# HW2

Haoluan Chen

10/17/2021

Name: Haoluan Chen
Student number: 1003994261
Department: Staitstial Science Program: Master of Statistics
Year: 1
Email: haoluan.chen@mail.utoronto.ca

## Q1

```
set.seed(1)
g <- function(x) {
  exp(-x^4/6)
}

h <- function(x){
  x^2
}
```

Let $\sigma^2 = 1$

```
M = 5000  # run length
B = 500 # amount of burn-in
X = runif(1)  # overdispersed starting distribution
sigma = 1  # proposal scaling
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);
}

cat("ran Metropolis algorithm for", M, "iterations, with burn-in", B, "\n");
```

```
## ran Metropolis algorithm for 5000 iterations, with burn-in 500
```

```
cat("acceptance rate =", numaccept/M, "\n");
```

```
## acceptance rate = 0.7114
```

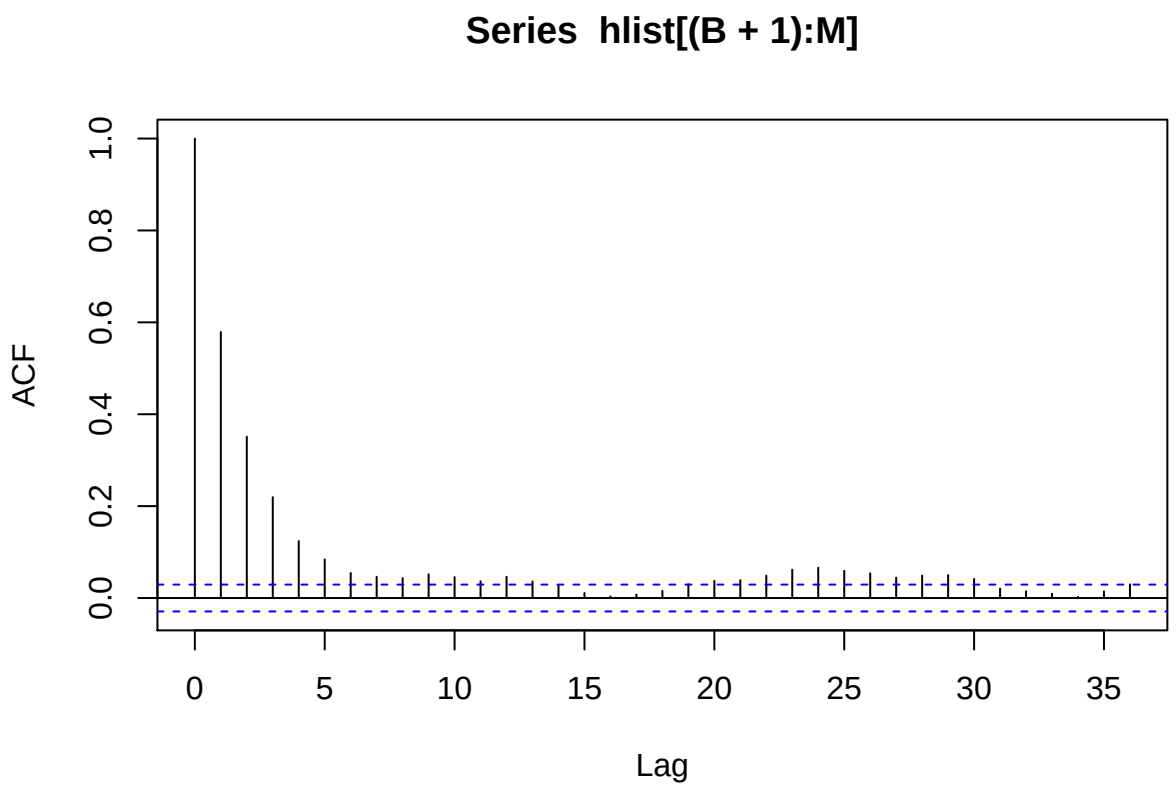
```
u = mean(hlist[(B+1):M])
cat("mean of h is about", u, "\n")
```

```
## mean of h is about 0.8514412
```

```
se1 =  sd(hlist[(B+1):M]) / sqrt(M-B)
cat("iid standard error would be about", se1, "\n")
```

```
## iid standard error would be about 0.01407701
```

```
acf(hlist[(B+1):M])
```




Looking at the acf plot, we see that there is no significant correlation between values of lag 15.

```
varfact <- function(xxx) { 2 * sum(acf(xxx, lag.max = 15, plot=FALSE)$acf) - 1 }
thevarfact = varfact(hlist[(B+1):M])
se = se1 * sqrt( thevarfact )
cat("varfact = ", thevarfact, "\n")
```

```
## varfact =  4.510536
```

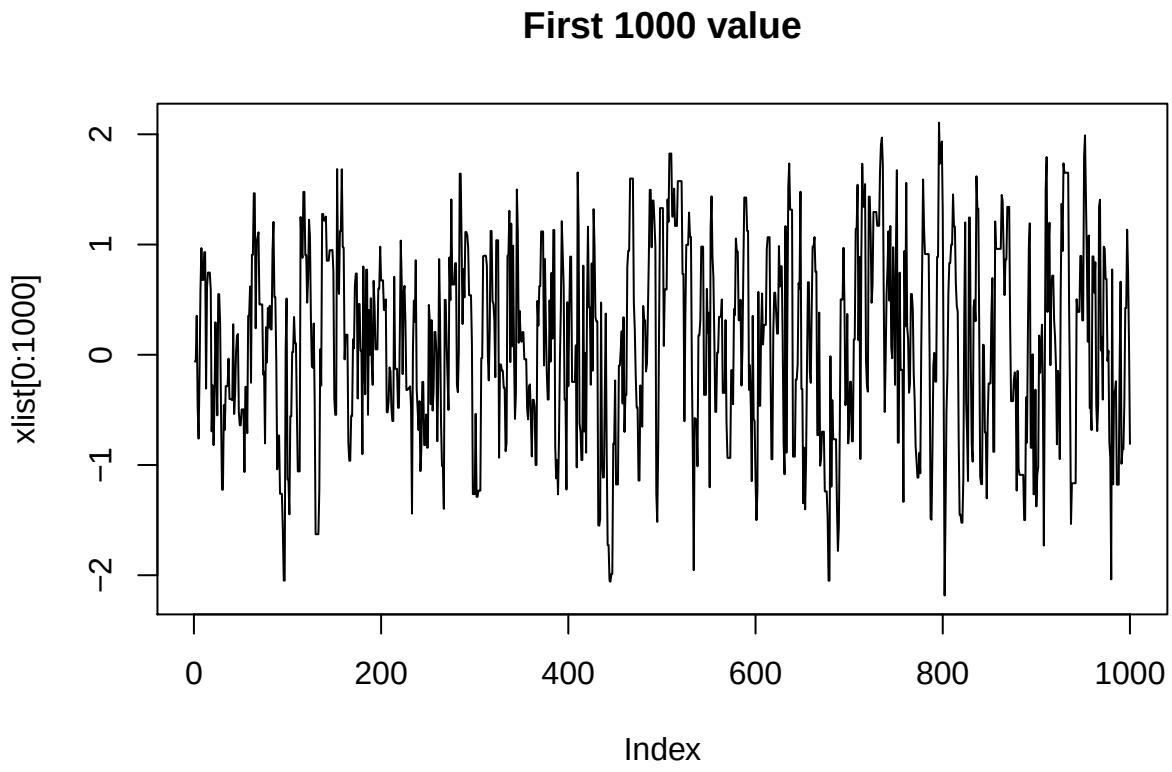
```
cat("true standard error is about", se, "\n")
```

```
## true standard error is about 0.02989678
```

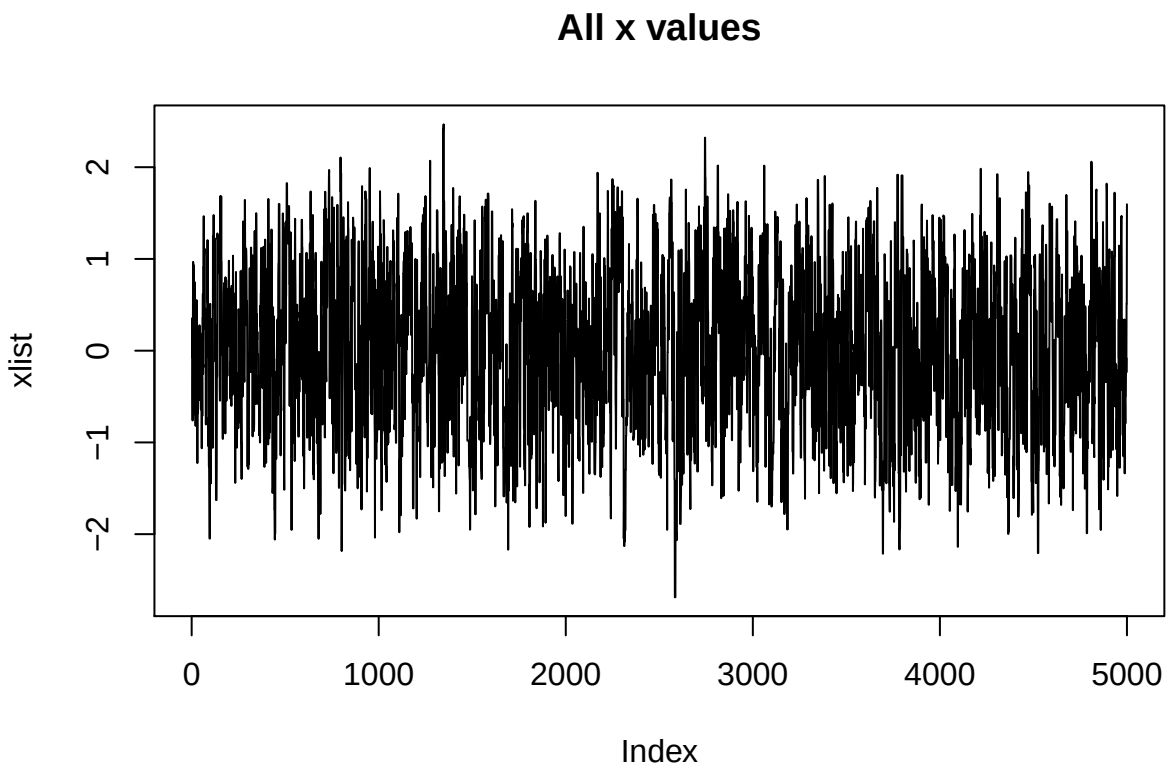
```
cat("approximate 95% confidence interval is (", u - 1.96 * se, ",",
                        u + 1.96 * se, ")\n\n")
```

```
## approximate 95% confidence interval is ( 0.7928435 , 0.9100389 )
```

```
plot(xlist[0:1000], type='l', main = "First 1000 value")
```




```
plot(xlist, type='l', main = "All x values")
```




From above output, we see that using $\sigma^2 = 1$, we have a good mixing for the x values after first 500 values, So I think using burning-in of 500 is good. My estimate is 0.8554118 with 72% acceptance rate. Since there is no significant correlation between values of lag 15, I used lag of 15 to calculate the varfact and I got variance of 0.0029 and my confident interval is kind of wide.

Let $\sigma^2 = 0.1$

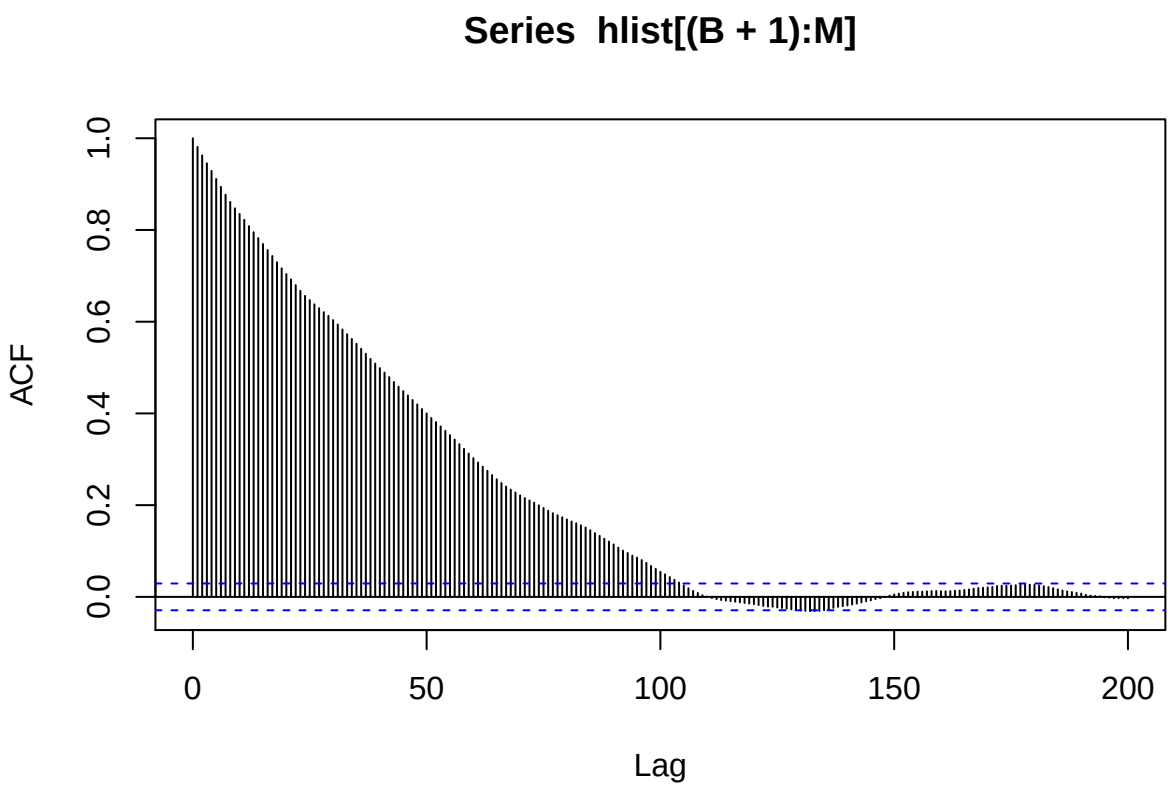
```
M = 5000  # run length
B = 500  # amount of burn-in
X = runif(1)  # overdispersed starting distribution
sigma = 0.1  # proposal scaling
xlist = rep(0,M)  # for keeping track of chain values
hlist = rep(0,M)  # for keeping track of h function values
numaccept = 0;
for (i in 1:M) {
    Y = X + sigma * 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);
}

cat("ran Metropolis algorithm for", M, "iterations, with burn-in", B, "\n");
```

```
## ran Metropolis algorithm for 5000 iterations, with burn-in 500
cat("acceptance rate =", numaccept/M, "\n");
```

```
## acceptance rate = 0.968
u = mean(hlist[(B+1):M])
cat("mean of h is about", u, "\n")
```

```
## mean of h is about 0.9406154
se1 =  sd(hlist[(B+1):M]) / sqrt(M-B)
cat("iid standard error would be about", se1, "\n")
```

```
## iid standard error would be about 0.01405283
acf(hlist[(B+1):M], lag.max = 200)
```




```
varfact <- function(xxx) { 2 * sum(acf(xxx, plot=FALSE, lag.max = 200)$acf) - 1 }
thevarfact = varfact(hlist[(B+1):M])
se = se1 * sqrt( thevarfact )
cat("varfact = ", thevarfact, "\n")
```

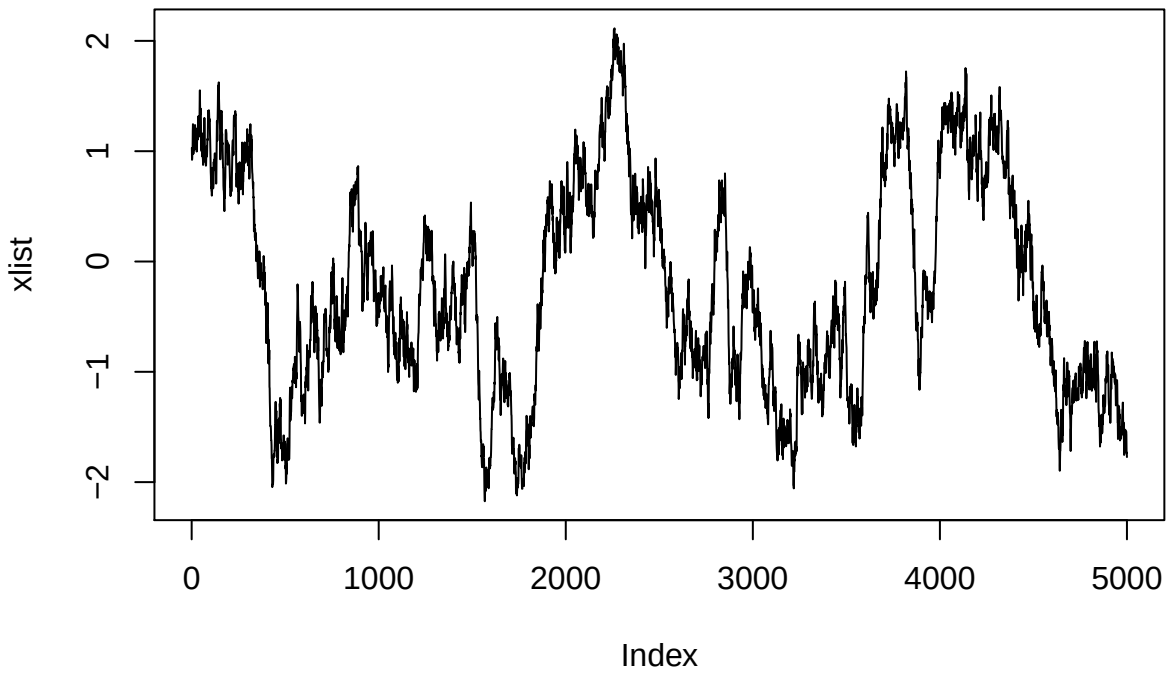
```
## varfact =  87.85672
cat("true standard error is about", se, "\n")
```

```
## true standard error is about 0.1317199
cat("approximate 95% confidence interval is (", u - 1.96 * se, ",",
                        u + 1.96 * se, ")\n\n")
```

```
## approximate 95% confidence interval is ( 0.6824444 , 1.198786 )
```

```
plot(xlist[0:1000], type='l')
```


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




From above output, we see that using $\sigma^2 = 0.1$, we do not have a good mixing for the x values. The acceptance rate is around 0.95, which is too high. The ACF varies a lot when I rerun the code, and the variance is not stable. The 95% confidence interval is very wide. So $\sigma^2 = 0.1$ does not estimate the expected value well.

