Assignment 4

STAT34800

Seung Chul Lee

Problem A

1. Derive EM Algorithm

Let Z be the latent variable such that $Z_{ik} = 1$ if ith observation came from component k. Then, $\mathbb{P}(Z_{ik} = 1) = \pi_k$. Then, the complete data likelihood is

$$p(X, Z \mid S, P) = \prod_{i=1}^{I} \prod_{k=1}^{2} (\pi_k Mult(X_i; s_i, P_k))^{Z_i k}$$

$$\Rightarrow \log p(X, Z \mid S, P) = \sum_{i=1}^{I} \sum_{k=1}^{2} Z_{ik} \bigg(\log \pi_k + \log \frac{s_i!}{X_{i1}! \dots X_{iJ}!} \prod_{j=1}^{J} P_{kj}^{X_{ij}} \bigg)$$

Then, the E step is

$$\begin{aligned} Q(\pi, P) &= \mathbb{E}_{Z|X,\pi,P}[\log p(X, Z \mid \pi, s, P)] \\ &= \sum_{i=1}^{I} \sum_{k=1}^{2} \mathbb{E}[Z_{ik}] \bigg(\log \pi_k + \log \frac{s_i!}{X_{i1}! \dots X_{iJ}!} + \sum_{j=1}^{J} X_{ij} \log P_{kj} \bigg) \\ &= \sum_{i=1}^{I} \sum_{k=1}^{2} w_{ik} \bigg(\log \pi_k + \log \frac{s_i!}{X_{i1}! \dots X_{iJ}!} + \sum_{j=1}^{J} X_{ij} \log P_{kj} \bigg) \end{aligned}$$

where w_{ik} is the responsibilities such that $\sum_k w_{ik} = 1$. In order to initialize the algorithm, we should set some initial values $\pi_k^{(0)}$ and $P_k^{(0)}$.

Now, for the M step.

$w_{ik}^{(0)}$: By definition, $w_{ik} = \mathbb{E}_{Z|X,\pi^{(0)},P^{(0)}}[Z_{ik}]$. Therefore, using law of total probability and our estimated values for π and P,

$$w_{ik}^{(0)} = \frac{\pi_k^{(0)} Mult(X_i; s_i, P_k^{(0)})}{\sum_{k'} \pi_{k'}^{(0)} Mult(X_i; s_i, P_{k'}^{(0)})}$$

π_k: Since we have the constraint that $\sum_k \pi_k = 1$, using Lagrangian gives

$$\mathcal{L}(\pi) = Q(\pi, P) + \lambda \bigg(\sum_{k}^{2} \pi_k - 1 \bigg)$$

where λ is the Lagrange multiplier.

$$\Rightarrow \frac{\partial \mathcal{L}}{\partial \pi_k} = \sum_{i=1}^{I} \frac{w_{ik}^{(0)}}{\pi_k} - \lambda$$

Setting the derivative to zero yields

$$\pi_k^{(1)} = \frac{\sum_{i=1}^{I} w_{ik}^{(0)}}{-\lambda}$$

Using our constraint,

$$-\lambda = \sum_k \sum_i w_{ik}^{(0)} = \sum_i 1 = I$$

$$\therefore \pi_k^{(1)} = \frac{1}{I} \sum_{i=1}^{I} w_{ik}^{(0)}$$

P_{kj}**:** Again, we have the constraint that $\sum_j P_{kj} = 1$. Using Lagrangian once more gives

$$\mathcal{L}(P_k) = Q(\pi, P) + \lambda\left(\sum_{j=1}^{J} P_{kj} - 1\right)$$

where λ is the Lagrange multiplier.

$$\Rightarrow \frac{\partial \mathcal{L}}{\partial P_{kj}} = \sum_{i=1}^{I} w_{ik}^{(0)} \frac{X_{ij}}{P_{kj}} + \lambda = \frac{1}{P_{kj}} \sum_{i=1}^{I} w_{ik}^{(0)} X_{ij} + \lambda$$

Setting the derivative to zero yields

$$P_{kj}^{(1)} = \frac{\sum_{i=1}^{I} w_{ik}^{(0)} X_{ij}}{-\lambda}$$

Using our constraint,

$$-\lambda = \sum_{j=1}^{J} \sum_{i=1}^{I} w_{ik}^{(0)} X_{ij} = \sum_{i=1}^{I} w_{ik}^{(0)} s_i$$

$$\therefore P_{kj}^{(1)} = \frac{\sum_{i=1}^{I} w_{ik}^{(0)} X_{ij}}{\sum_{i=1}^{I} w_{ik}^{(0)} s_i}$$

Then, repeat the two-step process (E and M) using $\pi_k^{(1)}$ and $P_k^{(1)}$ as the new initial values until convergence.

2. Implement EM Algorithm

```
# Import data
cell = read.csv("/Users/eelrice/Desktop/School/3. 2022S/34800/Assignments/HW4/cell_data.csv",
                sep = ",")
```

I first define the function `log_lik()` that takes in the values of the multinomial log likelihood values and the current π values and returns the incomplete log-likelihood[1] to determine convergence as it was done in the vignette. I then define the function `multinom_mixture_em()` that takes in the data `x` ($I \times J$ matrix), initial values of P `p_init` ($2 \times J$ matrix with P_k as each column), and initial values of π `pi_init` (a vector of length 2, i.e., (π_1, π_2)) to run the EM algorithm. The function returns a list of the log likelihoods in each iteration, π vector at convergence, P ($2 \times J$ matrix) at convergence, W matrix of the responsibilities at convergence ($I \times 2$ matrix). Note that I formulate my function assuming two components to avoid `for` loops and reduce the computational weight of my function. However, it can easily be extended to mixtures with more than 2 components by replacing lines for calculating `L` and `p_next` with appropriate loops.

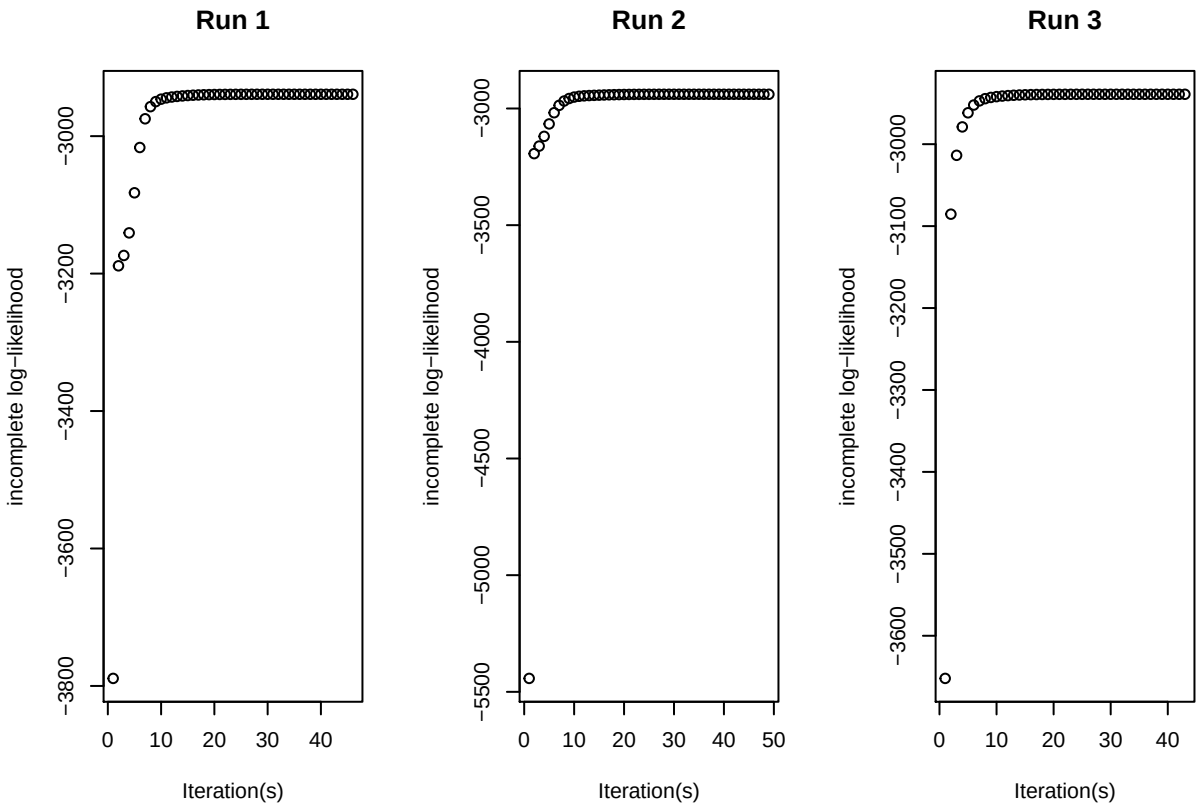
[1]
$$\ell(\pi, P; X) = \sum_i \log \sum_k \pi_k Mult(X_i; s_i, P_k)$$

```r
# Define function for calculating incomplete log likelihood
log_lik <- function(L, pi_curr){
  # assuming L is log likelihoods from multinomial
  L[, 1] = L[, 1] + log(pi_curr[1])
  L[, 2] = L[, 2] + log(pi_curr[2])
  return(sum(apply(L, 1, logsum)))
}

# Define function to run the EM algorithm
multinom_mixture_em = function(x, p_init, pi_init){
  # Initialize values
  p_curr = p_init
  pi_curr = pi_init
  s = rowSums(x) # get s_i
  n = dim(x)[1]

  # Initialize log likelihood using initial values
  L = matrix(nrow = nrow(x), ncol = length(pi_curr))
  L[, 1] = apply(x, 1, FUN = dmultinom, prob = p_curr[1, ], log = T)
  L[, 2] = apply(x, 1, FUN = dmultinom, prob = p_curr[2, ], log = T)

  log_liks = c()
  ll = log_lik(L, pi_curr)
  log_liks = c(log_liks, ll)
  delta = 1 # initialize convergence criterion

  while(delta > 1e-5){
    # E step
    w_curr = L
    for(i in 1:ncol(L)){ # multiply likelihood by component weights
      w_curr[, i] = w_curr[, i] + log(pi_curr[i])
    }
    w_curr = exp(w_curr - apply(w_curr, 1, logsum)) # responsibilities

    # M Step
    pi_next = colSums(w_curr) / sum(w_curr) # pi
    p_next = matrix(0, nrow = 2, ncol = ncol(x)) # p
    for(j in 1:ncol(x)){
      p_next[1, j] = sum(w_curr[, 1] * x[, j])
      p_next[2, j] = sum(w_curr[, 2] * x[, j])
    }
    p_next[1, ] = p_next[1, ] / sum(w_curr[, 1] * s)
    p_next[2, ] = p_next[2, ] / sum(w_curr[, 2] * s)

    # Multinomial log likelihood with new p
    L[, 1] = apply(x, 1, FUN = dmultinom, prob = p_next[1, ], log = T)
    L[, 2] = apply(x, 1, FUN = dmultinom, prob = p_next[2, ], log = T)

    ll = log_lik(L, pi_next) # calculate log likelihood with new pi
    log_liks = c(log_liks, ll) # accumulate
    # update convergence criterion
    delta = log_liks[length(log_liks)] - log_liks[length(log_liks)-1]
```

```
    # Pass on to iterate the next step
    pi_curr = pi_next
    p_curr = p_next
  }
  return(list(loglike = log_liks, pi = pi_curr, p = p_curr, w = w_curr))
}
```

