STA3431 Project

Estimate Inbreeding Coefficient Using Monte Carlo Mehtods

Fangming Liao; Jinda Yang

November.26.2018

1001374997; 1002736612
will.liao@mail.utoronto.ca; jinda.yang@mail.utoronto.ca
1st in Master of Biostatistics
Dalla Lana School of Public Health
University of Toronto

Contents

1 Introduction

The Hardy-Weinberg(HW) Equilibrium, states that, in a large random-mating population, assuming no selection, mutation, migration, the allele frequencies and the geno-type frequencies are constant from generation to another, and there is a simple relation between geno-type and allele frequencies [1]. This law is significant as many approaches in human genetics rely on the presence of Hardy-Weinberg Equilibrium. In particular, at a bi-allelic marker, the frequencies of the two alleles (A or B) are p and q, where $p = 1 - q$, the expected geno-type frequencies are: $AA : p^2, AB : 2pq, BB : q^2$.

2 Motivation

However, in practice the assumptions are often violated, and to test the departure from Hardy Weinberg Equilibrium we can simply calculate the expected geno-type frequencies and compare it with the observed ones using a chi-squared test.

Alternatively, a number of methods have emerged for obtaining point estimates of f, a parameter measuring departure from HW caused by inbreeding. And the reasons why such methods are unsatisfactory are discussed by Ayres and Balding [2].

If inbreeding (i.e. selection) is the main violation of HW assumptions, causing deviation from HW, the inbreeding model may be appropriate [8], where p_{ij}, the relative frequency of the geno-type A_iA_j is:

$$\begin{aligned} p_{ii} &= p_i(f + (1 - f)p_i) \\ p_{ij} &= 2p_ip_j(1 - f) \end{aligned} \tag{1}$$

where p_i denotes the frequency of allele A_i, and f is the inbreeding coefficient. When $f = 0$, equation (1) gives the HW proportions. When $f = 1$, the maximum value, hetero-zygotes never arise. The value of f can be negative and it is bounded by below by the requirement that the population frequencies of each homo-zygote be non-negative, which lead to:

$$f \geq \frac{-p_{min}}{1 - p_{min}} \tag{2}$$

where p_{min} is the smallest frequency. [2]

In some models for population subdivision, f can be interpreted as the probability that an individual's two genes are identical by descent [3], in which case it is constrained to be non-negative. Nei Chesser [4] and Robertson Hill [5] proposed their point estimators for the inbreeding coefficient, but these estimators do not explicitly take account of the inbreeding model and may, in the multi-allelic case, give estimates that conflict with the bound (2).

Ayres Balding [2] proposed the maximum likelihood estimator, which respect the bound under the inbreeding model. Assuming random sampling of genotypes, the likelihood is:

$$P(n_{ij}|f, p_1, ..., p_k) = C_1 \prod_{i=1}^{k} (p_i(f + (1-f)p_i)^{n_{ii}} \prod_{j=i+1}^{k} (2p_i p_j(1-f))^{n_{ij}} \tag{3}$$

where C_1 is a constant. For $k = 2$, equation (3) is readily maximized [6] to obtain

$$\hat{f}_{mle} = 1 - \frac{2n_{12}n}{(2n_{11} + n_{12})(n_{12} + 2n_{22})} \tag{4}$$

However, for $k > 2$, the likelihood cannot be maximized analytically, but numerical methods [5] and EM algorithm [7] can be employed.

The likelihood function obtained by setting each parameter other than f (i.e. $p_i's$) equal to its MLE $\hat{p_i}$ provides a measure of the support given by the data to different possible values for f, but it ignores uncertainty in the p_i [2]). While this problem can be solved by integration over the joint distribution of pi, leading to marginal likelihood of f, the exact integration can be unfeasible, and we can approximate the integration by Markov Chain Monte Carlo(MCMC) algorithms.

3 Methodology

3.1 Metropolis-Hastings-within-Gibbs

Let's take a look at the full joint density:

$$
\begin{aligned}
\pi(n_{ij}) &= P(n_{ij}|f, p_1, ..., p_k) \\
&= C_1 \prod_{i=1}^{k} (p_i(f + (1-f)p_i)^{n_{ii}} \prod_{j=i+1}^{k} (2p_i p_j(1-f))^{n_{ij}} \\
&= C_1 \prod_{i=1}^{k} (p_i(f + (1-f)p_i)^{n_{ii}} [(2p_i p_{i+1}(1-f))^{n_{i,i+1}} \dots (2p_i p_k(1-f))^{n_{i,k}}] \\
&= C_1 [(p_1(f + (1-f)p_1))^{n_{11}} (2p_1 p_2(1-f))^{n_{12}} \dots (2p_1 p_k(1-f))^{n_{1k}}] \\
&\quad [(p_2(f + (1-f)p_2))^{n_{22}} (2p_2 p_3(1-f))^{n_{23}} \dots (2p_2 p_k(1-f))^{n_{2k}}] \\
&\quad \dots \\
&\quad [(p_{k-1}(f + (1-f)p_{k-1}))^{n_{k-1,k-1}} (2p_{k-1} p_k(1-f))^{n_{k-1,k}}] \\
&\quad [(p_k(f + (1-f)p_k))^{n_{kk}}]
\end{aligned}
\tag{5}
$$

This is a product of a lot of numbers between 0 and 1, thus, taking log of this joint density can make computation easier and avoid the problem of extremely small value:

$$
\begin{aligned}
log(\pi(n_{ij})) &= log(P(n_{ij}|f, p_1, ..., p_k)) \\
&= n_{11}log(p_1(f + (1-f)p_1)) + n_{12}log(2p_1p_2(1-f)) + \ldots + n_{1k}log(2p_1p_k(1-f)) \\
&\quad + n_{22}log(p_2(f + (1-f)p_2)) + n_{23}log(2p_2p_3(1-f)) \ldots n_{2k}log(2p_2p_k(1-f)) \\
&\quad + \ldots \\
&\quad + n_{k-1,k-1}log(p_{k-1}(f + (1-f)p_{k-1}) + n_{k-1,k}log(2p_{k-1}p_k(1-f)) \\
&\quad + n_{kk}log(p_k(f + (1-f)p_k)) \\
&= \sum_{i=1}^{k} n_{ii}log(p_i(f + (1-f)p_i) + \sum_{i=1}^{k}\sum_{j=i+1}^{k} n_{ij}log(2p_ip_j(1-f))
\end{aligned}
\tag{6}
$$

The proposal function for p_i:

$$p_u^t \sim Unif[max(0, p_u^{t-1} - \epsilon_p), min(p_u^{t-1} + \epsilon_p, p_u^{t-1} + p_v^{t-1})]$$

$$p_v^t = p_u^{t-1} + p_v^{t-1} - p_u^t \tag{7}$$

The proposal function for f:

$$f \sim Unif(max(\frac{-p_{min}^t}{1 - p_{min}^t}, f - \epsilon_f), min(f + \epsilon_f, 1)) \tag{8}$$

where p_{min} is the minimum p_i at step t, ϵ_f

The whole idea is that we will first update a pair of p_u and p_v, then setting the new proposed $p_u^t + p_v^t$ = previous $p_u + p_v$ guarantees that $\sum_{i=1}^{k} p_i = 1$.

Then we accept/reject our proposed p_u and p_v with Metropolis-Hastings rule, where the full joint density function and proposed function are from equation (5) and (7).

Next we propose f with distribution in (8).(Note that only when we accept the p, we can propose new f). During the whole process, we adjust the ϵ_p in order to change the ϵ_f to control our acceptance rate, the (positive) value of ϵ_p is chosen to obtain reasonable acceptance rates:, if ϵ_p is too large, the chain will 'sticks' too much in one place and, hence, converge very slowly; if too small, the chain will make frequent but very small moves and again will converge slowly.

Since our joint density is very complicated, so we deal with it by taking logarithms, e.g. accept iff $log(U_n) < log(A_n)$, where $U_n \sim Unif[0,1]$ and $A_n = \frac{g(f^{new},P^{new})q(f^{old},P^{old})}{g(f^{old},P^{old})q(f^{new},P^{new})}$.

3.2 Gibbs Sampler

Instead of using regular Component-wise Metropolis-Hastings algorithm, we also tried to propose each coordinate according to its conditional distribution, conditioned on all other coordinates. From the full joint distribution (5), the conditional distributions of $f, p_1, ..., p_k$ are computed as:

$$g(f|p_1,...,p_k,\{n_{ij}\}) = \prod_{i=1}^{k}[f+(1-f)p_i]^{n_{ii}} \prod_{j=i+1}^{k}(1-f)^{n_{ij}} \quad for\ \frac{-p_{min}}{1-p_{min}} \le f \le 1 \tag{9}$$

$$g(p1|f,p_2,...,p_k,\{n_{ij}\}) = p_1(f+(1-f)p_1)^{n_{11}}(p_1(1-f))^{n_{12}} \quad for\ 0 \le p_1 \le 1 \tag{10}$$

$$g(p2|f,p_2,...,p_k,\{n_{ij}\}) = p_2(f+(1-f)p_2)^{n_{22}}(p_2(1-f))^{n_{12}+n_{23}} \quad for\ 0 \le p_2 \le 1 \tag{11}$$

$$g(p3|f,p_2,...,p_k,\{n_{ij}\}) = p_3(f+(1-f)p_3)^{n_{33}}(p_3(1-f))^{n_{34}+n_{23}} \quad for\ 0 \le p_3 \le 1 \tag{12}$$

$$g(p4|f,p_2,...,p_k,\{n_{ij}\}) = p_4(f+(1-f)p_4)^{n_{44}}(p_4(1-f))^{n_{34}+n_{45}} \quad for\ 0 \le p_4 \le 1 \tag{13}$$

$$g(p5|f,p_2,...,p_k,\{n_{ij}\}) = p_5(f+(1-f)p_5)^{n_{55}}(p_5(1-f))^{n_{56}+n_{45}} \quad for\ 0 \le p_5 \le 1 \tag{14}$$

$$g(p6|f,p_2,...,p_k,\{n_{ij}\}) = p_6(f+(1-f)p_6)^{n_{66}}(p_6(1-f))^{n_{56}} \quad for\ 0 \le p_6 \le 1 \tag{15}$$

Using a systematically scan, we propose each p_i according to its conditional density and normalize them like we did for the initial values to ensure the sum is 1. In this case, we always accept our proposal, and then we update f using its conditional distribution.

