334305 0.8254704 0.8373696 0.8178089 0.8036043
##  [8] 0.8302489 0.7893847 0.8323556 0.8337604 0.8298366 0.8225612 0.8228413
## [15] 0.8398019 0.8200117 0.8433623 0.8318023 0.8844229 0.8071353
```

```
cat("Mean: ", mean(estimates), "\n")
```

```
## Mean:  0.8272782
```

```
cat("Standard error: ", sd(estimates)/sqrt(20))
```

```
## Standard error:  0.004176594
```

As we can see most of the estimates for h are between 0.82 and 0.84. However, there are still occasional estimates that are way off like 0.79. This algorithm is moderately accurate.

Sigma = 5

Let's try to increase σ so that the acceptance rate is closer to the optimal acceptance rate 0.234. Note that this value is calculated for high-dimensional Metropolis and might not be the best for 1-dimensional case. We also do not change the number of burn-in's in this case because when we increase σ, we generally won't need more burn-in's to reach a stable distribution.

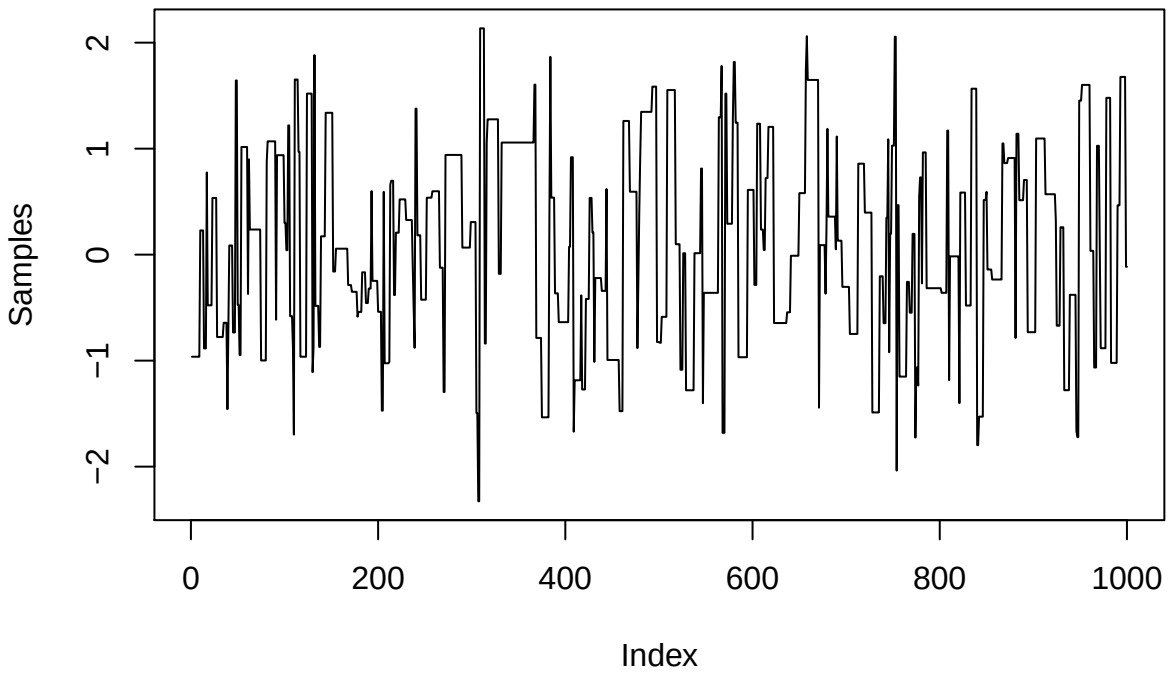
```
set.seed(1234)
result = run_Metropolis(11000,1000,5,TRUE)
```

```
## ran Metropolis algorithm for 11000 iterations, with burn-in 1000
## acceptance rate = 0.2402727
## mean of h is about 0.8492065
## iid standard error would be about 0.009386935
## varfact =  7.790412
## true standard error is about 0.02620016
## approximate 95% confidence interval is ( 0.7978542 , 0.9005588 )
```

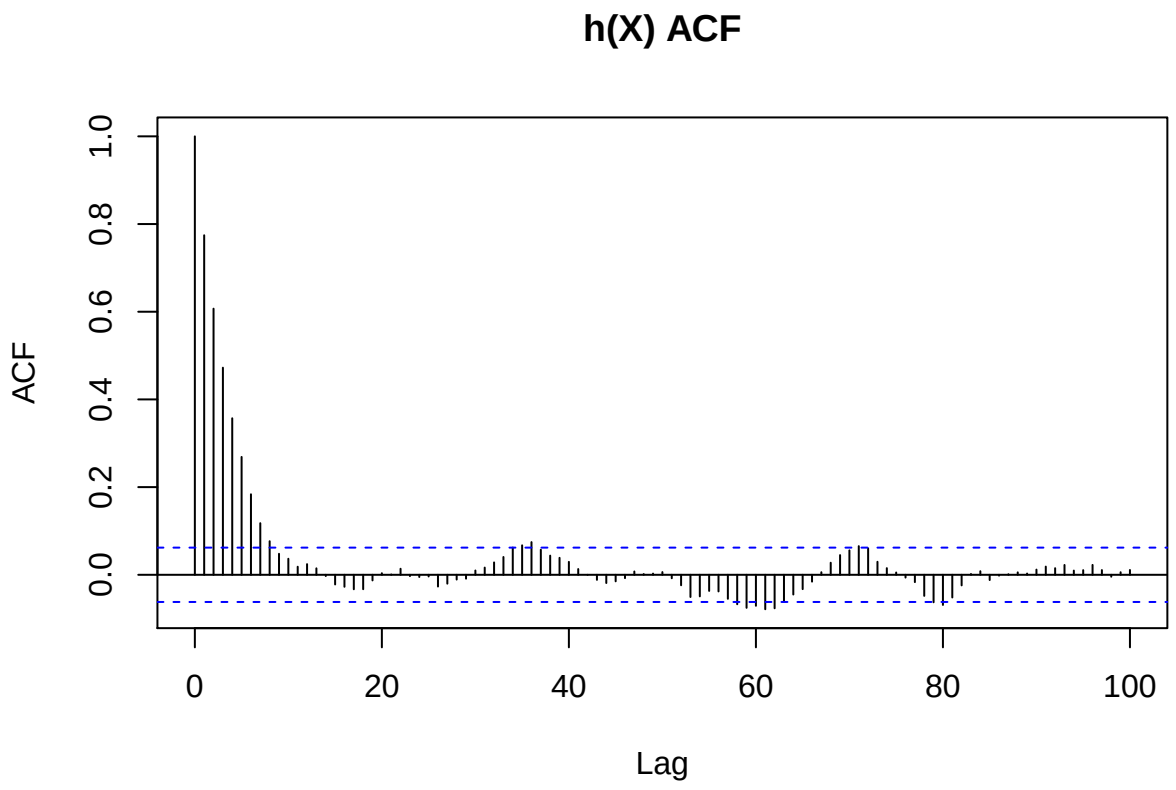
After a few tries, we found out that $\sigma = 5$ gets us an acceptance rate close to 0.234. The varfact is about 2 times as large as we had before and the standard error is larger, resulting in a larger confidence interval. This is due to the fact that with a smaller acceptance rate, x is more likely to be rejected and would plateau more often, leading to higher correlation between $h(x)$'s that are close.

Let's see the mixing of the samples to verify our analysis.

```
plot(result[[1]][1001:2000], type='l', ylab="Samples", xlab='Index')
```




```
acf(result[[2]][1001:2000], plot=TRUE, lag.max=100, main = "h(X) ACF")
```



h values plateaus more often and the acf function comes down a lot slower than that with $\sigma = 1$. More correlations between close estimates because of rejection.

Now let's re-run the algorithm for a few times and see the accuracy.

```
set.seed(1234)
estimates = c(20)
for (i in 1:20) {
  estimates[i] = run_Metropolis(11000,1000,5,FALSE)[[3]]
}

cat("The estimates are:\n")
```

```
## The estimates are:
```

```
estimates
```

```
##  [1] 0.8492065 0.8463992 0.8315751 0.8550199 0.8060011 0.8320218 0.7847565
##  [8] 0.8175990 0.8443568 0.8274081 0.7992952 0.8387760 0.7926659 0.8348756
## [15] 0.8106713 0.8355221 0.8660420 0.8343015 0.8261351 0.8478666
```

```
cat("Mean: ", mean(estimates), "\n")
```

```
## Mean:  0.8290248
```

```
cat("Standard error: ", sd(estimates)/sqrt(20))
```

```
## Standard error:  0.00479421
```

As expected, the algorithm is not as accurate using $\sigma = 5$. Extreme values like 0.79 and 0.86 are more frequent and the standard error is slightly larger.

Sigma = 2

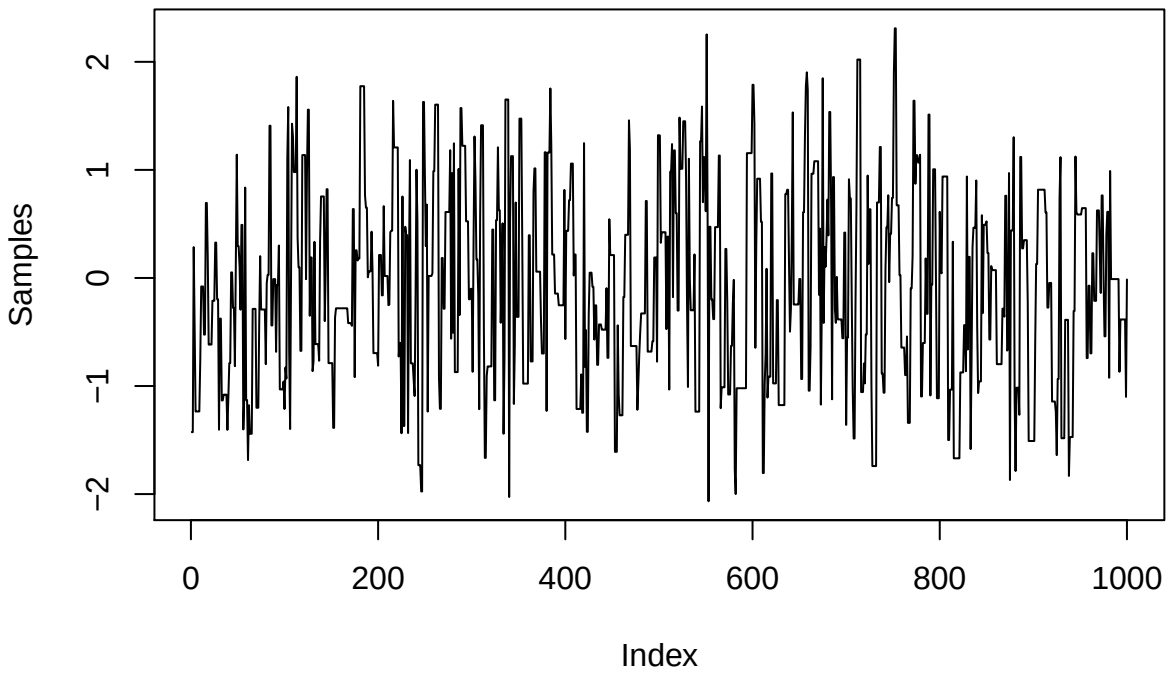
Let's try with rejection rate close to 0.5. As the value of σ should be between 1 and 5, we don't change the number of burn-in's.