I first check whether the function performs well with a simulated data. I generate a mixture of two multinomial distributions with component weights 0.7 and 0.3. I derive the parameters p by sampling from a Dirichlet distribution. I initialize the algorithm with $\pi^{(0)} = (0.5, 0.5)$ and some arbitrary choices for P of my liking and run the algorithm three times to overcome any issues with the initializations.

```
# Generate parameters
pis = c(0.7, 0.3)
set.seed(123)
p1 = rdirichlet(1, rep(1.5, 4))
p2 = rdirichlet(1, rep(1.5, 4))

# Simulate a multinomial mixture
x = matrix(nrow = 1000, ncol = 4)
truth = rep(0, 1000)
set.seed(1234)
for(i in 1:1000){
  if(runif(1) < 0.7){
    x[i, ] = rmultinom(1, 4, prob = p1)
    truth[i] = 1
  }else {
    x[i, ] = rmultinom(1, 4, prob = p2)
    truth[i] = 2
  }
}

# Initialize p and pi
pi_init_sim = c(0.5, 0.5)
p_init_sim1 = rbind(c(0.25, 0.25, 0.25, 0.25), c(0.3, 0.2, 0.1, 0.4))
p_init_sim2 = rbind(c(0.1, 0.1, 0.5, 0.3), c(0.7, 0.1, 0.1, 0.1))
p_init_sim3 = rbind(c(0.2, 0.2, 0.3, 0.3), c(0.05, 0.5, 0.3, 0.15))

# Run the EM algorithm
sim_run1 = multinom_mixture_em(x, p_init_sim1, pi_init_sim)
sim_run2 = multinom_mixture_em(x, p_init_sim2, pi_init_sim)
sim_run3 = multinom_mixture_em(x, p_init_sim3, pi_init_sim)

# Plot log likelihood
par(mfrow = c(1, 3))
plot(sim_run1$loglike, main = "Run 1", xlab = "Iteration(s)",
     ylab = "incomplete log-likelihood")
plot(sim_run2$loglike, main = "Run 2", xlab = "Iteration(s)",
     ylab = "incomplete log-likelihood")
plot(sim_run3$loglike, main = "Run 3", xlab = "Iteration(s)",
     ylab = "incomplete log-likelihood")
```



I first plot the incomplete log likelihood for each iteration from the three runs on the simulated data. I confirm that the likelihood increases monotonically. As we have learned in class, EM algorithm is fast to find a reasonable value but takes quite long to converge afterwards. In fact, all three runs resulted in more than 40 iterations but only the first few points are noticeably different. This suggests that my defined function is properly implemented.

```
# Component weights
pi_hat = as.data.frame(rbind(sim_run1$pi, sim_run2$pi, sim_run3$pi, pis))
colnames(pi_hat) = c("pi1", "pi2")
rownames(pi_hat) = c("run1", "run2", "run3", "Truth")
pi_hat
```

	pi1	pi2
run1	0.2839501	0.7160499
run2	0.2839482	0.7160518
run3	0.2839431	0.7160569
Truth	0.7000000	0.3000000

I find that the component weights are very close to the truth. Because the EM algorithm is agnostic about the ordering of the components, it can yield π in the reverse order.

```
# Estimated p
p1_hat = as.data.frame(rbind(sim_run1$p[2, ], sim_run2$p[2, ],
                             sim_run3$p[2, ], p1))
p2_hat = as.data.frame(rbind(sim_run1$p[1, ], sim_run2$p[1, ],
                             sim_run3$p[1, ], p2))
```

```
colnames(p1_hat) = c("p11", "p12", "p13", "p14")
rownames(p1_hat) = c("run1", "run2", "run3", "Truth")
colnames(p2_hat) = c("p21", "p22", "p23", "p24")
rownames(p2_hat) = c("run1", "run2", "run3", "Truth")
p1_hat
```

	p11	p12	p13	p14
run1	0.1223854	0.5969400	0.0094079	0.2712668
run2	0.1223856	0.5969393	0.0094082	0.2712670
run3	0.1223861	0.5969373	0.0094090	0.2712676
Truth	0.1227633	0.6029315	0.0057095	0.2685957

The estimated values for P_1 are not exact but very accurate nonetheless.

```
p2_hat
```

	p21	p22	p23	p24
run1	0.2143544	0.1076288	0.3275698	0.3504469
run2	0.2143546	0.1076273	0.3275712	0.3504469
run3	0.2143549	0.1076234	0.3275749	0.3504468
Truth	0.2367727	0.1038945	0.3075487	0.3517841

Similarly, the estimated values for P_2 are also reasonably close. Again, this confirms that the algorithm is properly implemented.

Finally, I check the false classification rate. I first define the function `false_rate()` that calculates the proportion of falsely classified observations given predicted and ground truth values. Because the ordering was switched for all three runs, I do a little modification to compare the correct values.

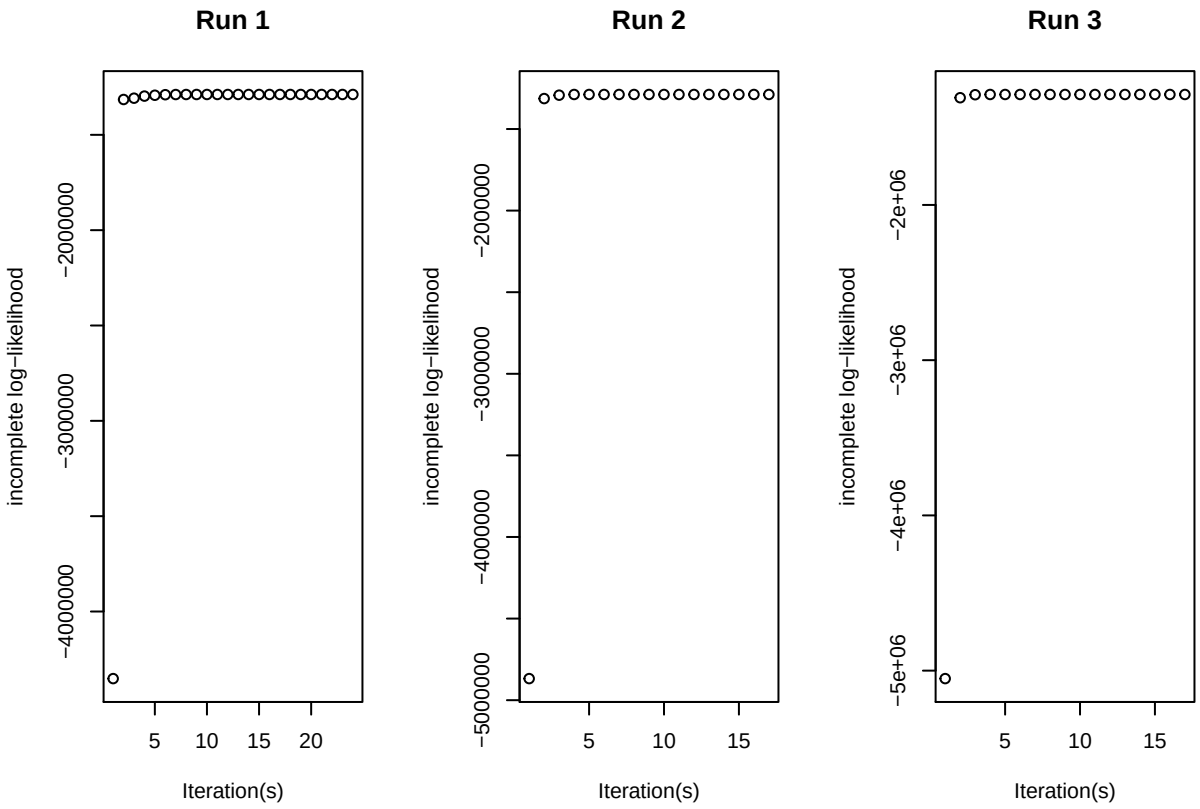
```
# Check classification
false_rate = function(x, ref){
  return(sum(x != ref) / length(x))
}

sim_class1 = as.numeric(apply(sim_run1$w, 1, which.max) == 2)
sim_class2 = as.numeric(apply(sim_run2$w, 1, which.max) == 2)
sim_class3 = as.numeric(apply(sim_run3$w, 1, which.max) == 2)
ref = as.numeric(truth == 1)

c(false_rate(sim_class1, ref), false_rate(sim_class2, ref),
  false_rate(sim_class3, ref))
```

```
## [1] 0.069 0.069 0.069
```

All three runs yielded false classification rate of approximately 7%, which seems very good considering the speed and simplicity.

I now run the EM algorithm on the cell data. I use three different initial values of P, all sampled from a Dirichlet distribution with all $\alpha_i = 1.5$. I first plot the log likelihood as with the simulated data.

```r
# Run the EM algorithm on cell data
set.seed(12345)
p_init1 = rbind(rdirichlet(1, rep(1.5, 500)), rdirichlet(1, rep(1.5, 500)))
p_init2 = rbind(rdirichlet(1, rep(1.5, 500)), rdirichlet(1, rep(1.5, 500)))
p_init3 = rbind(rdirichlet(1, rep(1.5, 500)), rdirichlet(1, rep(1.5, 500)))
pi_init = c(0.5, 0.5)

cell_run1 = multinom_mixture_em(cell[, -1], p_init1, pi_init)
cell_run2 = multinom_mixture_em(cell[, -1], p_init2, pi_init)
cell_run3 = multinom_mixture_em(cell[, -1], p_init3, pi_init)

par(mfrow = c(1, 3))
plot(cell_run1$loglike, main = "Run 1", xlab = "Iteration(s)",
     ylab = "incomplete log-likelihood")
plot(cell_run2$loglike, main = "Run 2", xlab = "Iteration(s)",
     ylab = "incomplete log-likelihood")
plot(cell_run3$loglike, main = "Run 3", xlab = "Iteration(s)",
     ylab = "incomplete log-likelihood")
```



Once again, I confirm that the log likelihood increases monotonically with each iterations, and that the algorithm finds a reasonable value quite quickly but takes longer to converge.

```r
pi_hat2 = as.data.frame(rbind(cell_run1$pi, cell_run2$pi, cell_run3$pi))
colnames(pi_hat2) = c("pi1", "pi2")
rownames(pi_hat2) = c("run1", "run2", "run3")
pi_hat2
```

	pi1	pi2
run1	0.7854148	0.2145852
run2	0.2145881	0.7854119
run3	0.2145879	0.7854121

Note that, depending on the initialization, the π_k can come in a reverse order even between the runs. I find that the performance is much worse compared to the actual component weights of 0.5 and 0.5.

```
p1_hat2 = as.data.frame(rbind(cell_run1$p[1, ], cell_run2$p[2, ],
                              cell_run3$p[2, ]))
p2_hat2 = as.data.frame(rbind(cell_run1$p[2, ], cell_run2$p[1, ],
                              cell_run3$p[1, ]))
rownames(p1_hat2) = c("run1", "run2", "run3")
rownames(p2_hat2) = c("run1", "run2", "run3")
p1_hat2[, 1:5]
```

	V1	V2	V3	V4	V5
run1	0.0248434	0.0246994	0.0209945	0.0201253	0.0183499
run2	0.0248433	0.0246994	0.0209945	0.0201253	0.0183499
run3	0.0248433	0.0246994	0.0209945	0.0201253	0.0183499

Looking at just the first 5 P_{1j}, I find that all three runs have estimated roughly the same probabilities.

```
p2_hat2[, 1:5]
```

	V1	V2	V3	V4	V5
run1	0.0386145	0.0203483	0.0176473	0.0179558	0.0172083
run2	0.0386145	0.0203483	0.0176473	0.0179558	0.0172084
run3	0.0386145	0.0203483	0.0176473	0.0179558	0.0172084

The same is true for P_{2j}. I now check the classification rates.

```
cell_class1 = apply(cell_run1$w, 1, which.max)
cell_class2 = apply(cell_run2$w, 1, which.max)
cell_class3 = apply(cell_run3$w, 1, which.max)

ref1 = c(rep(1, 1000), rep(2, 1000))
ref2 = c(rep(2, 1000), rep(1, 1000))

c(false_rate(cell_class1, ref2), false_rate(cell_class2, ref1),
  false_rate(cell_class3, ref1))
```

```
## [1] 0.298 0.298 0.298
```

I find that the false classification rate is equal for all three runs at approximately 30%. This is surprisingly high and I try to further investigate what is going on in the data.

```r
# Cytotoxin
c(false_rate(cell_class1[1:1000], ref2[1:1000]),
  false_rate(cell_class2[1:1000], ref1[1:1000]),
  false_rate(cell_class3[1:1000], ref1[1:1000]))
```

```
## [1] 0.584 0.584 0.584
```

```r
# Naive
false_rate(cell_class1[1001:2000], ref2[1001:2000]); false_rate(cell_class2[1001:2000], ref1[1001:2000])
```

```
## [1] 0.012
```

```
## [1] 0.012
```

```
## [1] 0.012
```

I find the interesting result that the algorithm has a much lower misclassification rate for naive cells, with more than 99% accuracy. However, for cytotoxin cells, the algorithm incorrectly classified over 50% of the observations. My guess is that the underlying mixture densities are structured in such a way that the density of cytotoxin cells are more likely to be mistaken as the density of naive cells. Below is a graphical illustration of my guess.