Let $\sigma^2 = 10$

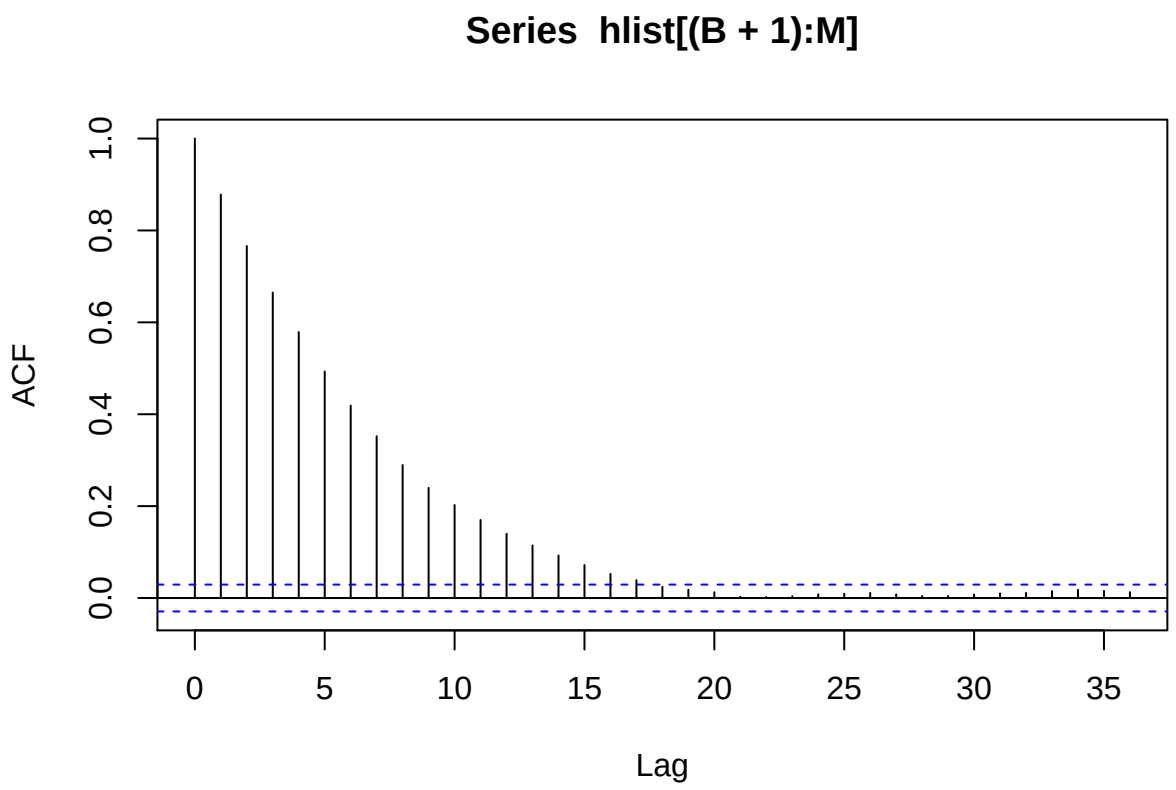
```
M = 5000  # run length
B = 500  # amount of burn-in
X = runif(1)  # overdispersed starting distribution
sigma = 10  # proposal scaling
xlist = rep(0,M)  # for keeping track of chain values
hlist = rep(0,M)  # for keeping track of h function values
numaccept = 0;
for (i in 1:M) {
    Y = X + sigma * 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);
}

cat("ran Metropolis algorithm for", M, "iterations, with burn-in", B, "\n");
```

```
## ran Metropolis algorithm for 5000 iterations, with burn-in 500
cat("acceptance rate =", numaccept/M, "\n");
```

```
## acceptance rate = 0.1184
u = mean(hlist[(B+1):M])
cat("mean of h is about", u, "\n")
```

```
## mean of h is about 0.8904353
se1 =  sd(hlist[(B+1):M]) / sqrt(M-B)
cat("iid standard error would be about", se1, "\n")
```

```
## iid standard error would be about 0.01369853
acf(hlist[(B+1):M])
```



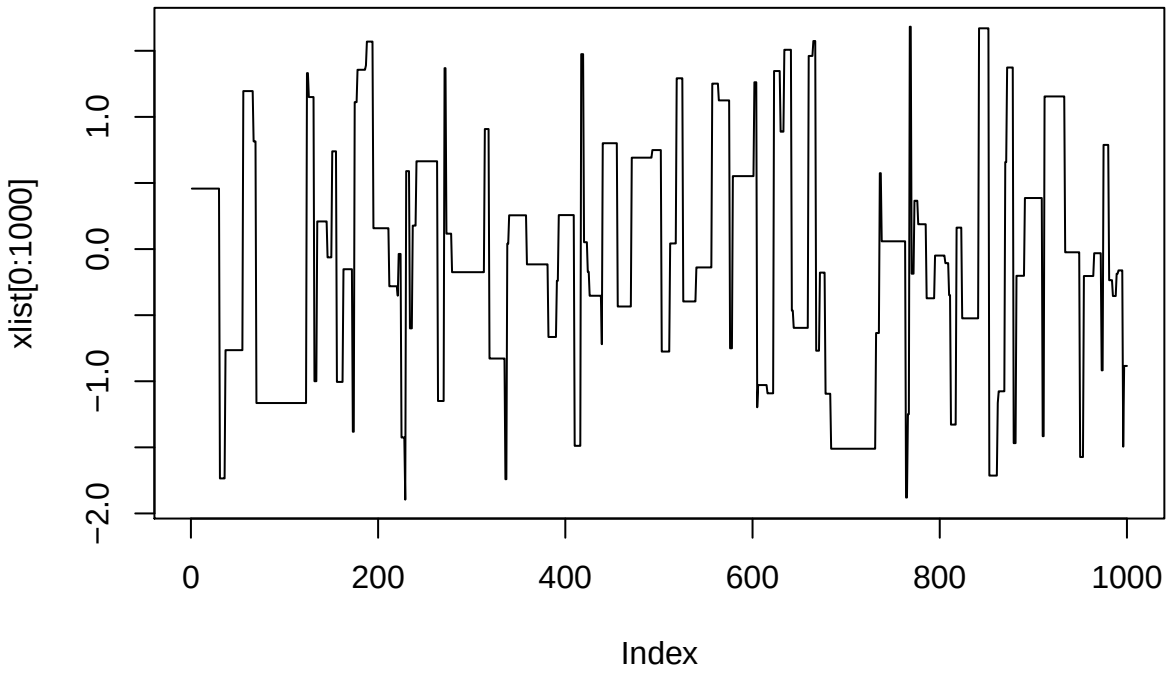

```
varfact <- function(xxx) { 2 * sum(acf(xxx, plot=FALSE, lag.max = 36)$acf) - 1 }
thevarfact = varfact(hlist[(B+1):M])
se = se1 * sqrt( thevarfact )
cat("varfact = ", thevarfact, "\n")
```

```
## varfact =  12.52651
cat("true standard error is about", se, "\n")
```

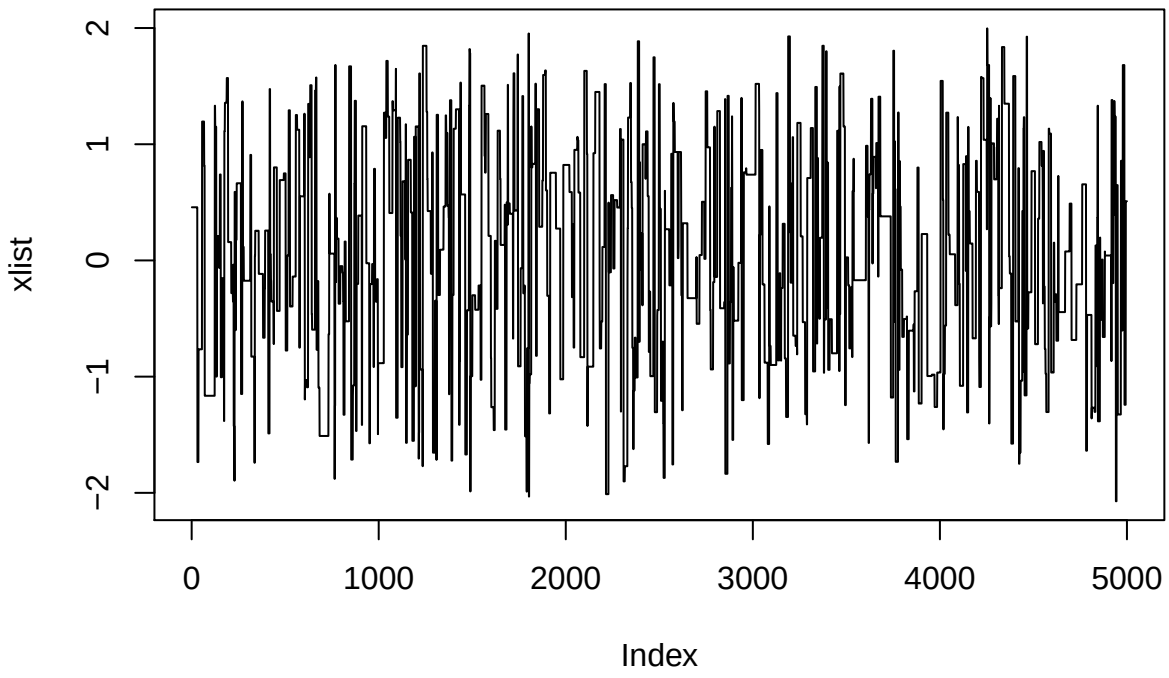
```
## true standard error is about 0.04848296
cat("approximate 95% confidence interval is (", u - 1.96 * se, ",",
                        u + 1.96 * se, ")\n\n")
```

```
## approximate 95% confidence interval is ( 0.7954087 , 0.9854619 )
```

```
plot(xlist[0:1000], type='l')
```




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




From above output, we see that using $\sigma^2 = 10$, we have a okay mixing for the x values after first 500 values, So I think using burning-in of 500 is good. However, the acceptance rate is quite low(11%) and we see that sometimes x is stuck in a value for a while, which is not good. Since there is no significant correlation between values of lag 36, I used lag of 36 to calculate the varfact. The confident interval is wider than using $\sigma^2 = 1$ but narrower than using $\sigma^2 = 0.1$.

Therefore, based on the plots and the confident interval, it seems like using $\sigma^2 = 1$ is the best.

# Q2

```
A = 4
B = 2
C = 6
D = 1

g <- function(x){
  if ( (x[1]<0) || (x[1]>1) || (x[2]<0) || (x[2]>1) || (x[3]<0) ||
       (x[3]>1) || (x[4]<0) || (x[4]>1) || (x[5]<0) || (x[5]>1))
      return(0)
  else
    x[1]^{A+6}*2^{x[2]+3} * (1 + cos(x[1] + 2*x[2] + 3*x[3] + 4*x[4] + (B + 3)*x[5]))*
    exp((C-12)*x[4]^2)*exp(-(D+2)*(x[4]-3*x[5])^2)
}

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

```
}

M = 11000  # run length
B = 2000  # amount of burn-in
X  = c(runif(1), runif(1) , runif(1), runif(1), runif(1)) # overdispersed starting distribution
sigma = 0.1  # proposal scaling
x1list = rep(0,M)  # for keeping track of chain values
x2list = rep(0,M)
x3list = rep(0,M)
x4list = rep(0,M)
x5list = rep(0,M)

hlist = rep(0,M)  # for keeping track of h function values
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);
}

cat("ran Metropolis algorithm for", M, "iterations, with burn-in", B, "\n");
```

```
## ran Metropolis algorithm for 11000 iterations, with burn-in 2000
```

```
cat("acceptance rate =", numaccept/M, "\n");
```

```
## acceptance rate = 0.2013636
```

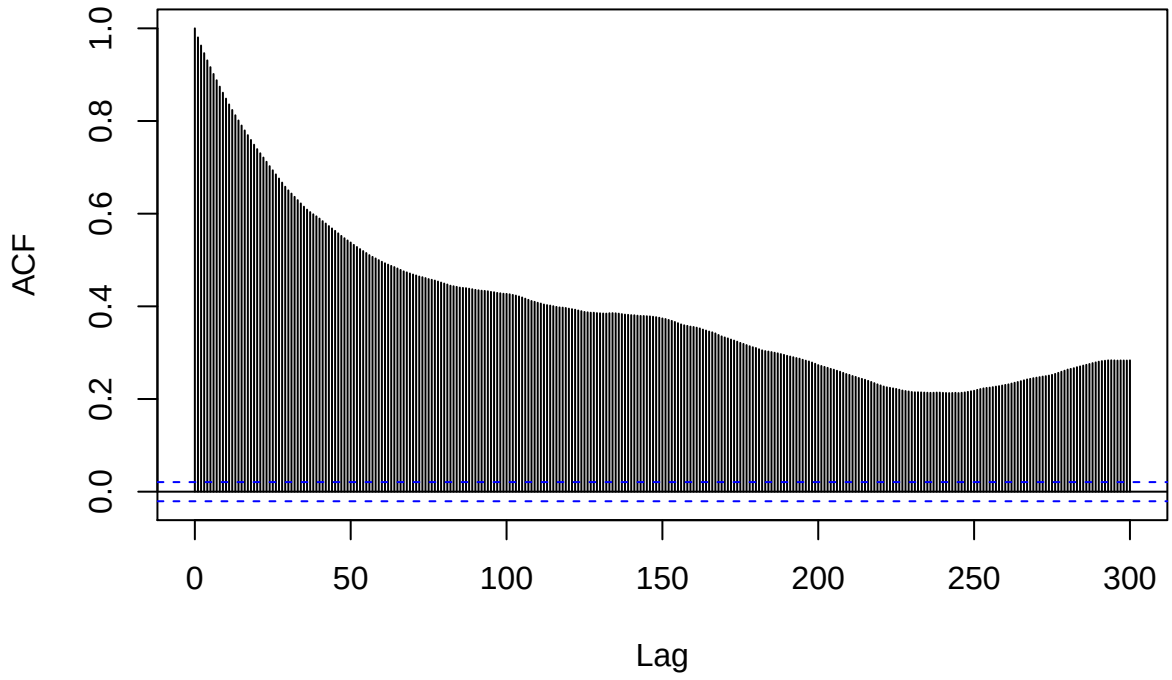
```
u = mean(hlist[(B+1):M])
cat("mean of h is about", u, "\n")
```

```
## mean of h is about 0.5711264
```

```
se1 =  sd(hlist[(B+1):M]) / sqrt(M-B)
cat("iid standard error would be about", se1, "\n")
```

```
## iid standard error would be about 0.001507882
```

```
acf(hlist[(B+1):M], lag.max = 300)
```




```
varfact <- function(xxx) { 2 * sum(acf(xxx, plot=FALSE, lag.max = 300)$acf) - 1 }
thevarfact = varfact(hlist[(B+1):M])
se = se1 * sqrt( thevarfact )
cat("varfact = ", thevarfact, "\n")
```

```
## varfact =  238.7681
```

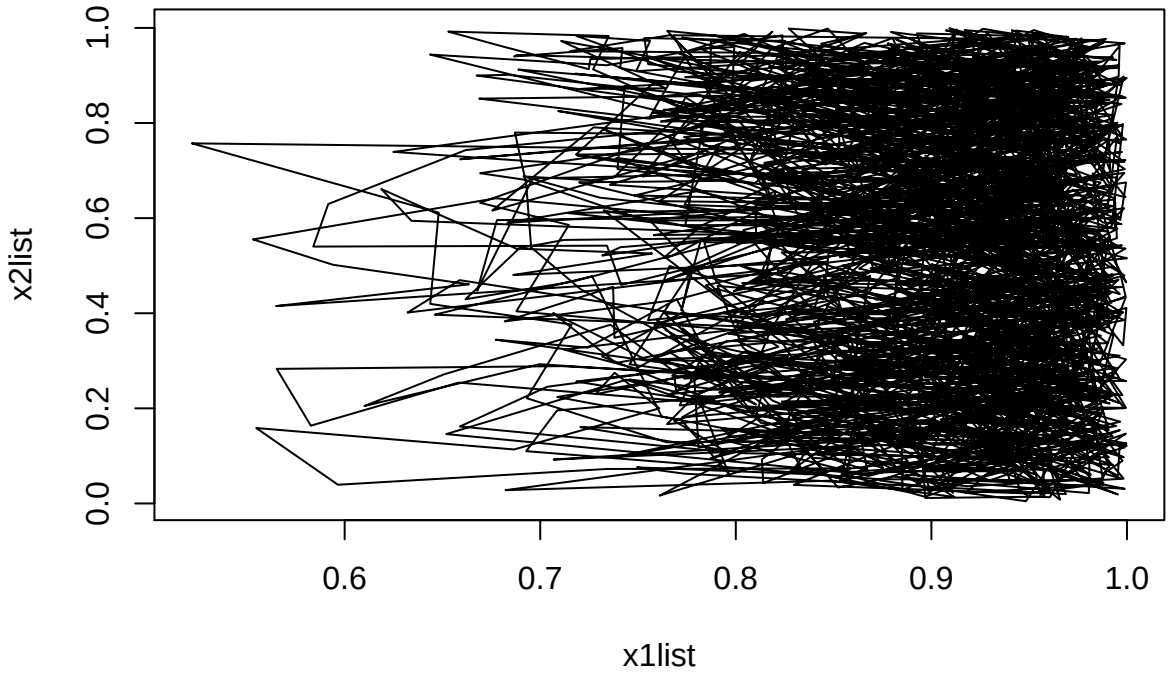
```
cat("true standard error is about", se, "\n")
```

```
## true standard error is about 0.02329998
```

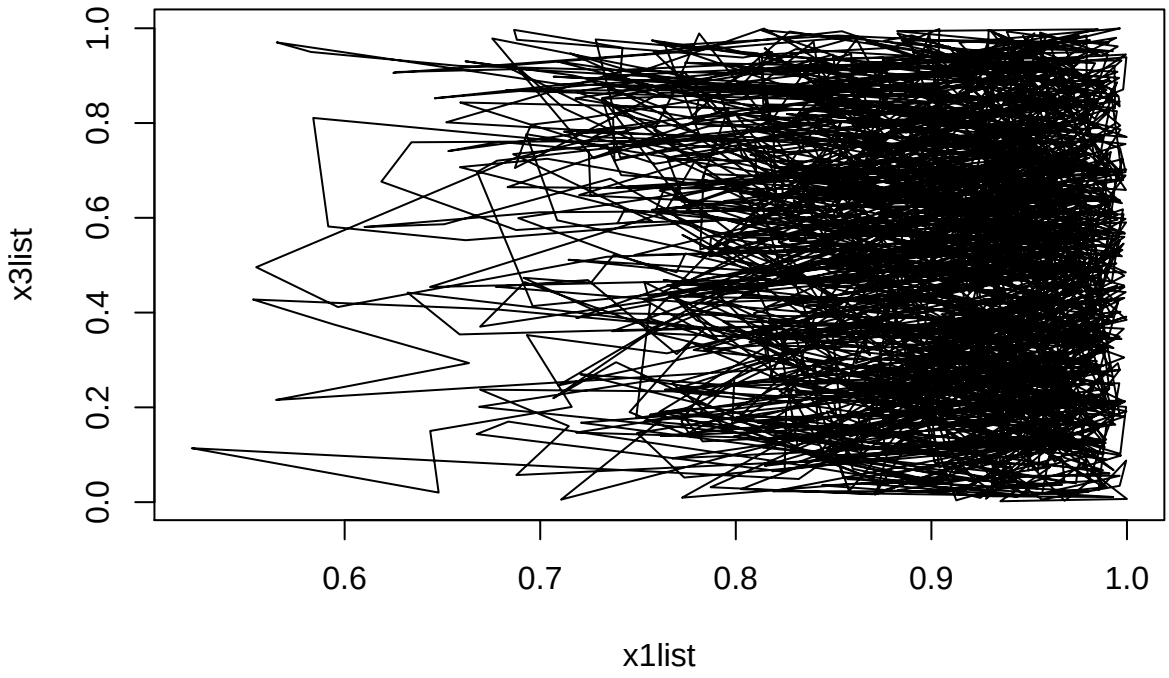
```
cat("approximate 95% confidence interval is (", u - 1.96 * se, ",",
                                                u + 1.96 * se, ")\n\n")
```