3.3 Independence Sampler

As the performance of M-H algorithm being discussed in the results section, we notice it works quite well. Hence we thought it might be a good idea to use a special case of M-H algorithm, the independence sampler, to see if it can provide us with a more efficient approach.

The full joint distribution is stated before as (5), the proposal distribution for the moving function f is:

$$f \sim Unif(max(\frac{-p_{min}^t}{1-p_{min}^t}, f-\epsilon_f), min(f+\epsilon_f, 1)) \tag{16}$$

where p_{min} is the minimum p_i at step t, ϵ_f

Since our joint density is very complicated and can be really small for the value, we deal with it by taking logarithms, e.g. accept if $log(U_n) < log(A_n)$, where $U_n \sim Unif[0,1]$ and $A_n = \frac{g(Y_n)q(X_{n-1})}{g(X_{n-1})q(Y_n)}$. Thus, the proposed states Y_n are independent of their previous states X_{n-1}.

In implementation, we just ignore the simple case where k = 2 and only consider when k = 6, where MCMC is more likely to be required as numeric methods are implausible.

3.4 Other Monte Carlo methods

Importance sampling seems to be impossible if we don't have detailed information about the sample group, as we would not be able to find the kernel of distribution of allele frequencies and inbreeding coefficient, $(f, p_1, ..., p_k)$, to sample from, so it's too inefficient. If, we insist to generate parameters from some general distribution, e.g. uniform, the results are too inefficient, and in some way make no sense as in figure 2.

Rejection Sampler also seem to be unreasonable, as it's to hard to find the suitable K and $f(x)$ to bound our joint density function, even for simplest case

where k = 2.

4 Simulations

4.1 Dataset1: Biallelic Site

When a specific locus in a genome that contains two observed alleles, then this site is called biallelic site, and in our study, $k = 2$. Suppose the inbreeding coefficient, f, among our observed sample is 0.05, and there are 200 people in our sample, with an allele frequency of $p_1 = 0.25, p_2 = 0.75$, the genotype frequencies can be simulated by equation (1) using the inbreeding model. Then, the phenotype frequencies are computed as $n_{ij} = n \times p_{ij}$, which is our observed data. However, this simulation usually gives us phenotype frequencies that are not integers, so we approximate them to give us a more practical observation.

4.2 Dataset2: Multiallelic Site

When a specific locus in a genome that contains three or more observed alleles, then this site is called multiallelic site, and in our study, we particularly consider $k = 6$. Still suppose the inbreeding coefficient, f, among our observed sample is 0.05, and there are 1000 people in our sample, with an allele frequency of $p_i = (0.02, 0.06, 0.075, 0.085, 0.21, 0.55)$ for $i = 1, 2, ..., 6$, the genotype frequencies can be simulated by equation (1) using the inbreeding model. Similarly to $k = 2$, the phenotype frequencies are computed as $n_{ij} = n \times p_{ij}$, which is our observed data. Also, this simulation usually gives us phenotype frequencies that are not integers, so we approximate them again to give us a more practical observation.

5 Results

5.1 MLE on Dataset1

When k = 2, we can use equation (4) for our MLE estimation. When simulating the data, we used the nearest integer of n_{ij}, e.g. 1 for 1.36, to get the first estimate, and since this value is below the exact value of n_{ij}, the estimate of f will be smaller/larger; then we use the next nearest integer of $n_{ij} + 1$, e.g 2 for 1.36 + 1, to get the second estimate festimate2. Averaging the first and second estimator would give us a final MLE estimate with a more narrow error than only using one of them. The final estimate we get is around 0.05281472 when n = 200. Notice: if we do not round n_{ij} as an integer, we can get 0.05 exactly.

5.2 M-H Algorithm on Dataset1

Here's a table of acceptance rate by different values of ϵ_p

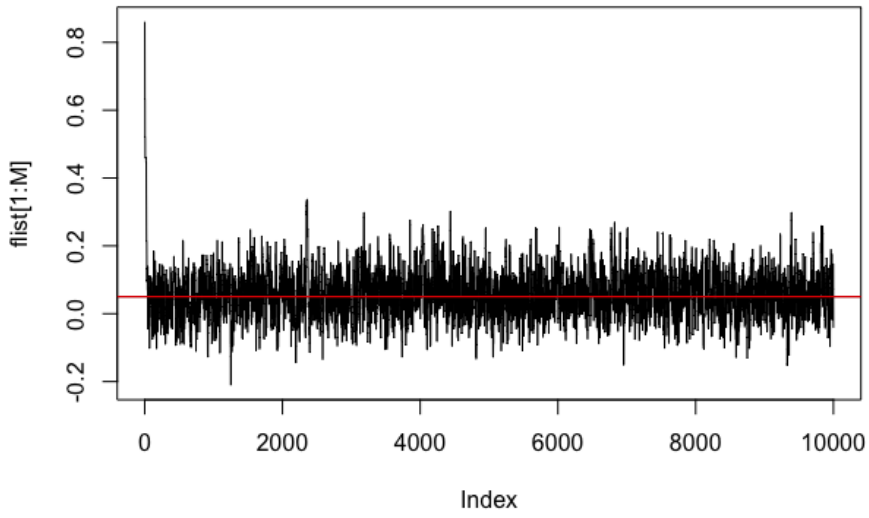
$\epsilon_\mathbf{p}$	**Acceptance Rate**	**Standard Error**
0.01	0.7974	0.004333330
0.02	0.6348	0.002964014
0.03	0.4908	0.002345893
0.04	0.3822	0.002274158
0.05	0.2859	0.002076836
0.06	0.2158	0.002260030
0.07	0.1629	0.002124728
0.08	0.1289	0.002367256
0.09	0.1040	0.002805974
0.10	0.0867	0.003053877

Table 1: Acceptance rate by different ϵ_p when $k = 2$

We notice when $\epsilon_p = 0.02$ to 0.05, the acceptance rates are away from 0 and 1, as well as relative low standard error from 0.03 to 0.08. By setting $\epsilon_p = 0.03$, the algorithm was performed for 10000 iterations with a length of 1000 "burn-in", and we give our estimate for f:

$$\hat{f} = \frac{1}{M-B} \sum_{i=B+1}^{M} f_i = 0.05230045$$

and a 95% Confidence interval for this estimator is: $(0.04777199, 0.05682892)$ and this Confidence interval covers our true theoretical value 0.05, which is good.



From the graph we notice that its good 'mixing', low uncertainty and the chain converges very quickly, and stays pretty close to the true value of f (around 0.05).

5.3 M-H Algorithm on Dataset2

Here's a table of acceptance rate by different values of ϵ_p

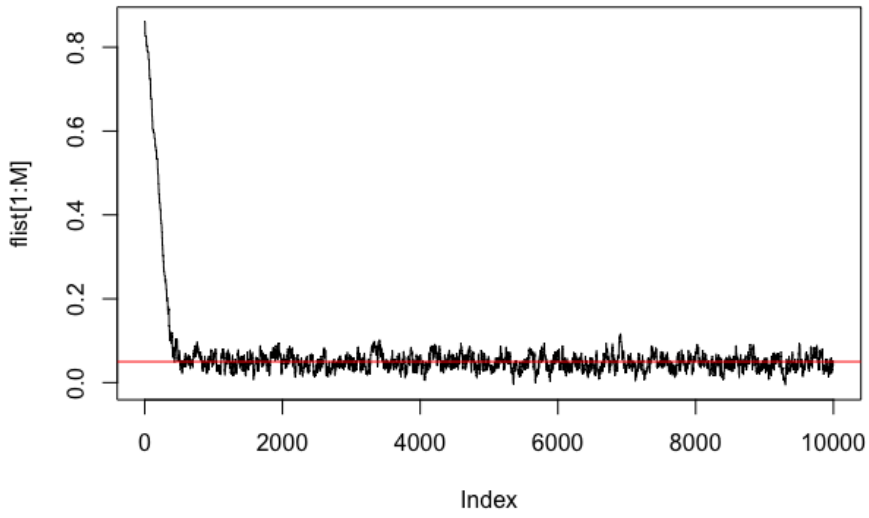
We notice when $\epsilon_p = 0.006$ or 0.014, the acceptance rates are away from 0 and 1, as well as relative low standard error from 0.01 to 0.02. By setting $\epsilon_p = 0.01$, the algorithm was performed for 10000 iterations with a length of 1000 "burn-in", and we give our estimate for f:

$\epsilon_{\mathbf{p}}$	**Acceptance Rate**	**Standard Error**
0.001	0.7862	0.0134081678
0.004	0.7275	0.0016441060
0.006	0.6513	0.0011857324
0.007	0.6091	0.0011977114
0.008	0.5566	0.0011500971
0.010	0.4821	0.0010797890
0.012	0.4186	0.0010551926
0.014	0.3565	0.0008867617
0.017	0.2925	0.0008183769
0.019	0.2604	0.0008544226

Table 2: Acceptance rate by different ϵ_p when $k = 6$

$$\hat{f} = \frac{1}{M - B} \sum_{i=B+1}^{M} f_i = 0.05072752$$

and a 95% Confidence interval for this estimator is: $(0.04837185, 0.05308320)$ and this Confidence interval covers our true theoretical value 0.05, which is good.