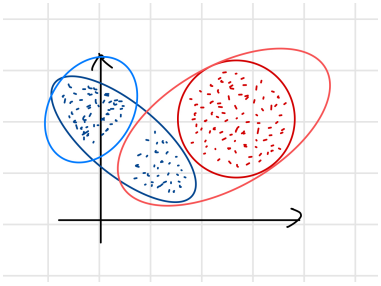


Figure 1: Graphical Illustration

The red circle and the dark blue oval are the actual densities, whereas the light blue and orange ovals are the EM estimates. This is a very rough sketch, but I feel this is an idea of what is going on under the hood.

Problem B

1. easyMCMC

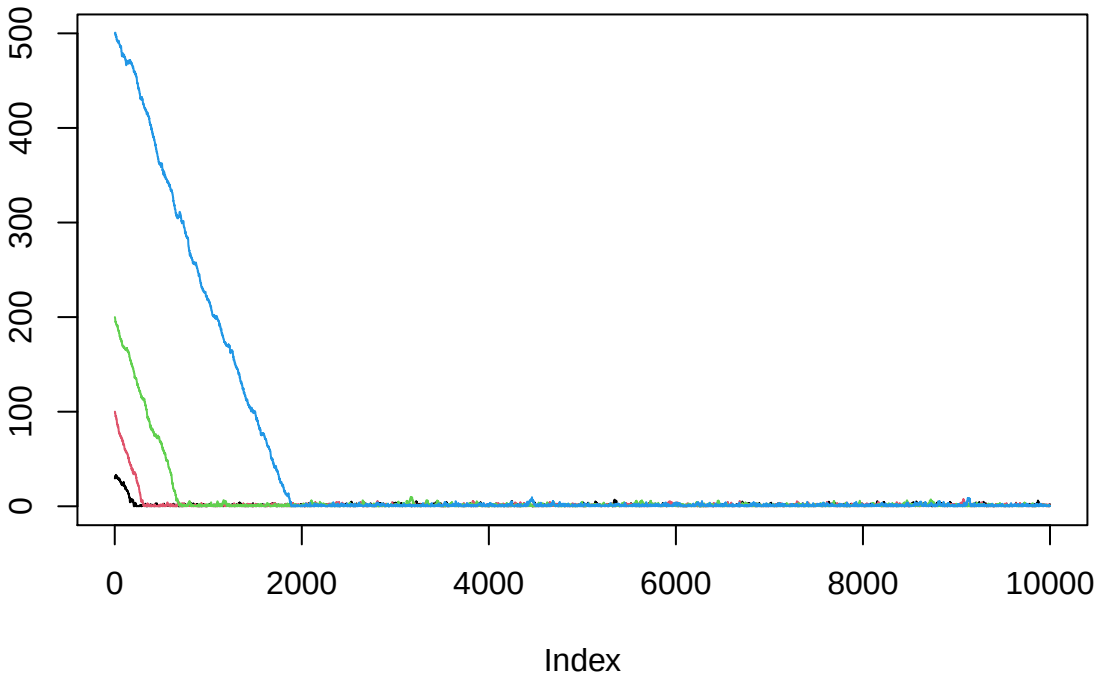
I first define the function `easyMCMC` as in the vignette.

```r
# Define easyMCMC
easyMCMC = function(log_target, niter, startval, proposalsd){
  x = rep(0,niter)
  x[1] = startval
  for(i in 2:niter){
    currentx = x[i-1]
    proposedx = rnorm(1,mean=currentx,sd=proposalsd)
    A = exp(log_target(proposedx) - log_target(currentx))
```

```
    if(runif(1)<A){
      x[i] = proposedx # accept move with probabily min(1,A)
    } else {
      x[i] = currentx # otherwise "reject" move, and stay where we are
    }
  }
  return(x)
}

# Define log target function
log_exp_target = function(x){
  return(dexp(x,rate=1, log=TRUE))
}
```

a. Starting Points I try four different starting points, namely 30, 100, 200, and 500. These would undoubtedly be unusual under an exponential density. I also increase the number of iterations to 10000 for a better illustration.

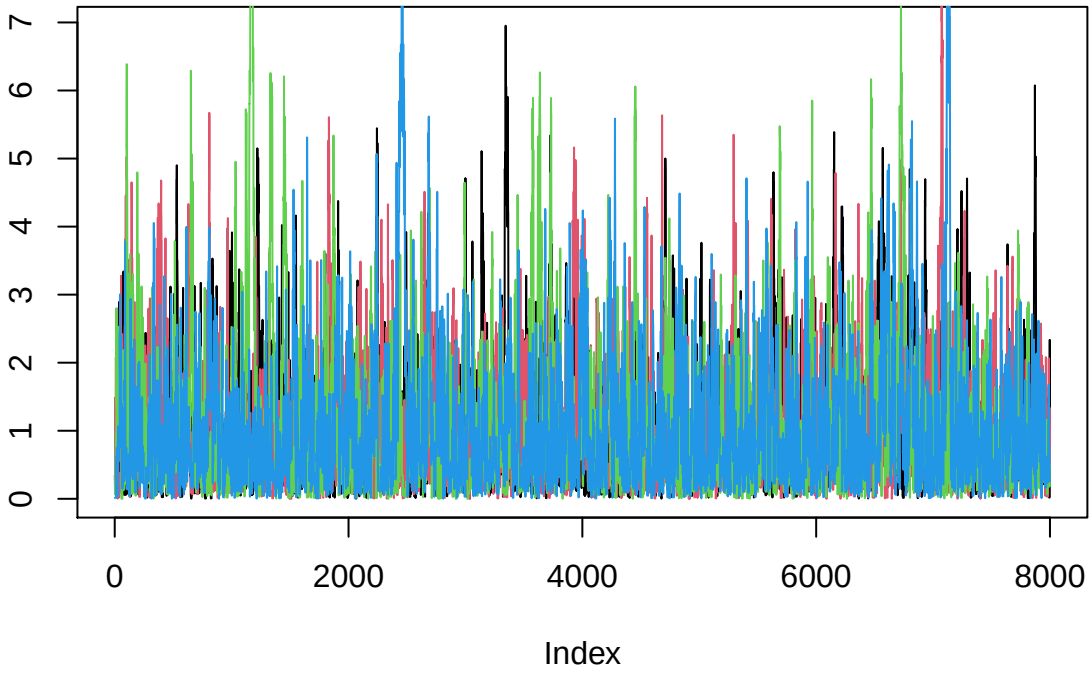
```
set.seed(123) # for reproducibility
z_st1 = easyMCMC(log_exp_target, 10000, 30, 1)
z_st2 = easyMCMC(log_exp_target, 10000, 100, 1)
z_st3 = easyMCMC(log_exp_target, 10000, 200, 1)
z_st4 = easyMCMC(log_exp_target, 10000, 500, 1)

plot(z_st1, type = "l", ylim = c(0, 500), ylab = "")
lines(z_st2, col = 2)
lines(z_st3, col = 3)
lines(z_st4, col = 4)
```



Note that, because I start at extremely large values, the Metropolis-Hastings algorithm correctly accepts all the proposals to move downward and the chain stays at the more usual values of the density. I try removing

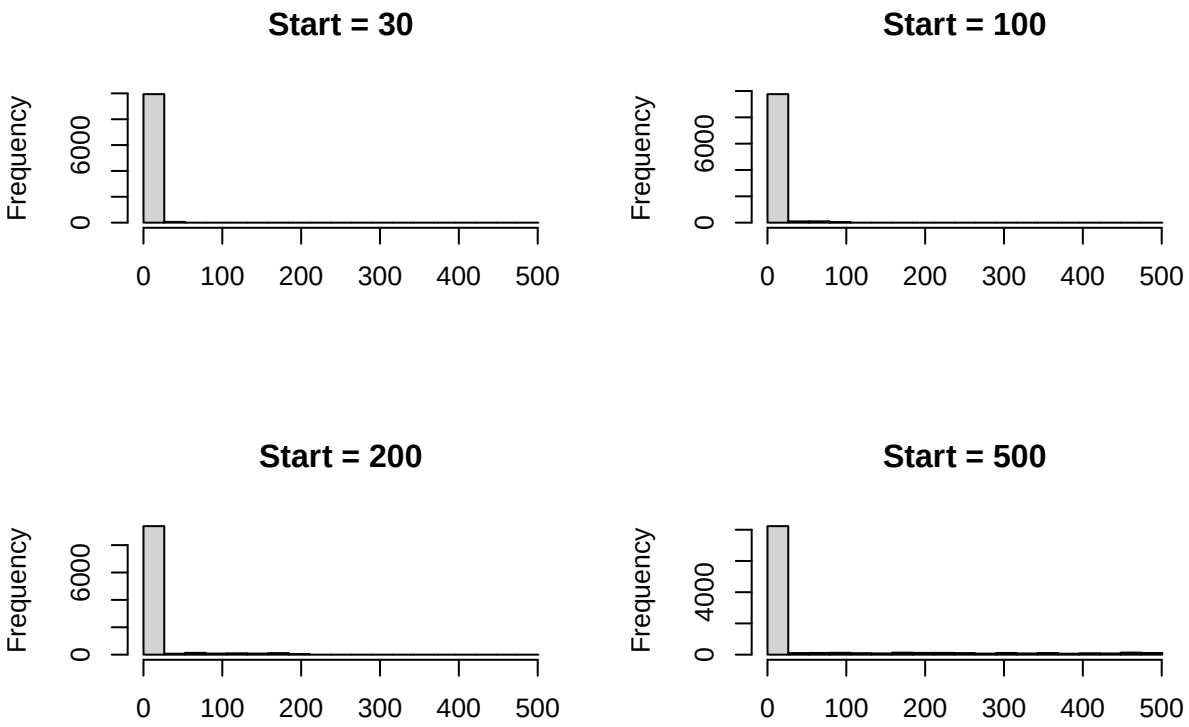
the first 2000 observations to see whether the lines look more reasonable with burn-in.

```
plot(z_st1[-c(1:2000)], type = "l", ylab = "")
lines(z_st2[-c(1:2000)], col = 2)
lines(z_st3[-c(1:2000)], col = 3)
lines(z_st4[-c(1:2000)], col = 4)
```



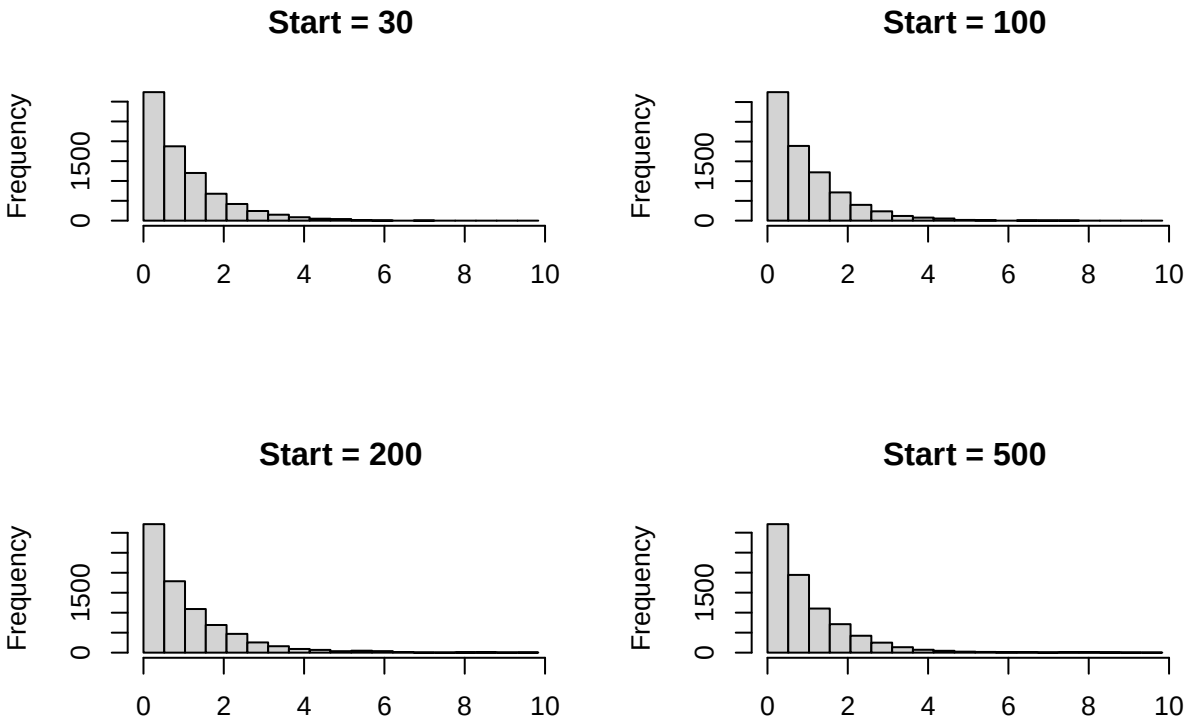
The lines look much more similar as the Markov chain seems to have converged to its stationary distribution by the 2000th iteration. I now plot the histograms for each case.

```
par(mfrow = c(2, 2))
maxz = max(c(z_st1, z_st2, z_st3, z_st4))
hist(z_st1, breaks = seq(0, maxz, length = 20), main = "Start = 30",
     xlab = "")
hist(z_st2, breaks = seq(0, maxz, length = 20), main = "Start = 100",
     xlab = "")
hist(z_st3, breaks = seq(0, maxz, length = 20), main = "Start = 200",
     xlab = "")
hist(z_st4, breaks = seq(0, maxz, length = 20), main = "Start = 500",
     xlab = "")
```



Without burn-in, the densities are too skewed to be seen as an exponential density. I plot the histograms again with burn-in.