```
set.seed(1234)
result = run_Metropolis(11000,1000,2,TRUE)
```

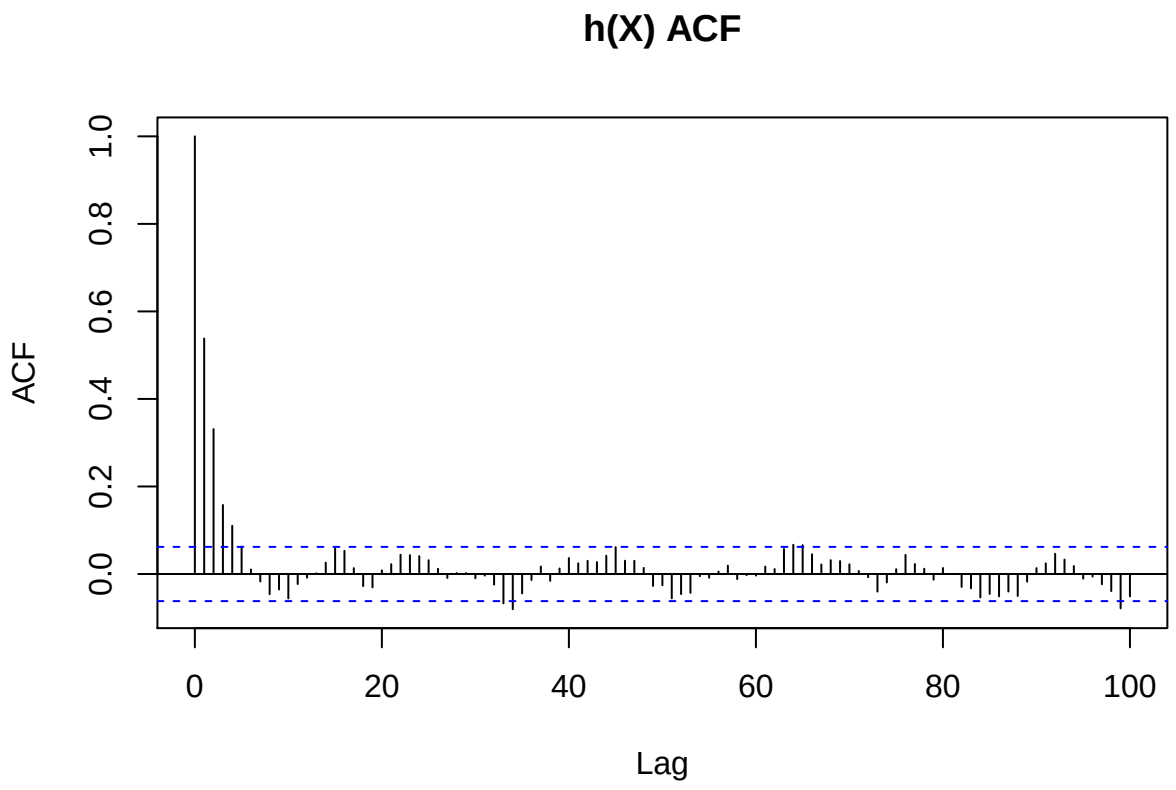
```
## ran Metropolis algorithm for 11000 iterations, with burn-in 1000
## acceptance rate = 0.5115455
## mean of h is about 0.8589294
## iid standard error would be about 0.009205505
## varfact =  3.748634
## true standard error is about 0.01782314
## approximate 95% confidence interval is ( 0.823996 , 0.8938627 )
```

After a few tries, we found that $\sigma = 2$ gives us close to 0.5 acceptance rate. The varfact and standard error are both relatively low. Let's see the mixing of the samples.

```
plot(result[[1]][1001:2000], type='l', ylab="Samples", xlab='Index')
```




```
acf(result[[2]][1001:2000], plot=TRUE, lag.max=100, main = "h(X) ACF")
```



This is definitely an improvement to $\sigma = 5$. The samples seems to be mixing pretty nicely and the 'acf' function reduces to 0 faster.

Now let's re-run the algorithm for a few times and see the accuracy.

```
set.seed(1234)
estimates = c(20)
for (i in 1:20) {
  estimates[i] = run_Metropolis(11000,1000,2,FALSE)[[3]]
}

cat("The estimates are:\n")
```

```
## The estimates are:
```

```
estimates
```

```
##  [1] 0.8589294 0.8303130 0.7925044 0.8204465 0.8118771 0.8312510 0.8487686
##  [8] 0.8209151 0.8412488 0.8274949 0.8302168 0.8474817 0.8211806 0.8343943
## [15] 0.8349640 0.8228879 0.8607069 0.7972749 0.8285716 0.8066843
```

```
cat("Mean: ", mean(estimates), "\n")
```

```
## Mean:  0.8284056
```

```
cat("Standard error: ", sd(estimates)/sqrt(20))
```

```
## Standard error:  0.004045285
```

As expected, the standard error is smaller than that with $\sigma = 5$. This is the most accurate estimation so far.

Conclusion

Judging by the standard error of single estimate and across multiple iid estimates, $\sigma = 0.2$ seems to be the best choice. Let's run Metropolis for more iterations and get a better estimate.

```
result = run_Metropolis(101000,1000,2,TRUE)
```

```
## ran Metropolis algorithm for 101000 iterations, with burn-in 1000
## acceptance rate = 0.5047822
## mean of h is about 0.8313728
## iid standard error would be about 0.002868534
## varfact =  3.754821
## true standard error is about 0.005558462
## approximate 95% confidence interval is ( 0.8204782 , 0.8422674 )
```

Question 2

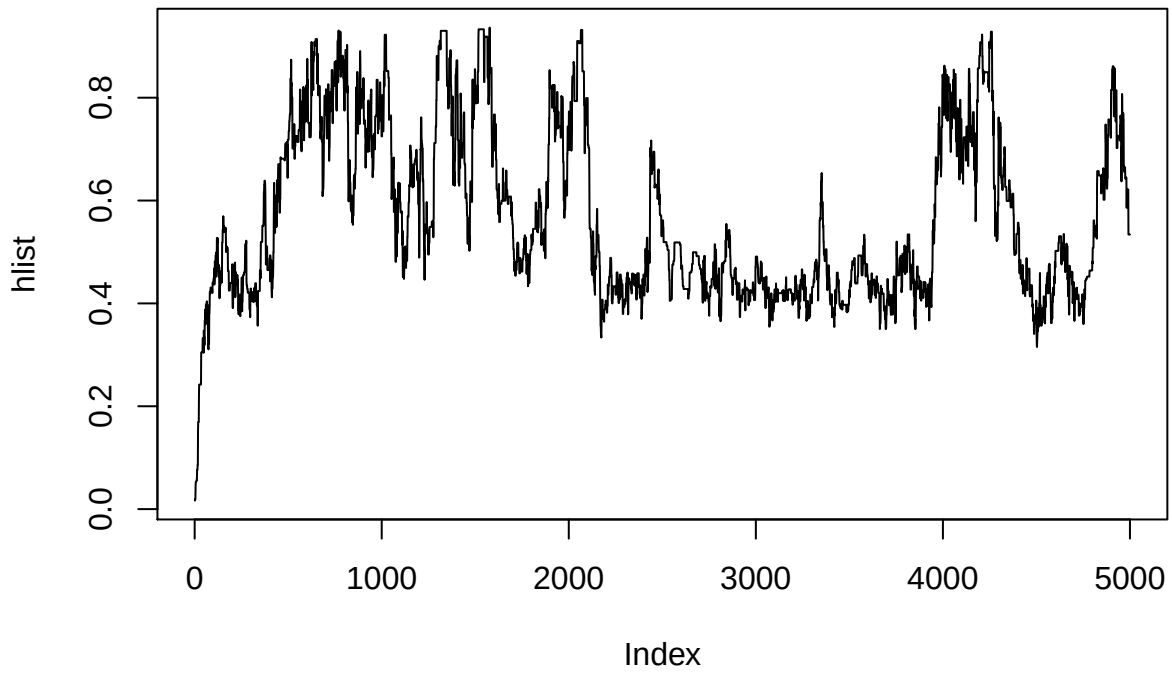
Let's borrow the function definitions from HW#1. Remember we need to make sure that the density function g returns 0 for all points outside the region $0 < x_i < 1$.

```
in_bound <- function(x) {
  return(0<=x && x<=1)
}

g <- function(x) {
  for (xi in x) {
    if (! in_bound(xi)) {
      return(0)
    }
  }
  return(x[1]^14 * 2^(x[2]+3) * (1 + cos(x[1] + 2*x[2] + 3*x[3] + 4*x[4] + 8*x[5])) * exp(-8*x[4]^2) * 
}

h <- function(x) {
  return((x[1] + x[2]^2) / (2 + x[3]*x[4] + x[5]))
}
```

The starting point for each dimension is selected from a uniform distribution between 0 and 1. This is because we know density function is only defined on this region thus 'import region' is sure to be covered.

Let's try to find out the appropriate number of burn-in's with a relatively small scaling factor 0.05 and initial value X_0 close to origin. This will elongate the the burn-in.

```r
set.seed(123)
N = 5000
X = runif(n=5, min=0, max=0.1)
sigma = 0.05  # proposal scaling
x1list = x2list = x3list = x4list = x5list = hlist = rep(0,N)  # for keeping track of values

for (i in 1:N) {
    Y = X + sigma * rnorm(5)  # proposal value (dim=2)
    U = runif(1)  # for accept/reject
    alpha = g(Y) / g(X)  # for accept/reject
    if (U < alpha) {
      X = Y  # accept proposal
    }
    hlist[i] = h(X)
}

plot(hlist, type='l')
```



Using $\sigma = 0.05$, it seems that a 1000-step burn-in is more than enough. However, the samples are not well mixed, so we ought to run it for longer.

Now let's define the Metropolis algorithm.

```r
run_Metropolis_5_dim <- function(M, B, sigma, print_result=TRUE) {
  X = runif(n=5, min=0, max=1)  # overdispersed starting distribution
  x1list = x2list = x3list = x4list = x5list = hlist = rep(0,M)
```

```
  numaccept = 0

  for (i in 1:M) {
    Y = X + sigma * rnorm(5)  # proposal value
    U = runif(1)  # for accept/reject
    alpha = g(Y) / g(X)  # for accept/reject
    if (U < alpha) {
      X = Y  # accept proposal
      numaccept = numaccept + 1
    }
    x1list[i] = X[1]
    x2list[i] = X[2]
    x3list[i] = X[3]
    x4list[i] = X[4]
    x5list[i] = X[5]
    hlist[i] = h(X)
  }

  u = mean(hlist[(B+1):M])

  if (print_result) {
    cat("ran Metropolis algorithm for", M, "iterations, with burn-in", B, "\n")
    cat("acceptance rate =", numaccept/M, "\n")
    cat("mean of h is about", u, "\n")
    se1 =  sd(hlist[(B+1):M]) / sqrt(M-B)
    cat("iid standard error would be about", se1, "\n")

    varfact <- function(xxx) { 2 * sum(acf(xxx, plot=FALSE, lag.max=get_lag_max(hlist))$acf) - 1 }
    thevarfact = varfact(hlist[(B+1):M])
    se = se1 * sqrt( thevarfact )
    cat("varfact = ", thevarfact, "\n")
    cat("true standard error is about", se, "\n")
    cat("approximate 95% confidence interval is (", u - 1.96 * se, ",",
        u + 1.96 * se, ")\n\n")
  }