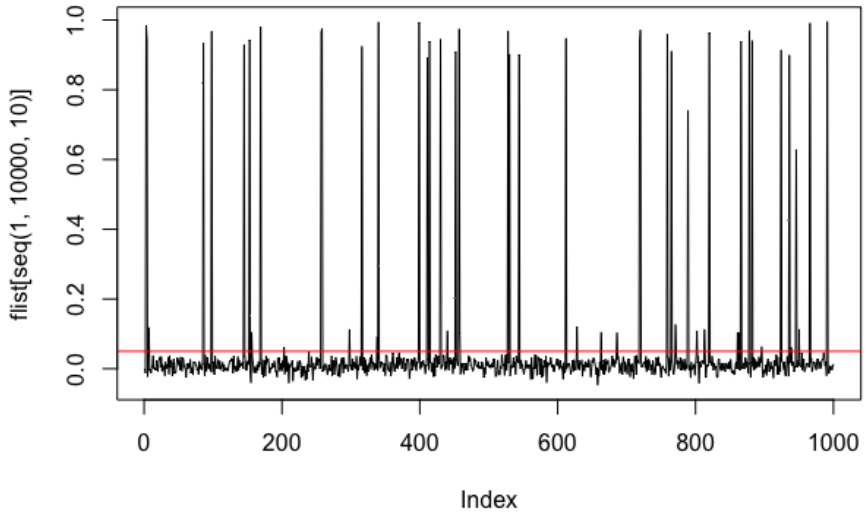


Seeing from the graph we found that the chain converges within 1000 itera-

tions, and it stays close to the true f value as well, And good 'mixing' and pretty low uncertainty.

5.4 Gibbs Sampler

$$\hat{f} = \frac{1}{M-B} \sum_{i=B+1}^{M} f_i = 0.03558$$

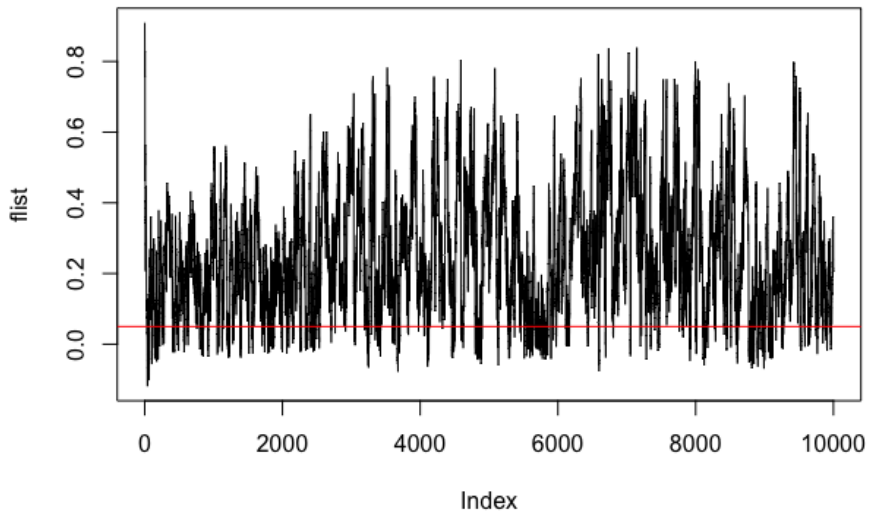


Seeing from the graph we notice that, the chain converges, to around 0.03, performing not well as a little bit away from our true value (0.05). Also, we can see that it has very high uncertainty, which is mainly due to the fact that the coefficient f is correlated with our p_i. When we were using the conditional distribution to generate p_i, problems arises, which will be discussed in details in the discussion section.

5.5 Independence Sampler

The result was not good, whereas the chain does not actually converge although it converges to our true value at the beginning for a moment. Even when we tuned the eps.p, it does not make any better.

$$\hat{f} = \frac{1}{M-B} \sum_{i=B+1}^{M} f_i = 0.2897391$$

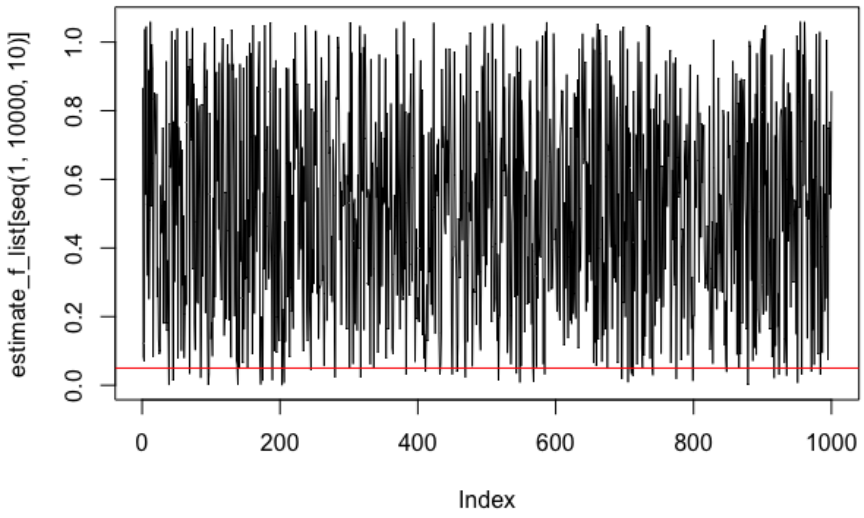


Seeing the graph and the results, we can see that Independence Sampler performs poorly, converges far away from our goal (red line – 0.05), and it has very high uncertainty.

The reason of this failure might be that Y_n are i.i.d and independent of X_{n-1}, actually they are correlated by looking the formula (1). We know that in Genetic field, people's Genes are all correlated, just isolated some genotypes will definitely affect the inbreeding coefficient, which could be the main part for this algorithm's failure.

5.6 Importance Sampling

$$\hat{f} = 0.5205245$$



The Importance Sampling performs quite bad even for the simplest case when k = 2, as stated in section 3.4 the distribution of f is really hard to determine, that's the main reason causing this results. Moreover, we found that Importance sampler could be used for some simple, low-dimention function, but impossible to implement for high-dimention, complicate functions.

6 Conclusion

Over all the MCMC that we use, Metropolis-Hastings-within-Gibbs performed the best, in both low-dimension and high-dimension case, providing us with a pretty accurate estimate to the true parameter. However, the proposal distribution requires prior knowledge of the parameters.

However, all other algorithms were unsatisfactory. For Independent Sampler,

the estimate was far away from the true value, and gives a large standard error. On the other hand, both of the Importance Sampling and Rejection Sampler performed pretty bad, as K and f(x) are hard to find.

Gibbs Sampler provided an estimate that is more close true value comparing with other unsatisfactory MCMC algorithms, but still with a high uncertainty. Nevertheless, the Gibbs sampler itself is a very powerful and efficient MCMC algorithm as long as we have a reasonable conditional distribution.

7 Discussion

For the Gibbs sampler, since the conditional distribution for each parameter is very unique, so we have to generate them from their density function. Firstly we considered to use a MCMC method to generate the sample, but later we managed to generate a sample each time by looking for the root of $U_n = F(x)$ where $U_n \sim Unif[0,1]$ and $F(x)$ is the cumulative distribution for the parameters conditional distribution, which is an approach of inverse CDF.

However, as the conditional distribution is in form of product and power of number of observation, it can get significantly small. To avoid the problem of zero in the denominator when updating f, we set the parameters to 10^20 if the computer recognize it as a zero, especially for $p_4, ..., p_6$.

Even though, the value of above parameters gets close to zero frequently, causing f close to 1 frequently as shown in the graph. Unfortunately, we didn't manage to solve this problem by taking log of the conditional distribution, as the scale of CDF changes if the density changes, making it really difficult and inaccurate to find roots for $(U_n) = log(F(x))$.

Moreover, the R package, *distr*, was employed to generate random samples from the conditional distribution. But it spends super long time make the density as a distribution when the power of the function is big (e.g. > 4).

8 Acknowledgements

We thank Dr. Andrew Paterson at the Hospital of Sick Children, Toronto, for providing help in our interpretation of the departures from Hardy-Weinberg equilibrium.

References

[1] Hardy HG (1908) Mendelian proportions in a mixed population. Science 28:49–50

[2] Ayres, K. L. Balding, D. J.(1998) Measuring departures from Hardy–Weinberg: a Markov chain Monte Carlo method for estimating the inbreeding coefficient. Heredity 80, 769–777.