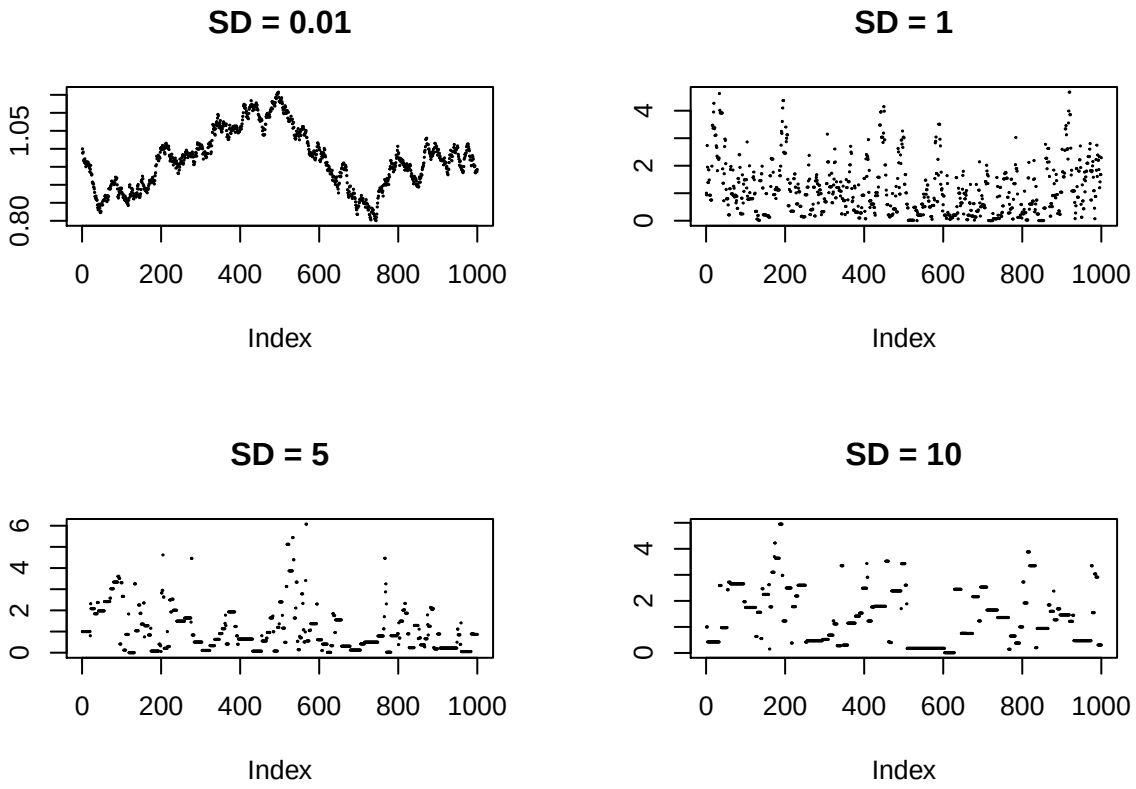
```
par(mfrow = c(2, 2))
maxz = max(c(z_st1[-c(1:2000)], z_st2[-c(1:2000)], z_st3[-c(1:2000)],
             z_st4[-c(1:2000)]))
hist(z_st1[-c(1:2000)], breaks = seq(0, maxz, length = 20),
     main = "Start = 30", xlab = "")
hist(z_st2[-c(1:2000)], breaks = seq(0, maxz, length = 20),
      main = "Start = 100", xlab = "")
hist(z_st3[-c(1:2000)], breaks = seq(0, maxz, length = 20),
      main = "Start = 200", xlab = "")
hist(z_st4[-c(1:2000)], breaks = seq(0, maxz, length = 20),
      main = "Start = 500", xlab = "")
```



With burn-in, the histograms appear to be a much more reasonable representation of the exponential density.

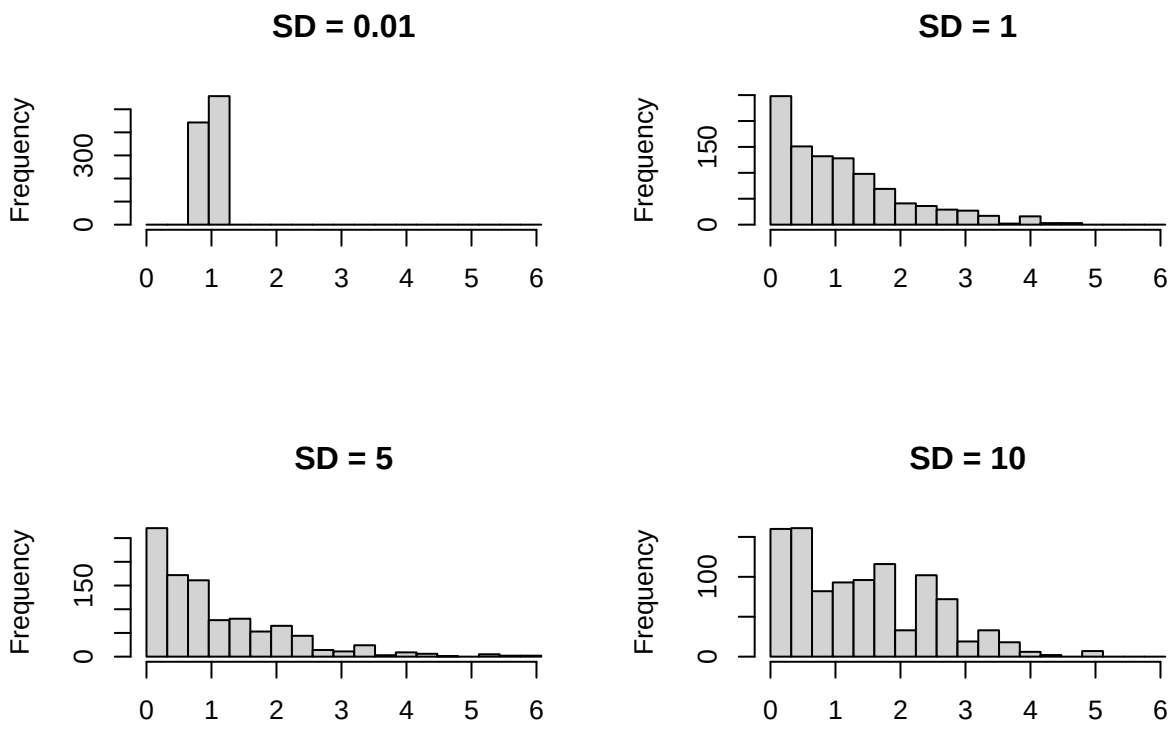
b. Proposal Standard Deviation I try four different values of the proposal standard deviation, namely 0.01, 1, 5, and 10. I return to 1000 iterations, as was done in the vignette, and a starting point of 1 for better comparison.

```
set.seed(1234)
z_psd1 = easyMCMC(log_exp_target, 1000, 1, 0.01)
z_psd2 = easyMCMC(log_exp_target, 1000, 1, 1)
z_psd3 = easyMCMC(log_exp_target, 1000, 1, 5)
z_psd4 = easyMCMC(log_exp_target, 1000, 1, 10)

par(mfrow = c(2, 2))
plot(z_psd1, cex = 0.1, ylab = "", main = "SD = 0.01")
plot(z_psd2, cex = 0.1, ylab = "", main = "SD = 1")
plot(z_psd3, cex = 0.1, ylab = "", main = "SD = 5")
plot(z_psd4, cex = 0.1, ylab = "", main = "SD = 10")
```



Consistent with what we have covered during class, having a proposal standard deviation of 0.01 proposes changes that are too small and thus does not converge to the stationary distribution fast enough. In fact, the range of the y axis in the graph suggests that it has not explored much of the parameter space. On the contrary, having proposal standard deviation of 10 has the problem that it rejects most of the changes throughout the algorithm, but once it does, there is a massive jump. A somewhat similar, but not as extreme, problem exists with the case when proposal standard deviation is 5. I now try plotting the histograms.

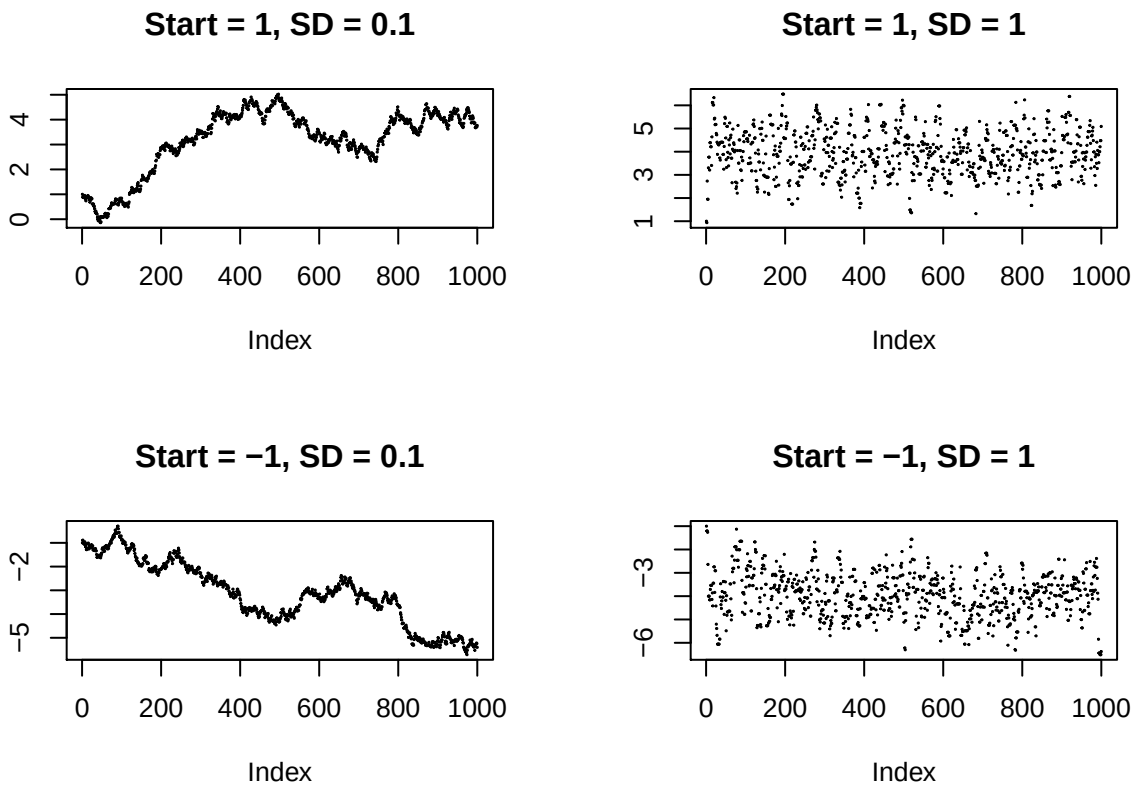
```
par(mfrow = c(2, 2))
maxz = max(c(z_psd1, z_psd2, z_psd3, z_psd4))
hist(z_psd1, breaks = seq(0, maxz, length = 20), main = "SD = 0.01",
     xlab = "")
hist(z_psd2, breaks = seq(0, maxz, length = 20), main = "SD = 1", xlab = "")
hist(z_psd3, breaks = seq(0, maxz, length = 20), main = "SD = 5", xlab = "")
hist(z_psd4, breaks = seq(0, maxz, length = 20), main = "SD = 10",
     xlab = "")
```



Again, the histograms for extreme standard deviations are not accurate representations of the exponential density. The first with 0.01 stayed mostly around the initial starting point of 1. The third and fourth histograms seem to have more extreme values, due to the jumps in the paths.