```
## approximate 95% confidence interval is ( 0.5254585 , 0.6167944 )
```

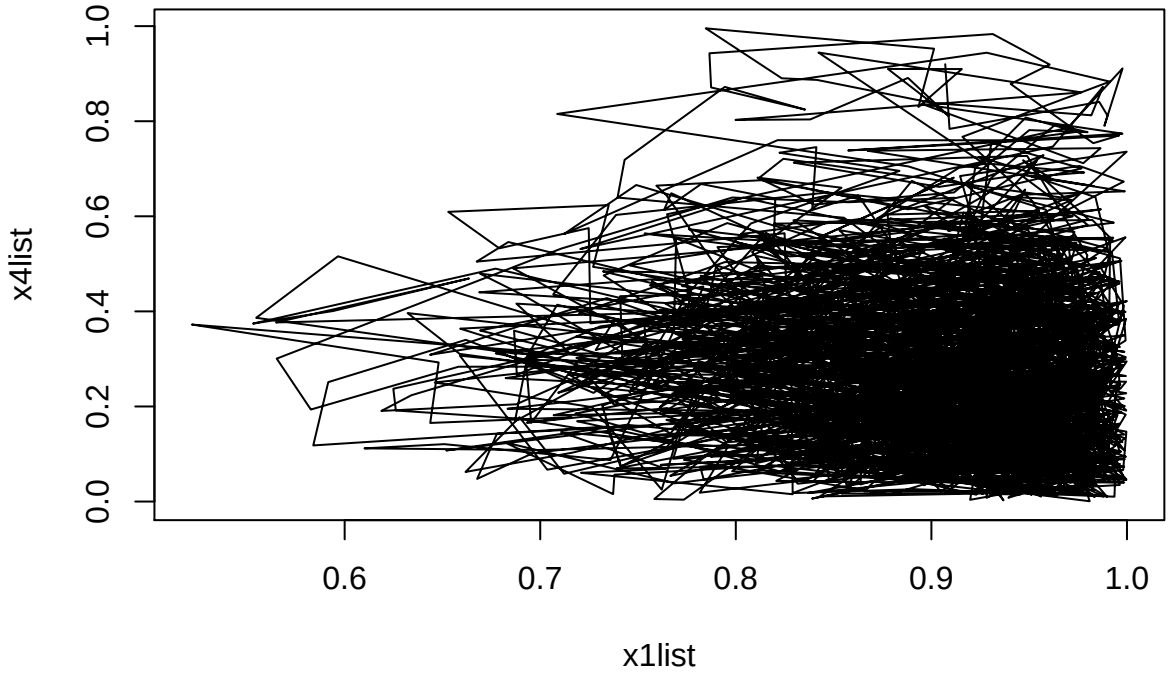
```
plot(x1list, x2list, type='l')
```



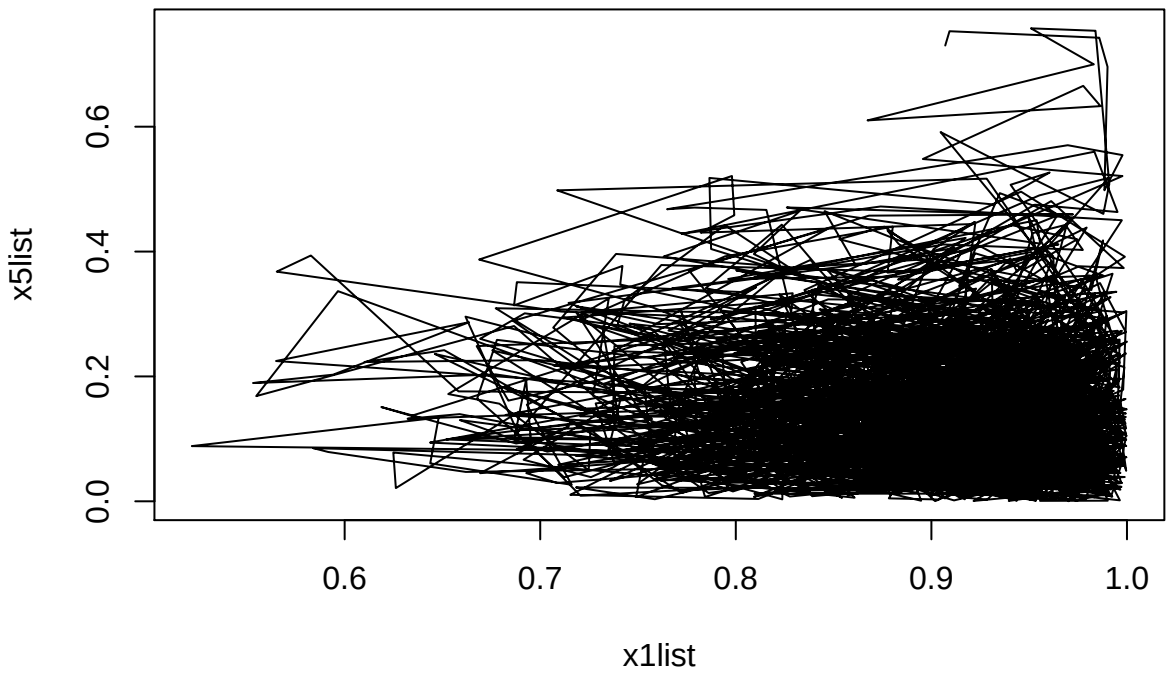

```
plot(x1list, x3list, type='l')
```



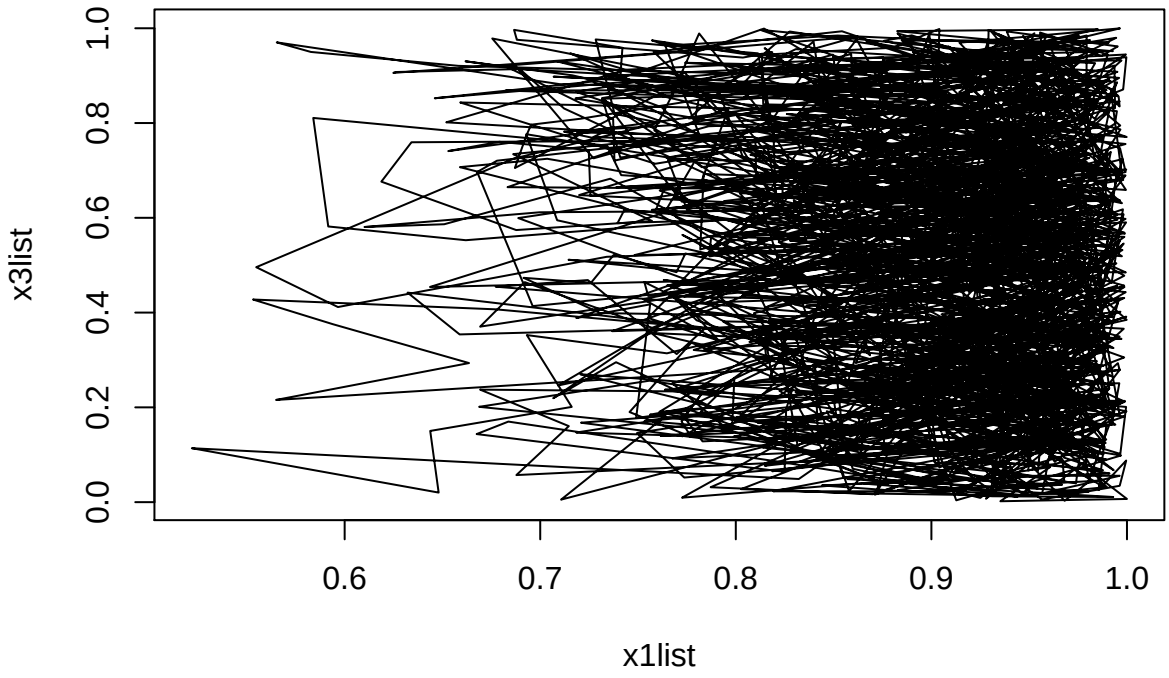

```
plot(x1list, x4list, type='l')
```



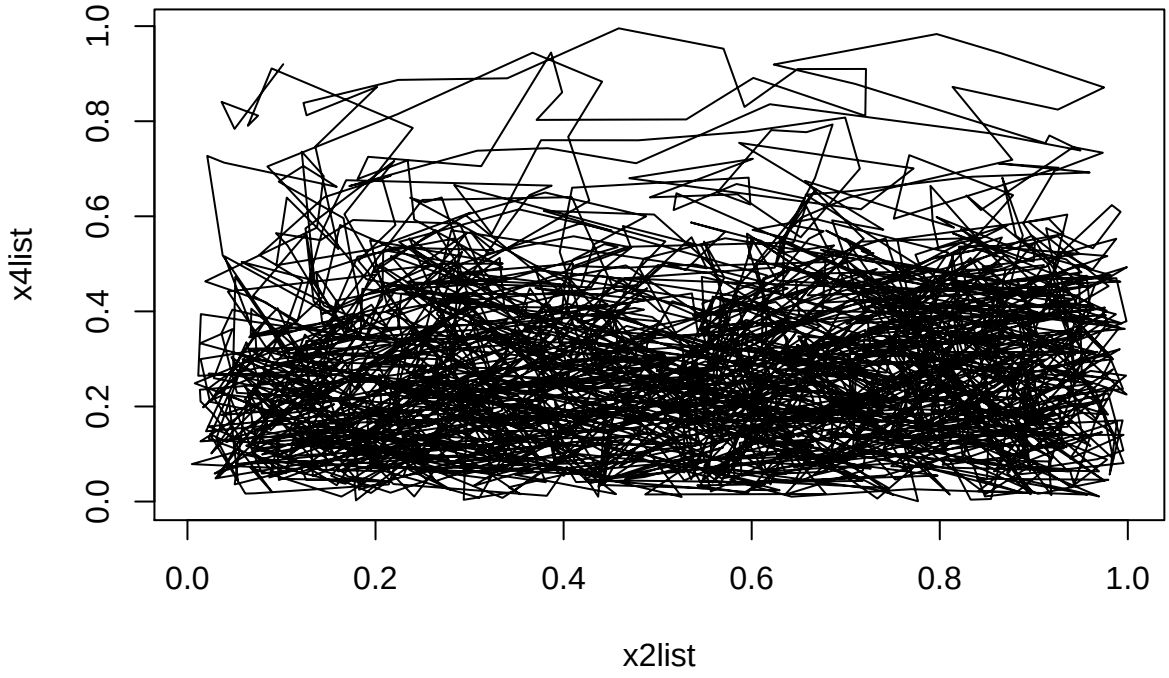

```
plot(x1list, x5list, type='l')
```



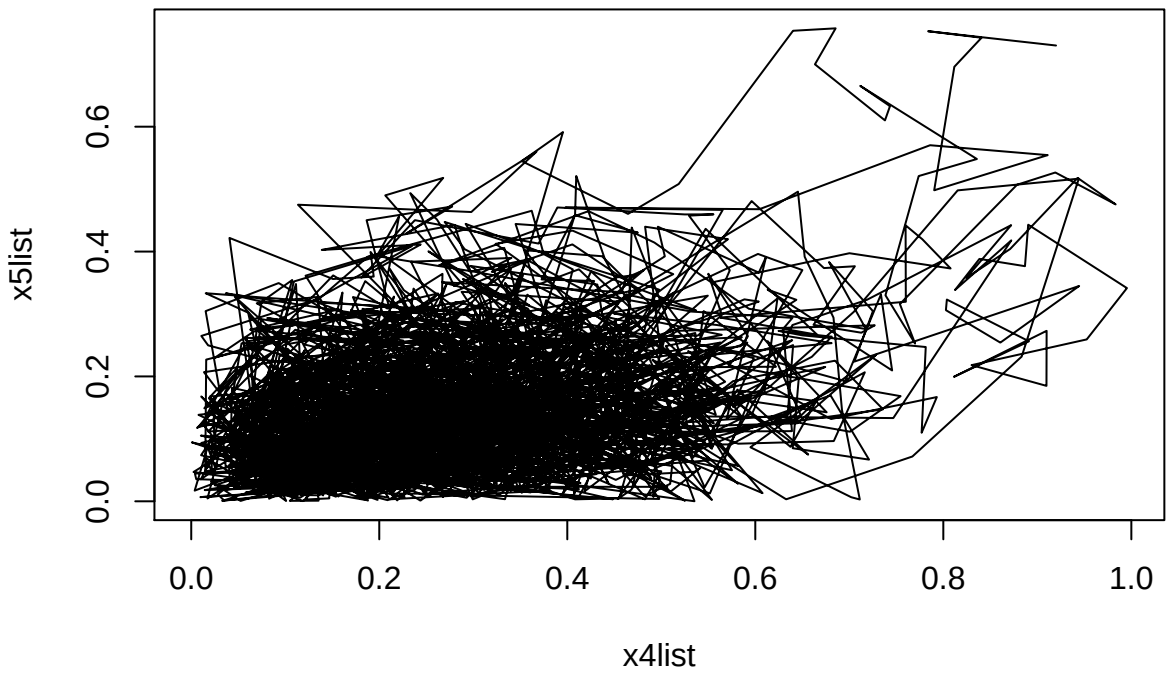

```
plot(x1list, x3list, type='l')
```



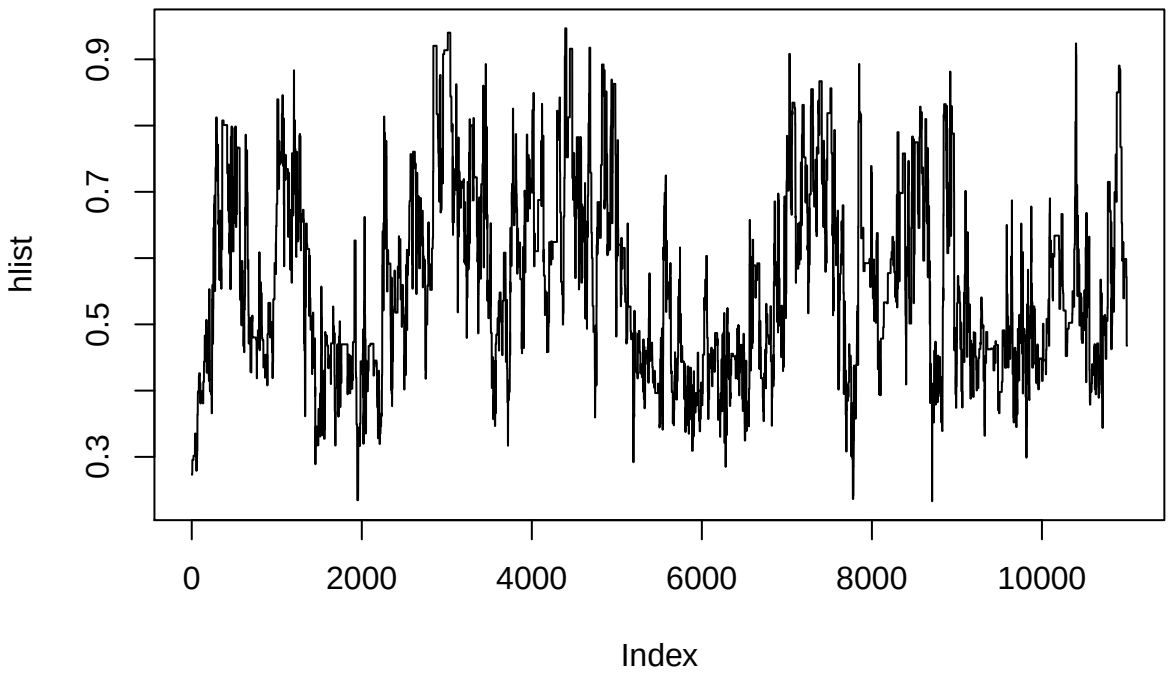

```
plot(x2list, x4list, type='l')
```




```
plot(x4list, x5list, type='l')
```




```
# plot(x1list, type='l')
# plot(x2list, type='l')
# plot(x1list, x2list, type='p')
plot(hlist, type = "l")
```




From above output, we see that we have a good mixing for the x value and our estimate values after first 2000 values, So I think using burning-in of 2000 is good. The acceptance rate is around 20%, which is close to optimal acceptance rate(23.4%)[Roberts et al., Ann Appl Prob 1997; Roberts and Rosenthal, Stat Sci 2001]. The "relevant" number of lag varies quite a lot for each time I ran the program. I see that the there is no significant correlation between 150 lag to 600 lag in different run. So I am chose lag of 300 to calculate varfact. The confident interval is around (0.56, 0.63), it is not vary wide, so it is okay. The estimate is pretty close to what I got for HW1, so I think the estimate is good.

Comparing to HW1, I think there are several advantages. Firstly there is no need to choose a function and k value, and show that it is larger than our objective function. Secondly, we are able to calculate standard error in a single run(no need to run multiple times).

There are also disadvantages. Using metropolis algorithm, we have to choose the value of $signma^2$ and burning-in and "relevant" lag for standard error , which can vary a lot every time I run the code, which leads to uncertainty in the final estimate and the confidence interval.

Trying different sigma:

```
M = 11000  # run length
B = 2000  # amount of burn-in
X  = c(runif(1), runif(1) , runif(1), runif(1), runif(1)) # overdispersed starting distribution
sigma = 0.01  # proposal scaling
x1list = rep(0,M)  # for keeping track of chain values
x2list = rep(0,M)
x3list = rep(0,M)
x4list = rep(0,M)
x5list = rep(0,M)
```