[3] Crow, J. F. and Kimura, M. (1970). An Introduction to Population Genetics Theory. Harper Row, New York.

[4] Nei, M. and Chesser, R. K. (1983). Estimation of fixation indices and gene diversities. Ann Hum Genet, 47: 253–259.

[5] Robertson, A. and Hill, W. G. (1984). Deviations from Hardy–Weinberg proportions: sampling variances and use in estimation of inbreeding coefficients. Genetics, 107: 703–718.

[6] Weir, B. S. (1996). Genetic Data Analysis II Sinauer Associates, Sunderland, MA.

[7] Hill, W. G., Babiker, H. A., Ranford-Cartwright, L. C. and Andwalliker, D. (1995). Estimation of inbreeding coefficients from genotypic data on multiple alleles, and application to estimation of clonality in malaria parasites. Genet Res, 65: 53–61.

[8] Malécot, G. (1969). The Mathematics of Heredity, W. H. Freeman, San Francisco.

Appendix - R code and outputs

```
# Proposal distribution for p_i: u,v randomly chosen from 1,...,k
# p_u^{t} \sim Unif[max(0,p_u^{t-1}-\epsilon_p),
# min(p_u^{t-1}+\epsilon_p, p_u^{t-1}+p_v^{t-1})]
# then p_v^{t}=p_u^{t-1}+p_v{t-1}-p_u{t}
qq.p <- function(x,u,v,eps.p){
  1/(min(x[u]+eps.p, x[u]+x[v]) - max(0, x[u]-eps.p))
}

# Proposal distribution for f: Unif(max(-p_{min}^t)/(1-p_{min}^t,f-\epsilon_f))
qq.f <- function(x,y,k,eps.f){
  1/(min(x[1]+eps.f, 1) - max(-min(y[2:(k+1)])/(1-min(y[2:(k+1)])), x[1]-eps.f))
}
```

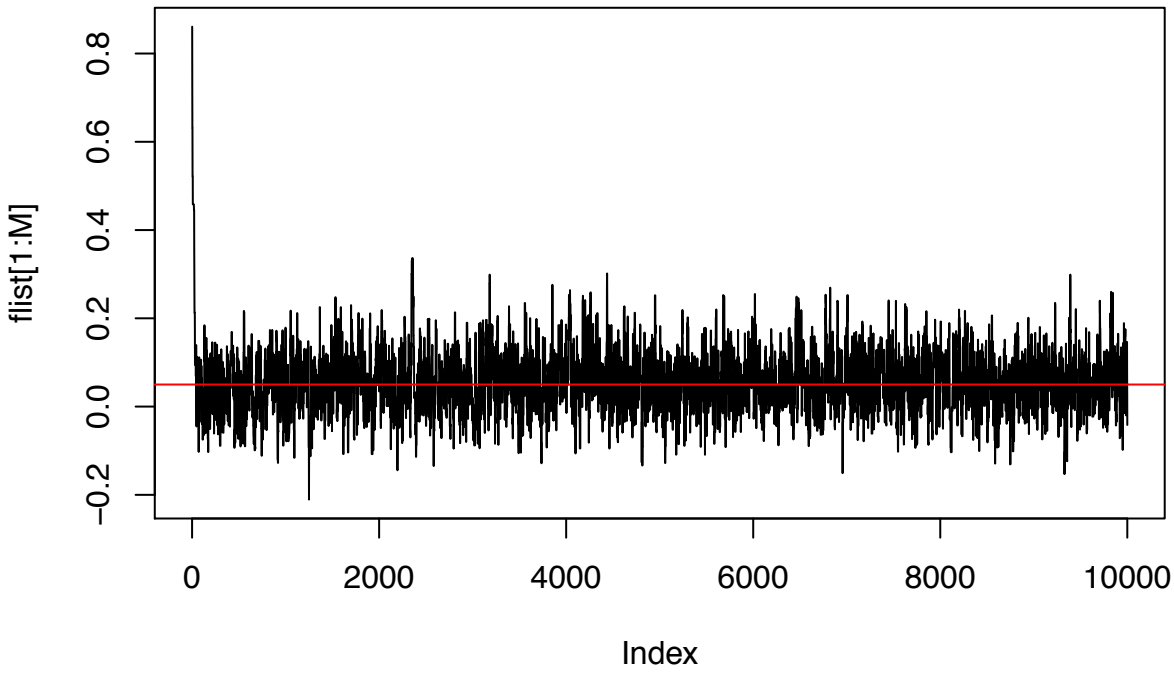
```
##### MH Algorithm for k=2
MHk2 <- function(eps = 0.05){
  # log of the joint distribution function, the reason we do that is mainly due to
  # the number is too small, by doing so, we can enlarge our number and avoid flowing
  # for k = 2
  log.g = function(X,n11,n12,n22){
    n11*(log(X[2])+log(X[1]+(1-X[1])*X[2])) +
      n12*log(2*X[2]*X[3]*(1-X[1]))+
      n22*(log(X[3])+log(X[1]+(1-X[1])*X[3]))
  }
  ### data simulation
  eps.p <- eps
  k  = 2    # number of alleles
  n  = 200  # sample size
  f  = 0.05 # true inbreeding coef.
  # true allele frequencies
  p1 = 0.25
  p2 = 0.75

  p11 = p1*(f+(1-f)*p1)
  p22 = p2*(f+(1-f)*p2)
  p12 = 2*p1*p2*(1-f)
  n12 = round(p12*n)
  n11 = round(p11*n)
  n22 = n-n11-n12

  #### the algo
  # initial values
  X    <- rep(0,3)
  X[1] <- runif(1)
  a = runif(1)
  b = runif(1)
  # in this way we can gurantee that p1+p2 = 1
  p1 = a/(a+b)
  p2 = b/(a+b)
  X[2] = p1
  X[3] = p2 # overdispersed value
```

```
  M  =10000
  B = 1000 # burn value
  #since the eps.f should satisfy eps.f > (k^2)*eps.p/((k - 1)(k - 1 - k*eps.p))
  eps.f =((k^2)*eps.p/((k-1)*(k-1-k*eps.p)))+0.0001

  numaccept = 0
  flist = rep(0,M)

  for (i in 1:M) {
    Y = X

    r = sample(c(2,3),2) # propose p1 or p2
    u = r[1]
    v = r[2]

    Y[u] = runif(1,max(0,Y[u] - eps.p),min(Y[u]+eps.p,Y[u]+Y[v]))
    Y[v] = 1-Y[u] # p1+p2 = 1 when k=2

    U  = runif(1) # for accept/reject
    alpha = log.g(Y,n11,n12,n22) + log(qq.p(X,u,v,eps.p)) -
      log.g(X,n11,n12,n22) - log(qq.p(Y,u,v,eps.p))

    if(log(U) < alpha){
      X = Y
      # now we update the f when p' is accepted
      Z = X

      Z[1] = runif(1,max(-min(Y[2:3])/(1-min(Y[2:3])), X[1]-eps.f),min(X[1]+eps.f, 1))

      W = runif(1) # for accept/reject

      beta = log.g(Z,n11,n12,n22) + log(qq.f(X,Z,k,eps.f)) -
        log.g(X,n11,n12,n22) - log(qq.f(Z,X,k,eps.f))

      if(log(W)<beta){
        X = Z
        numaccept = numaccept + 1
      }
    }
    flist[i] = X[1]
  }
  estmean = mean(flist[(B+1):M])
  se1 =  sd(flist[(B+1):M]) / sqrt(M-B)
  varfact <- function(xxx) { 2 * sum(acf(xxx, plot=FALSE)$acf) - 1 }
  se2 = se1 * sqrt( varfact(flist[(B+1):M]) )
  ci = c(estmean - 1.96*se2, estmean + 1.96*se2)
  return(list(numaccept/M, estmean, ci, flist, M, B, se2))
}

# test which epsilon gives the best acceptance rate (e.g. away from 0 and 1)
set.seed(9999)
epslist  <- seq(0.01,0.1,0.01)
acclist = meanlist = cilblist = ciublist= selist = rep(0,10)
```

```r
for (i in 1:10){
  result <- MHk2(epslist[i])
  acclist[i]  <- result[[1]]
  meanlist[i] <- result[[2]]
  cilblist[i] <- result[[3]][1]
  ciublist[i] <- result[[3]][2]
  selist[i]   <- result[[7]]
}
results <- cbind(epslist, acclist,meanlist,cilblist,ciublist, selist)
results <- as.data.frame(results)
colnames(results) <- c("Epsilon", "Acceptance Rate", "Mean", "Lower bound of CI",
                       "Upper bound of CI", "Standard Error")
results
```

```
##    Epsilon Acceptance Rate       Mean Lower bound of CI Upper bound of CI
## 1     0.01          0.7974 0.05627718        0.04778385        0.06477050
## 2     0.02          0.6348 0.05262606        0.04681659        0.05843553
## 3     0.03          0.4908 0.05113373        0.04653577        0.05573168
## 4     0.04          0.3822 0.04802755        0.04357020        0.05248490
## 5     0.05          0.2859 0.05354149        0.04947089        0.05761208
## 6     0.06          0.2158 0.05882788        0.05439823        0.06325754
## 7     0.07          0.1629 0.05523329        0.05106883        0.05939776
## 8     0.08          0.1289 0.06051190        0.05587208        0.06515172
## 9     0.09          0.1040 0.06342502        0.05792531        0.06892473
## 10    0.10          0.0867 0.06286519        0.05687959        0.06885079
##    Standard Error
## 1     0.004333330
## 2     0.002964014
## 3     0.002345893
## 4     0.002274158
## 5     0.002076836
## 6     0.002260030
## 7     0.002124728
## 8     0.002367256
## 9     0.002805974
## 10    0.003053877
```

```r
# when epsilon = 0.02 or 0.03 the algo seems to perform better
set.seed(9999)
result <- MHk2(0.03)
flist <- result[[4]]
M <- result[[5]]
B <- result[[6]]
plot(flist[1:M], type = "l")
abline(h=0.05, col="red")
```




```
# mean
estmean = mean(flist[(B+1):M])
estmean
```

```
## [1] 0.05575833
```

```
# 95$ CI
se2 = result[[7]]
ci = c(estmean - 1.96*se2, estmean + 1.96*se2)
ci
```

```
## [1] 0.05123021 0.06028644
```

```
############  MH for k=6
MHk6 <- function(eps = 0.05){
  # log of the joint distribution function, the reason we do that is mainly due to
  # the number is too small, by doing so, we can enlarge our number and avoid flowing
  # for k = 6
  log.g = function(X,N){
    dens = 0
    k = length(X) - 1
    for (i in 1:k) {
      if (i < k){
        for (j in (i+1):k){
          dens = dens + N[i,j]*log(2*X[i+1]*X[j+1]*(1-X[1]))
        }
      }
      dens = dens + N[i,i]*log(X[i+1]*(X[1]+(1-X[1])*X[i+1]))
    }
    return(dens)
  }
  # data simulation
  eps.p = eps
  n <- 1000 # sample sizes
```