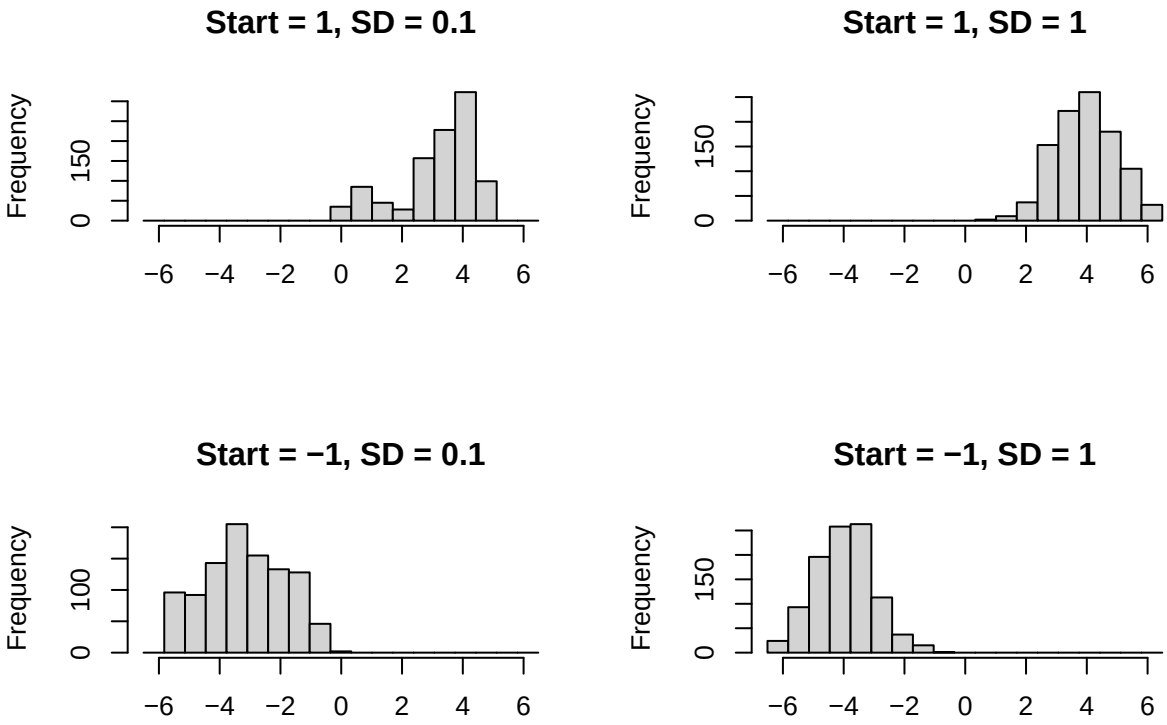
c. Different Target I change the target to the following given function and run the `easyMCMC`. I try using proposal standard deviations of 0.1 and 1 as suggested in the problem. Pretending to be agnostic about the distribution, I choose 1 and -1 as the starting points.

```
log_target_bimodal = function(x){
  log(0.8 * dnorm(x, -4, 1) + 0.2 * dnorm(x, 4, 1))
}

set.seed(1234)
z_new1 = easyMCMC(log_target_bimodal, 1000, 1, 0.1)
z_new2 = easyMCMC(log_target_bimodal, 1000, 1, 1)
z_new3 = easyMCMC(log_target_bimodal, 1000, -1, 0.1)
z_new4 = easyMCMC(log_target_bimodal, 1000, -1, 1)

par(mfrow = c(2, 2))
plot(z_new1, cex = 0.1, main = "Start = 1, SD = 0.1", ylab = "")
plot(z_new2, cex = 0.1, main = "Start = 1, SD = 1", ylab = "")
plot(z_new3, cex = 0.1, main = "Start = -1, SD = 0.1", ylab = "")
plot(z_new4, cex = 0.1, main = "Start = -1, SD = 1", ylab = "")
```



Plotting the paths shows that the distribution should look roughly concentrated around 4 if I start at 1 and concentrated around -4 if I start at -1. Again, with proposal standard deviation of 0.1, the proposed movements seem too small to attain a convergence to the stationary distribution.

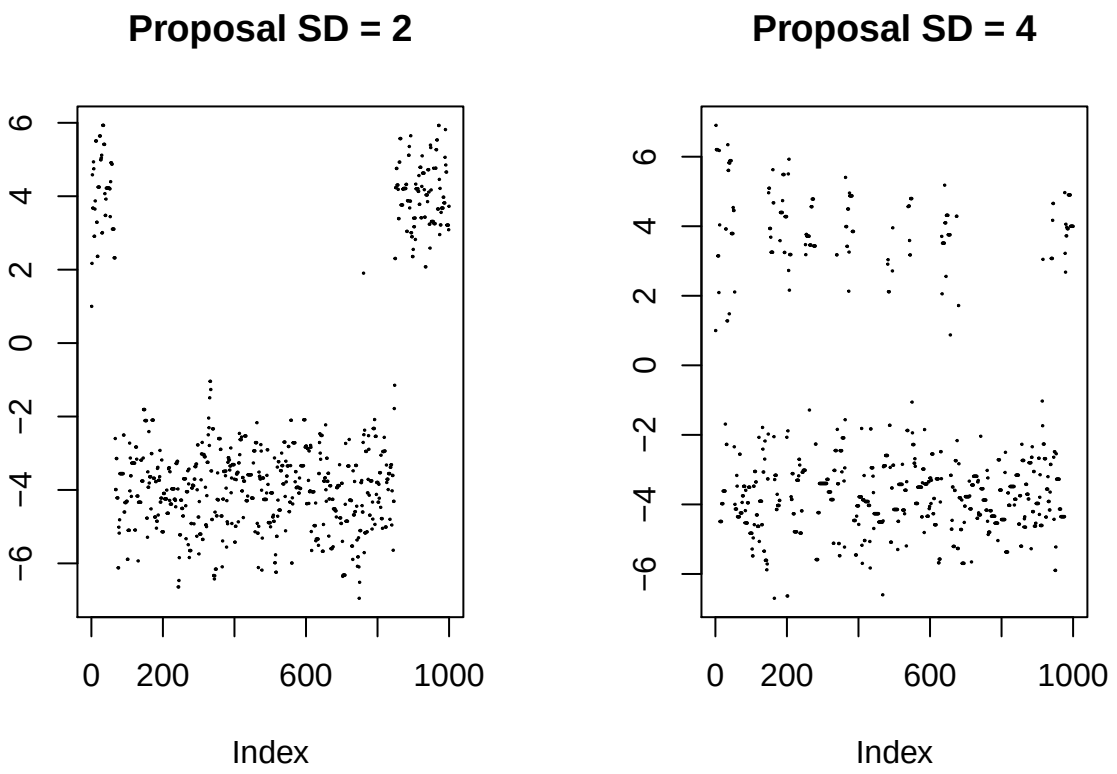
```
par(mfrow = c(2, 2))
maxz = max(c(z_new1, z_new2, z_new3, z_new4))
minz = min(c(z_new1, z_new2, z_new3, z_new4))
hist(z_new1, main = "Start = 1, SD = 0.1", xlab = "",
     breaks = seq(minz, maxz, length = 20))
hist(z_new2, main = "Start = 1, SD = 1", xlab = "",
     breaks = seq(minz, maxz, length = 20))
hist(z_new3, main = "Start = -1, SD = 0.1", xlab = "",
     breaks = seq(minz, maxz, length = 20))
hist(z_new4, main = "Start = -1, SD = 1", xlab = "",
     breaks = seq(minz, maxz, length = 20))
```



The histograms confirm this as well. I observe a clear divide in the histograms with starting point of 1 and those with starting point of -1.

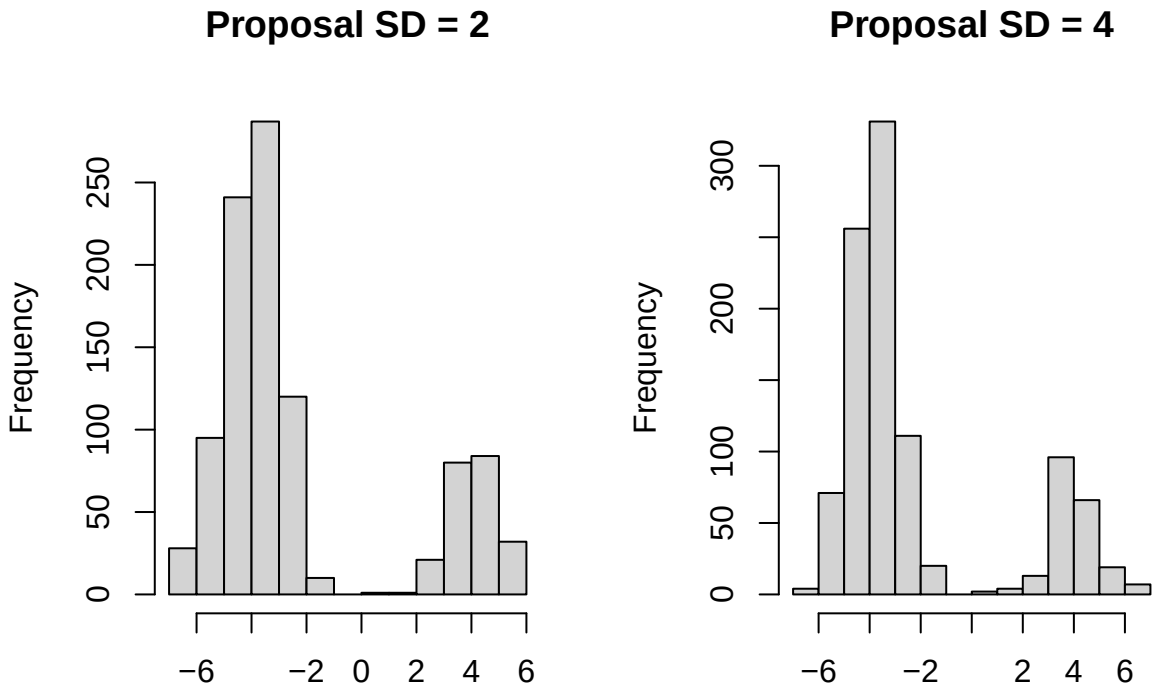
I now try using a larger proposal standard deviation, namely 2 and 4.

```
set.seed(12345)
z_new5 = easyMCMC(log_target_bimodal, 1000, 1, 2)
z_new6 = easyMCMC(log_target_bimodal, 1000, 1, 4)

par(mfrow = c(1,2))
plot(z_new5, cex = 0.1, main = "Proposal SD = 2", ylab = "")
plot(z_new6, cex = 0.1, main = "Proposal SD = 4", ylab = "")
```



Note that the increased standard deviation allows jumps between the modes and will probably capture the bimodal nature of the density.

```
par(mfrow = c(1, 2))
hist(z_new5, main = "Proposal SD = 2", xlab = "")
hist(z_new6, main = "Proposal SD = 4", xlab = "")
```



The histograms confirm the bimodal structure. It also seems to accurately capture the mixture proportions

of 0.8 and 0.2, yielding a higher mode at -4.

In conclusion, it is critical to thoroughly check the resulting density by trying different values of starting points and proposal standard deviations. In practice, we will not know the true density and thus cannot confirm whether a certain starting point is appropriate for the stationary distribution, whether the proposal standard deviations are of the adequate size, or whether all modes have been found.