```
hlist = rep(0,M)  # for keeping track of h function values
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);
}

cat("ran Metropolis algorithm for", M, "iterations, with burn-in", B, "\n");
```

```
## ran Metropolis algorithm for 11000 iterations, with burn-in 2000
```

```
cat("acceptance rate =", numaccept/M, "\n");
```

```
## acceptance rate = 0.5329091
```

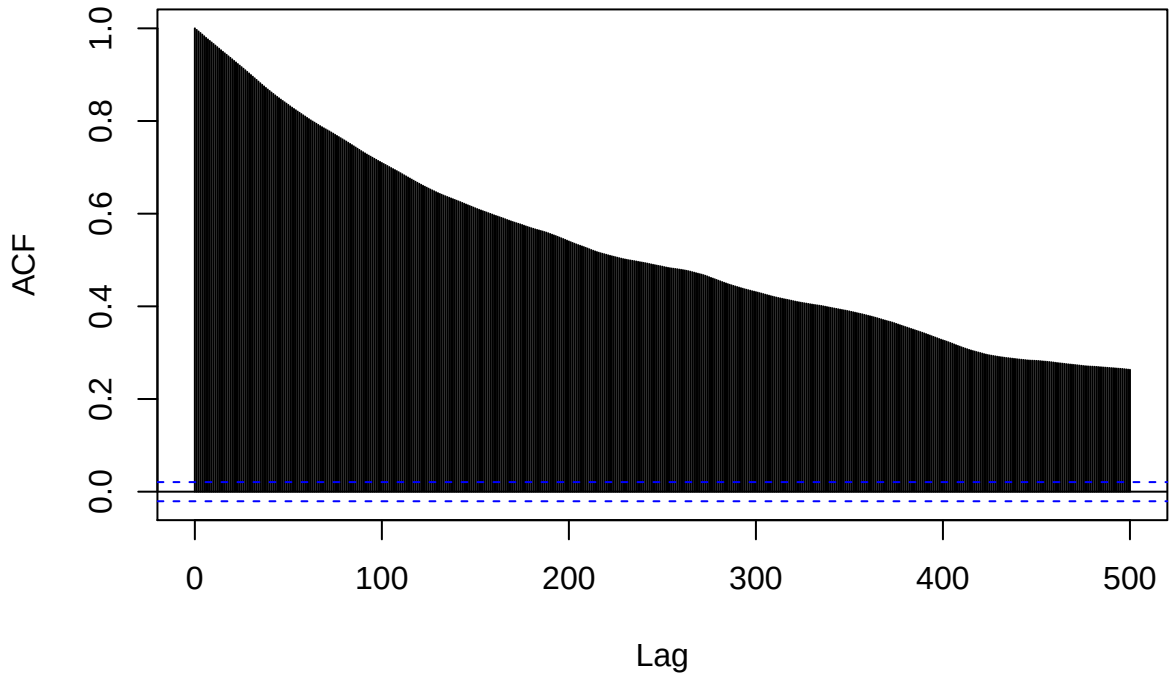
```
u = mean(hlist[(B+1):M])
cat("mean of h is about", u, "\n")
```

```
## mean of h is about 0.687166
```

```
se1 =  sd(hlist[(B+1):M]) / sqrt(M-B)
cat("iid standard error would be about", se1, "\n")
```

```
## iid standard error would be about 0.0008557721
```

```
acf(hlist[(B+1):M], lag.max = 500)
```




```
varfact <- function(xxx) { 2 * sum(acf(xxx, plot=FALSE, lag.max = 500)$acf) - 1 }
thevarfact = varfact(hlist[(B+1):M])
se = se1 * sqrt( thevarfact )
cat("varfact = ", thevarfact, "\n")
```

```
## varfact =  523.0388
```

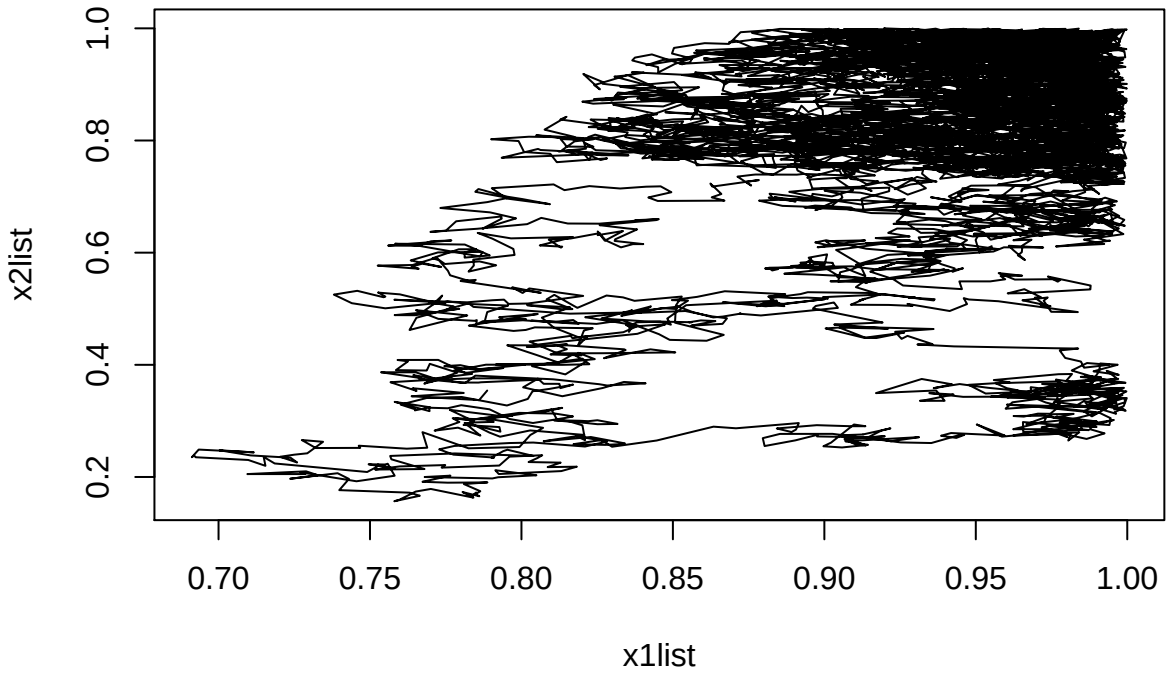
```
cat("true standard error is about", se, "\n")
```

```
## true standard error is about 0.01957154
```

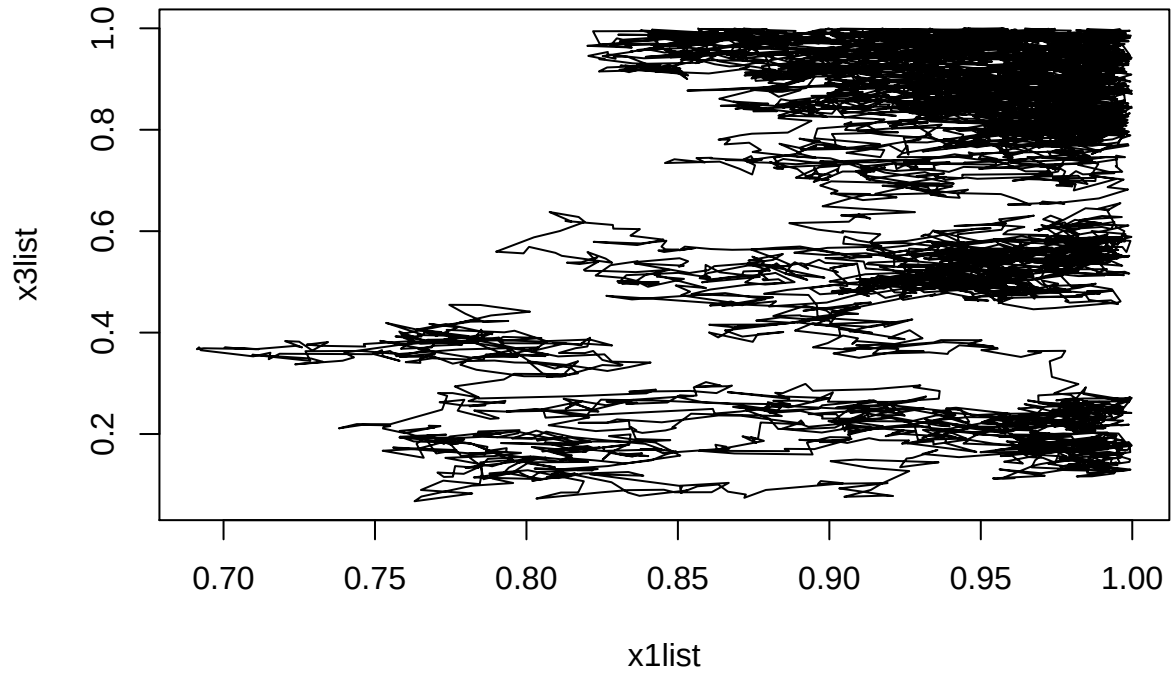
```
cat("approximate 95% confidence interval is (", u - 1.96 * se, ",",
                                                u + 1.96 * se, ")\n\n")
```

```
## approximate 95% confidence interval is ( 0.6488058 , 0.7255262 )
```

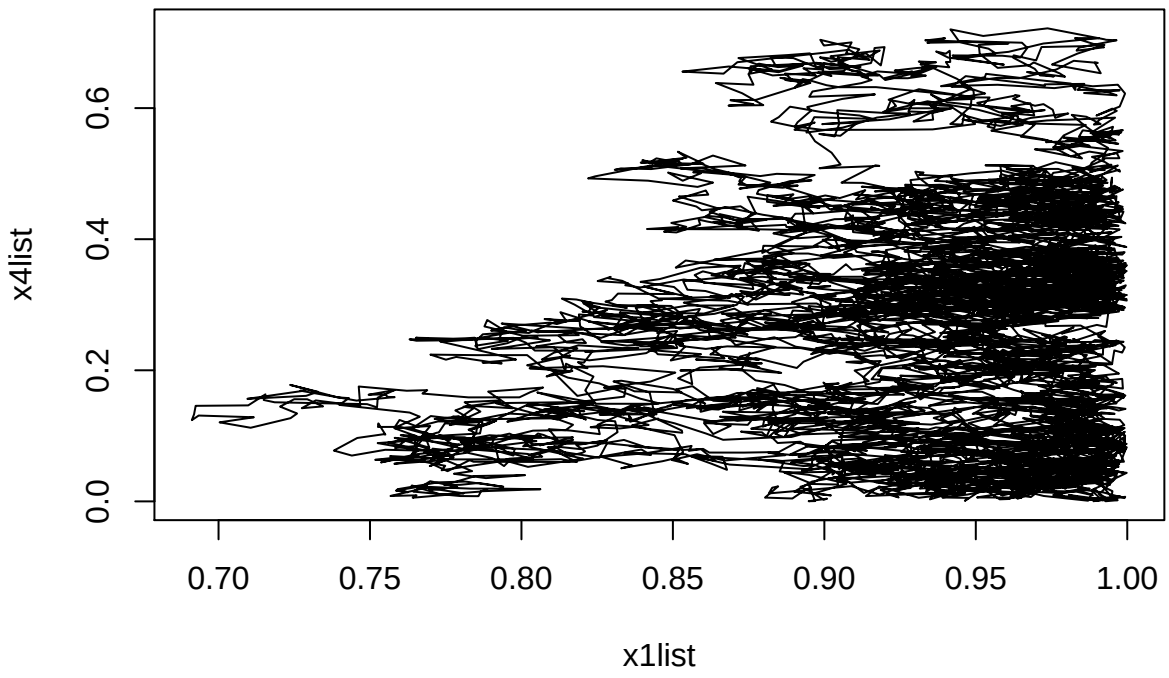
```
plot(x1list, x2list, type='l')
```



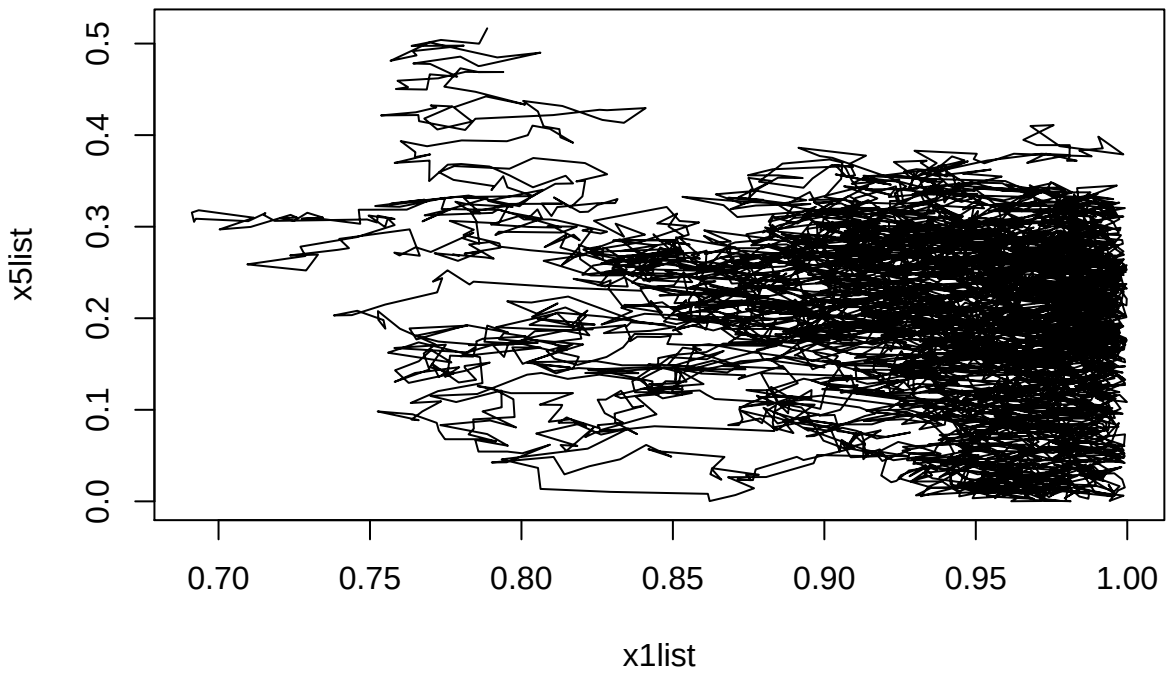

```
plot(x1list, x3list, type='l')
```



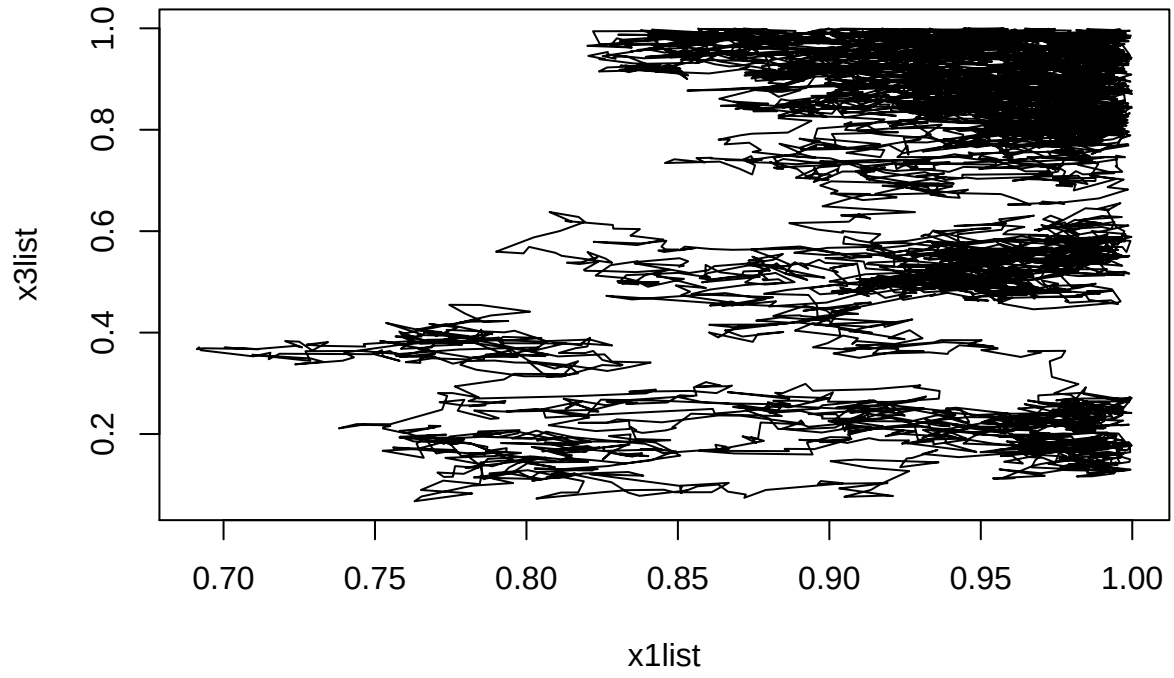

```
plot(x1list, x4list, type='l')
```



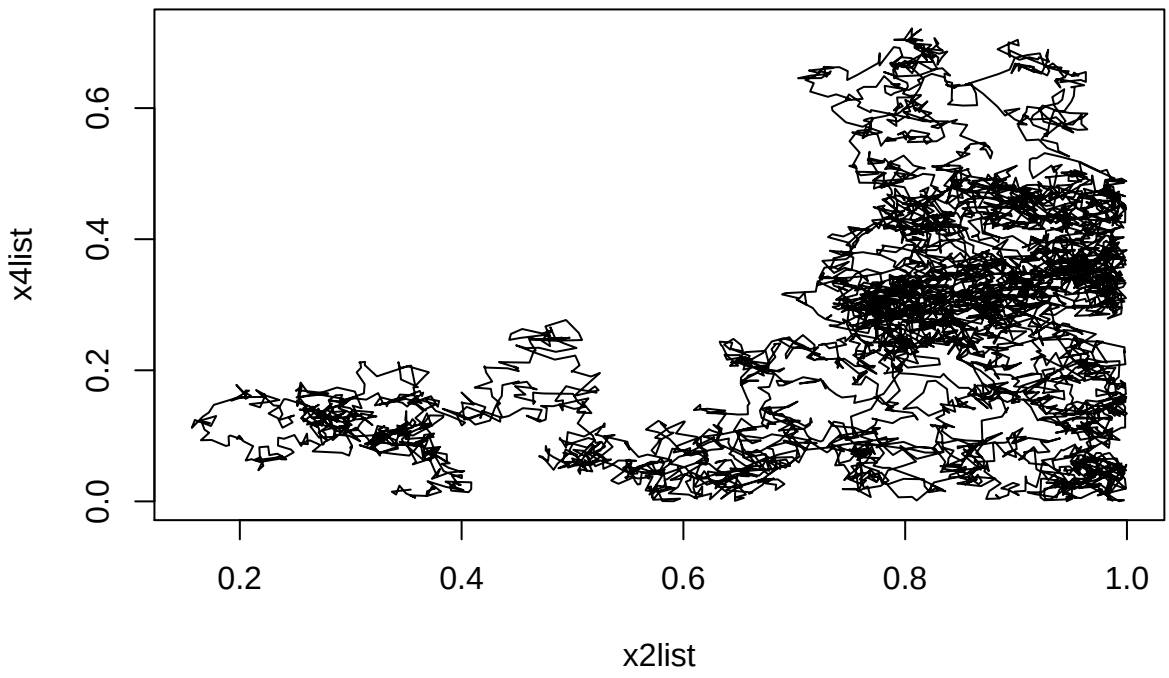