  return(list(x1list, x2list, x3list, x4list, x5list, hlist, u))
}
```

As the region within which $g(X)$ is valid is relatively small, we don't want a very large σ. Therefore more steps are needed for the samples to mix well. We will try to go for a 0.234 acceptance rate.

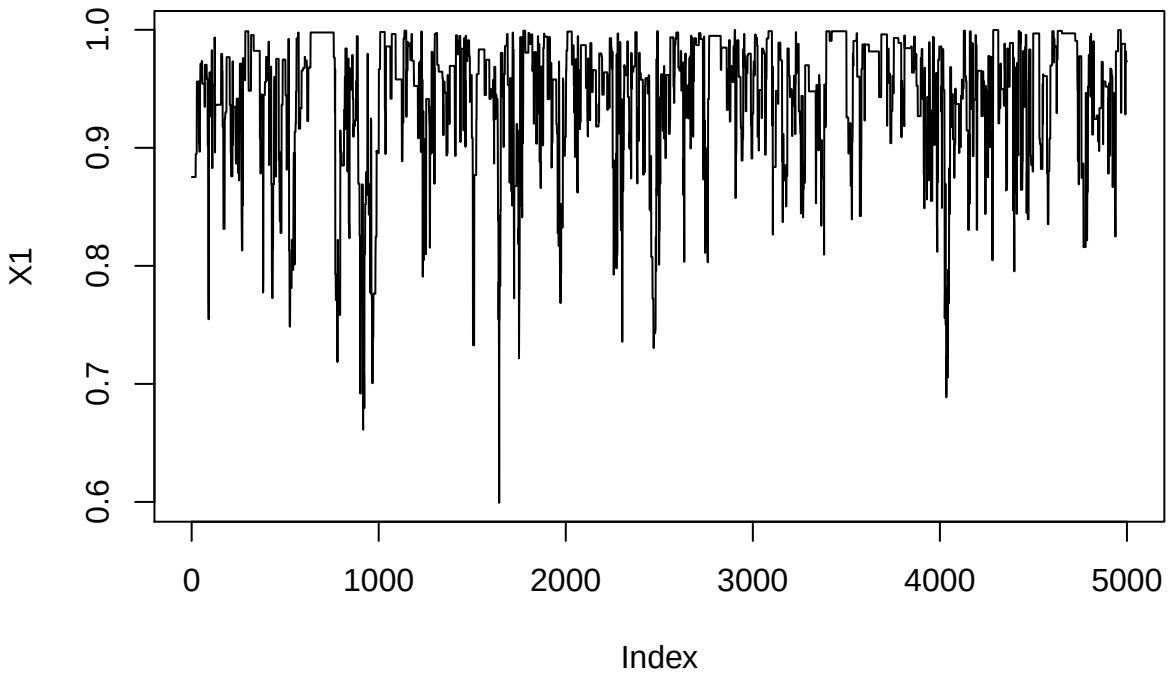
```
set.seed(1234)
result = run_Metropolis_5_dim(10^5 + 1000, 1000, 0.09, TRUE)
```

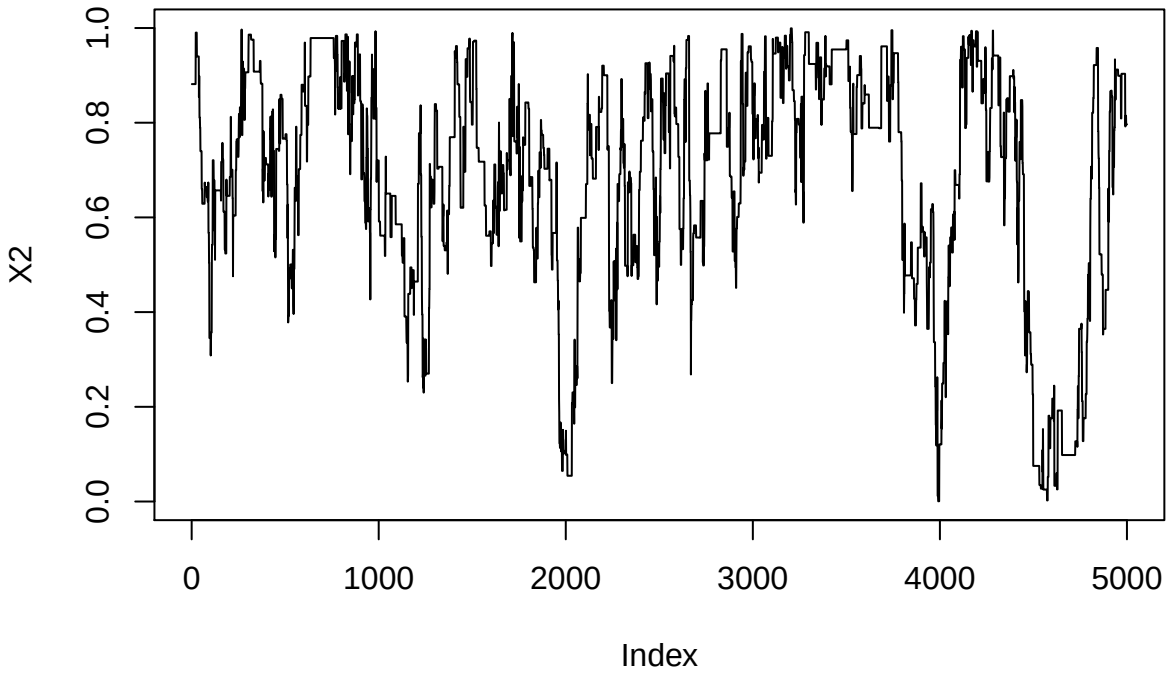
```
## ran Metropolis algorithm for 101000 iterations, with burn-in 1000
## acceptance rate = 0.2127525
## mean of h is about 0.6169371
## iid standard error would be about 0.0004858565
## varfact =  195.0813
## true standard error is about 0.006786031
## approximate 95% confidence interval is ( 0.6036364 , 0.6302377 )
```

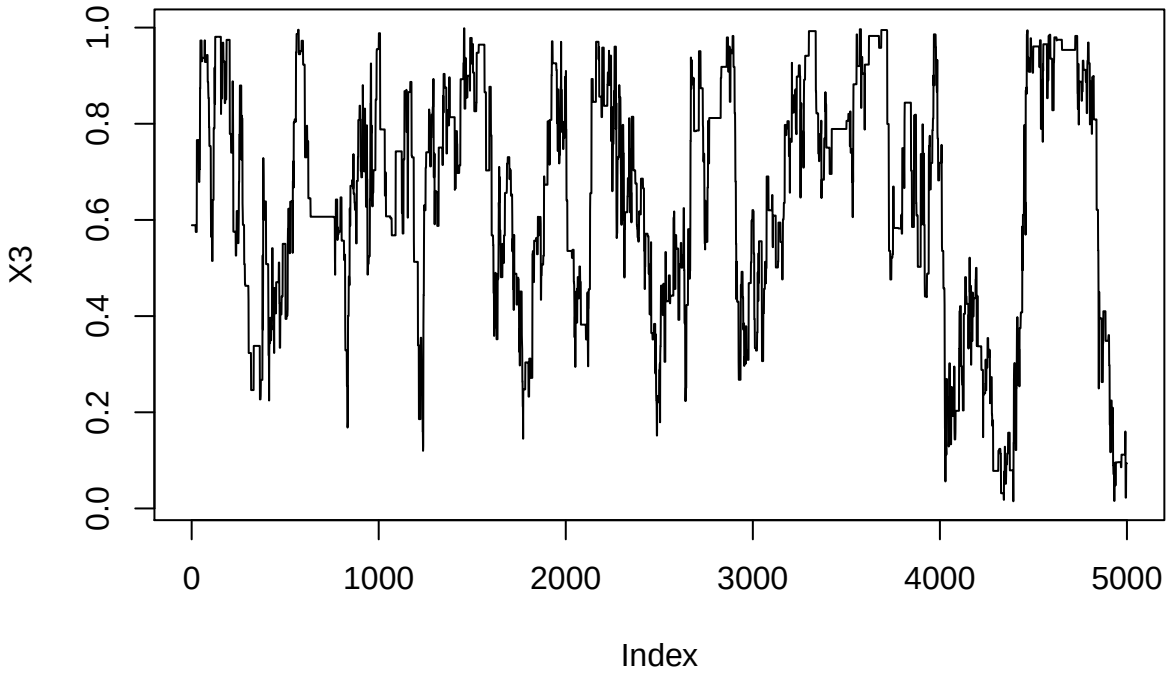
After a few runs, we found out that the best sigma is 0.09. We won't change the number of burn-in's as the analysis we did with $\sigma = 0.05$ should still holds (the higher the sigma, the faster we burn-in). We choose a sample size of 10^5. Varfact is still quite large but the standard error is relatively small.

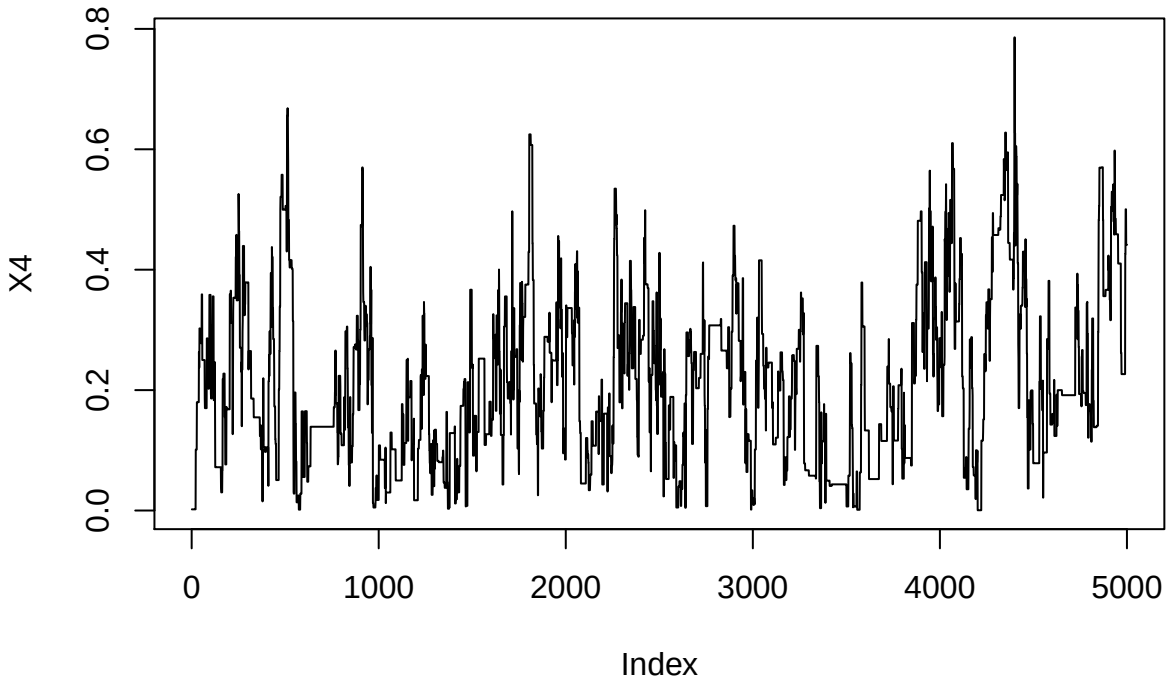
Let's look at the mixing of the values.

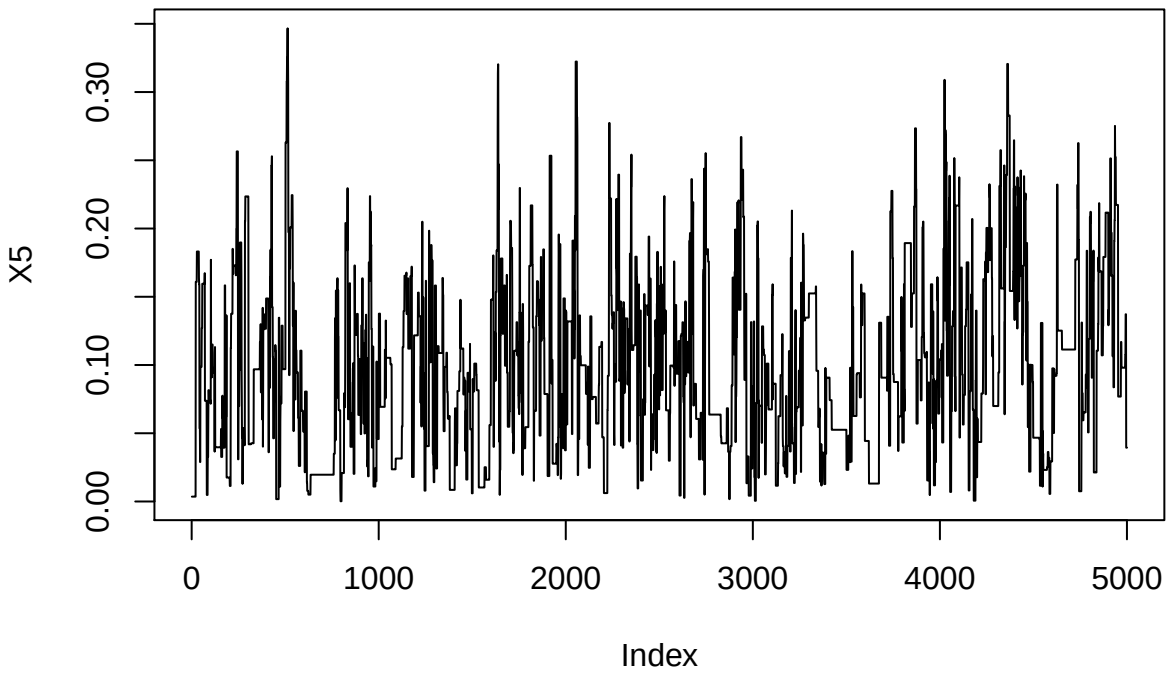
```
for (i in 1:5) {
  plot(result[[i]][1001: 6000], type='l', ylab=paste('X',i,sep=""))
}
```



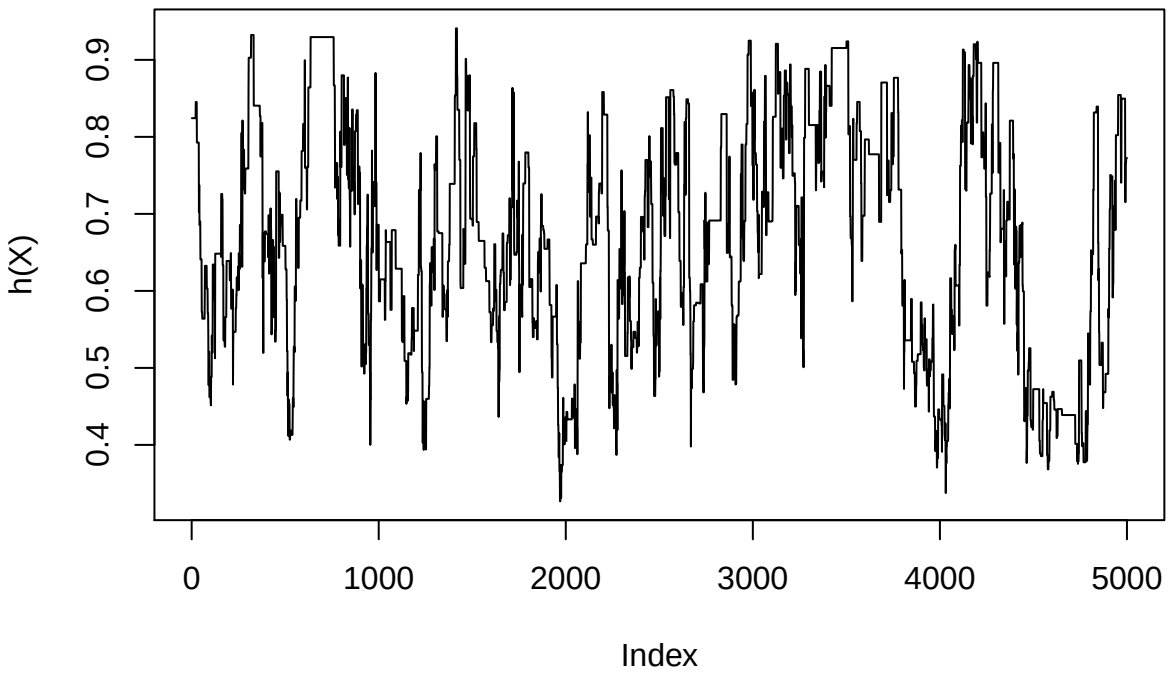

X2
0.0
0.2
0.4
0.6
0.8
1.0
0
1000
2000
3000
4000
5000
Index


X3
0.0
0.2
0.4
0.6
0.8
1.0
0
1000
2000
3000
4000
5000
Index


X4
0.0
0.2
0.4
0.6
0.8
0
1000
2000
3000
4000
5000
Index

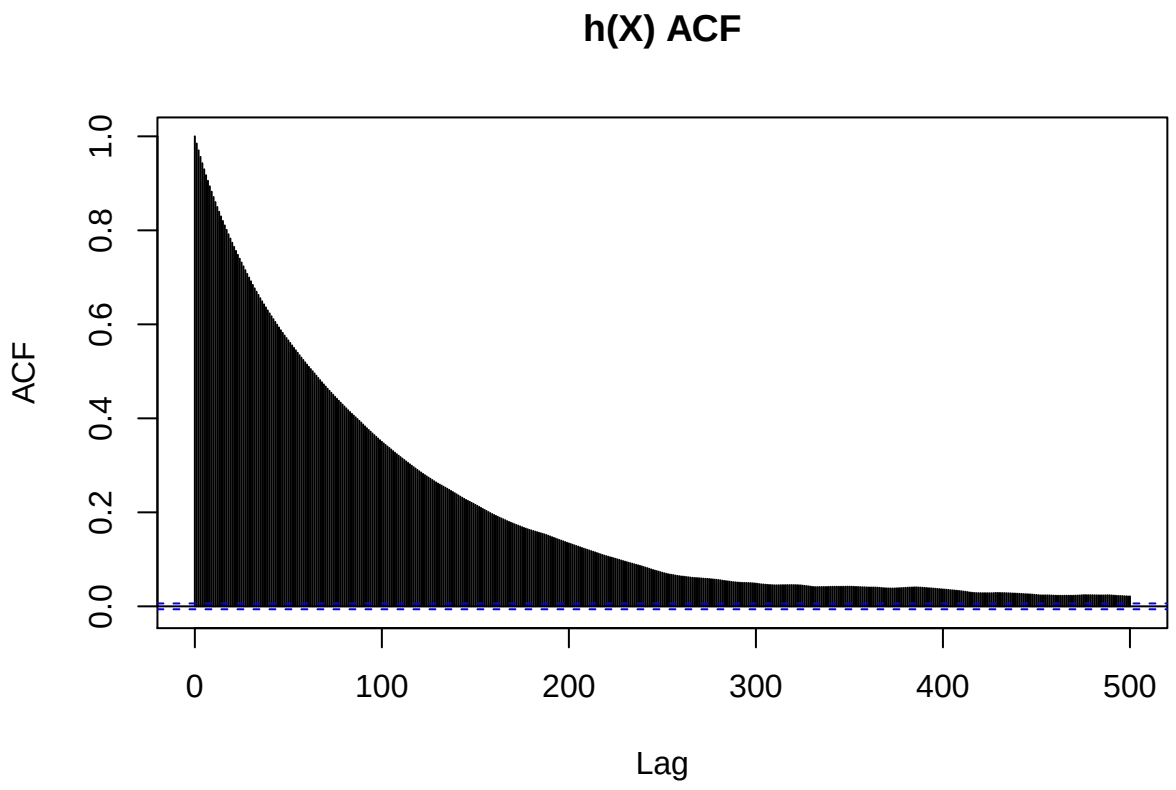