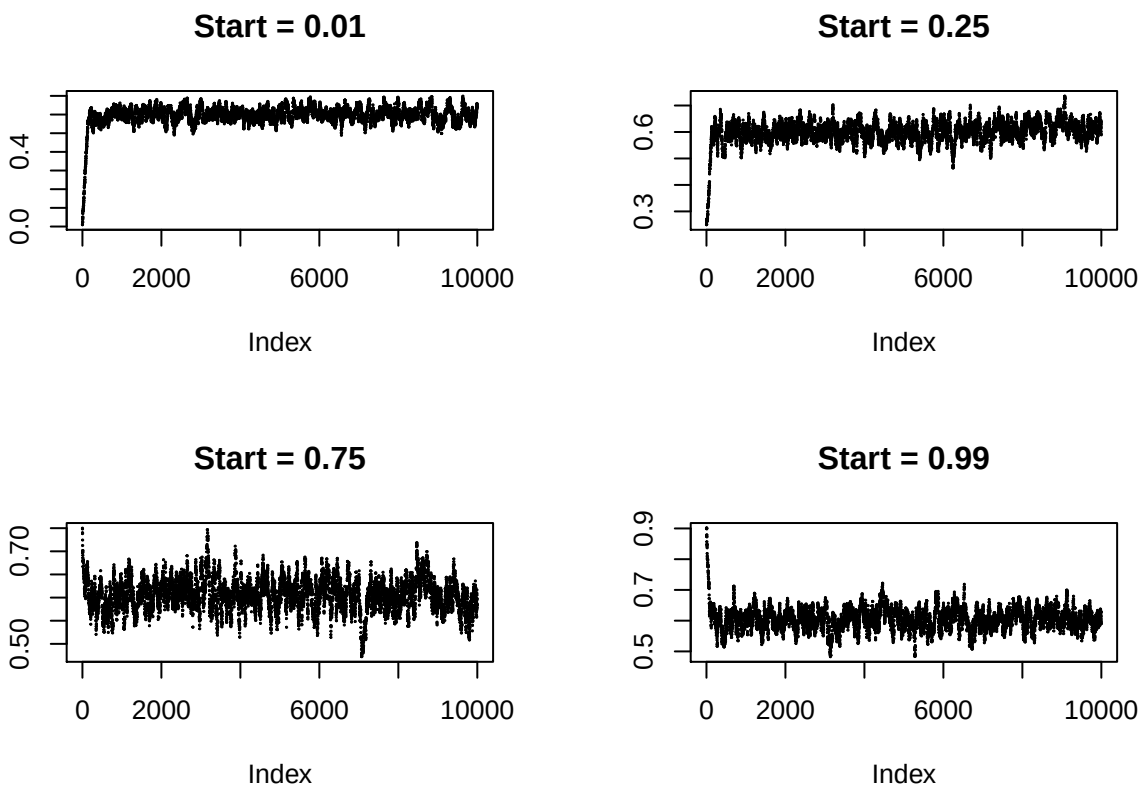
2. Investigate

I first copy over the functions defined in the vignette.

```
log_prior = function(p){
  if((p<0) || (p>1)){  # || here means "or"
    return(-Inf)}
  else{
    return(0)}
}

log_likelihood = function(p, nAA, nAa, naa){
  return((2*nAA)*log(p)  + nAa * log (2*p*(1-p)) + (2*naa)*log(1-p))
}

psampler = function(nAA, nAa, naa, niter, pstartval, pproposalsd){
  p = rep(0,niter)
  p[1] = pstartval
  for(i in 2:niter){
    currentp = p[i-1]
    newp = currentp + rnorm(1,0,pproposalsd)
    A = exp(log_prior(newp) + log_likelihood(newp,nAA,nAa,naa) -
              log_prior(currentp) - log_likelihood(currentp,nAA,nAa,naa))
    if(runif(1) < A){
      p[i] = newp # accept move with probability min(1,A)
    } else {
      p[i] = currentp # otherwise "reject" move, and stay where we are
    }
  }
  return(p)
}
```

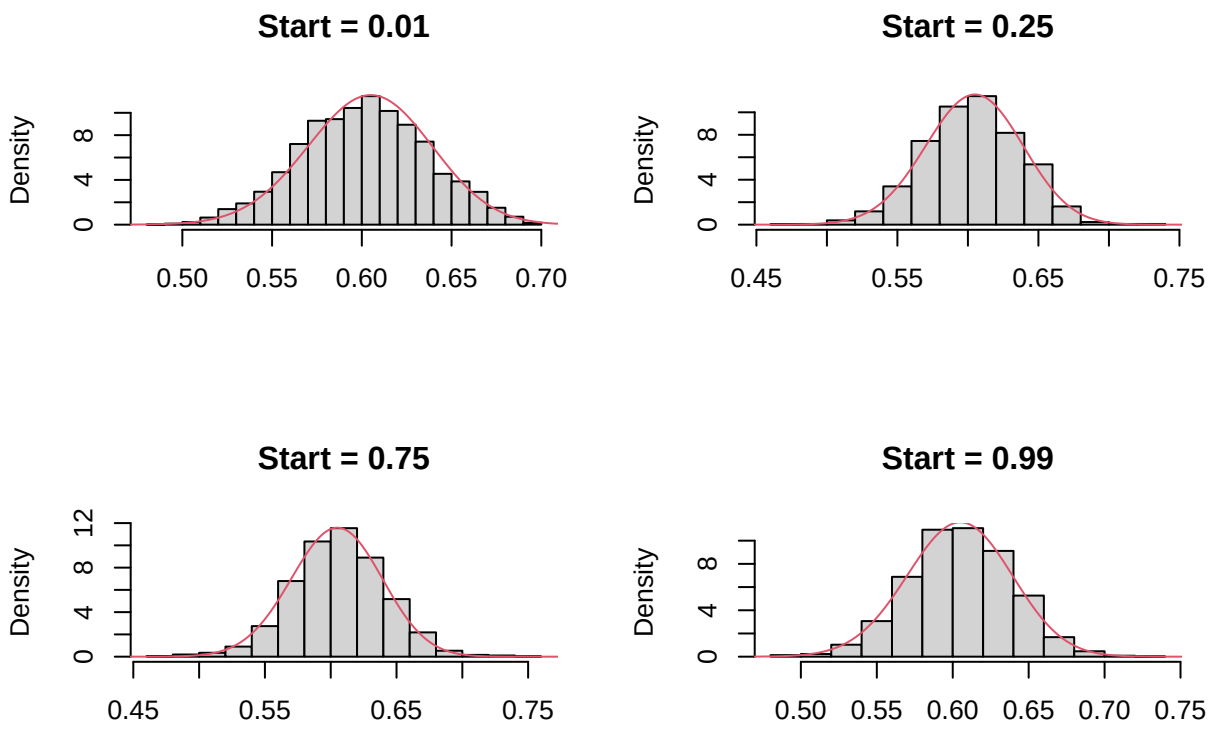
Similarly to the previous part, I first experiment with the starting points. I try namely 0.01, 0.25, 0.75, and 0.9, to see what *extreme* values can cause. Despite the wording, I know *ex ante* that $p \in (0, 1)$, so the values are not as extreme as in the previous problem.

```
set.seed(123) # for reproducibility
z1 = psampler(50, 21, 29, 10000, 0.01, 0.01)
z2 = psampler(50, 21, 29, 10000, 0.25, 0.01)
z3 = psampler(50, 21, 29, 10000, 0.75, 0.01)
z4 = psampler(50, 21, 29, 10000, 0.9, 0.01)

par(mfrow = c(2, 2))
plot(z1, cex = 0.1, main = "Start = 0.01", ylab = "")
plot(z2, cex = 0.1, main = "Start = 0.25", ylab = "")
plot(z3, cex = 0.1, main = "Start = 0.75", ylab = "")
plot(z4, cex = 0.1, main = "Start = 0.99", ylab = "")
```



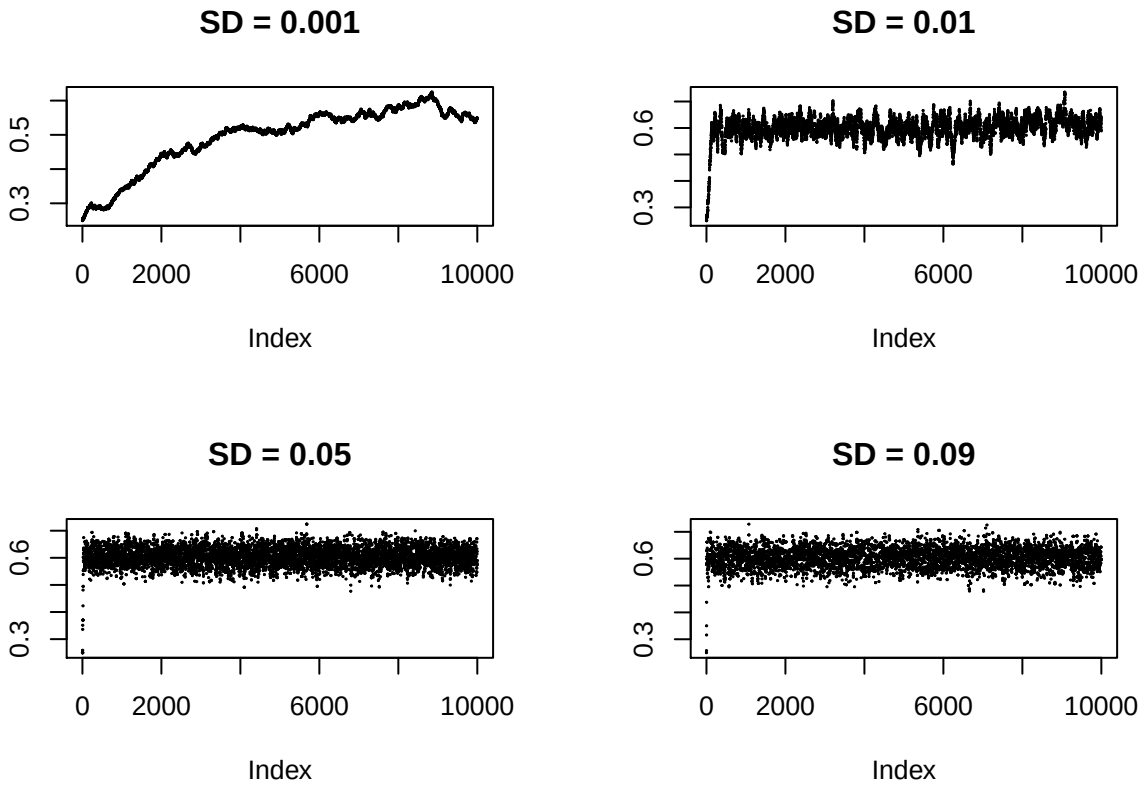
As with extreme starting values in the previous problem, I observe an initial ascent or descent for 0.01 and 0.99. The case with 0.25 also has a steep ascent at the beginning. This also calls for the use of burn-in. I try omitting the first 1000 observations this time. I also add the line for the theoretical density $Beta(121 + 1, 79 + 1)$ for comparison.

```
x = seq(0, 1, length = 1000)

par(mfrow = c(2, 2))
hist(z1[-c(1:1000)], main = "Start = 0.01", xlab = "", prob = T)
lines(x, dbeta(x, 122, 80), col = 2)
hist(z2[-c(1:1000)], main = "Start = 0.25", xlab = "", prob = T)
lines(x, dbeta(x, 122, 80), col = 2)
hist(z3[-c(1:1000)], main = "Start = 0.75", xlab = "", prob = T)
lines(x, dbeta(x, 122, 80), col = 2)
hist(z4[-c(1:1000)], main = "Start = 0.99", xlab = "", prob = T)
lines(x, dbeta(x, 122, 80), col = 2)
```