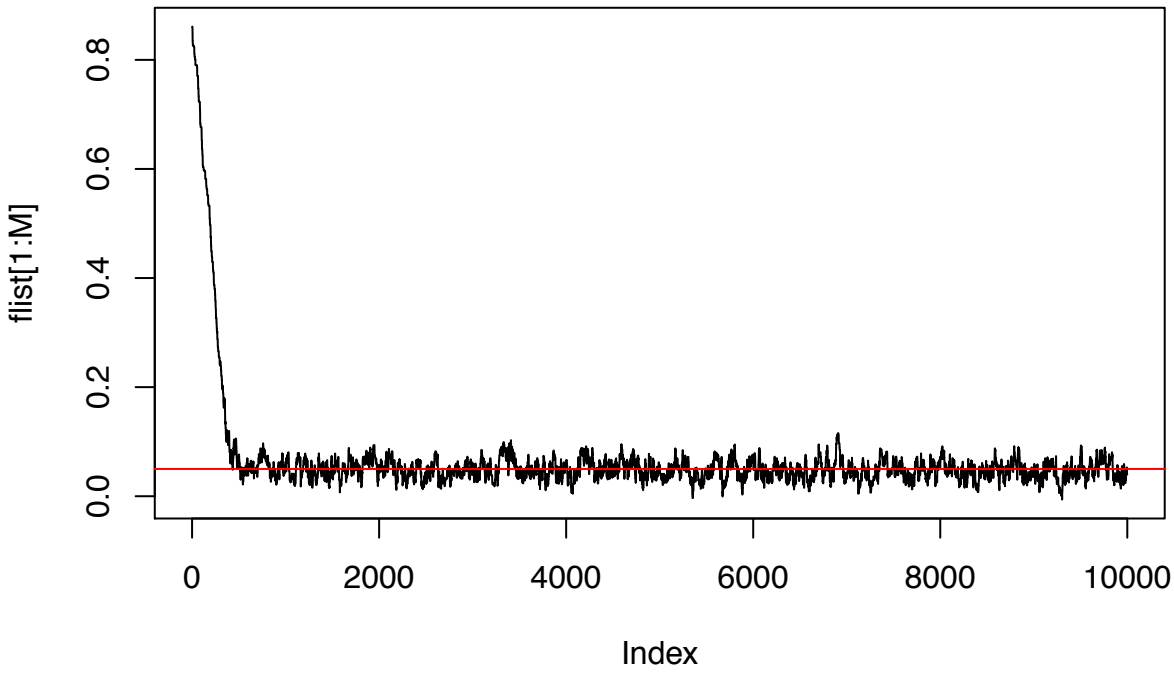
```r
k <- 6    # number of alleles
f <- 0.05 # true inbreeding coefficient
# allele frequencies when k = 6
k6 <- c(0.02,0.06,0.075,0.085,0.21,0.55)
# sum(k6) # this should be 1
# matrix of genotype frequencies
# pij represents the frequency of AiAj
P <- matrix(nrow = 6, ncol = 6)
for (i in 1:6){
  for (j in 1:6){
    if (i==j){
      P[i,j] <- k6[i]*(f+(1-f)*k6[i])
    }
    else {
      P[i,j] <- 2*k6[i]*k6[j]*(1-f)
    }
  }
}
# sum(P[upper.tri(P, diag = T)]) # this should be 1

# matrix of genotype counts
# Nij represents the number for AiAj
N <- round(P*n)
# sum(N[upper.tri(N,diag = T)])
# keep the total population to 1000
N[6,6] <- 316
# sum(N[upper.tri(N, diag = T)]) # this should be 1000 now

#### the algo
# initial values
X      <- rep(0,7)
X[1]   <- runif(1)
ps     <- runif(6)
X[2:7] <- ps/sum(ps) # make sure sum of p is 1

M = 10000
B = 1000 # burn value
#since the eps.f should satisfy eps.f > (k^2)*eps.p/((k - 1)(k - 1 - k*eps.p))
eps.f =((k^2)*eps.p/((k-1)*(k-1-k*eps.p)))+0.0001

numaccept = 0
flist = rep(0,M)

for (m in 1:M) {
  Y = X

  r = sample(c(2,3,4,5,6,7),2)
  u = r[1]
  v = r[2]

  Y[u] = runif(1,max(0,Y[u] - eps.p),min(Y[u]+eps.p,Y[u]+Y[v]))
  Y[v] = X[u] + X[v] - Y[u] # the pair sum should be the same
```

```
    U  = runif(1) # for accept/reject
    alpha = log.g(Y,N) + log(qq.p(X,u,v,eps.p)) -
      log.g(X,N) - log(qq.p(Y,u,v,eps.p))

    if(log(U) < alpha){
      X = Y
      # now we update the f when p' is accepted
      Z = X

      Z[1] = runif(1,max(-min(Y[2:7])/(1-min(Y[2:7])), X[1]-eps.f),min(X[1]+eps.f, 1))

      W = runif(1) # for accept/reject

      beta = log.g(Z,N) + log(qq.f(X,Z,k,eps.f)) -
        log.g(X,N) - log(qq.f(Z,X,k,eps.f))

      if(log(W)<beta){
        X = Z
        numaccept = numaccept + 1
      }
    }
    flist[m] = X[1]
  }
  estmean = mean(flist[(B+1):M])
  se1 =  sd(flist[(B+1):M]) / sqrt(M-B)
  varfact <- function(xxx) { 2 * sum(acf(xxx, plot=FALSE)$acf) - 1 }
  se2 = se1 * sqrt( varfact(flist[(B+1):M]) )
  ci = c(estmean - 1.96*se2, estmean + 1.96*se2)
  return(list(numaccept/M, estmean, ci, flist, M, B, se2))
}

# test which epsilon gives the best acceptance rate (e.g. away from 0 and 1)
set.seed(9999)
epslist = c(seq(0.001,0.01,0.001), seq(0.011,0.02,0.001))
acclist = meanlist = cilblist = ciublist = selist = rep(0,20)
for (i in 1:20){
  tryCatch({
    result <- MHk6(epslist[i])
    acclist[i]  <- result[[1]]
    meanlist[i] <- result[[2]]
    cilblist[i] <- result[[3]][1]
    ciublist[i] <- result[[3]][2]
    selist[i]   <- result[[7]]
  }, error=function(e){})
}
results <- cbind(epslist, acclist,meanlist,cilblist,ciublist,selist)
results <- as.data.frame(results)
colnames(results) <- c("Epsilon", "Acceptance Rate", "Mean", "Lower bound of CI",
                       "Upper bound of CI", "Standard Error")
results
```

```
##    Epsilon Acceptance Rate       Mean Lower bound of CI Upper bound of CI
## 1    0.001          0.7862 0.20591453        0.17963452        0.23219454
## 2    0.002          0.7905 0.09253044        0.07428352        0.11077736
```

```
## 3    0.003          0.7765 0.05535221        0.04968442        0.06102000
## 4    0.004          0.7275 0.04804688        0.04482444        0.05126933
## 5    0.005          0.6844 0.04675863        0.04383178        0.04968548
## 6    0.006          0.6513 0.04966548        0.04734145        0.05198952
## 7    0.007          0.6091 0.05062757        0.04828006        0.05297508
## 8    0.008          0.5566 0.05210278        0.04984859        0.05435697
## 9    0.009          0.5120 0.04966877        0.04748524        0.05185231
## 10   0.010          0.4821 0.04845844        0.04634206        0.05057483
## 11   0.011          0.4398 0.04882735        0.04683638        0.05081831
## 12   0.012          0.4336 0.04774121        0.04590411        0.04957831
## 13   0.013          0.3903 0.05092078        0.04899039        0.05285118
## 14   0.014          0.3649 0.04904919        0.04721943        0.05087894
## 15   0.015          0.3420 0.04794078        0.04616883        0.04971273
## 16   0.016          0.3143 0.04779441        0.04597184        0.04961698
## 17   0.017          0.2995 0.04909350        0.04734332        0.05084368
## 18   0.018          0.2695 0.04794010        0.04648569        0.04939451
## 19   0.019          0.2598 0.04988987        0.04812537        0.05165437
## 20   0.020          0.2460 0.04951500        0.04805553        0.05097446
##    Standard Error
## 1    0.0134081678
## 2    0.0093096521
## 3    0.0028917292
## 4    0.0016441060
## 5    0.0014932909
## 6    0.0011857324
## 7    0.0011977114
## 8    0.0011500971
## 9    0.0011140497
## 10   0.0010797890
## 11   0.0010157996
## 12   0.0009372959
## 13   0.0009848955
## 14   0.0009335483
## 15   0.0009040558
## 16   0.0009298810
## 17   0.0008929490
## 18   0.0007420458
## 19   0.0009002550
## 20   0.0007446267
```

```r
# when epsilon = 0.01 the algo seems to perform better
set.seed(9999)
result <- MHk6(0.01)
flist <- result[[4]]
M <- result[[5]]
plot(flist[1:M], type = "l")
abline(h=0.05,col="red")
```




```
# mean
estmean = mean(flist[(B+1):M])
estmean
```

```
## [1] 0.04797064
```

```
# 95$ CI
se2 = result[[7]]
ci = c(estmean - 1.96*se2, estmean + 1.96*se2)
ci
```

```
## [1] 0.04583234 0.05010895
```