```
plot(x1list, x5list, type='l')
```



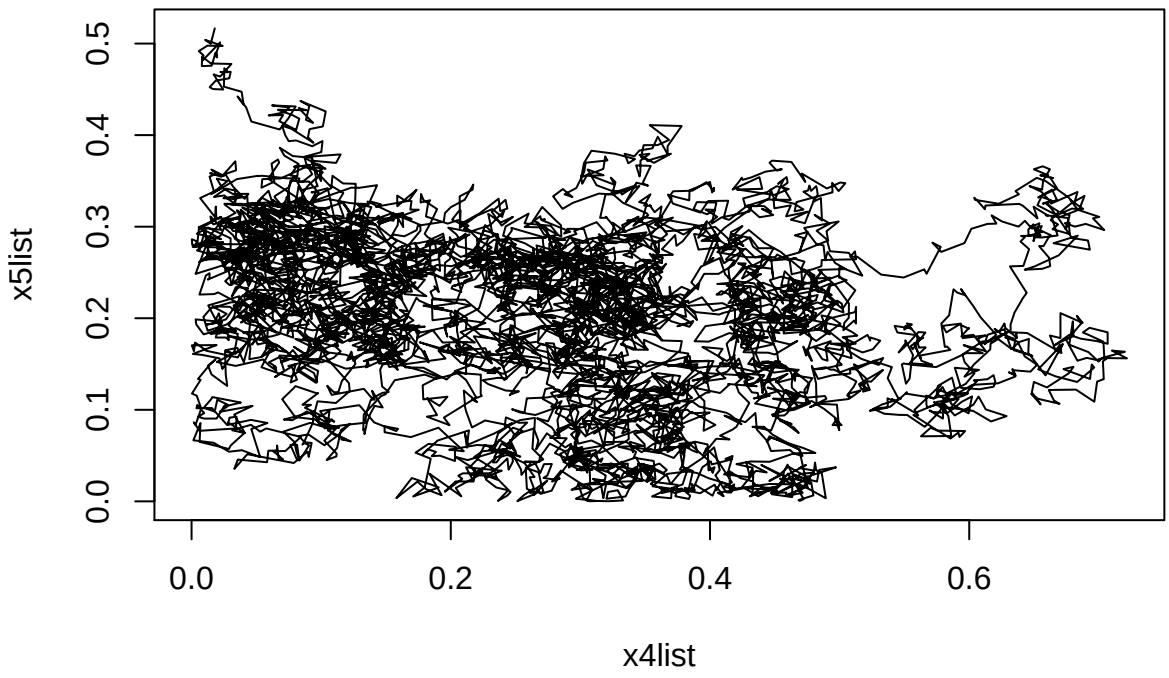

```
plot(x1list, x3list, type='l')
```



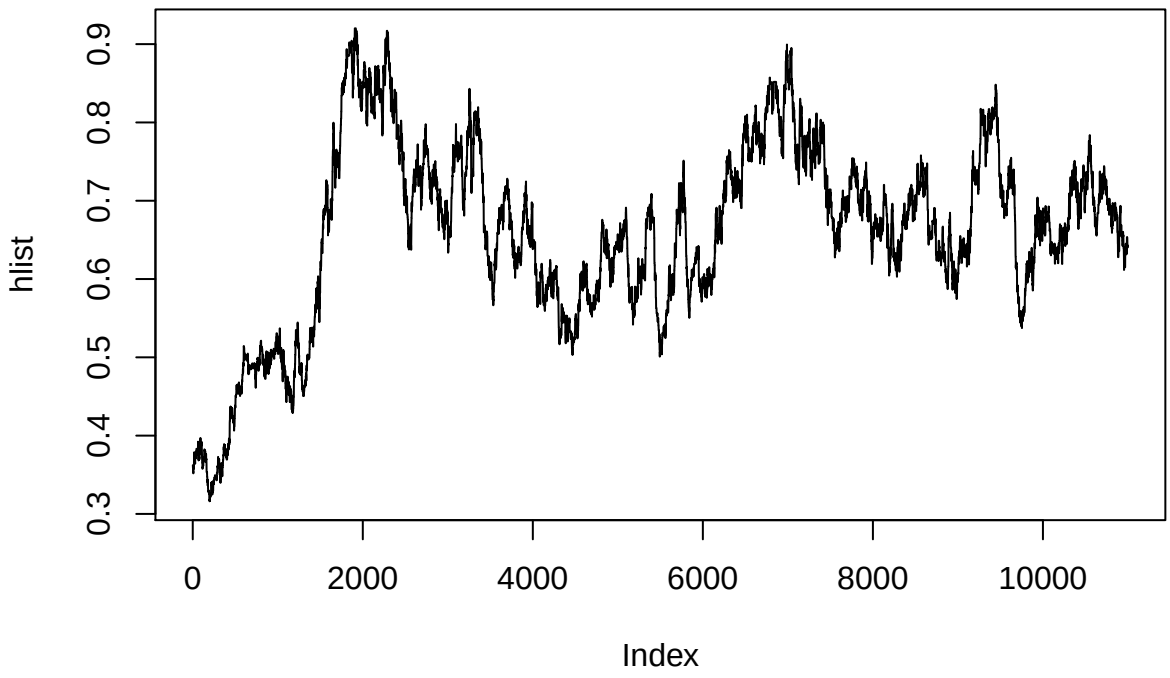

```
plot(x2list, x4list, type='l')
```




```
plot(x4list, x5list, type='l')
```




```
# plot(x1list, type='l')
# plot(x2list, type='l')
# plot(x1list, x2list, type='p')
plot(hlist, type = "l")
```




when $\sigma^2 = 0.01$, our acceptance rate may be too high for multidimensional metropolis algorithm, because we see very high correction between each iteration, and our h(x) and x does not have a good mix. Also the confidence interval is wider compared to using $\sigma^2 = 0.1$

```
M = 11000  # run length
B = 2000  # amount of burn-in
X  = c(runif(1), runif(1) , runif(1), runif(1), runif(1)) # overdispersed starting distribution
sigma = 1  # proposal scaling
x1list = rep(0,M)  # for keeping track of chain values
x2list = rep(0,M)
x3list = rep(0,M)
x4list = rep(0,M)
x5list = rep(0,M)

hlist = rep(0,M)  # for keeping track of h function values
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);
}

cat("ran Metropolis algorithm for", M, "iterations, with burn-in", B, "\n");
```

```
## ran Metropolis algorithm for 11000 iterations, with burn-in 2000
```

```
cat("acceptance rate =", numaccept/M, "\n");
```

```
## acceptance rate = 0.001
```

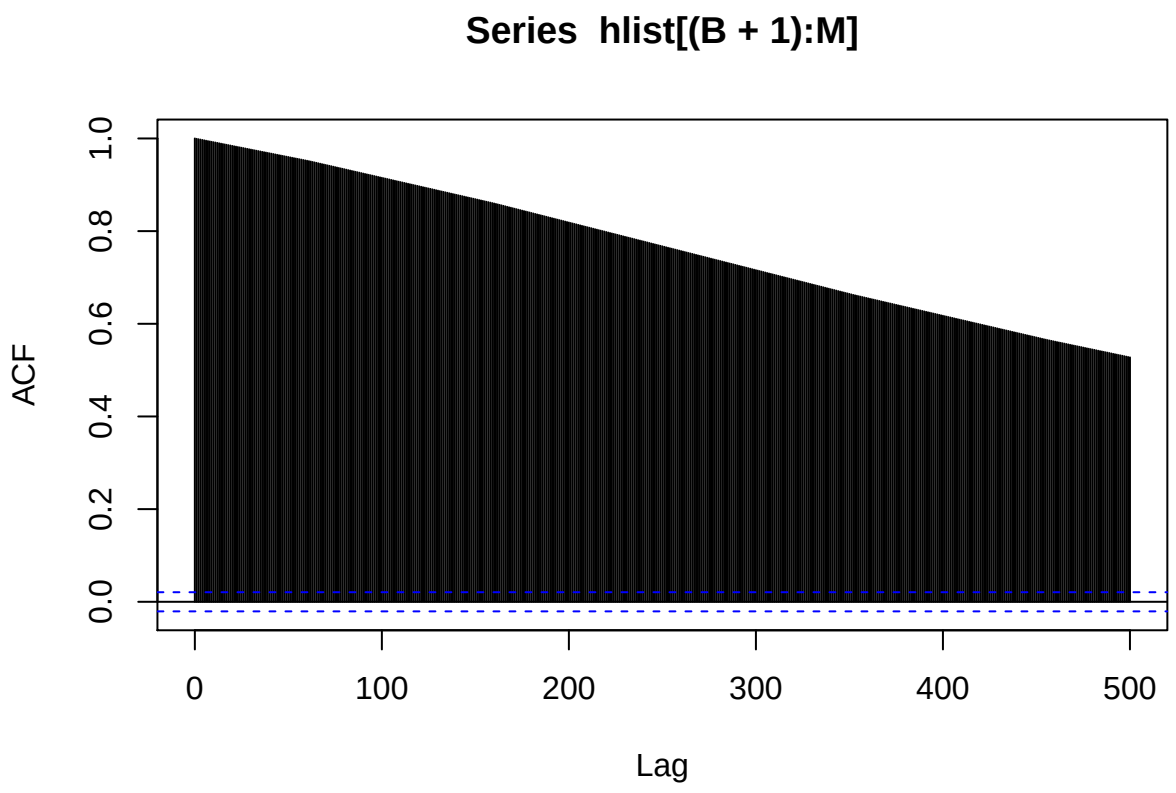
```
u = mean(hlist[(B+1):M])
cat("mean of h is about", u, "\n")
```

```
## mean of h is about 0.4896871
```

```
se1 =  sd(hlist[(B+1):M]) / sqrt(M-B)
cat("iid standard error would be about", se1, "\n")
```

```
## iid standard error would be about 0.001551317
```

```
acf(hlist[(B+1):M], lag.max = 500)
```



```
varfact <- function(xxx) { 2 * sum(acf(xxx, plot=FALSE, lag.max = 500)$acf) - 1 }
thevarfact = varfact(hlist[(B+1):M])
se = se1 * sqrt( thevarfact )
cat("varfact = ", thevarfact, "\n")
```

```
## varfact =  766.7561
```

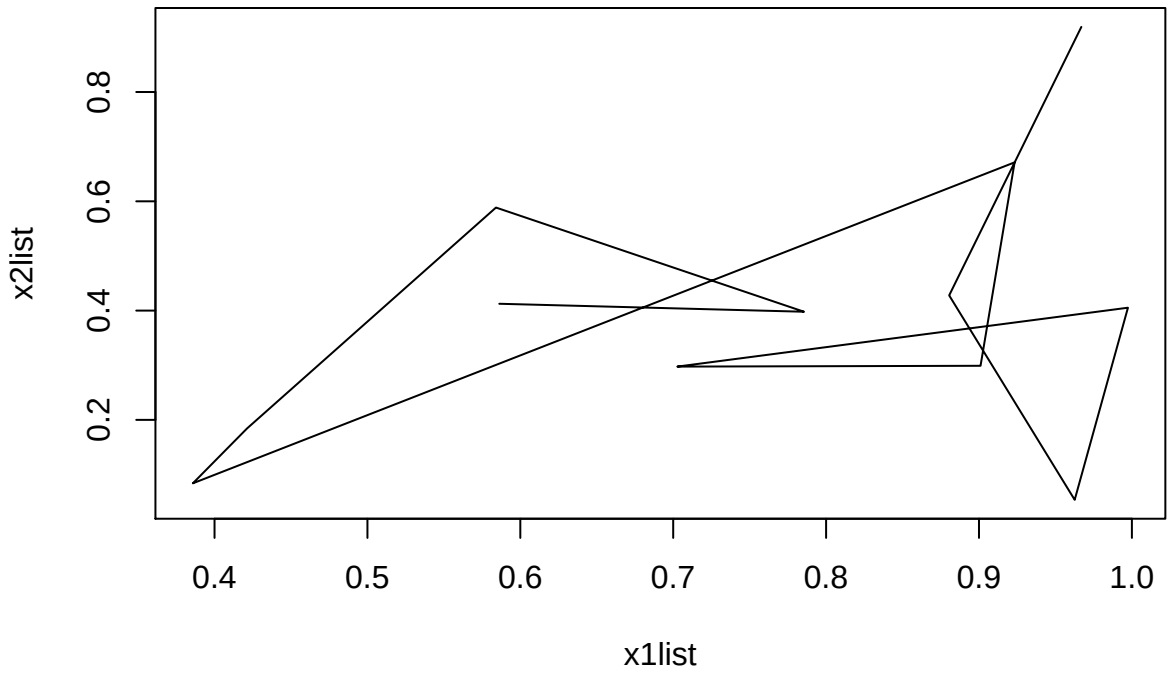
```
cat("true standard error is about", se, "\n")
```

```
## true standard error is about 0.04295653
```

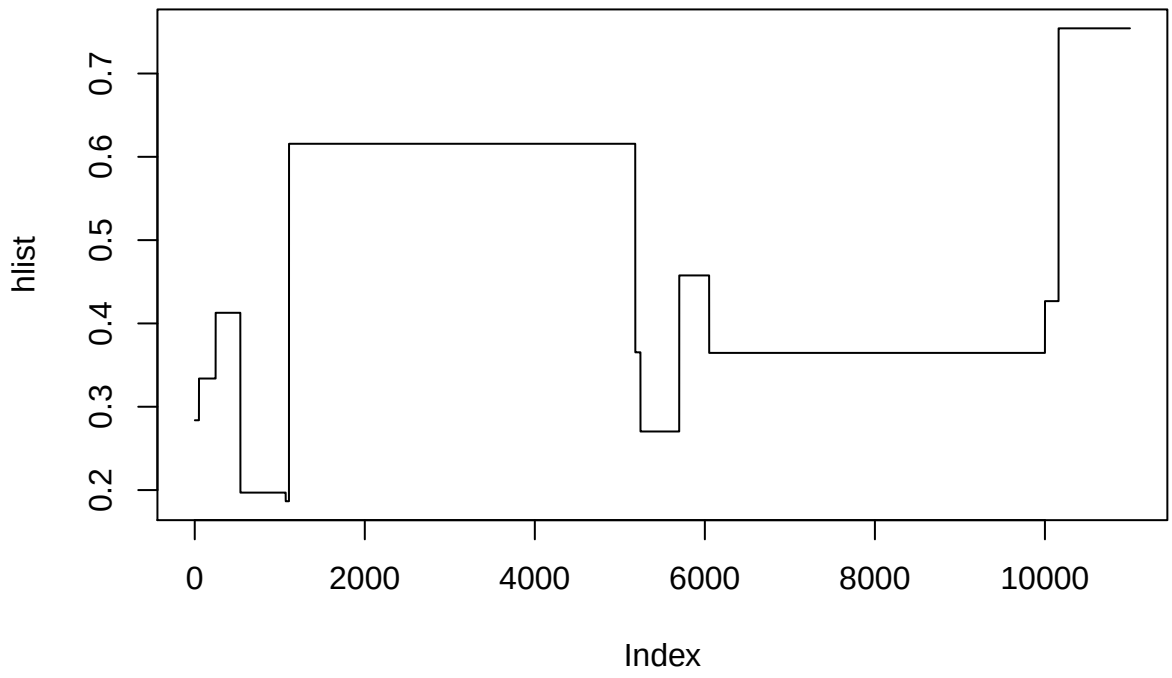
```
cat("approximate 95% confidence interval is (", u - 1.96 * se, ",",
                                                u + 1.96 * se, ")\n\n")
```

```
## approximate 95% confidence interval is ( 0.4054923 , 0.5738819 )
```

```
plot(x1list, x2list, type='l')
```




```
# plot(x1list, type='l')
# plot(x2list, type='l')
# plot(x1list, x2list, type='p')
plot(hlist, type = "l")
```




Using $\sigma^2 = 1$, acceptance rate is way too low. x and h(x) value stuck at one value for long time. This is not good.

Therefore, using $\sigma^2 = 0.1$ is the best.

#Q3

Let $\sigma^2 = 0.5$.