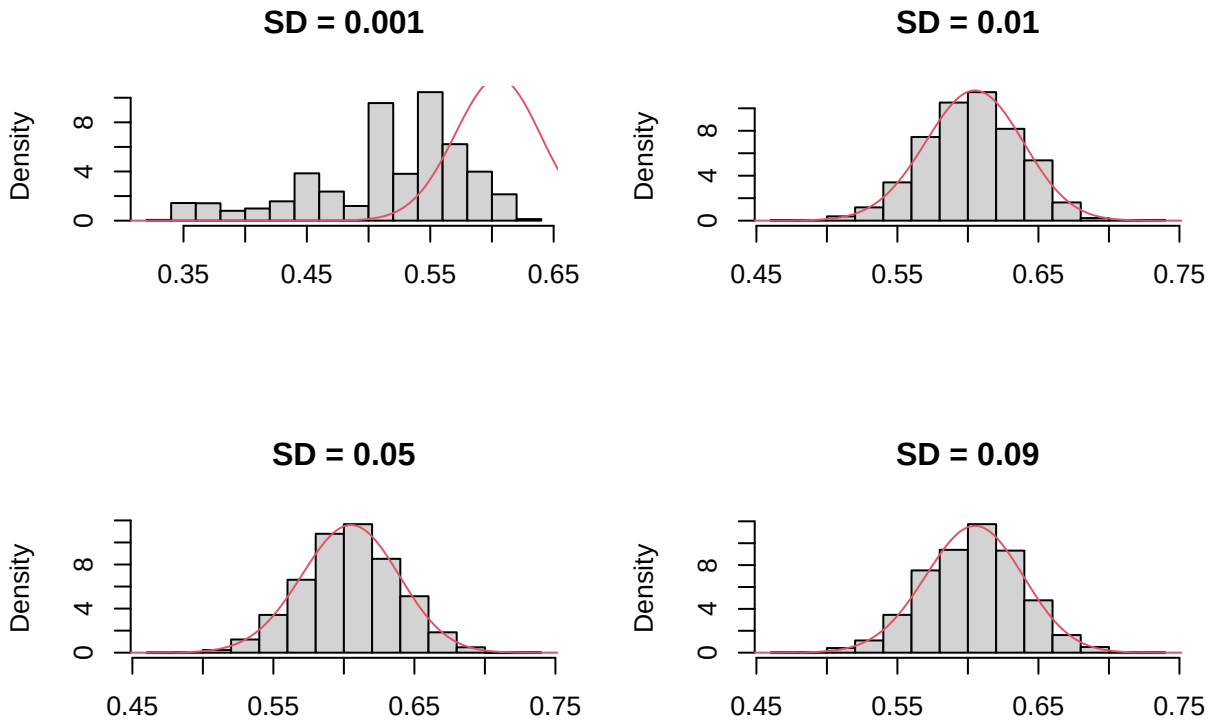
With burn-in, the density estimates for all starting points seem reasonably close. I now experiment with proposal standard deviations. To properly gauge the effect of the standard deviation, I equate the starting points at 0.25. I namely try 0.001, 0.01, 0.05 and 0.09.

```
set.seed(123) # for reproducibility
z_sd1 = psampler(50, 21, 29, 10000, 0.25, 0.001)
z_sd2 = psampler(50, 21, 29, 10000, 0.25, 0.01)
z_sd3 = psampler(50, 21, 29, 10000, 0.25, 0.05)
z_sd4 = psampler(50, 21, 29, 10000, 0.25, 0.09)

par(mfrow = c(2, 2))
plot(z_sd1, cex = 0.1, main = "SD = 0.001", ylab = "")
plot(z_sd2, cex = 0.1, main = "SD = 0.01", ylab = "")
plot(z_sd3, cex = 0.1, main = "SD = 0.05", ylab = "")
plot(z_sd4, cex = 0.1, main = "SD = 0.09", ylab = "")
```



It appears that the small standard deviation is still problematic for this example. However, for larger proposed standard deviations, because we have a proportion as our parameter, it seems to have limited effect. The last plot with 0.09 does seem to be less concentrated and possibly more "jumpy", but it suggests a reasonable degree of convergence (perhaps due to the large number of iterations). I now plot the histograms with the same burn-in as in the previous experiment.

```
par(mfrow = c(2, 2))
hist(z_sd1[-c(1:1000)], main = "SD = 0.001", xlab = "", prob = T)
lines(x, dbeta(x, 122, 80), col = 2)
hist(z_sd2[-c(1:1000)], main = "SD = 0.01", xlab = "", prob = T)
lines(x, dbeta(x, 122, 80), col = 2)
hist(z_sd3[-c(1:1000)], main = "SD = 0.05", xlab = "", prob = T)
lines(x, dbeta(x, 122, 80), col = 2)
hist(z_sd4[-c(1:1000)], main = "SD = 0.09", xlab = "", prob = T)
lines(x, dbeta(x, 122, 80), col = 2)
```



Confirming my observation of the Markov chain movements, the first plot is quite off. The chain could not move far away from the initial point of 0.25 fast enough. However, for the other plots, they seem quite accurate. This suggests that larger proposed standard deviations (albeit slower than what would be optimal) are faster, in terms of convergence to the stationary distribution, than extremely small proposed standard deviations.

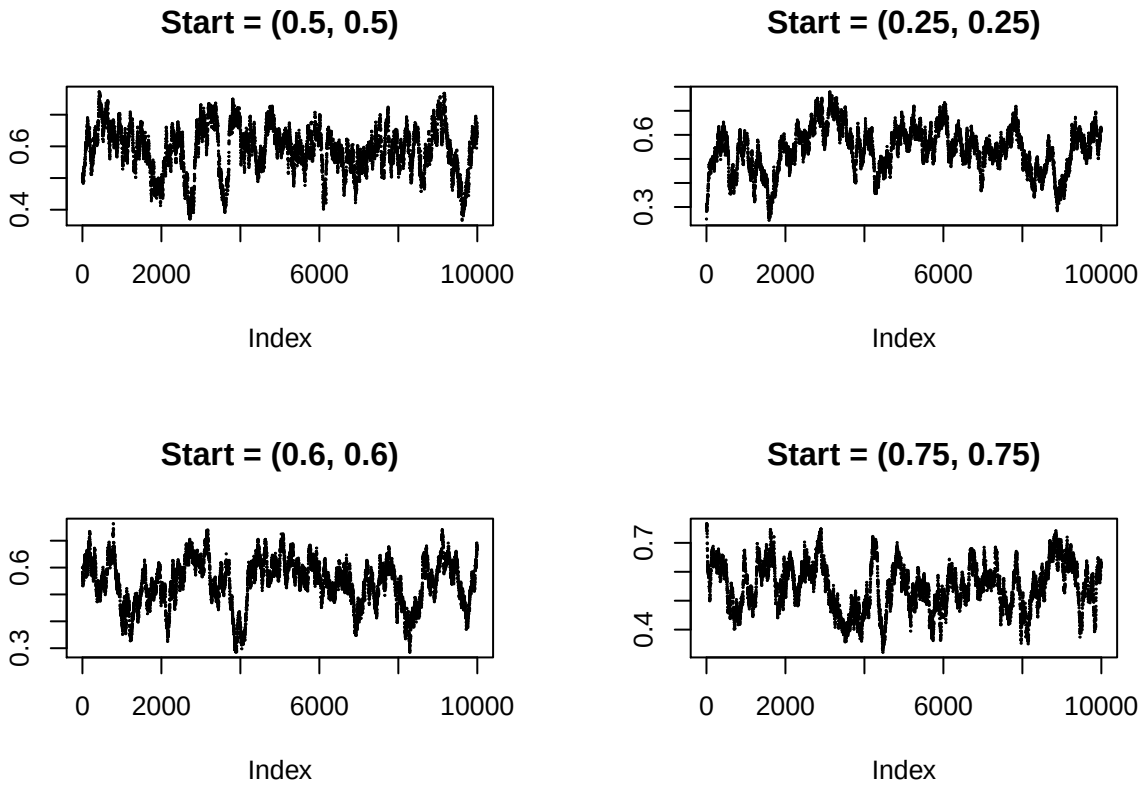
3. Joint Distribution

```
log_lik_fp = function(p, f, nAA, nAa, naa){
  return(nAA*log(f*p + (1-f)*p**2) + nAa*log(2*p*(1-p)*(1-f)) +
           naa*log(f*(1-p) + (1-f)*(1-p)**2))
}

fpsampler = function(nAA, nAa, naa, niter, fstartval, pstartval, fproposalsd, pproposalsd){
  f = rep(0, niter)
  p = rep(0, niter)
  f[1] = fstartval
  p[1] = pstartval
  for(i in 2:niter){
    currentf = f[i-1]
    currentp = p[i-1]
    newf = currentf + rnorm(1, 0, fproposalsd) # proposed f
    newp = currentp + rnorm(1, 0, pproposalsd) # proposed p

    # Rejection criterion
    A = exp(log_prior(newp) + log_prior(newf) +
              log_lik_fp(newp, newf, nAA, nAa, naa) -
              log_prior(currentp) - log_prior(currentf) -
              log_lik_fp(currentp, currentf, nAA, nAa, naa))
```

```
    # Acceptance/Rejection step
    if(runif(1) < A){
      p[i] = newp
      f[i] = newf
    }else{
      p[i] = currentp
      f[i] = currentf
    }
  }
  return(list(f = f,p = p)) # return a list with two elements named f and p
}
```

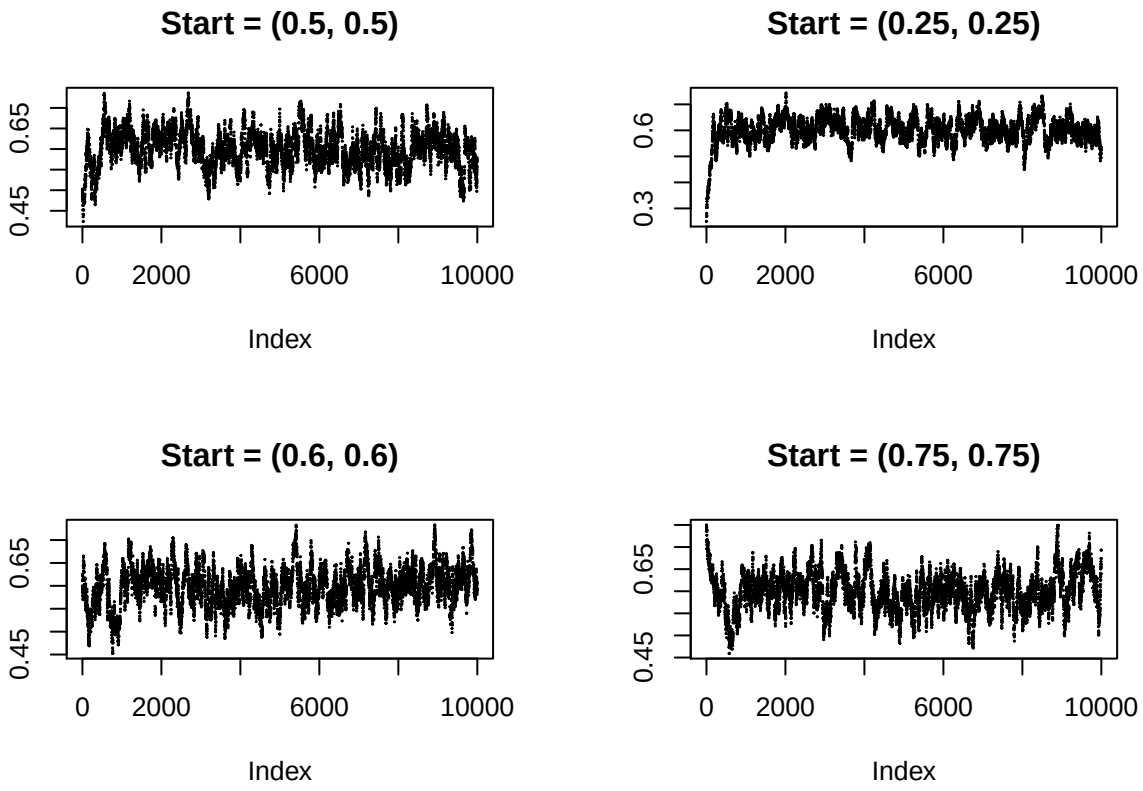
I now run the sampler and plot the paths for both f and p. I arbitrarily choose four different starting points in the parameter space and employ 0.01 for both proposed standard deviations.

```
# Run the sampler
set.seed(1234) # for reproducibility
z1 = fpsampler(50, 21, 29, 10000, 0.5, 0.5, 0.01, 0.01)
z2 = fpsampler(50, 21, 29, 10000, 0.25, 0.25, 0.01, 0.01)
z3 = fpsampler(50, 21, 29, 10000, 0.6, 0.6, 0.01, 0.01)
z4 = fpsampler(50, 21, 29, 10000, 0.75, 0.75, 0.01, 0.01)

# Plot paths for f
par(mfrow = c(2, 2))
plot(z1$f, cex = 0.1, main = "Start = (0.5, 0.5)", ylab = "")
plot(z2$f, cex = 0.1, main = "Start = (0.25, 0.25)", ylab = "")
plot(z3$f, cex = 0.1, main = "Start = (0.6, 0.6)", ylab = "")
plot(z4$f, cex = 0.1, main = "Start = (0.75, 0.75)", ylab = "")
```