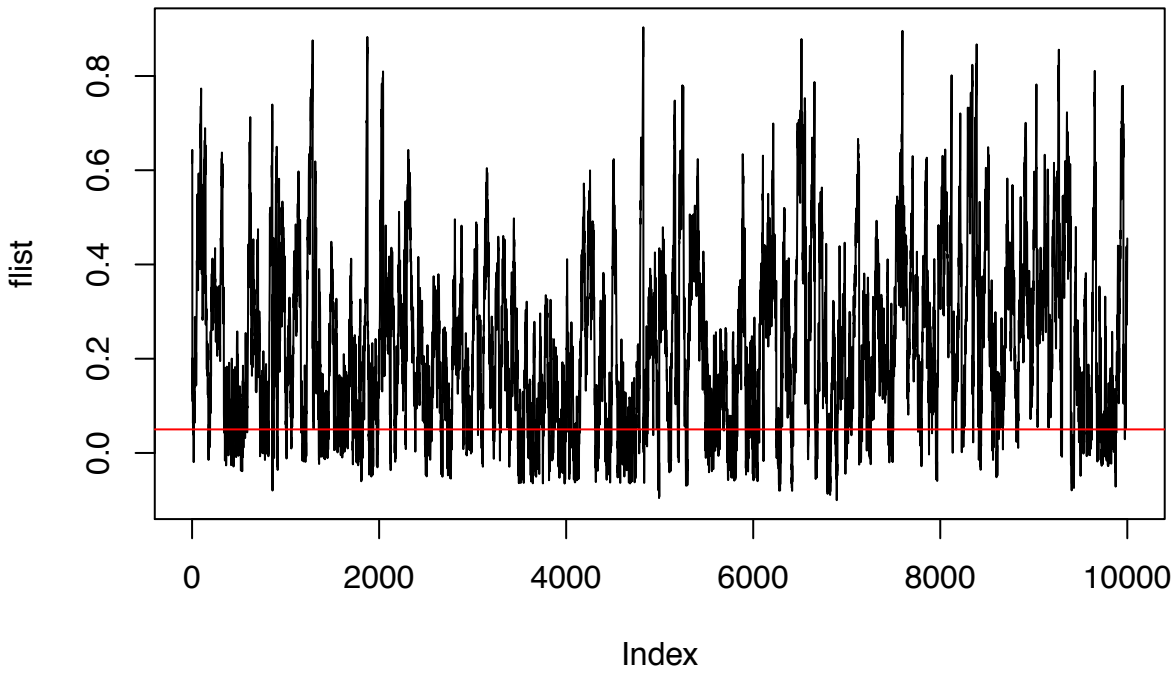
```
######## independence sampling
# Proposal distribution for f: Unif(max(-p_{min}^t)/(1-p_{min}^t,f-\epsilon_f))
qq.f <- function(x,y,k,eps.f){
  1/(min(x[1]+eps.f, 1) - max(-min(y[2:(k+1)])/(1-min(y[2:(k+1)])), x[1]-eps.f))
}
# log of the joint distribution function, the reason we do that is mainly due to
# the number is too small, by doing so, we can enlarge our number and avoid flowing
# for k = 6

eps.p = eps = 0.05
log.g = function(X,N){
  dens = 0
  k = length(X) - 1
  for (i in 1:k) {
    if (i < k){
      for (j in (i+1):k){
        dens = dens + N[i,j]*log(2*X[i+1]*X[j+1]*(1-X[1]))
      }
    }
    dens = dens + N[i,i]*log(X[i+1]*(X[1]+(1-X[1])*X[i+1]))
  }
```

```
  return(dens)
}

# data simulation
n <- 1000 # sample sizes
k <- 6    # number of alleles
f <- 0.05 # true inbreeding coefficient
# allele frequencies when k = 6
k6 <- c(0.02,0.06,0.075,0.085,0.21,0.55)
# sum(k6) # this should be 1
# matrix of genotype frequencies
# pij represents the frequency of AiAj
P <- matrix(nrow = 6, ncol = 6)
for (i in 1:6){
  for (j in 1:6){
    if (i==j){
      P[i,j] <- k6[i]*(f+(1-f)*k6[i])
    }
    else {
      P[i,j] <- 2*k6[i]*k6[j]*(1-f)
    }
  }
}
# sum(P[upper.tri(P, diag = T)]) # this should be 1

# matrix of genotype counts
# Nij represents the number for AiAj
N <- round(P*n)
# sum(N[upper.tri(N,diag = T)])
# keep the total population to 1000
N[6,6] <- 316
# sum(N[upper.tri(N, diag = T)]) # this should be 1000 now

#### the algo
# initial values
X      <- rep(0,7)
X[1]   <- runif(1)
ps     <- runif(6)
X[2:7] <- ps/sum(ps) # make sure sum of p is 1

M = 10000
B = 1000 # burn value
#since the eps.f should satisfy eps.f > (k^2)*eps.p/((k - 1)(k - 1 - k*eps.p))
eps.f =((k^2)*eps.p/((k-1)*(k-1-k*eps.p)))+0.0001

numaccept = 0
flist = rep(0,M)

for (i in 1:M) {
  Y = X
  X[1] = runif(1,max(-min(Y[2:7])/(1-min(Y[2:7])), X[1]-eps.f),min(X[1]+eps.f, 1))

  Z = rep(0,7)
```

```
  Z[1] = runif(1)
  pz = runif(6)
  Z[2:7] = pz/sum(pz)
  W = Z
  Z[1] = runif(1,max(-min(W[2:7])/(1-min(W[2:7])), Z[1]-eps.f),min(Z[1]+eps.f, 1))

  U  = runif(1) # for accept/reject
  alpha = log.g(Z,N) + log(qq.f(X,Y,k,eps.f)) -
    log.g(X,N) - log(qq.f(Z,W,k,eps.p))

  if(log(U) < alpha){
    X = Z
    numaccept = numaccept + 1
  }

  flist[i] = X[1]
}
estmean = mean(flist[(B+1):M])
estmean
```