```
plot(result[[6]][1001:6000], type='l', ylab="h(X)")
```



We can see that both the samples and the estimates are mixed pretty well. None of the values have long plateaus. Let take a look at the 'acf'.

```
acf(result[[6]], lag.max = 500, main = "h(X) ACF")
```



As expected, the 'acf' function goes down really slowly. 2 reasons: σ is too small and rejection rate is pretty high. Let's also see how accurate the multi-dimensional Metropolis is.

```
set.seed(1234)
estimates = c(20)
for (i in 1:20) {
  estimates[i] = run_Metropolis_5_dim(10^5 + 1000, 1000, 0.09, FALSE)[[7]]
}

cat("The estimates are:\n")
```

```
## The estimates are:
```

```
estimates
```

```
##  [1] 0.6169371 0.6220863 0.6137634 0.6284997 0.6159683 0.6128396 0.6108968
##  [8] 0.6243432 0.6249094 0.6176597 0.6155481 0.6080012 0.6149200 0.6185227
## [15] 0.6133516 0.6282422 0.6117533 0.6254217 0.6086215 0.6131432
```

```
cat("Mean: ", mean(estimates), "\n")
```

```
## Mean:  0.6172715
```

```r
cat("Standard error: ", sd(estimates)/sqrt(20))
```

```
## Standard error:  0.001407324
```

The result is moderately accurate.

However, as shown in the line plot for the sample in 5 dimensions, each dimension has a different variance. Instead of using the same scaling factor for all dimensions, why not try a different factor by their standard deviation? Let's see if that gives us better result.

```r
sdlist = c(5)

for (i in 1:5) {
  sdlist[i] = sd(result[[i]][1001:length(result[[i]])])
}

run_Metropolis_5_dim_scale <- function(M, B, sigma, sdlist, print_result=TRUE) {
  X = runif(n=5, min=0, max=1)  # overdispersed starting distribution
  x1list = x2list = x3list = x4list = x5list = hlist = rep(0,M)
  numaccept = 0

  for (i in 1:M) {
    Y = X + sigma * sdlist * rnorm(5)  # proposal value
    U = runif(1)  # for accept/reject
    alpha = g(Y) / g(X)  # for accept/reject
    if (U < alpha) {
      X = Y  # accept proposal
      numaccept = numaccept + 1
    }
    x1list[i] = X[1]
    x2list[i] = X[2]
    x3list[i] = X[3]
    x4list[i] = X[4]
    x5list[i] = X[5]
    hlist[i] = h(X)
  }

  u = mean(hlist[(B+1):M])

  if (print_result) {
    cat("ran Metropolis algorithm for", M, "iterations, with burn-in", B, "\n")
    cat("acceptance rate =", numaccept/M, "\n")
    cat("mean of h is about", u, "\n")
    se1 =  sd(hlist[(B+1):M]) / sqrt(M-B)
    cat("iid standard error would be about", se1, "\n")

    varfact <- function(xxx) { 2 * sum(acf(xxx, plot=FALSE, lag.max=get_lag_max(hlist))$acf) - 1 }
    thevarfact = varfact(hlist[(B+1):M])
    se = se1 * sqrt( thevarfact )
    cat("varfact = ", thevarfact, "\n")
    cat("true standard error is about", se, "\n")
    cat("approximate 95% confidence interval is (", u - 1.96 * se, ",",
        u + 1.96 * se, ")\n\n")
```