```
M = 11000  # run length
B = 1000  # amount of burn-in
X  = c(runif(1), runif(1) , runif(1), runif(1), runif(1)) # overdispersed starting distribution
sigma = 0.5  # proposal scaling
x1list = rep(0,5*M)  # for keeping track of chain values
x2list = rep(0,5*M)
x3list = rep(0,5*M)
x4list = rep(0,5*M)
x5list = rep(0,5*M)

hlist = rep(0,5*M)  # for keeping track of h function values
numaccept = 0;


for (i in 1:M) {
  for (coord in 1:5){
    Y = X
    Y[coord] = X[coord]+ 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;
      }

    x1list[5*i - 5+coord] = X[1];
    x2list[5*i - 5+coord] = X[2];
    x3list[5*i - 5+coord] = X[3];
    x4list[5*i - 5+coord] = X[4];
    x5list[5*i - 5+coord] = X[5];
    hlist[5*i - 5+coord] = h(X);

  }
}

cat("ran Metropolis algorithm for", M, "iterations, with burn-in", B, "\n");
```

```
## ran Metropolis algorithm for 11000 iterations, with burn-in 1000
```

```
cat("acceptance rate =", numaccept/(5*M), "\n");
```

```
## acceptance rate = 0.3118545
```

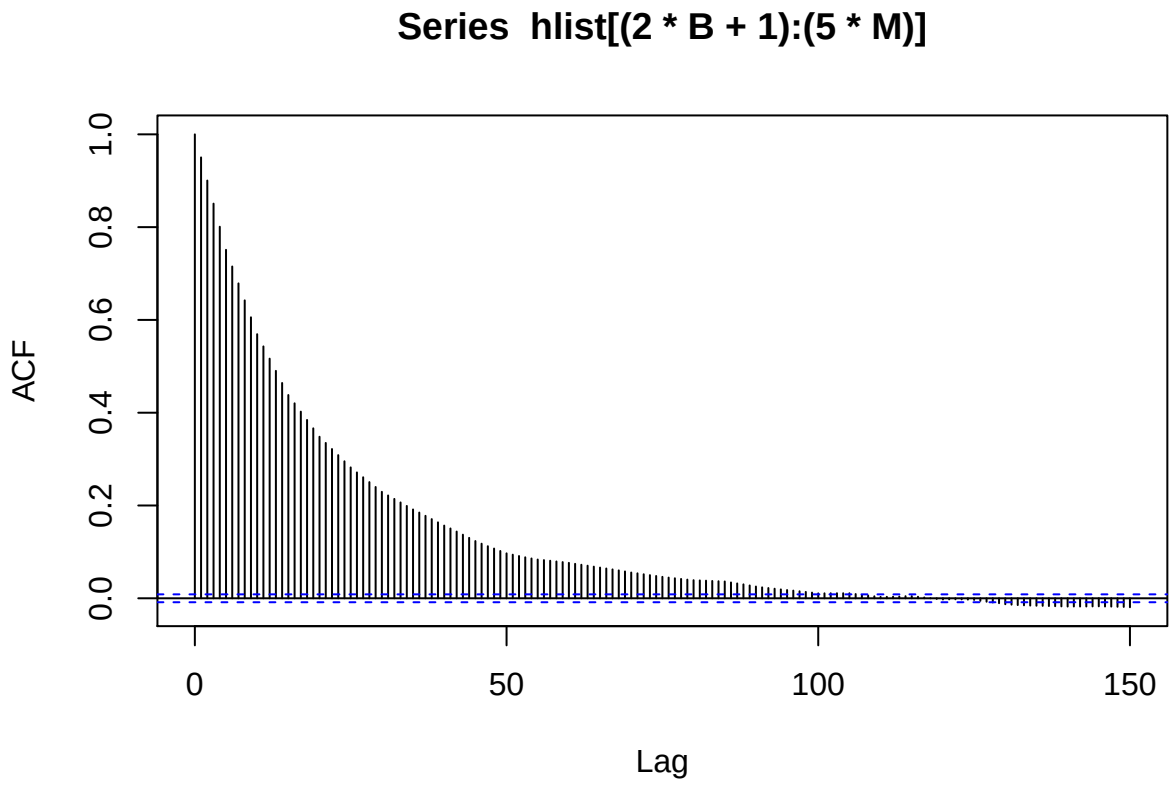
```
u = mean(hlist[(2*B+1):(5*M)])
cat("mean of h is about", u, "\n")
```

```
## mean of h is about 0.5808161
```

```
se1 =  sd(hlist[(2*B+1):(5*M)]) / sqrt(5*(M-B))
cat("iid standard error would be about", se1, "\n")
```

```
## iid standard error would be about 0.0006689432
```

```
acf(hlist[(2*B+1):(5*M)], lag.max = 150)
```




```
varfact <- function(xxx) { 2 * sum(acf(xxx, plot=FALSE, lag.max = 150)$acf) - 1 }
thevarfact = varfact(hlist[(2*B+1):(5*M)])
se = se1 * sqrt( thevarfact )
cat("varfact = ", thevarfact, "\n")
```

```
## varfact =  40.80999
```

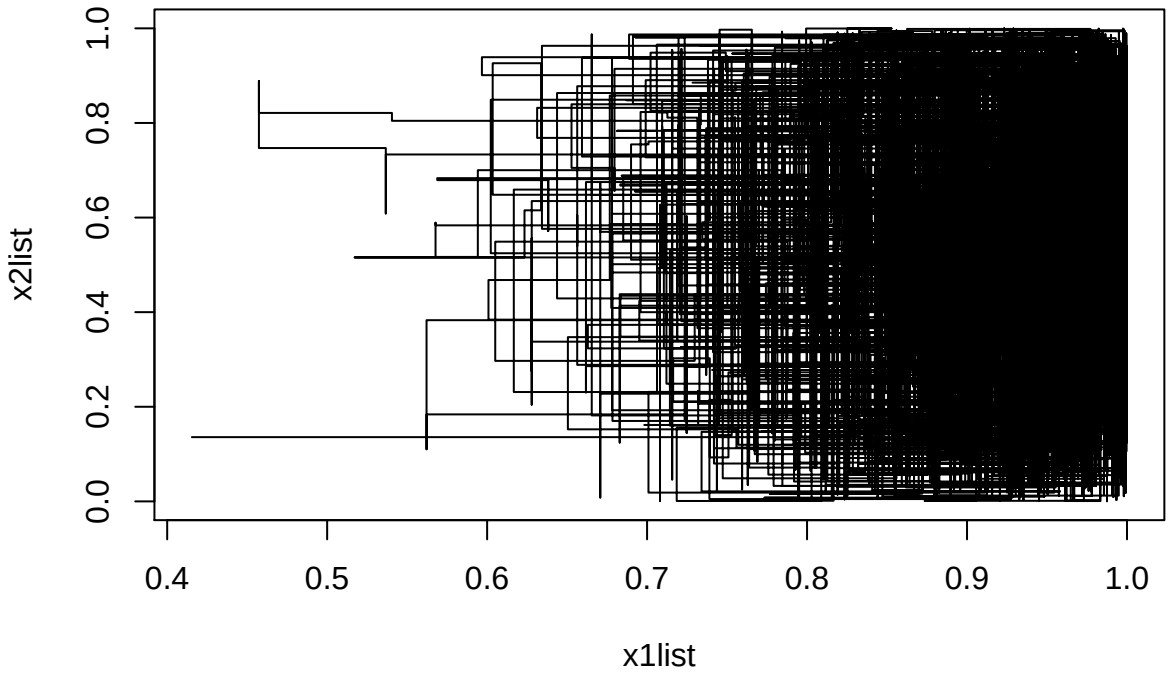
```
cat("true standard error is about", se, "\n")
```

```
## true standard error is about 0.00427339
```

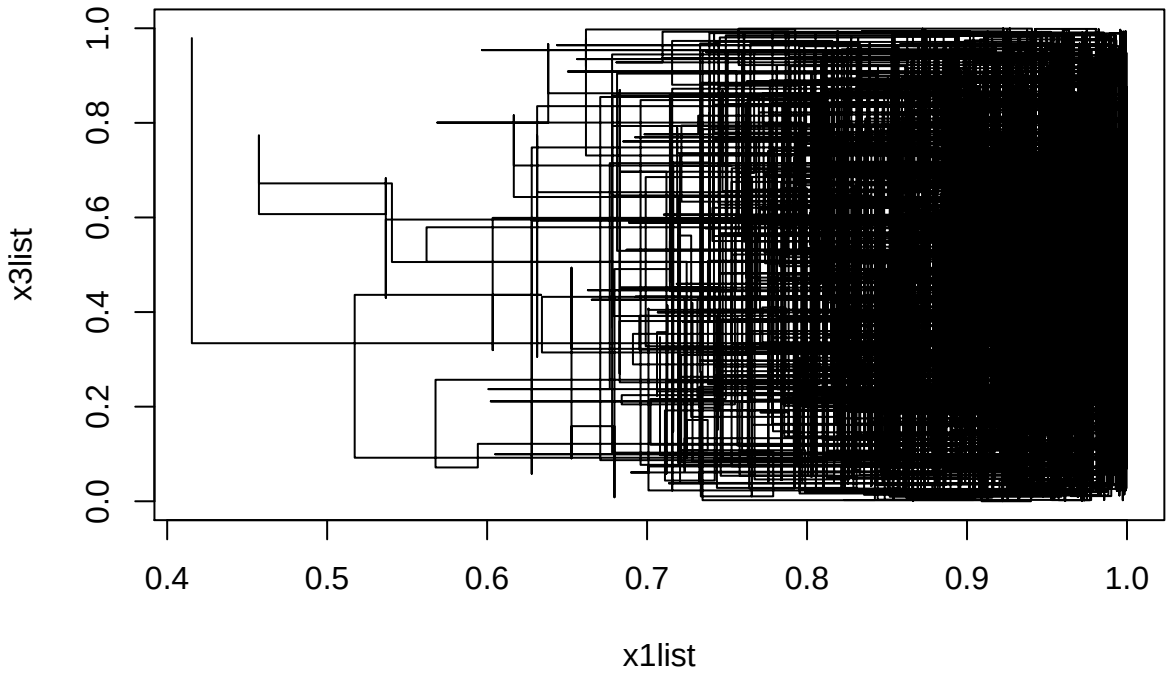
```
cat("approximate 95% confidence interval is (", u - 1.96 * se, ",",
                                                u + 1.96 * se, ")\n\n")
```

```
## approximate 95% confidence interval is ( 0.5724403 , 0.589192 )
```

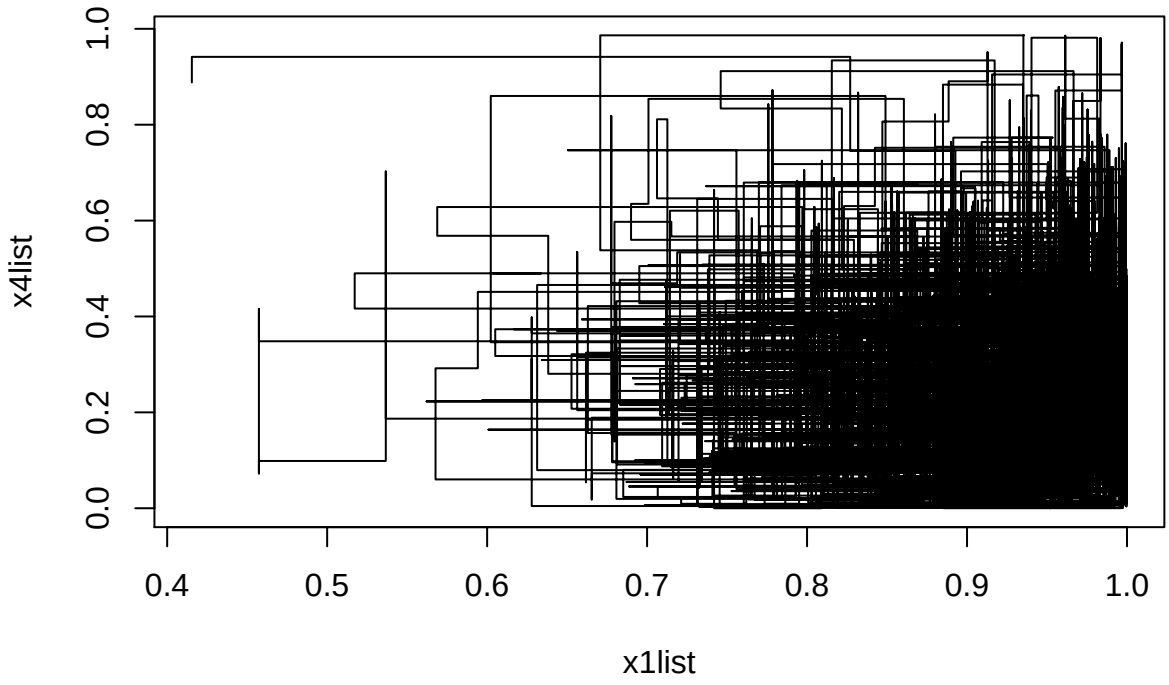
```
plot(x1list, x2list, type='l')
```



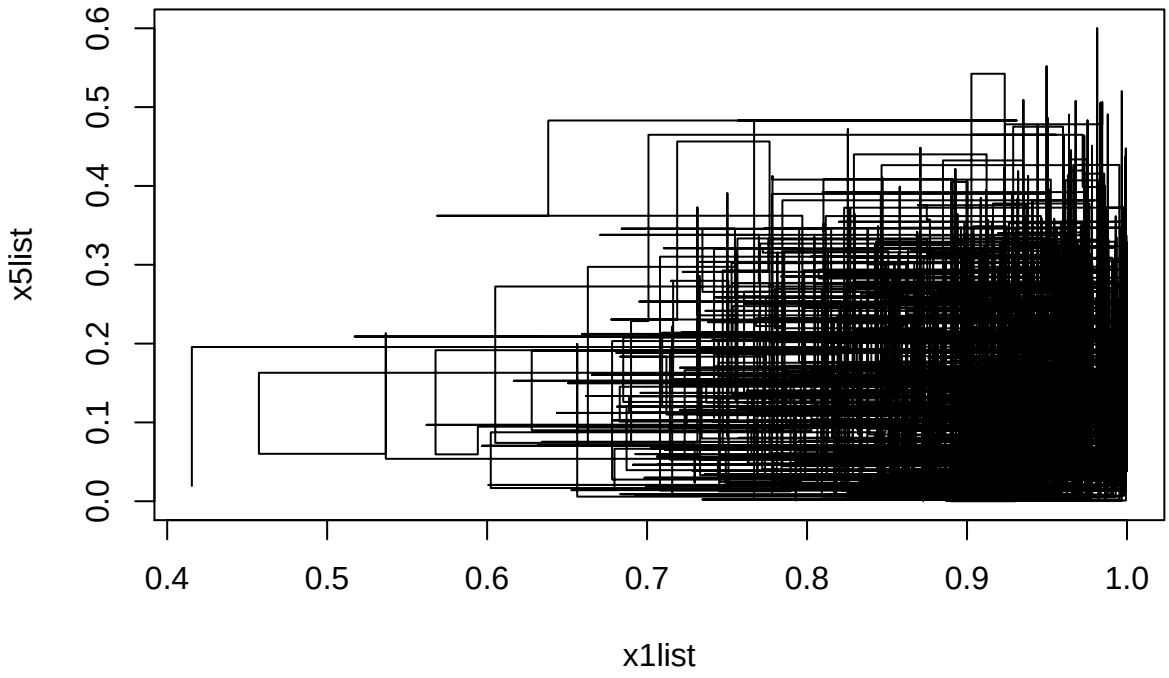

```
plot(x1list, x3list, type='l')
```



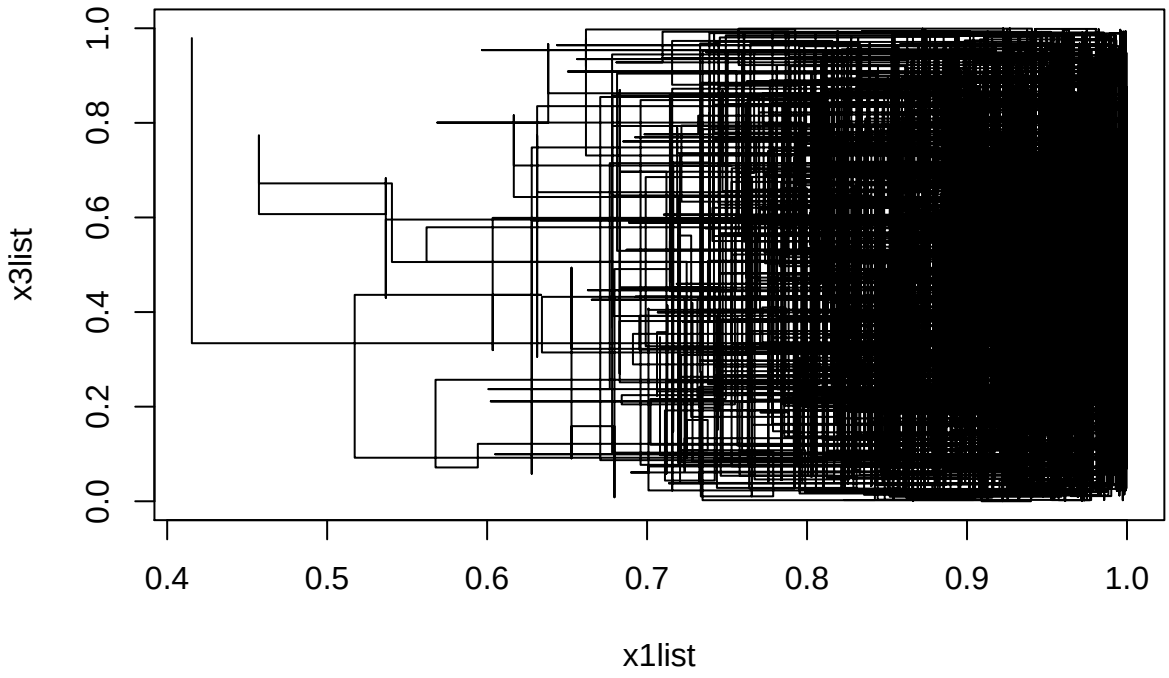

```
plot(x1list, x4list, type='l')
```



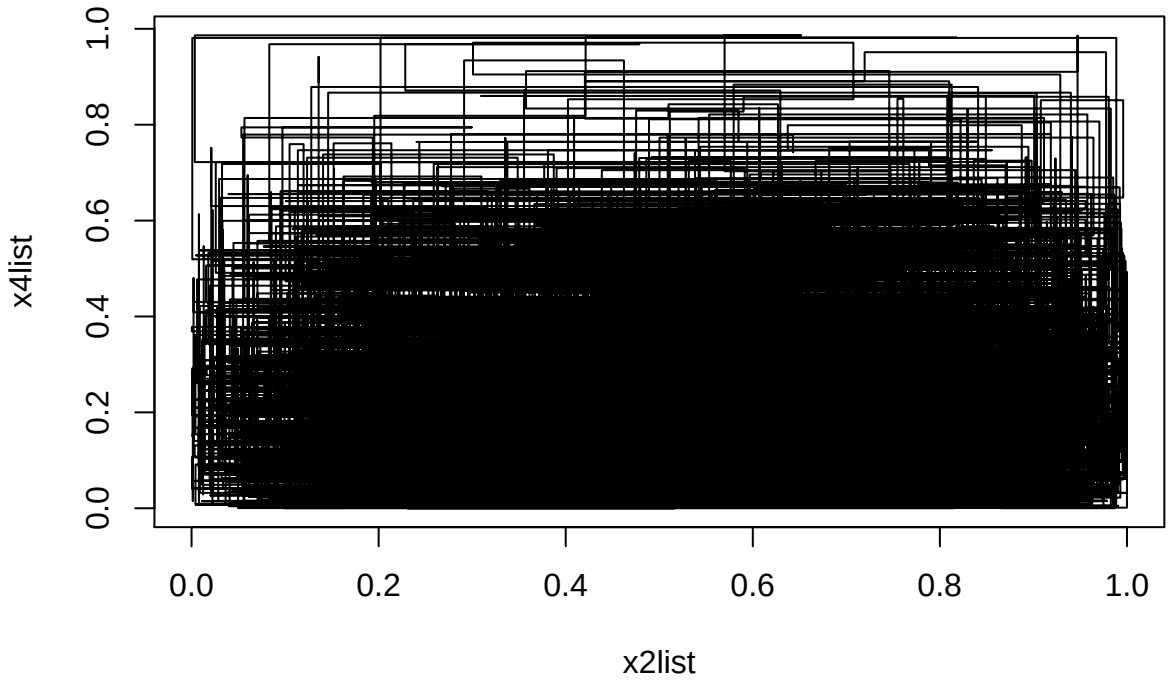