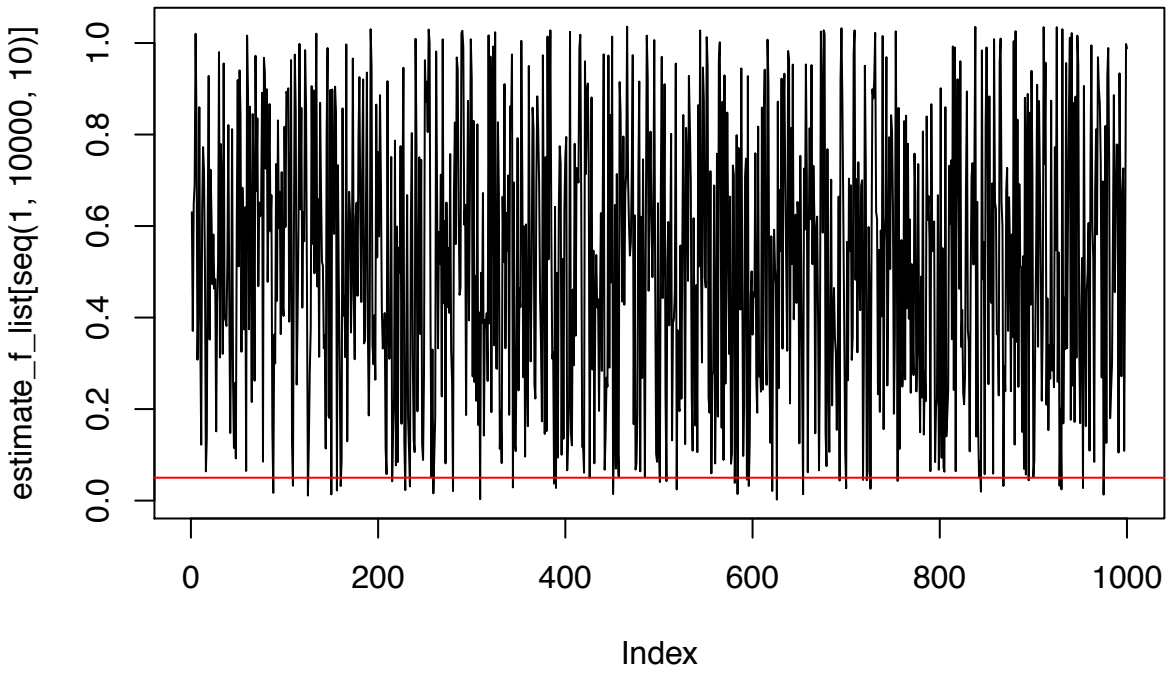
```
## [1] 0.2304864
```

```
se1 =  sd(flist[(B+1):M]) / sqrt(M-B)
varfact <- function(xxx) { 2 * sum(acf(xxx, plot=FALSE)$acf) - 1 }
se2 = se1 * sqrt( varfact(flist[(B+1):M]) )
ci = c(estmean - 1.96*se2, estmean + 1.96*se2)

plot(flist, type = 'l')
abline(h=0.05, col="red")
```




```
# importance sampling:

n = 200
```

```
k = 2
# due to hard to manipulate for the joint density function, we use log to deal with it.
h = function(f){
  return(f)
}

g = function(X){
  X[4]*(log(X[2])+log(X[1]+(1-X[1])*X[2])) +
    X[5]*log(2*X[2]*X[3]*(1-X[1]))+X[6]*(log(X[3])+log(X[1]+(1-X[1])*X[3]))
}

f_a = function(X){
  return(runif(1,max(-min(X[2:3])/(1-min(X[2:3])), X[1]-eps.f),min(X[1]+eps.f, 1)))

}
eps.p = 0.02
eps.f =((k^2)*eps.p/((k-1)*(k-1-k*eps.p)))+0.0001
f = runif(10000)
a = runif(10000)
b = runif(10000)
# in this way we can gurantee that p1+p2 = 1 for sure
p1 = a/(a+b)
p2 = b/(a+b)
p11 = p1*(f+(1-f)*p1)
p22 = p2*(f+(1-f)*p2)
p12 = 2*p1*p2*(1-f)
n12 = round(p12*n)
n11 = round(p11*n)
n22 = round(p22*n)
X  = c(f,p1,p2,n11,n12,n22)# overdispersed value

# for the choice of f, its hard to choose, so we pick the 1

numlist = g(X)+log(f) - log(f_a(X))
denomlist = g(X) - log(f_a(X))

estimate_f_list = exp(numlist)/exp(denomlist)
plot(estimate_f_list[seq(1,10000,10)],type = 'l')
abline(h=0.05, col="red")
```




```
estimate_f = mean(exp(numlist))/mean(exp(denomlist))
estimate_f
```

```
## [1] 0.5205245
```

```
#############MLE

# fisrst we consider the simplest case for k = 2,
# lets compare how the accurate value by giving f = 0.05
# (it should converge to this value) and the mle numerical value
# and the MCMC methods value, and
# then see how the Monte carlo methods works well or not.

# For this one, we need to see that k[1] is denote as p1,
# which is the frequency of allele A1
k2 = c(0.25,0.75) # sum is 1
f = 0.05
n = 200
#pij = 2*pi*pj*(1-f)
#pii = pi*(f+(1-f)*pi)
p1 = k2[1]
p2 = k2[2]
p11 = p1*(f+(1-f)*p1)
p22 = p2*(f+(1-f)*p2)
p12 = 2*p1*p2*(1-f)

# note here since the nij are the number of people with genotype AiAj,
# they must be integer, so we use
# the way to remove the remainder and up by 1,
# and then take average for them to get the estimate.
# we find that, if we do not round them,
# we will get the exact the 0.05 we want, but its just the idea case.
n12 = round(p12*n)+1
```

```r
n11 = round(p11*n)+1
n22 = round(p22*n)+1


festimate = 1-(2*n12*n)/((2*n11+n12)*(n12+2*n22))

festimate1 = festimate

n12 = round(p12*n)
n11 = round(p11*n)
n22 = round(p22*n)

festimate = 1-(2*n12*n)/((2*n11+n12)*(n12+2*n22))

festimate2 = festimate

festimate_final = (festimate1+festimate2)/2
festimate_final
```

```
## [1] 0.05281472
```

```r
# the reason we retest the way that the paper use is that,
# even we use the same formula, the way we deal
# with the remainder will be huge difference for the estimate
# and that's the reason why it can explain when
# we use the MCMC methods, it can still have some standard error.
```

```r
############################## Gibbs Sampler
####### k = 6

###### data simulation
n <- 1000 # sample sizes
k <- 6    # number of alleles
f <- 0.05 # true inbreeding coefficient
k6 <- c(0.02,0.06,0.075,0.085,0.21,0.55)
P <- matrix(nrow = 6, ncol = 6)
for (i in 1:6){
  for (j in 1:6){
    if (i==j){
      P[i,j] <- k6[i]*(f+(1-f)*k6[i])
    }
    else {
      P[i,j] <- 2*k6[i]*k6[j]*(1-f)
    }
  }
}
N <- round(P*n)
N[6,6] <- 316

#### the algo
# initial values
set.seed(9999)
X      <- rep(0,7)
X[1]   <- runif(1)
ps     <- runif(6)
```

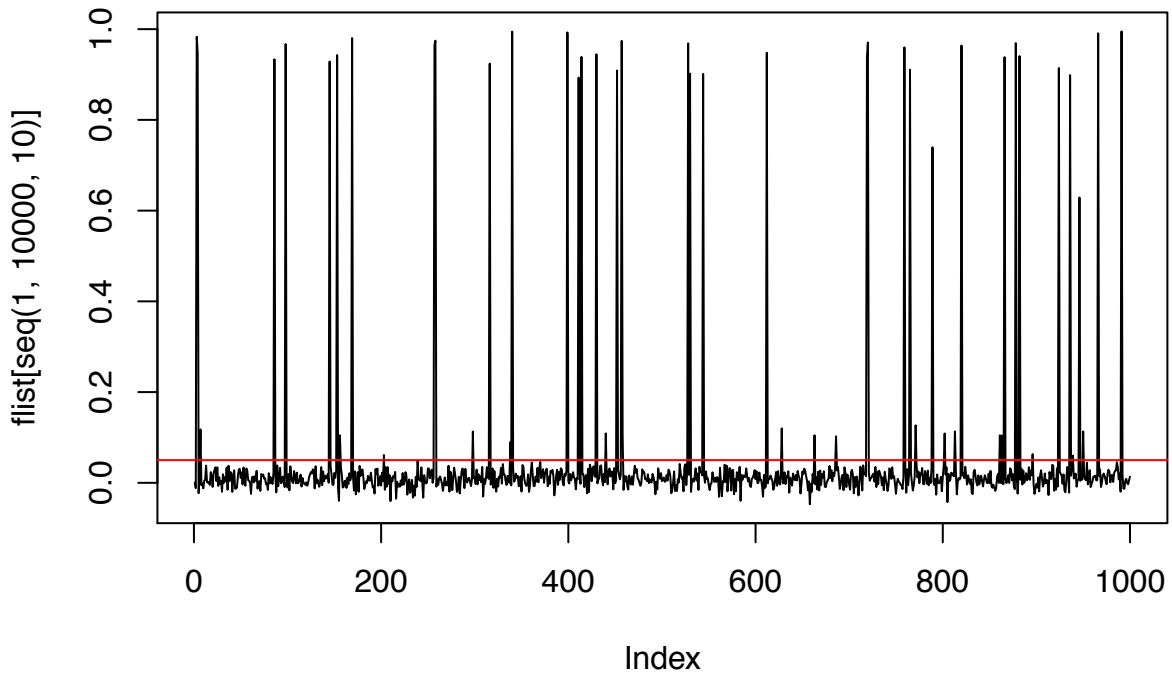
```r
X[2:7] <- ps/sum(ps) # make sure sum of p is 1

M = 10000
B = 1000 # burn value
flist  = rep(0,M)
p1list = rep(0,M)
p3list = rep(0,M)
p5list = rep(0,M)

set.seed(9999)
# systematic-scan
for (m in 1:M) {
  Y = X
  ###### update function for p1,...,p6
  # where we generate samples from their density function
  # For each of the conditional function,
  # firstly integrate over the domain to find a constant,
  # that makes the integration 1.
  # Then integrate to get the CDF
  # generate sample according to CDF by finding roots of
  # u = cdf where u ~ unif[0,1]
  pdfp1 = function(x){
    if(x < 0 || x > 1)
      return(0)
    else
      return(x*(X[1]+(1-X[1])*x)^N[1,1]*(x*(1-X[1]))^N[1,2])
  }
  c1 <- integrate(pdfp1,0,1)[[1]]
  if (c1>0){
    cdfp1 = function(x,u){
      return(integrate(pdfp1,0,x)[[1]]/c1 - u);
    }
    Y[2] <- uniroot(cdfp1, c(0,1), tol = 0.0001, u = runif(1))$root
  } else Y[2] <- 1e-20


  pdfp2 = function(x){
    if(x < 0 || x > 1)
      return(0)
    else
      return(x*(X[1]+(1-X[1])*x)^N[2,2]*(x*(1-X[1]))^(N[1,2]+N[2,3]))
  }
  c2 <- integrate(pdfp2,0,1)[[1]]
  if (c2>0){
    cdfp2 = function(x,u){
      return(integrate(pdfp2,0,x)[[1]]/c2 - u);
    }
    Y[3] <- uniroot(cdfp2, c(0,1), tol = 0.0001, u = runif(1))$root
  } else Y[3] <- 1e-20


  pdfp3 = function(x){
    if(x < 0 || x > 1)
```

```
    return(0)
  else
    return(x*(X[1]+(1-X[1])*x)^N[3,3]*(x*(1-X[1]))^(N[3,4]+N[2,3]))
}
c3 <- integrate(pdfp3,0,1)[[1]]
if (c3>0){
  cdfp3 = function(x,u){
    return(integrate(pdfp3,0,x)[[1]]/c3 - u);
  }
  Y[4] <- uniroot(cdfp3, c(0,1), tol = 0.0001, u = runif(1))$root
} else Y[4] <- 1e-20


pdfp4 = function(x){
  if(x < 0 || x > 1)
    return(0)
  else
    return(x*(X[1]+(1-X[1])*x)^N[4,4]*(x*(1-X[1]))^(N[3,4]+N[4,5]))
}
c4 <- integrate(pdfp4,0,1)[[1]]
if (c4>0){
  cdfp4 = function(x,u){
    return(integrate(pdfp4,0,x)[[1]]/c4 - u);
  }
  Y[5] <- uniroot(cdfp4, c(0,1), tol = 0.0001, u = runif(1))$root
} else Y[5] <- 1e-20


pdfp5 = function(x){
  if(x < 0 || x > 1)
    return(0)
  else
    return(x*(X[1]+(1-X[1])*x)^N[5,5]*(x*(1-X[1]))^(N[5,6]+N[4,5]))
}
c5 <- integrate(pdfp5,0,1)[[1]]
if (c5>0){
  cdfp5 = function(x,u){
    return(integrate(pdfp5,0,x)[[1]]/c5 - u);
  }
  Y[6] <- uniroot(cdfp5, c(0,1), tol = 0.0001, u = runif(1))$root
} else Y[6] <- 1e-20


pdfp6 = function(x){
  if(x < 0 || x > 1)
    return(0)
  else
    return(x*(X[1]+(1-X[1])*x)^N[6,6]*(x*(1-X[1]))^N[5,6])
}
c6 <- integrate(pdfp6,0,1)[[1]]
if (c6>0){
  cdfp6 = function(x,u){
    return(integrate(pdfp6,0,x)[[1]]/c6 - u);
```