```
  }

  return(list(x1list, x2list, x3list, x4list, x5list, hlist, u))
}
```

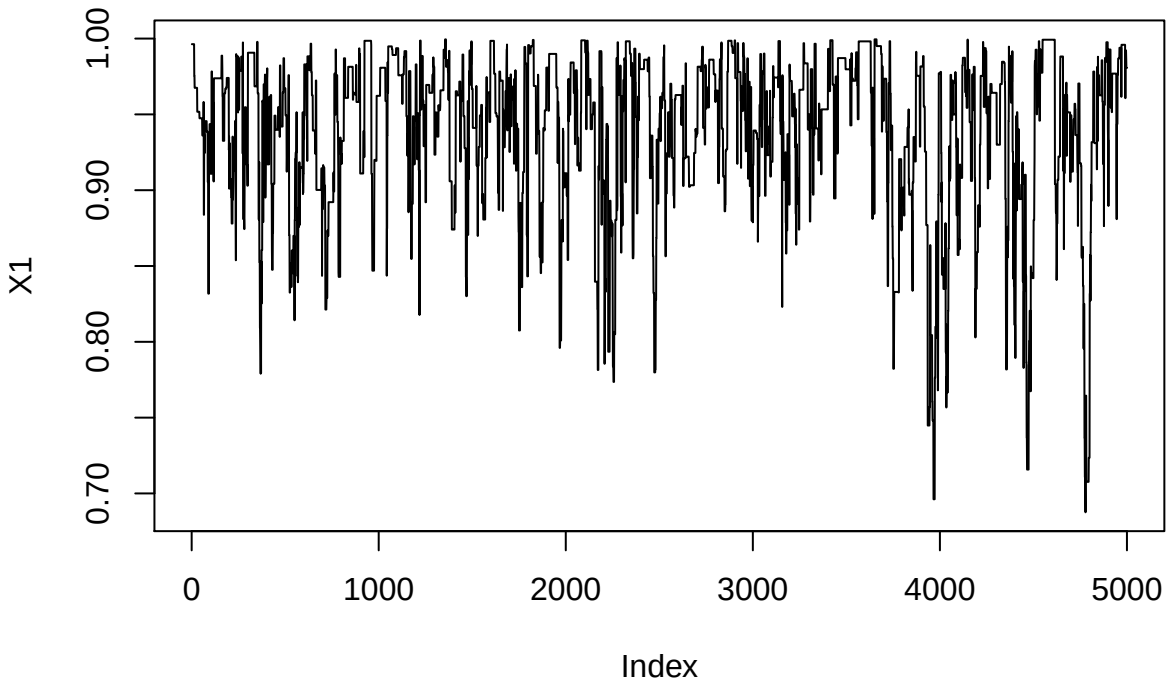
This method is correct because $q(X_1, X_2) = q(X_2, X_1)$. The rest of the proof is the same for normal Metropolis algorithm. Note this is not the exact definition with $Y_n \sim MVN(X_{n-1}, \sigma^2 I)$.

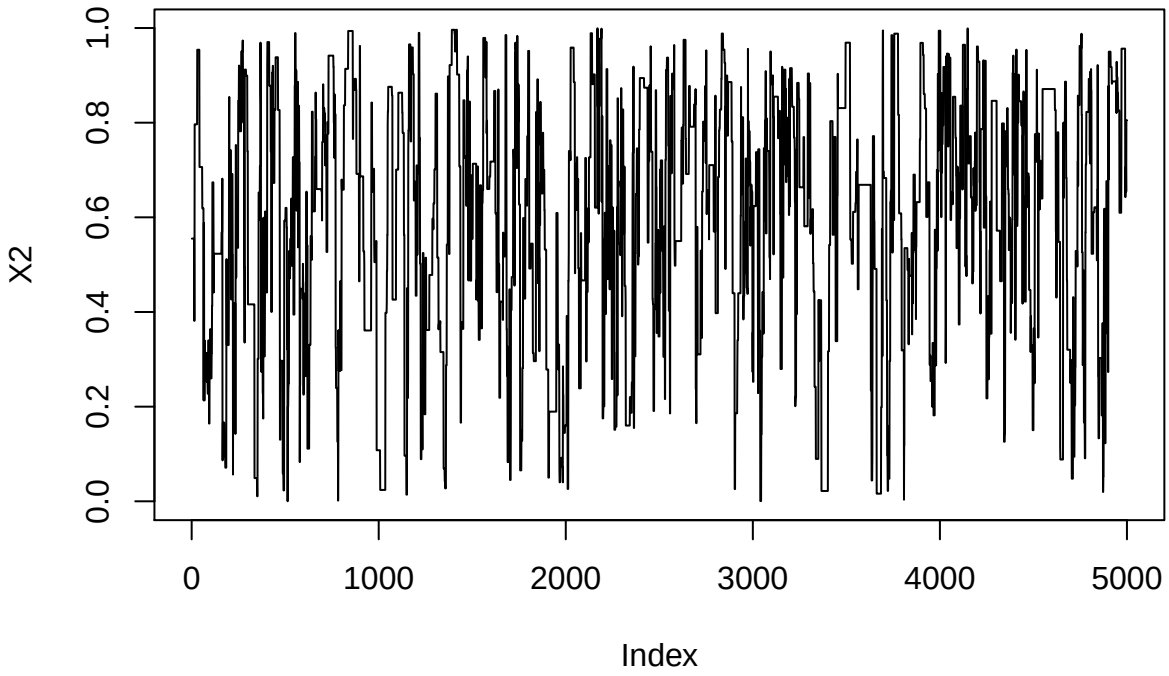
```
set.seed(1234)
result = run_Metropolis_5_dim_scale(10^5 + 1000, 1000, 0.9, sdlist, TRUE)
```

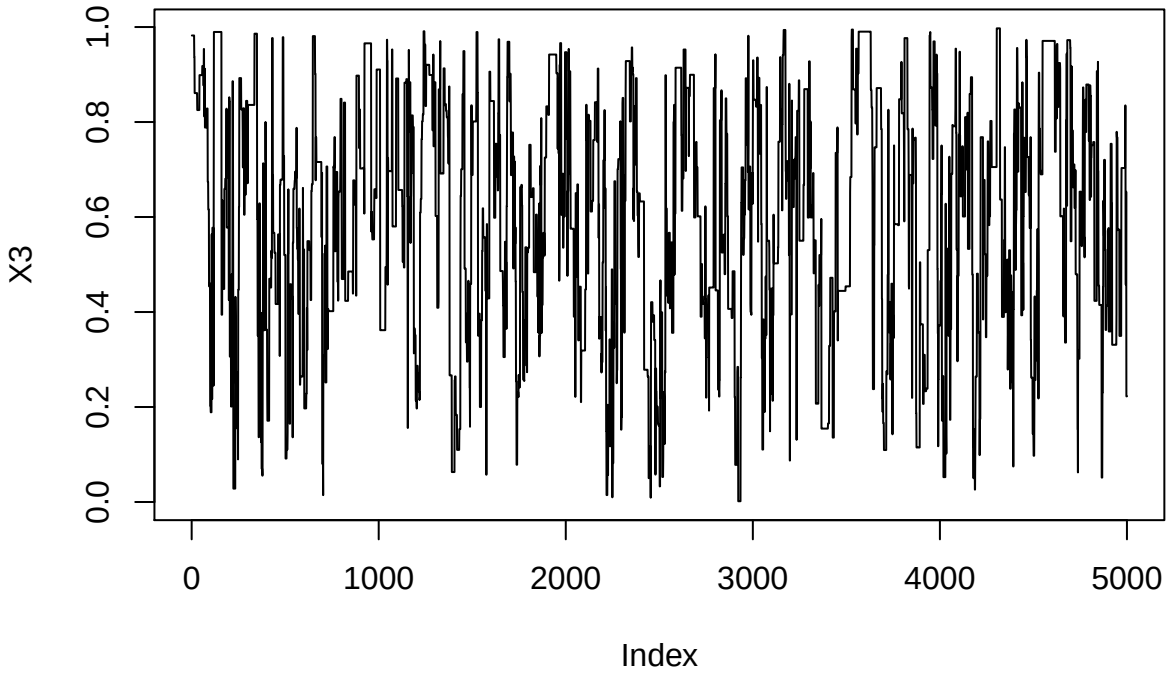
```
## ran Metropolis algorithm for 101000 iterations, with burn-in 1000
## acceptance rate = 0.2127228
## mean of h is about 0.6169798
## iid standard error would be about 0.0004711832
## varfact =  37.60423
## true standard error is about 0.002889403
## approximate 95% confidence interval is ( 0.6113166 , 0.6226431 )
```

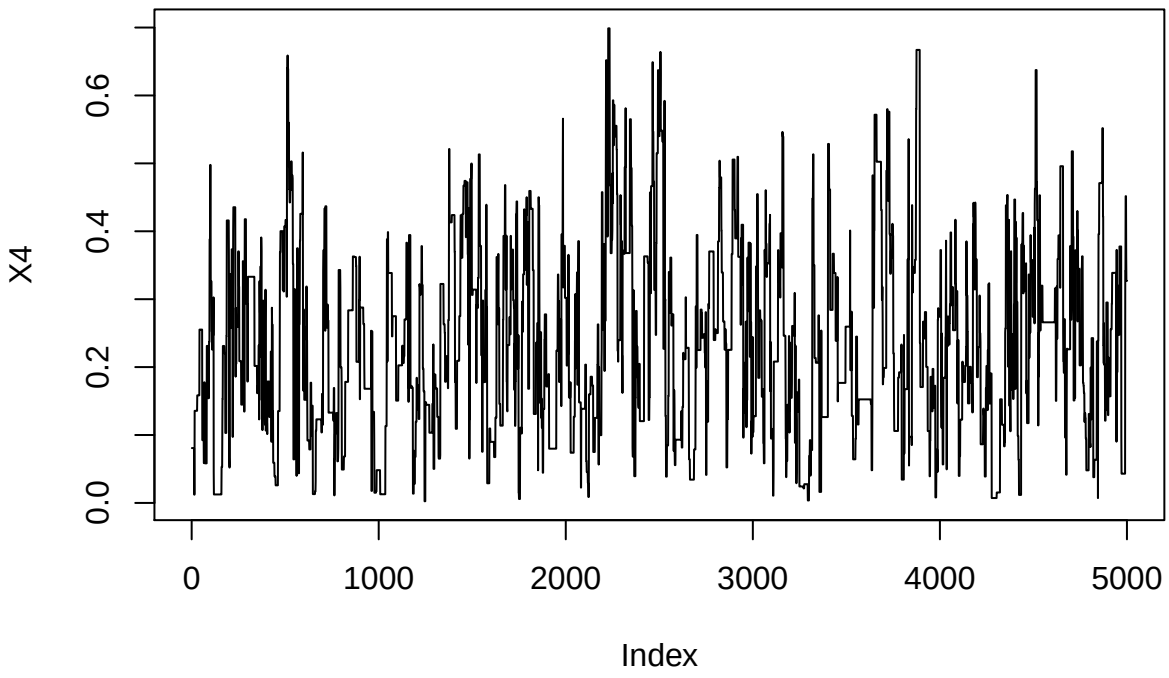
Nice! Varfact came down to just about 39. We are much more confidence about our prediction.

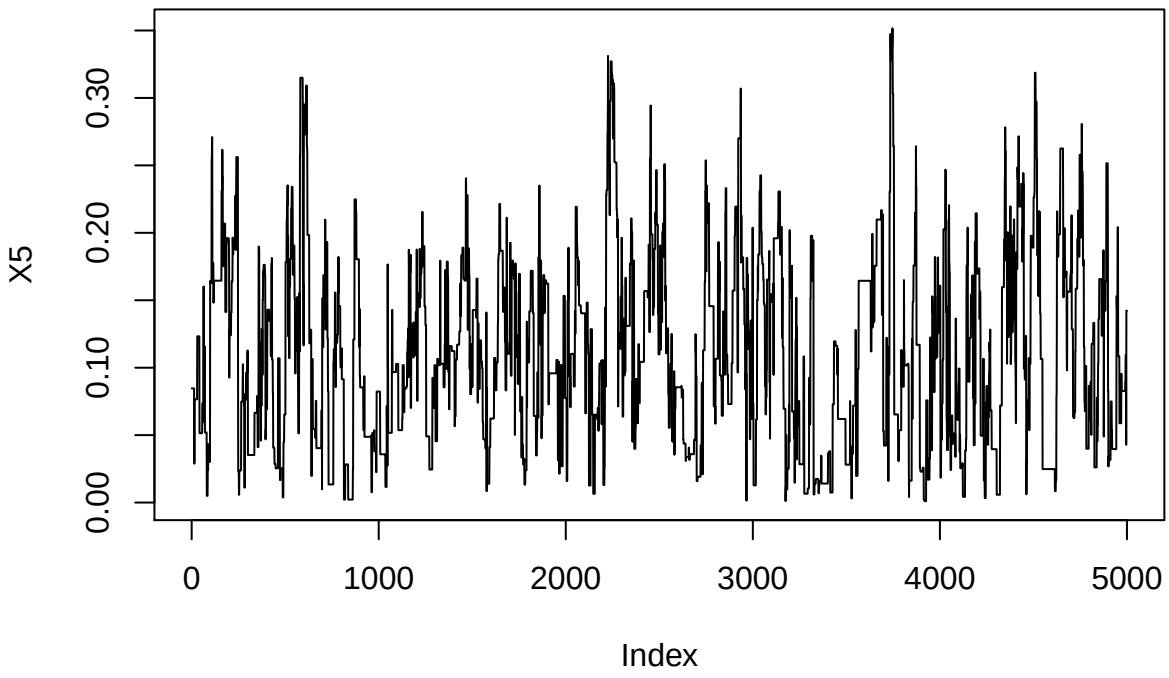
```
for (i in 1:5) {
  plot(result[[i]][1001: 6000], type='l', ylab=paste('X',i,sep=""))
}
```



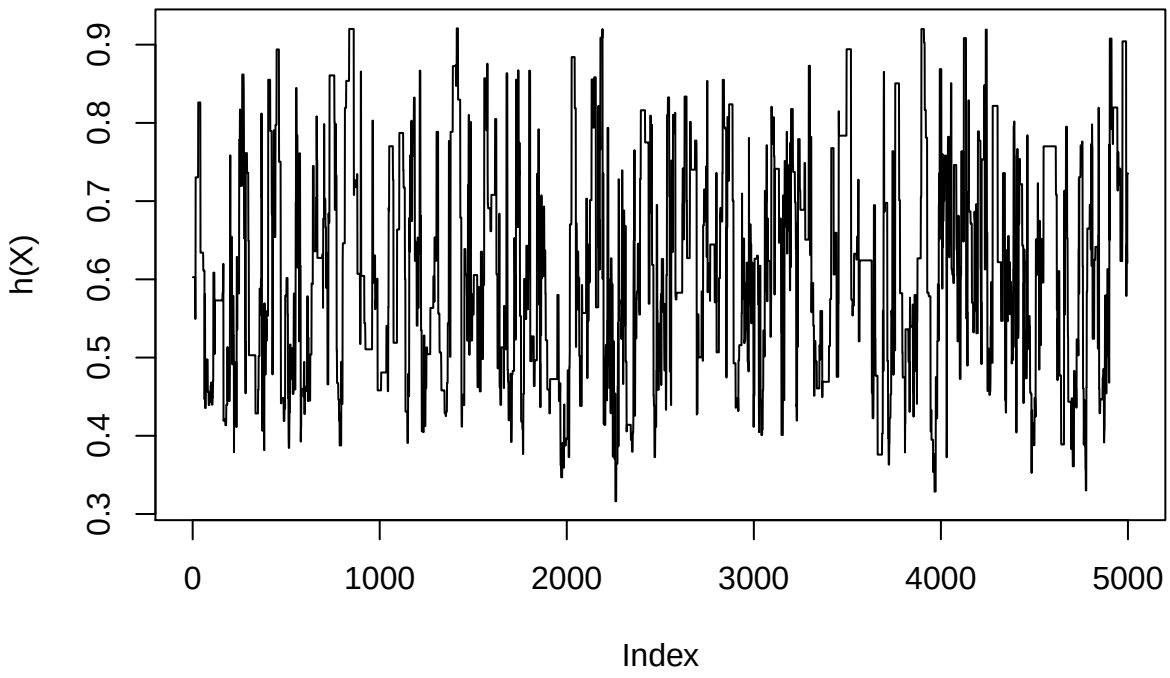

X2
0.0
0.2
0.4
0.6
0.8
1.0
0
1000
2000
3000
4000
5000
Index


X3
0.0
0.2
0.4
0.6
0.8
1.0
0
1000
2000
3000
4000
5000
Index


X4
0.0
0.2
0.4
0.6
0
1000
2000
3000
4000
5000
Index

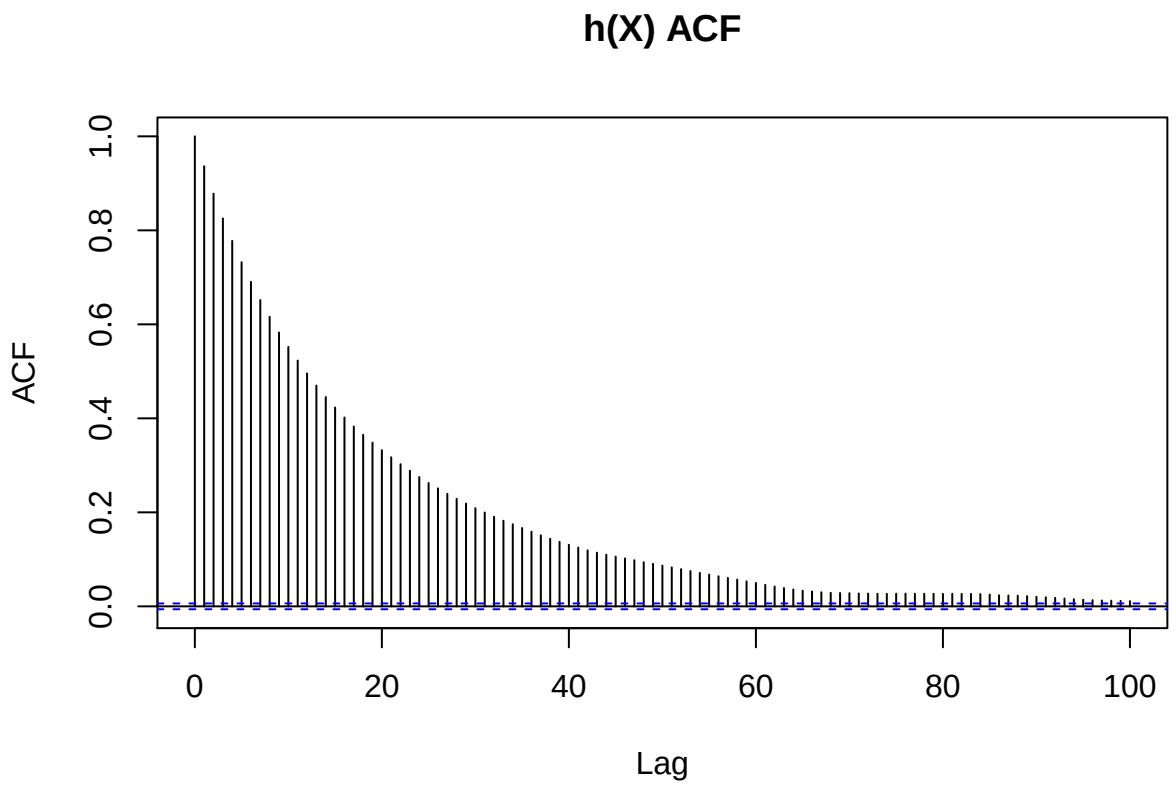