```
plot(x1list, x5list, type='l')
```



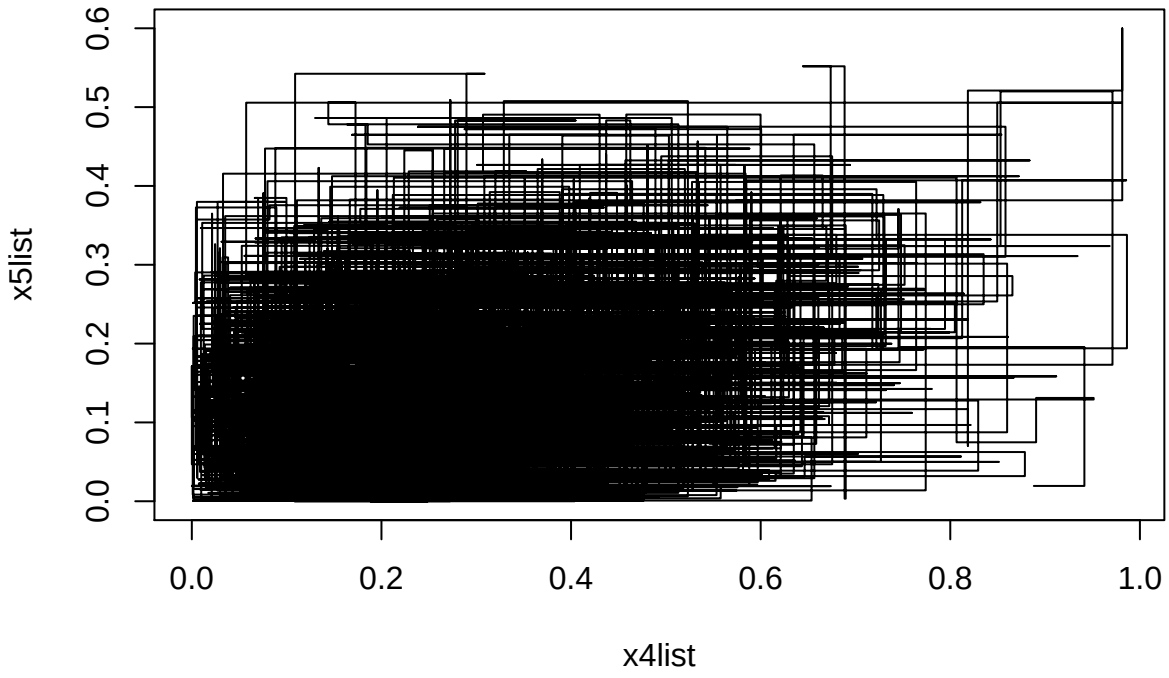

```
plot(x1list, x3list, type='l')
```



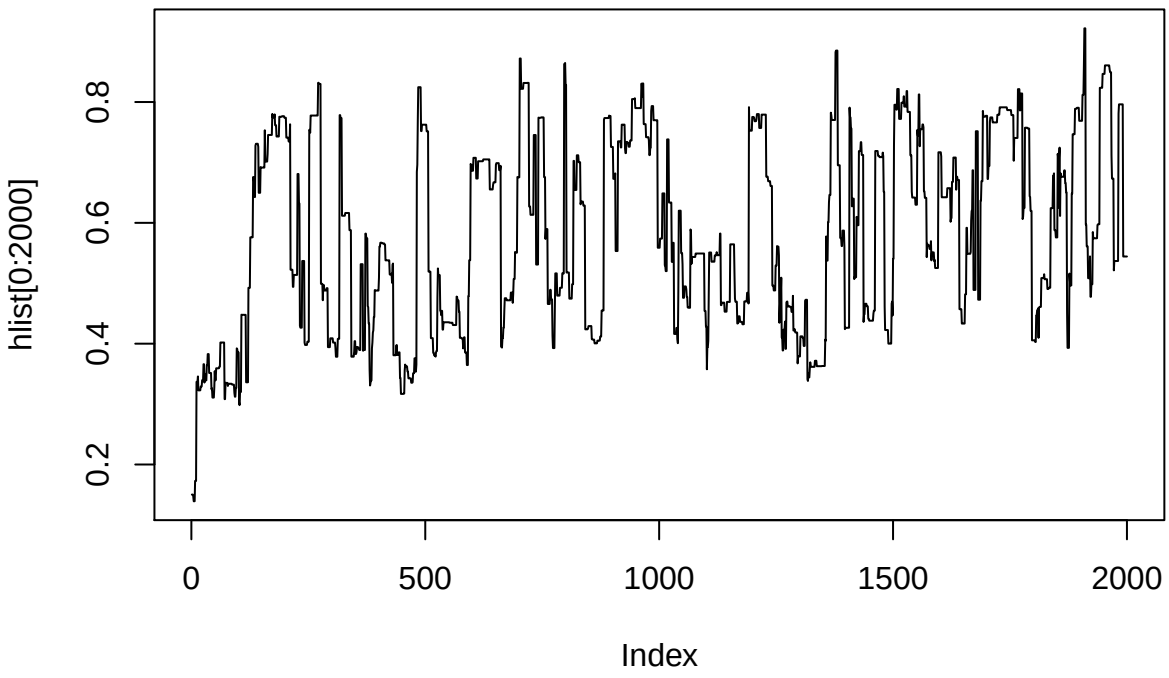

```
plot(x2list, x4list, type='l')
```



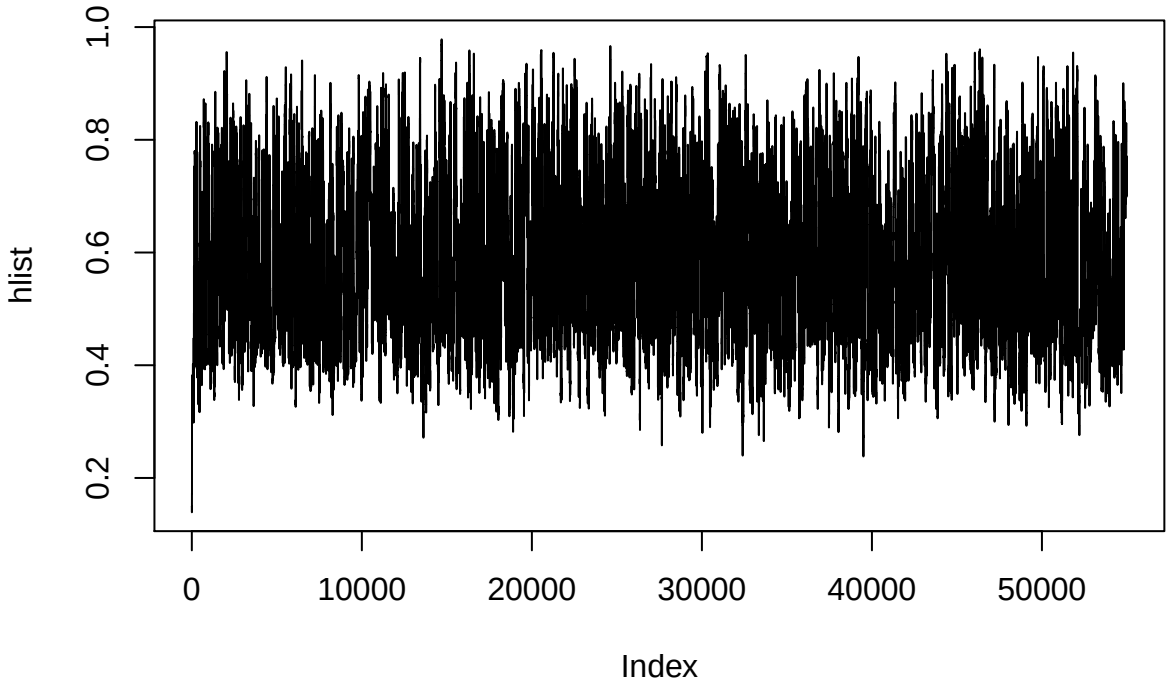

```
plot(x4list, x5list, type='l')
```




```
# plot(x1list, type='l')
# plot(x2list, type='l')
# plot(x1list, x2list, type='p')
plot(hlist[0:2000], type = "l")
```




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




From above output, we see that we have a good mixing for the x and h(x) value and our estimate values after first 1000 values, So I think using burning-in of 1000 is good. The acceptance rate is around 31%, The "relevant" number of lag varies is around 150. The confident interval is around (0.57, 0.59), which is much better compared to full-dimensional metropolis algorithm. The estimate is pretty close to what I got for HW1, so I think the estimate is good.

Let's try $\sigma^2 = 0.1$ (Best for full-dimensional metroplis algorithm in Q2)

```
M = 11000  # run length
B = 1000  # amount of burn-in
X  = c(runif(1), runif(1) , runif(1), runif(1), runif(1)) # overdispersed starting distribution
sigma = 0.1  # proposal scaling
x1list = rep(0,5*M)  # for keeping track of chain values
x2list = rep(0,5*M)
x3list = rep(0,5*M)
x4list = rep(0,5*M)
x5list = rep(0,5*M)

hlist = rep(0,5*M)  # for keeping track of h function values
numaccept = 0;


for (i in 1:M) {
  for (coord in 1:5){
    Y = X
    Y[coord] = X[coord]+ 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;
      }

    x1list[5*i - 5+coord] = X[1];
    x2list[5*i - 5+coord] = X[2];
    x3list[5*i - 5+coord] = X[3];
    x4list[5*i - 5+coord] = X[4];
    x5list[5*i - 5+coord] = X[5];
    hlist[5*i - 5+coord] = h(X);

  }
}

cat("ran Metropolis algorithm for", M, "iterations, with burn-in", B, "\n");
```

```
## ran Metropolis algorithm for 11000 iterations, with burn-in 1000
```

```
cat("acceptance rate =", numaccept/(5*M), "\n");
```

```
## acceptance rate = 0.6854
```

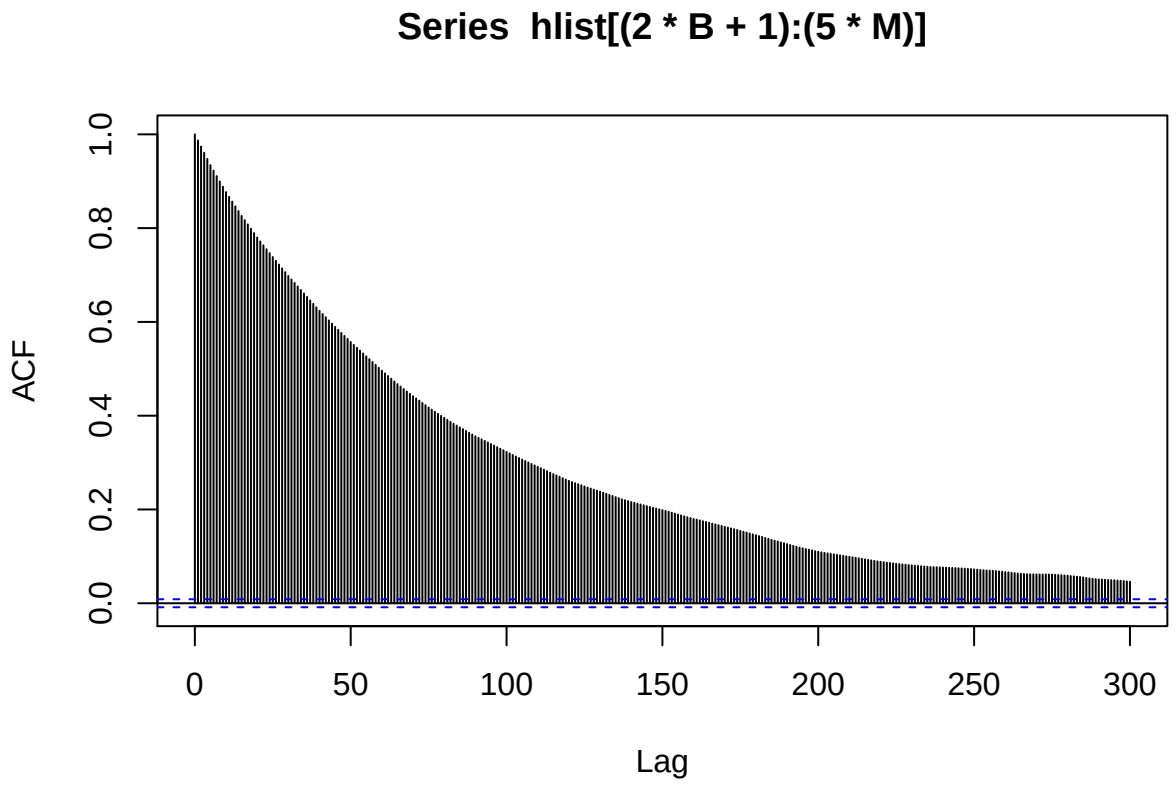
```
u = mean(hlist[(2*B+1):(5*M)])
cat("mean of h is about", u, "\n")
```

```
## mean of h is about 0.5780299
```

```
se1 =  sd(hlist[(2*B+1):(5*M)]) / sqrt(5*(M-B))
cat("iid standard error would be about", se1, "\n")
```

```
## iid standard error would be about 0.00066076
```

```
acf(hlist[(2*B+1):(5*M)], lag.max = 300)
```




```
varfact <- function(xxx) { 2 * sum(acf(xxx, plot=FALSE, lag.max = 300)$acf) - 1 }
thevarfact = varfact(hlist[(2*B+1):(5*M)])
se = se1 * sqrt( thevarfact )
cat("varfact = ", thevarfact, "\n")
```

```
## varfact =  172.9719
```

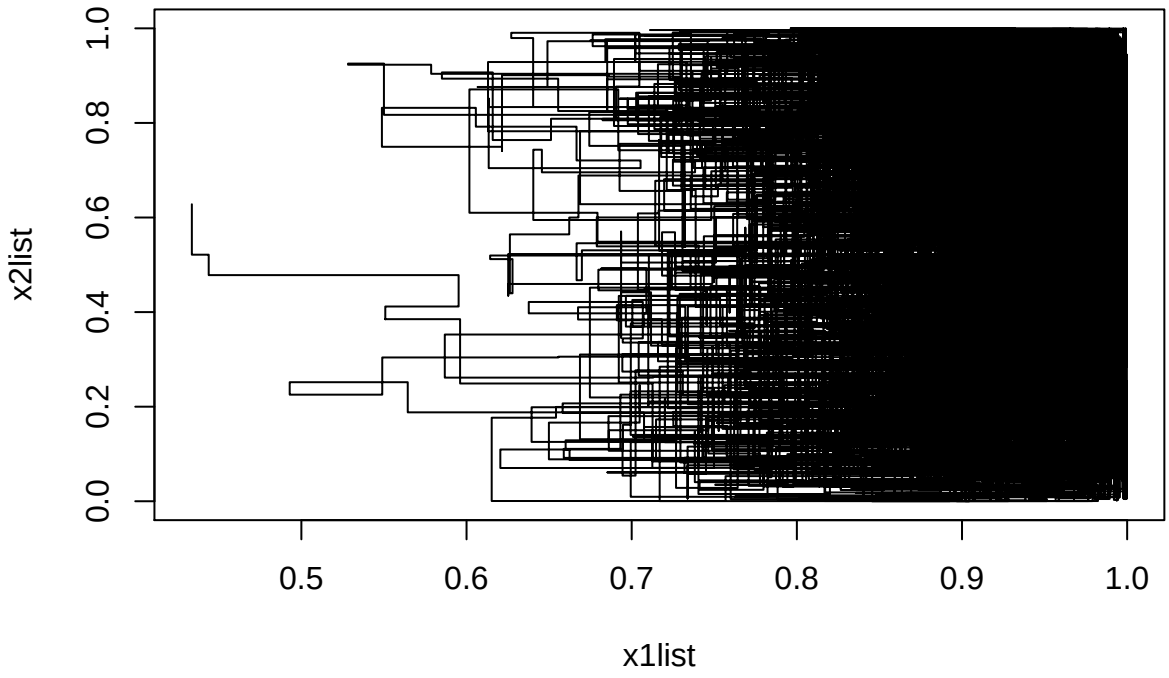
```
cat("true standard error is about", se, "\n")
```

```
## true standard error is about 0.008690236
```

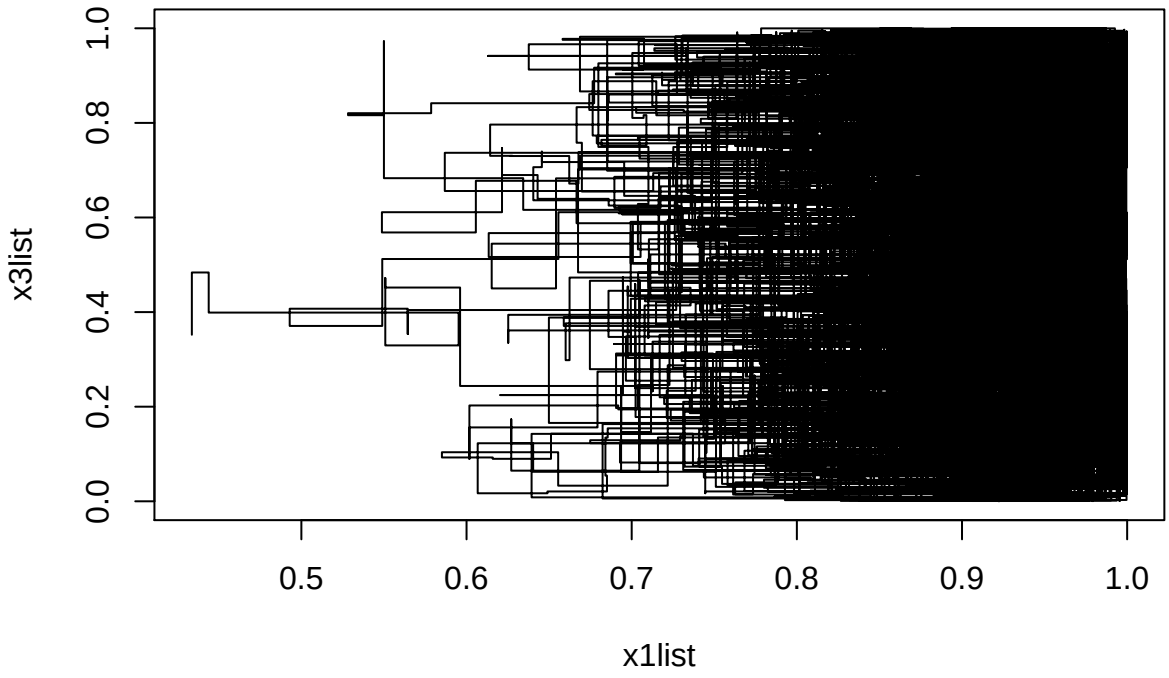
```
cat("approximate 95% confidence interval is (", u - 1.96 * se, ",",
                                                u + 1.96 * se, ")\n\n")
```