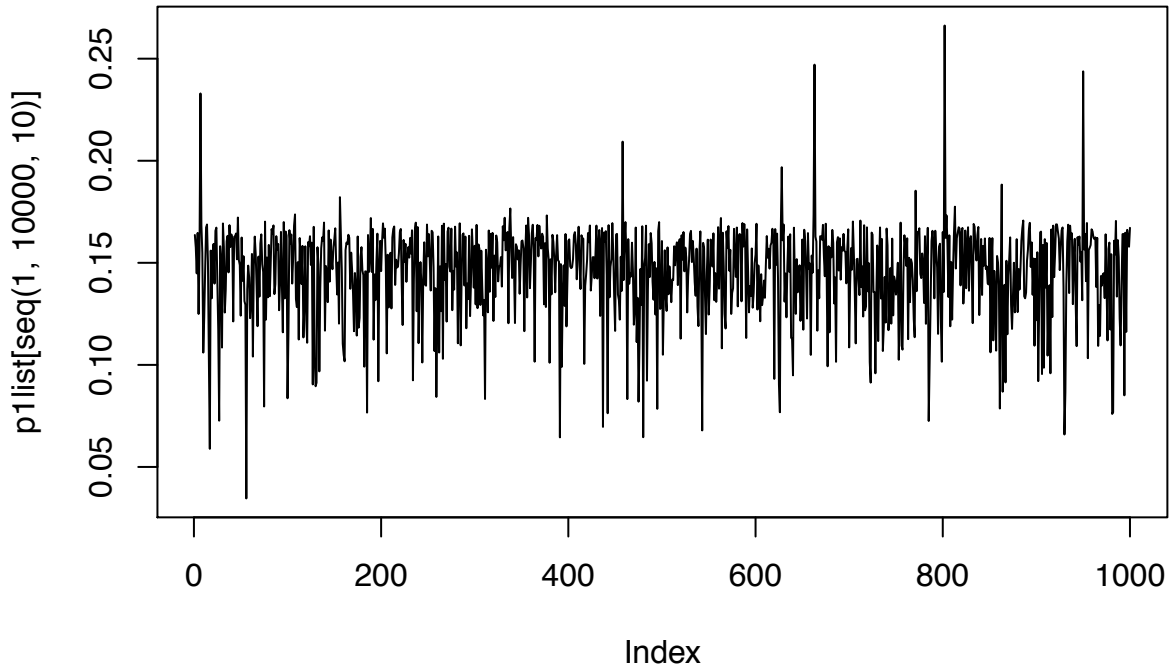
```
    }
    Y[7] <- uniroot(cdfp6, c(0,1), tol = 0.0001, u = runif(1))$root
  } else Y[7] <- 1e-20

  # like we did for the initial value,
  # make sure the sum is 1
  psum <- sum(Y[2:7])
  Y[2:7] <- Y[2:7]/psum
  X = Y


  # now we update the f
  Z = X
  # update function for f, similarly to pi
  pdff <- function(x){
    dens <- 1
    if(x < -min(X[2:7])/(1-min(X[2:7])) || x > 1)
      return(0)
    else
      for (i in 1:6){
        if (i < k){
          for (j in (i+1):6) {
            dens <- dens*(x+(1-x)*X[i+1])^N[i,i]*(1-x)^N[i,j]
          }
        }
      }
    dens
  }
  cf <- integrate(pdff,-min(X[2:7])/(1-min(X[2:7])),1)[[1]]
  cdff = function(x,u){
    return(integrate(pdff,-min(X[2:7])/(1-min(X[2:7])),x)[[1]]/cf - u);
  }
  Z[1] = uniroot(cdff, c(-min(X[2:7])/(1-min(X[2:7])),1), tol = 0.0001, u = runif(1))$root
  X = Z

  p1list[m] = X[2]
  p3list[m] = X[4]
  p5list[m] = X[6]
  flist[m]  = X[1]
}
(estmean = mean(flist[(B+1):M]))
```

```
## [1] 0.03558
```

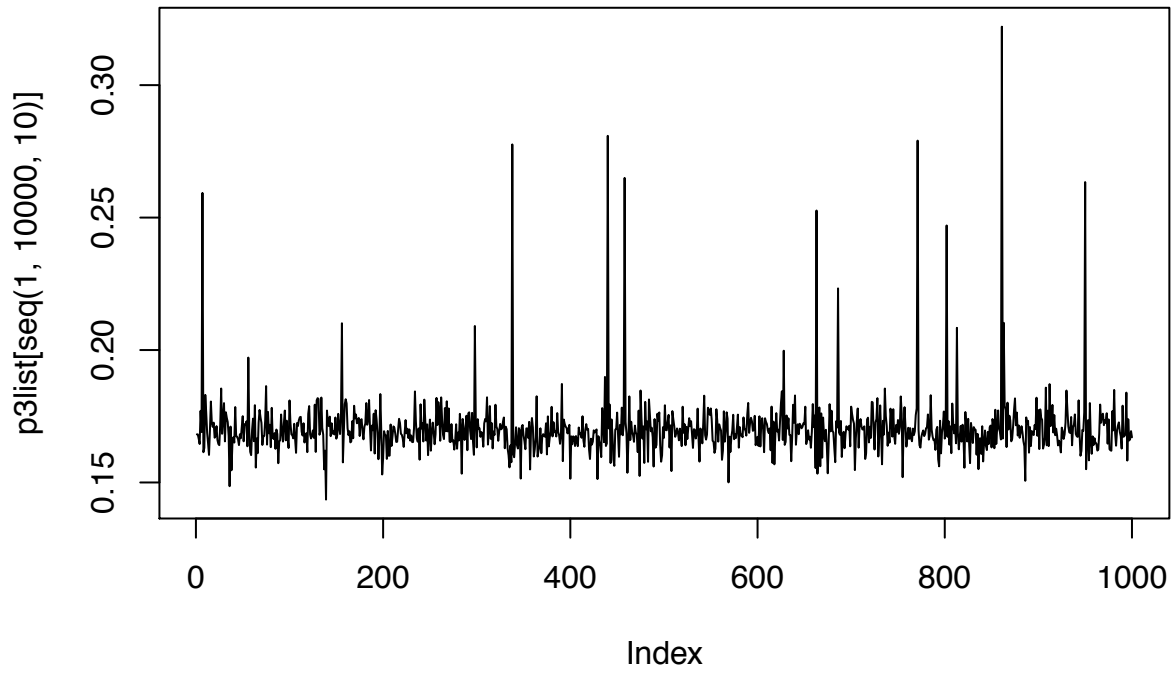
```
plot(flist[seq(1,10000,10)],type="l")
abline(h=0.05,col="red")
```




```
plot(p1list[seq(1,10000,10)],type="l")
```




```
plot(p3list[seq(1,10000,10)],type="l")
```




```
plot(p5list[seq(1,10000,10)],type="l")
```