```
plot(result[[6]][1001:6000], type='l', ylab="h(X)")
```



We can see there are much less plateaus in this case.

```
acf(result[[6]], lag.max = 100, main = "h(X) ACF")
```



As we expected, 'acf' function approaches 0 much faster. I assume this is because we have much less plateaus even with similar acceptance rate and each dimension can change 'faster' than before.

Let's see the accuracy.

```
set.seed(1234)
estimates = c(20)
for (i in 1:20) {
  estimates[i] = run_Metropolis_5_dim_scale(10^5 + 1000, 1000, 0.9, sdlist, FALSE)[[7]]
}

cat("The estimates are:\n")
```

```
## The estimates are:
```

```
estimates
```

```
##  [1] 0.6169798 0.6161014 0.6157942 0.6221819 0.6189430 0.6141124 0.6174062
##  [8] 0.6204320 0.6148992 0.6215503 0.6187630 0.6140891 0.6137045 0.6172681
## [15] 0.6126910 0.6166365 0.6119420 0.6212142 0.6179390 0.6203156
```

```
cat("Mean: ", mean(estimates), "\n")
```

```
## Mean:  0.6171482
```

```r
cat("Standard error: ", sd(estimates)/sqrt(20))
```

```
## Standard error:  0.0006772256
```

The standard error is 2 times smaller and there are much less extreme values. This is much more accurate.

Comparing to importance sampler, Metropolis algorithm doesn't require any hand-picked density function f. We also don't need to worry about cancelling terms in the denominator or finding easy-to-sample distributions. Directly comparing the two examples, we were able achieve similar accuracy with 10 times less iterations using the Metropolis algorithm with component wise scaling.

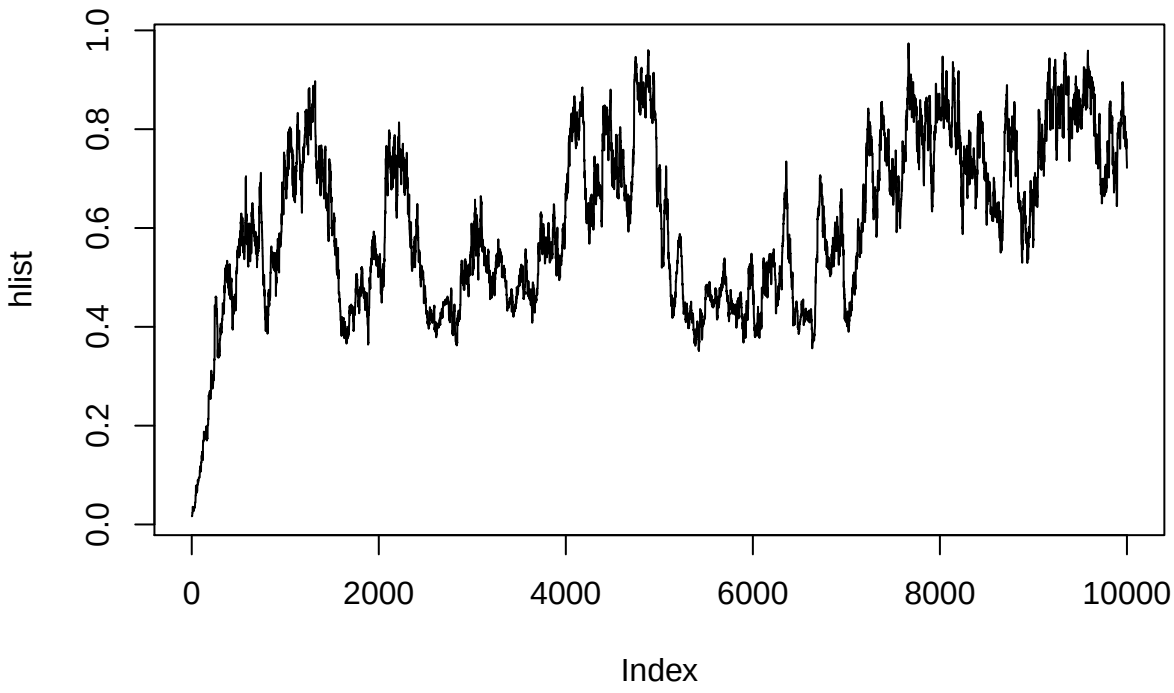
Comparing to rejection sampler, again, no need to handpick density function f. Also, all the samples in Metropolis are used, leading to better efficiency.

Metropolis algorithm also have some drawbacks. It requires more tuning, as the number of burn-in's and the scaling factor needs to be determined. Metropolis also suffers from high correlations between close estimates. increasing the true standard error.

Question 3

First, let's see how much burn-in's are needed. We expect it to take longer as only one coordinate moves at a time.

```r
set.seed(123)
N = 10000
X = runif(n=5, min=0, max=0.1)
sigma = 0.05  # proposal scaling
x1list = x2list = x3list = x4list = x5list = hlist = rep(0,N)  # for keeping track of values

for (i in 1:N) {
  coord = i %% 5 + 1 # systematic scan
  Y = X
  Y[coord] = X[coord] + sigma * rnorm(1)
  U = runif(1)  # for accept/reject
  alpha = g(Y) / g(X)  # for accept/reject
  if (U < alpha) {
    X = Y  # accept proposal
  }
  x1list[i] = X[1]
  x2list[i] = X[2]
  hlist[i] = h(X)
}

plot(hlist, type='l')
```



As expect, more burn-in's are required. Let's choose 4000 burn-in's just to be safe.

Again, let's formalize the findings from last question. First we create a function for getting a list of standard deviations. We will use the same component-wise algorithm so that the variance is more accurately reflected. We will choose sigma by acceptance rate.

Idea: maybe we should use higher acceptance rate? Intuitively that will help the variance to be more 'pronounced', but is that true? Let's try with 50% acceptance rate.

```
get_sdlist <- function(M, B, sigma) {
  X = runif(n=5, min=0, max=1)  # overdispersed starting distribution
  x1list = x2list = x3list = x4list = x5list = rep(0,M)
  numaccept = 0

  for (i in 1:M) {
    coord = i %% 5 + 1 # systematic scan
    Y = X
    Y[coord] = X[coord] + sigma * rnorm(1)
    U = runif(1)  # for accept/reject
    alpha = g(Y) / g(X)  # for accept/reject
    if (U < alpha) {
      X = Y  # accept proposal
      numaccept = numaccept + 1
    }
    x1list[i] = X[1]
    x2list[i] = X[2]
    x3list[i] = X[3]
```

```
    x4list[i] = X[4]
    x5list[i] = X[5]
  }

  cat("ran Metropolis algorithm for", M, "iterations, with burn-in", B, "\n")
  cat("acceptance rate =", numaccept/M, "\n")

  return(c(sd(x1list), sd(x2list), sd(x3list), sd(x4list), sd(x5list)))
}

set.seed(123)
sdlist = get_sdlist(10^5 + 4000, 4000, 0.2)
```

```
## ran Metropolis algorithm for 104000 iterations, with burn-in 4000
## acceptance rate = 0.5261442
```

Now, let's use the standard deviation we had before to conduct component-wise MCMC.

```
run_Metropolis_cmpnt_scale <- function(M, B, sigma, sdlist, print_result=TRUE) {
  X = runif(n=5, min=0, max=1)  # overdispersed starting distribution
  x1list = x2list = x3list = x4list = x5list = hlist = rep(0,M)
  numaccept = 0

  for (i in 1:M) {
    coord = i %% 5 + 1 # systematic scan
    Y = X
    Y[coord] = X[coord] + sigma * sdlist[coord] * rnorm(1)
    U = runif(1)  # for accept/reject
    alpha = g(Y) / g(X)  # for accept/reject
    if (U < alpha) {
      X = Y  # accept proposal
      numaccept = numaccept + 1
    }
    x1list[i] = X[1]
    x2list[i] = X[2]
    x3list[i] = X[3]
    x4list[i] = X[4]
    x5list[i] = X[5]
    hlist[i] = h(X)
  }

  u = mean(hlist[(B+1):M])

  if (print_result) {
    cat("ran Metropolis algorithm for", M, "iterations, with burn-in", B, "\n")
    cat("acceptance rate =", numaccept/M, "\n")
    cat("mean of h is about", u, "\n")
    se1 =  sd(hlist[(B+1):M]) / sqrt(M-B)
    cat("iid standard error would be about", se1, "\n")

    varfact <- function(xxx) { 2 * sum(acf(xxx, plot=FALSE, lag.max=get_lag_max(hlist))$acf) - 1 }
    thevarfact = varfact(hlist[(B+1):M])
    se = se1 * sqrt( thevarfact )
```

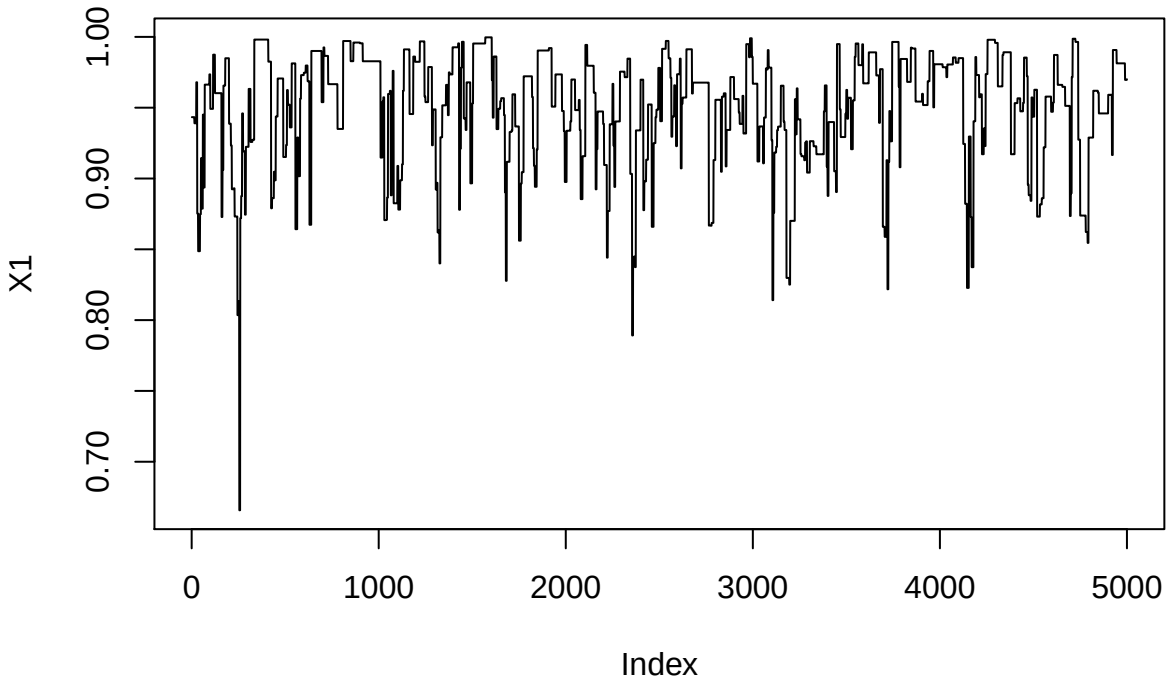
```
    cat("varfact = ", thevarfact, "\n")
    cat("true standard error is about", se, "\n")
    cat("approximate 95% confidence interval is (", u - 1.96 * se, ",",
        u + 1.96 * se, ")\n\n")
  }