```
## approximate 95% confidence interval is ( 0.5609971 , 0.5950628 )
```

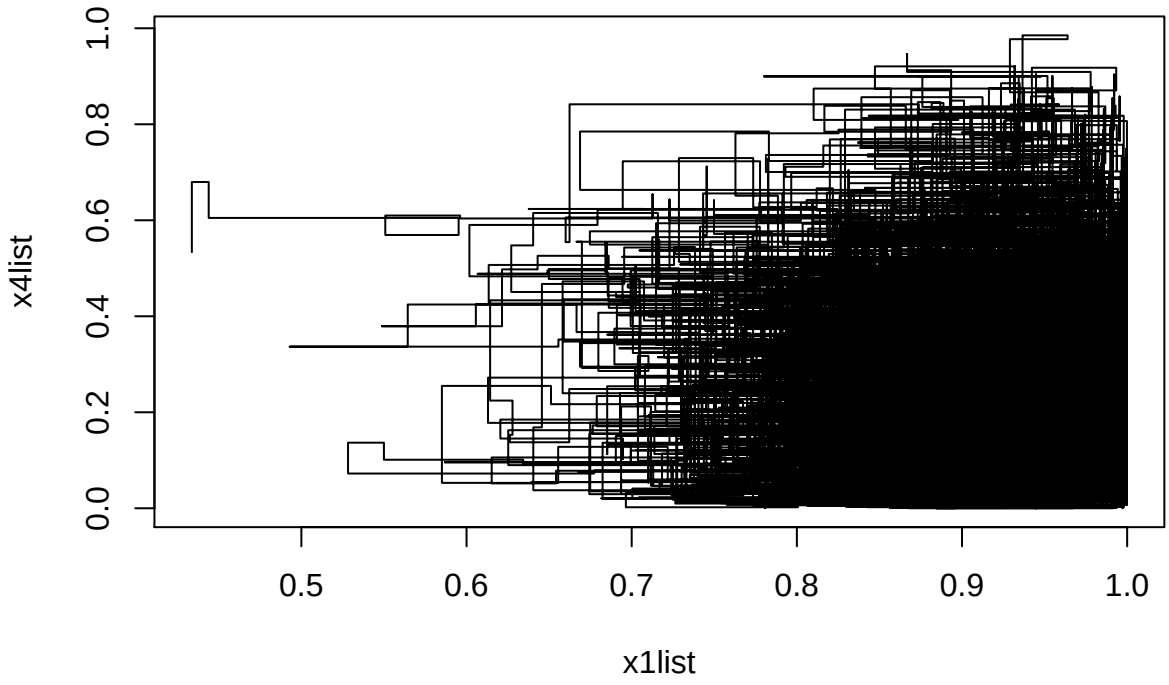
```
plot(x1list, x2list, type='l')
```



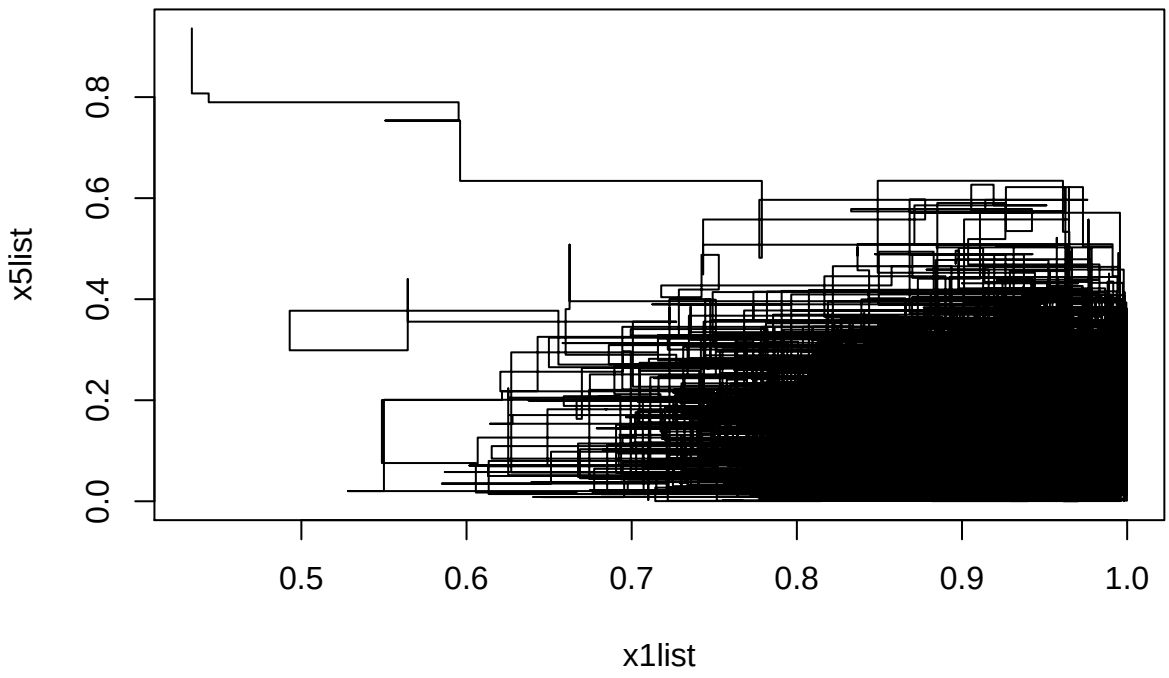

```
plot(x1list, x3list, type='l')
```



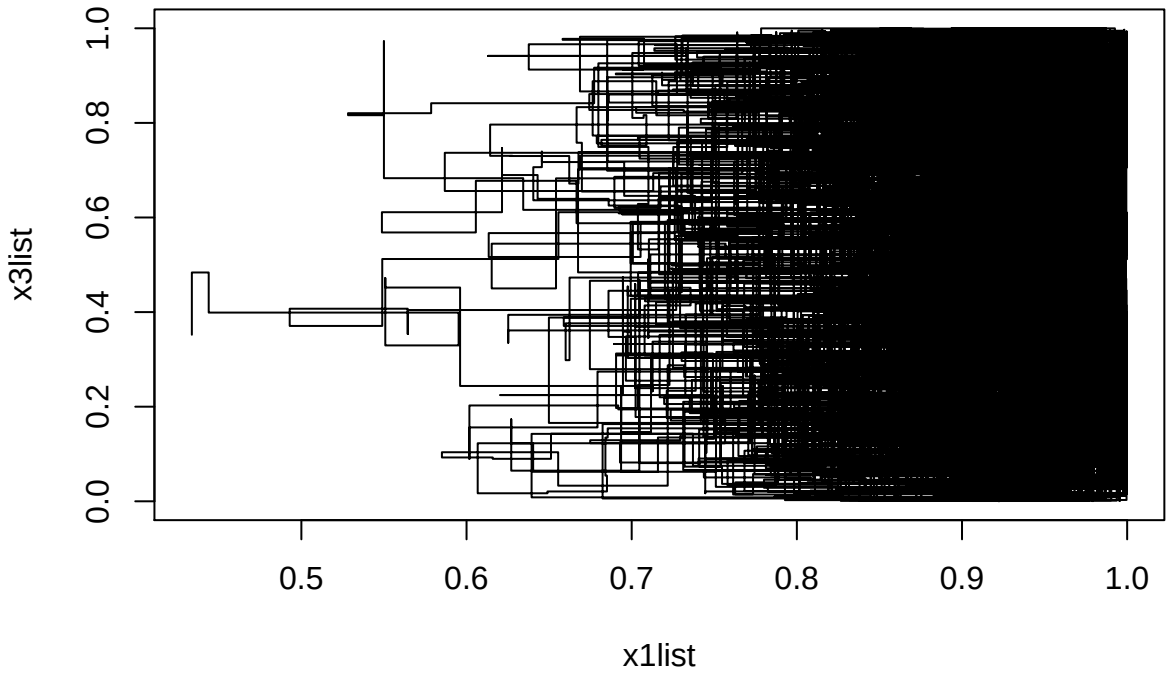

```
plot(x1list, x4list, type='l')
```



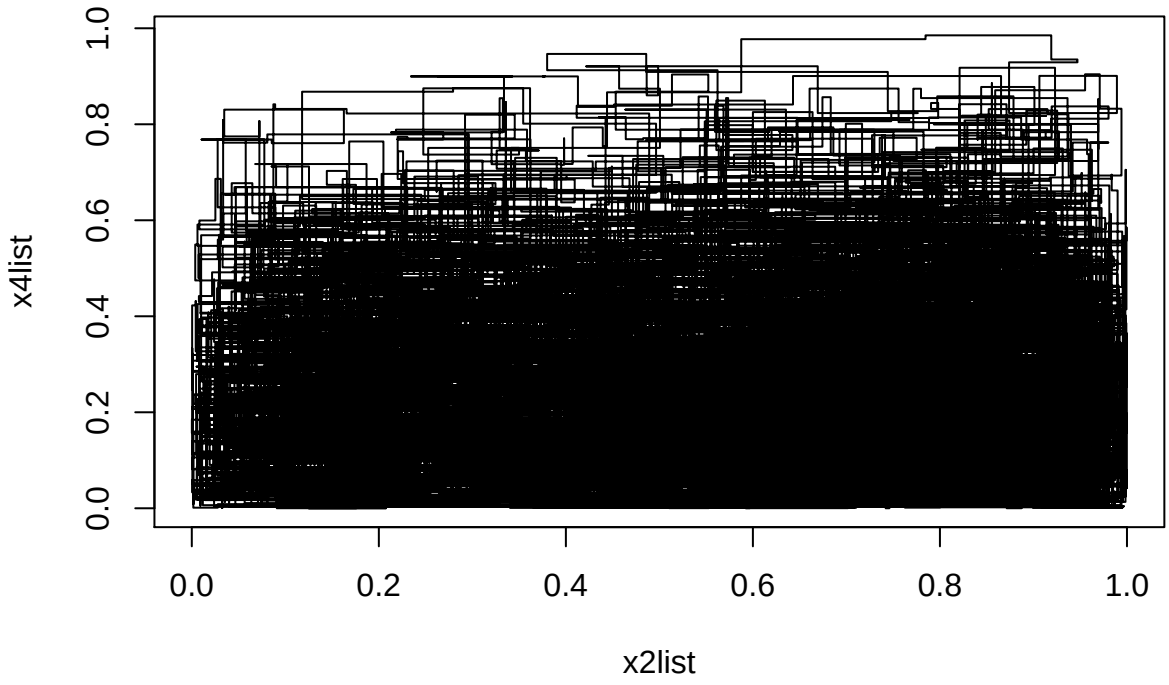

```
plot(x1list, x5list, type='l')
```



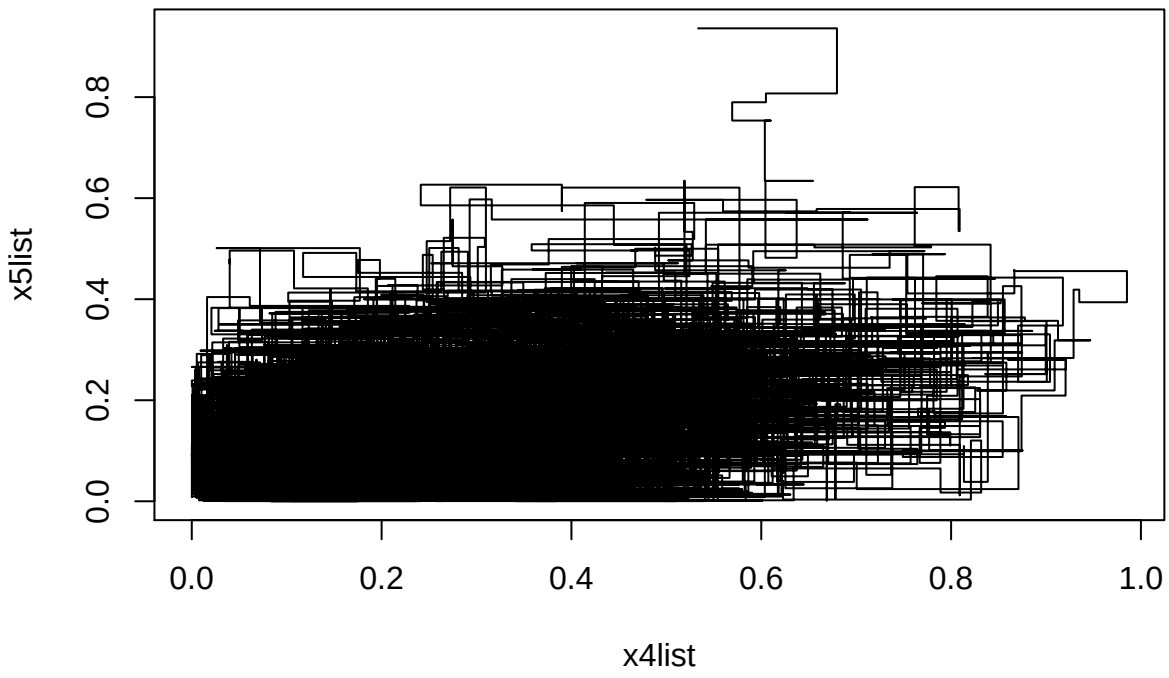

```
plot(x1list, x3list, type='l')
```



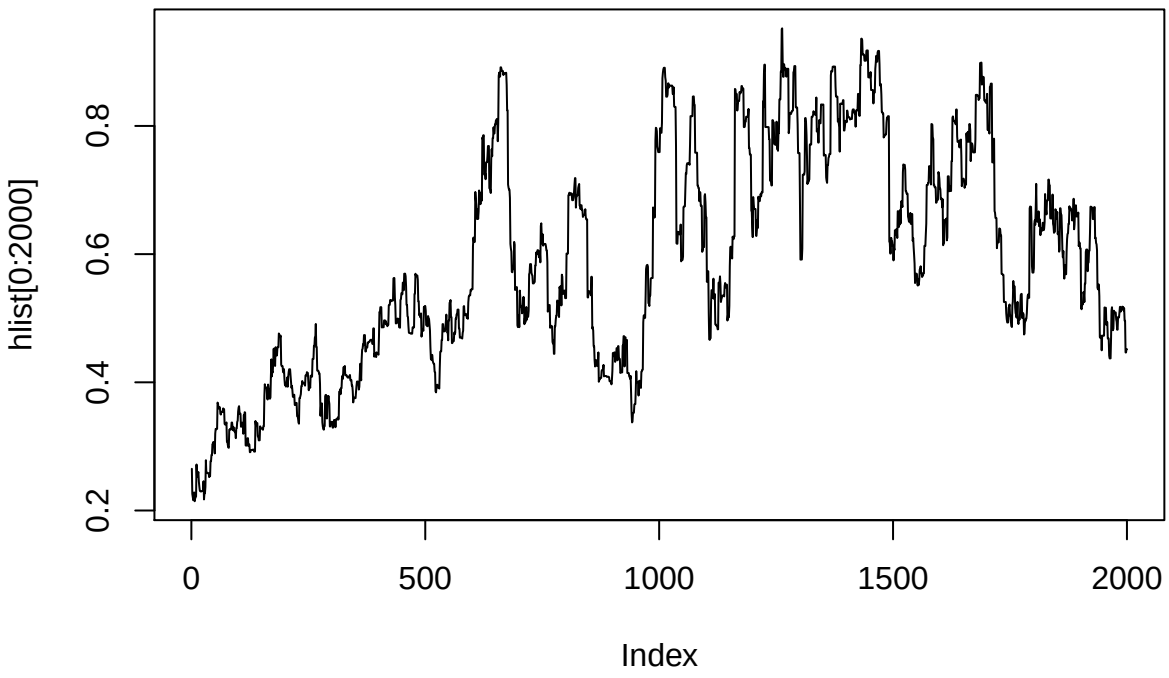

```
plot(x2list, x4list, type='l')
```



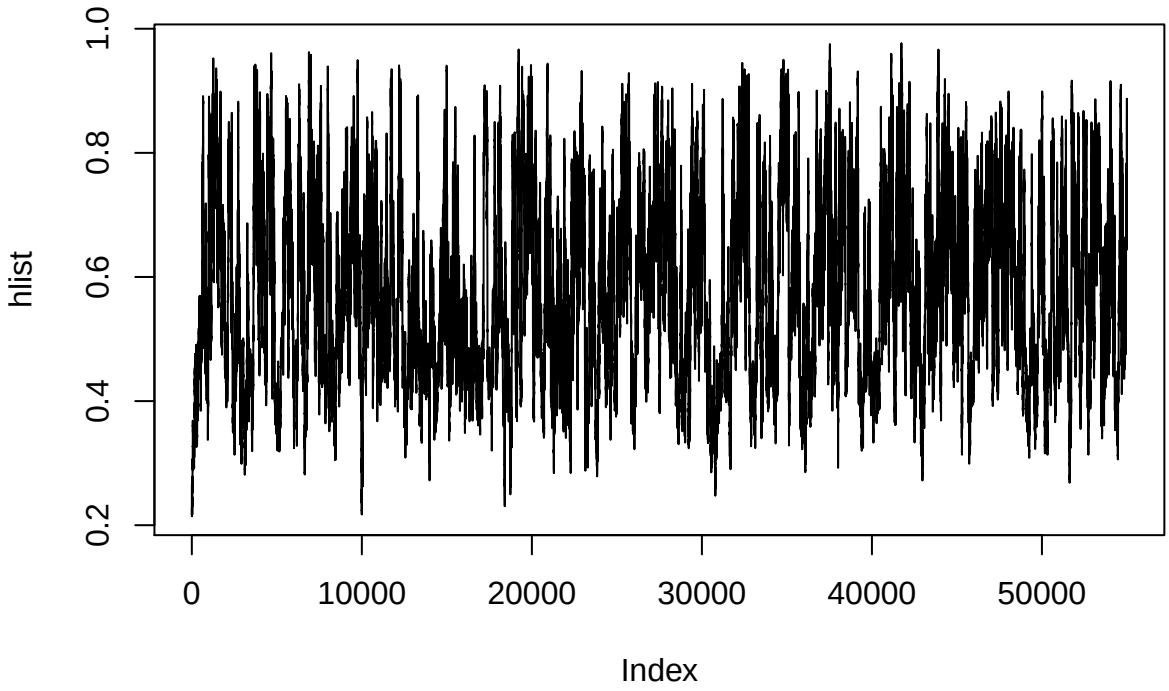

```
plot(x4list, x5list, type='l')
```




```
# plot(x1list, type='l')
# plot(x2list, type='l')
# plot(x1list, x2list, type='p')
plot(hlist[0:2000], type = "l")
```




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




We see that the acceptance rate is quite high(68%) and h(x) does not seem to have good mix for the first 2000 iteration. This means this metropolis algorithm requires higher burning-in. The "relevant" number of lag varies is much larger compared to using $\sigma^2 = 0.5$, which leads to wider confidence interval.

Therefore, using $\sigma^2 = 0.5$ and burn-in 1000 in componentwise metropolis algorithm is better estimating our objective function.

Also, it looks like componentwise metropolis algorithm is better than full-dimensional metropolis algorithm because of smaller variance and confidence interval.