  return(list(x1list, x2list, x3list, x4list, x5list, hlist, u))
}
```

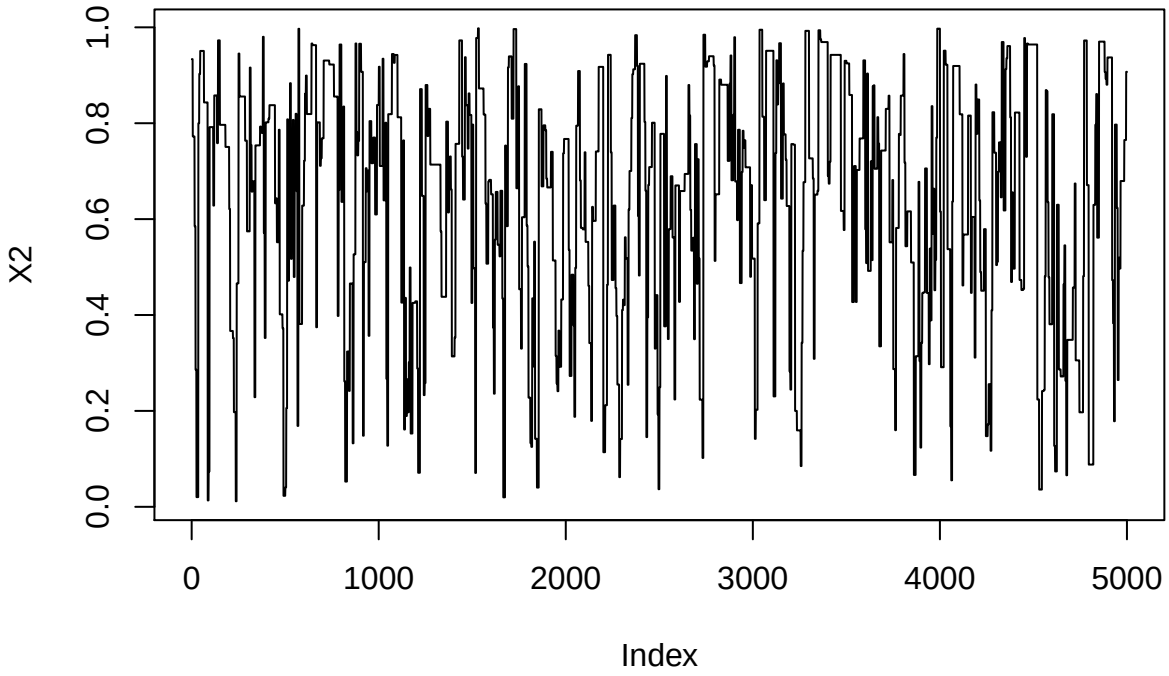
```
set.seed(1234)
result = run_Metropolis_cmpnt_scale(10^5 + 4000, 4000, 1.6, sdlist, TRUE)
```

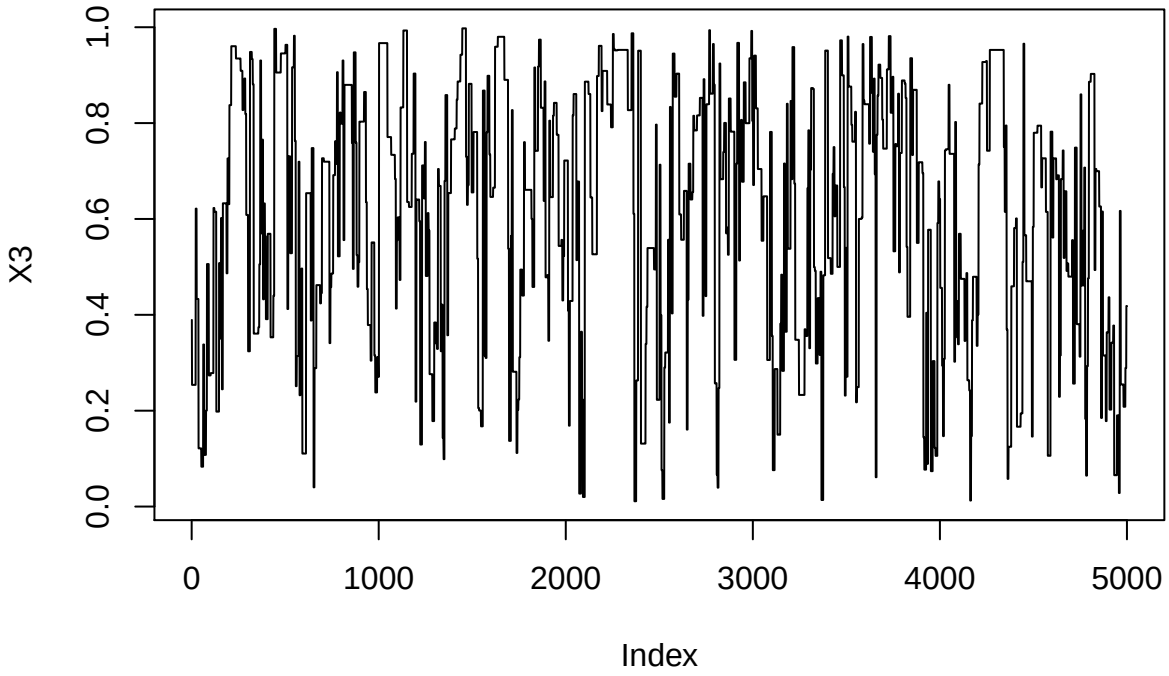
```
## ran Metropolis algorithm for 104000 iterations, with burn-in 4000
## acceptance rate = 0.5039231
## mean of h is about 0.6163739
## iid standard error would be about 0.0004781272
## varfact =  39.03123
## true standard error is about 0.002987099
## approximate 95% confidence interval is ( 0.6105192 , 0.6222287 )
```

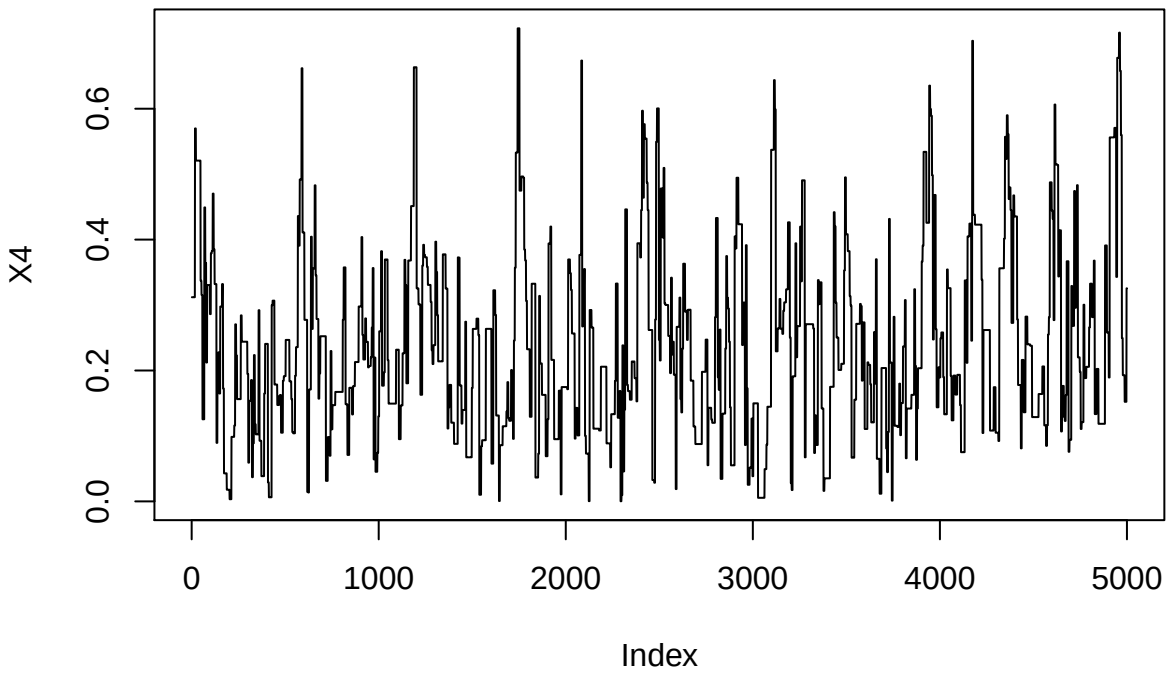
Aiming for 50% acceptance rate, the sigma we choose is much larger for component-wise MCMC. Note that σ here doesn't correspond to that in the original MCMC algorithm. With varfact at around 39, the true standard error is pretty small. We also have a pretty tight confidence interval.

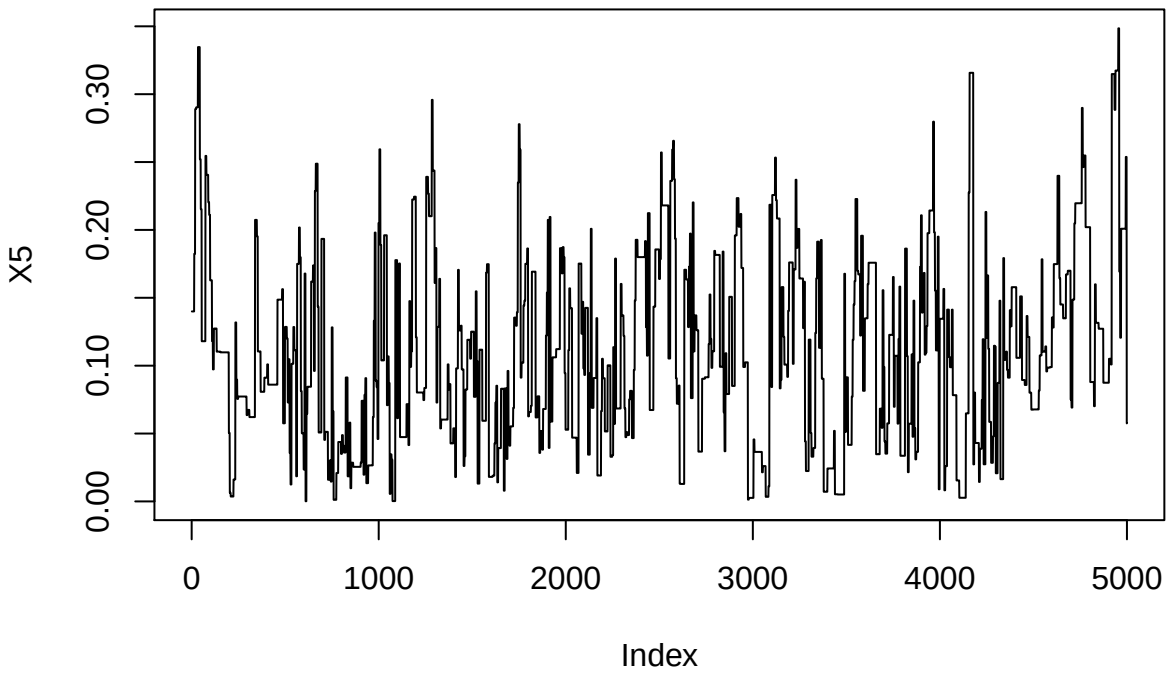
```
for (i in 1:5) {
  plot(result[[i]][1001: 6000], type='l', ylab=paste('X',i,sep=""))
}
```


X1
1.00
0.90
0.80
0.70
0
1000
2000
3000
4000
5000
Index

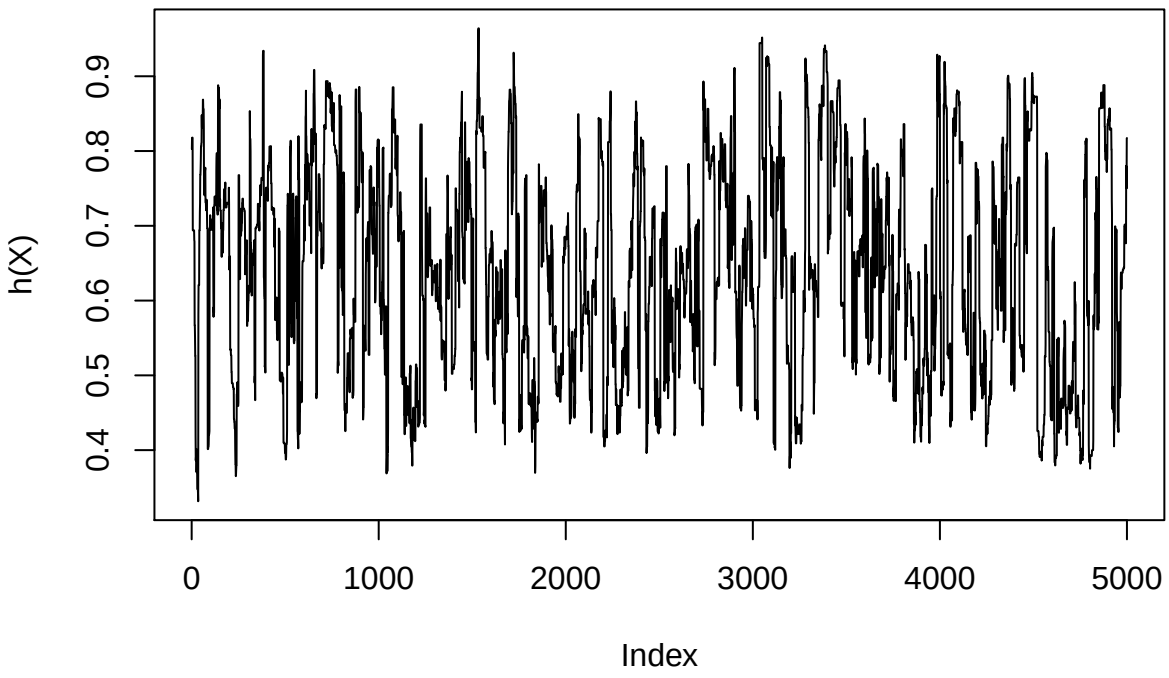

X2
0.0
0.2
0.4
0.6
0.8
1.0
0
1000
2000
3000
4000
5000
Index


X3
0.0
0.2
0.4
0.6
0.8
1.0
0
1000
2000
3000
4000
5000
Index


X4
0.0
0.2
0.4
0.6
0
1000
2000
3000
4000
5000
Index

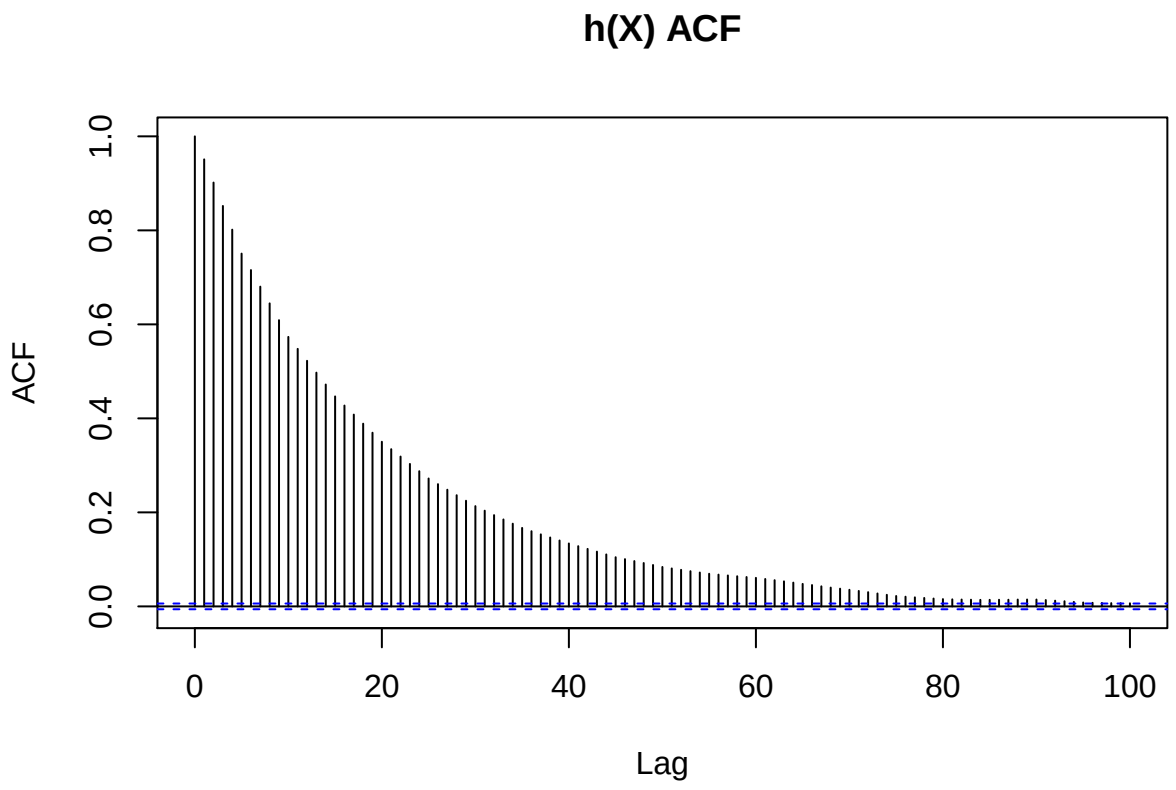



```
plot(result[[6]][1001:6000], type='l', ylab="h(X)")
```



Each dimension is plateauing due to fact that we are using systematic-scan. We can see that both the samples and the estimates are mixed pretty well. $h(X)$ is pretty evenly spread and have little to no plateauing. Let take a look at the 'acf'.

```
acf(result[[6]], lag.max = 100, main = "h(X) ACF")
```



Interesting. ACF comes down even faster than non-componentwise MCMC with component wise scaling. I suspect this is because we now have higher acceptance rate.

```
set.seed(1234)
estimates = c(20)
for (i in 1:20) {
  estimates[i] = run_Metropolis_cmpnt_scale(10^5 + 4000, 4000, 1.6, sdlist, FALSE)[[7]]
}

cat("The estimates are:\n")
```

```
## The estimates are:
```

```
estimates
```

```
##  [1] 0.6163739 0.6150108 0.6196538 0.6153548 0.6122143 0.6184561 0.6169165
##  [8] 0.6176260 0.6216244 0.6180214 0.6207198 0.6207929 0.6186033 0.6155354
## [15] 0.6182319 0.6216729 0.6210054 0.6154851 0.6166555 0.6159146
```

```
cat("Mean: ", mean(estimates), "\n")
```

```
## Mean:  0.6177934
```

```
cat("Standard error: ", sd(estimates)/sqrt(20))
```

```
## Standard error:  0.0005741919
```

Nice! The standard error for 20 i.i.d. estimates is even smaller than non-componentwise MCMC with scaling. We have a pretty accurate estimator.

Let's also compare this with normal component wise MCMC.

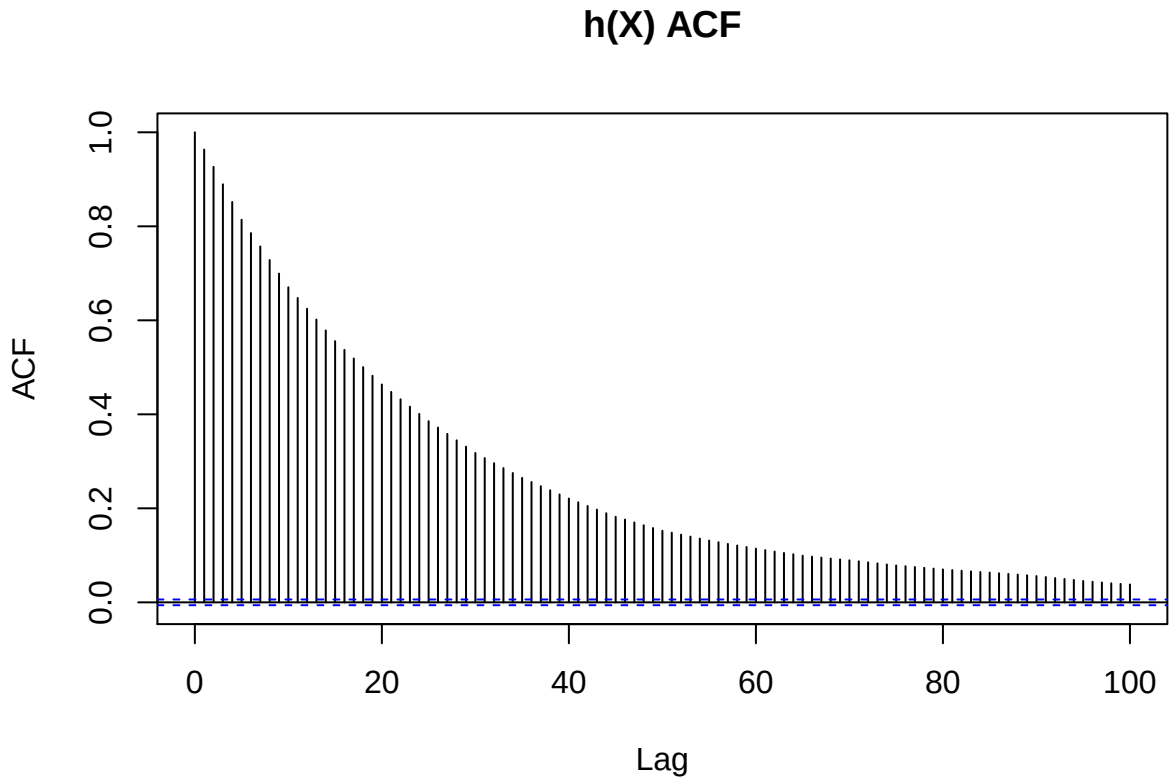
```
run_Metropolis_cmpnt <- function(M, B, sigma, print_result=TRUE) {
  X = runif(n=5, min=0, max=1)  # overdispersed starting distribution
  x1list = x2list = x3list = x4list = x5list = hlist = rep(0,M)
  numaccept = 0

  for (i in 1:M) {
    coord = i %% 5 + 1 # systematic scan
    Y = X
    Y[coord] = X[coord] + sigma * rnorm(1)
    U = runif(1)  # for accept/reject
    alpha = g(Y) / g(X)  # for accept/reject
    if (U < alpha) {
      X = Y  # accept proposal
      numaccept = numaccept + 1
    }
    x1list[i] = X[1]
    x2list[i] = X[2]
    x3list[i] = X[3]
    x4list[i] = X[4]
    x5list[i] = X[5]
    hlist[i] = h(X)
  }

  u = mean(hlist[(B+1):M])

  if (print_result) {
    cat("ran Metropolis algorithm for", M, "iterations, with burn-in", B, "\n")
    cat("acceptance rate =", numaccept/M, "\n")
    cat("mean of h is about", u, "\n")
    se1 =  sd(hlist[(B+1):M]) / sqrt(M-B)
    cat("iid standard error would be about", se1, "\n")

    varfact <- function(xxx) { 2 * sum(acf(xxx, plot=FALSE, lag.max=get_lag_max(hlist))$acf) - 1 }
    thevarfact = varfact(hlist[(B+1):M])
    se = se1 * sqrt( thevarfact )
    cat("varfact = ", thevarfact, "\n")
    cat("true standard error is about", se, "\n")
    cat("approximate 95% confidence interval is (", u - 1.96 * se, ",",
        u + 1.96 * se, ")\n\n")
  }

  return(list(x1list, x2list, x3list, x4list, x5list, hlist, u))
}
```

```
set.seed(1234)
result = run_Metropolis_cmpnt(10^5 + 4000, 4000, 0.25, TRUE)
```

```
## ran Metropolis algorithm for 104000 iterations, with burn-in 4000
## acceptance rate = 0.4707212
## mean of h is about 0.6193682
## iid standard error would be about 0.0004776875
## varfact =  59.68035
## true standard error is about 0.003690282
## approximate 95% confidence interval is ( 0.6121353 , 0.6266012 )
```

With the same acceptance rate, component wise scaling again has lower varfact. I assume this is because the number of rejection for each dimension is "not balanced". When every dimension changes with the same rate, there will be certain dimensions that are rejected more often. Therefore leading to worse results.

```
acf(result[[6]], lag.max = 100, main = "h(X) ACF")
```



The analysis reflects on 'acf' function, which gets to 0 a bit slower comparing to component wise scaling.