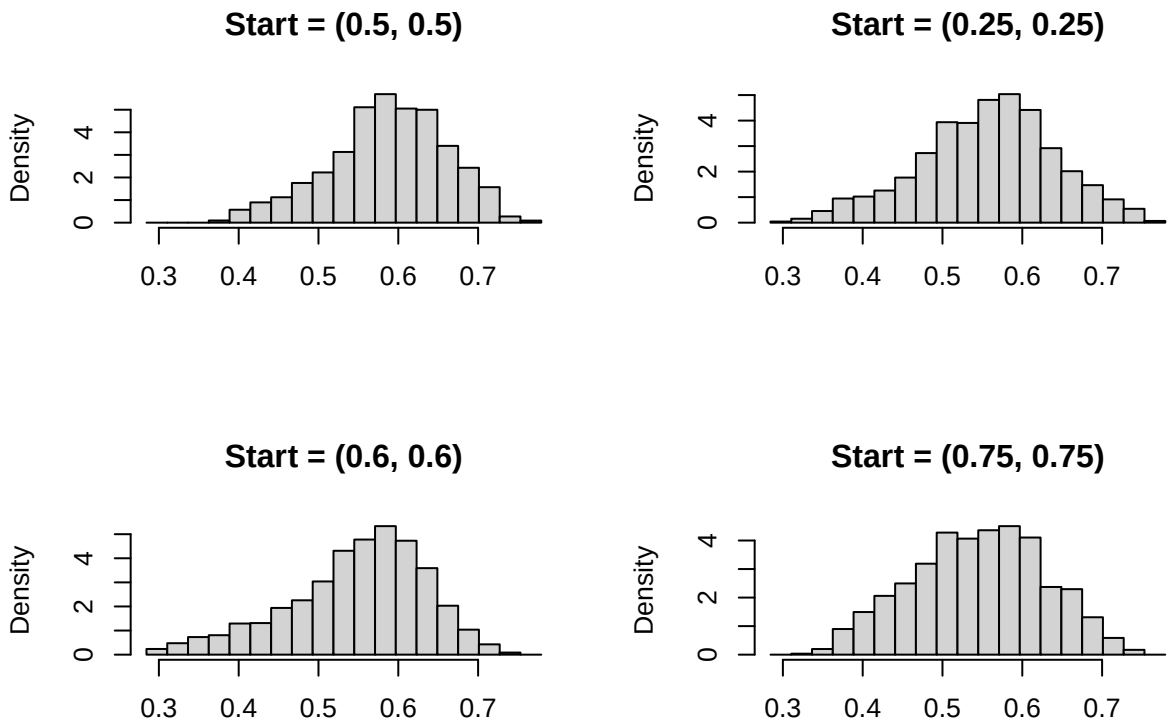
The mode seems to be around 0.5 or maybe higher for f.

```
# Plot paths for p
par(mfrow = c(2, 2))
plot(z1$p, cex = 0.1, main = "Start = (0.5, 0.5)", ylab = "")
plot(z2$p, cex = 0.1, main = "Start = (0.25, 0.25)", ylab = "")
plot(z3$p, cex = 0.1, main = "Start = (0.6, 0.6)", ylab = "")
plot(z4$p, cex = 0.1, main = "Start = (0.75, 0.75)", ylab = "")
```

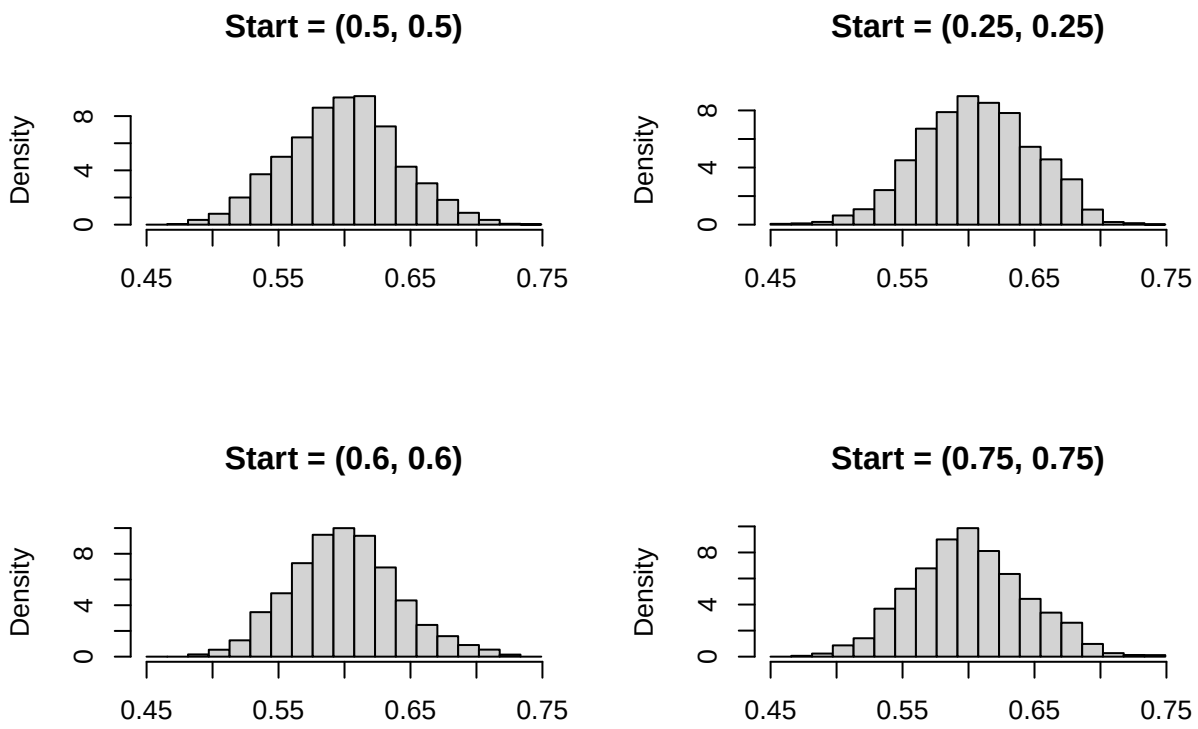


The mode seems to be around 0.6 for p. The paths, as always, seem to suggest the use of burn-in. To implement burn-in, I discard the first 2000 observations and plot the histograms for both f and p

```
# Histograms for f
maxz = max(c(z1$f[-c(1:2000)], z2$f[-c(1:2000)], z3$f[-c(1:2000)],
             z4$f[-c(1:2000)]))
minz = min(c(z1$f[-c(1:2000)], z2$f[-c(1:2000)], z3$f[-c(1:2000)],
             z4$f[-c(1:2000)]))

par(mfrow = c(2, 2))
hist(z1$f[-c(1:2000)], main = "Start = (0.5, 0.5)", xlab = "", prob = T,
     breaks = seq(minz, maxz, length = 20))
hist(z2$f[-c(1:2000)], main = "Start = (0.25, 0.25)", xlab = "", prob = T,
     breaks = seq(minz, maxz, length = 20))
hist(z3$f[-c(1:2000)], main = "Start = (0.6, 0.6)", xlab = "", prob = T,
     breaks = seq(minz, maxz, length = 20))
hist(z4$f[-c(1:2000)], main = "Start = (0.75, 0.75)", xlab = "", prob = T,
     breaks = seq(minz, maxz, length = 20))
```




```
# Histograms for p
maxz = max(c(z1$p[-c(1:2000)], z2$p[-c(1:2000)], z3$p[-c(1:2000)],
             z4$p[-c(1:2000)]))
minz = min(c(z1$p[-c(1:2000)], z2$p[-c(1:2000)], z3$p[-c(1:2000)],
             z4$p[-c(1:2000)]))

par(mfrow = c(2, 2))
hist(z1$p[-c(1:2000)], main = "Start = (0.5, 0.5)", xlab = "", prob = T,
     breaks = seq(minz, maxz, length = 20))
hist(z2$p[-c(1:2000)], main = "Start = (0.25, 0.25)", xlab = "", prob = T,
     breaks = seq(minz, maxz, length = 20))
hist(z3$p[-c(1:2000)], main = "Start = (0.6, 0.6)", xlab = "", prob = T,
     breaks = seq(minz, maxz, length = 20))
hist(z4$p[-c(1:2000)], main = "Start = (0.75, 0.75)", xlab = "", prob = T,
     breaks = seq(minz, maxz, length = 20))
```



The four histograms seem to correspond to the same stationary distributions for both f and p. I now calculate the posterior means as the point estimates and the relevant credible intervals.

```
# Posterior mean of f
post_mean = c(mean(z1$f[-c(1:2000)]), mean(z2$f[-c(1:2000)]),
              mean(z3$f[-c(1:2000)]), mean(z4$f[-c(1:2000)]))

# 90% Credible intervals for f
as.data.frame(cbind(post_mean, rbind(quantile(z1$f[-c(1:2000)], probs = c(0.05, 0.95)), quantile(z2$f[-
```

post_mean	5%	95%
0.5868343	0.4503868	0.7013112
0.5553327	0.3976160	0.6949708
0.5475584	0.3821691	0.6696884
0.5447540	0.4038070	0.6769762

The point estimate for f is roughly 0.55 (although there is some variation across the trials) and the 90% credible interval is roughly (0.4, 0.7).

```
# Posterior mean of p
post_mean = c(mean(z1$p[-c(1:2000)]), mean(z2$p[-c(1:2000)]),
              mean(z3$p[-c(1:2000)]), mean(z4$p[-c(1:2000)]))

# 90% Credible intervals for p
as.data.frame(cbind(post_mean, rbind(quantile(z1$p[-c(1:2000)], probs = c(0.05, 0.95)), quantile(z2$p[-
```

post_mean	5%	95%
0.5990214	0.5285408	0.6701267
0.6061386	0.5373048	0.6763245
0.5998077	0.5363954	0.6704773
0.6001947	0.5316934	0.6747325

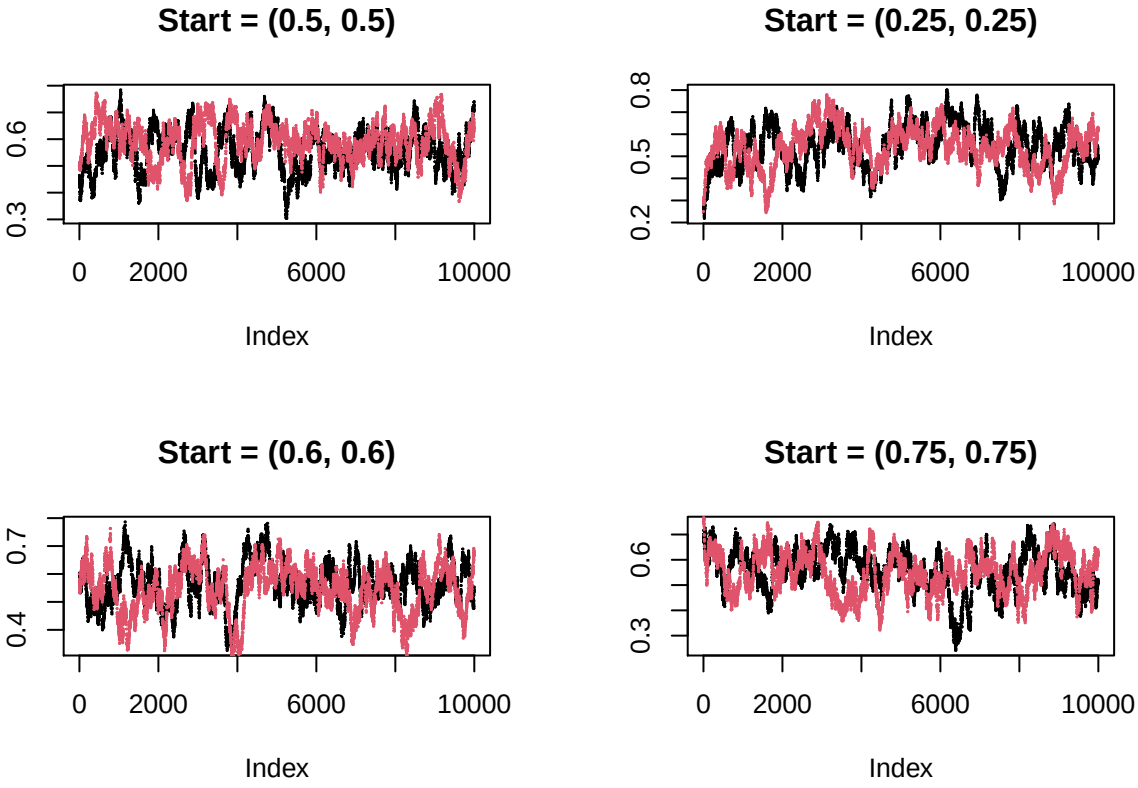
The point estimate for p is roughly 0.6 and the 90% credible interval is roughly (0.53, 0.67).

As a further exercise, I also try implementing the component-wise Metropolis-Hastings algorithm and double check my results. (I really wanted to see for myself whether the component-wise is more efficient than the joint MH.)

```
# Component-wise Metropolis-Hastings
fpsampler_cw = function(nAA, nAa, naa, niter, fstartval, pstartval, fproposalsd, pproposalsd){
  f = rep(0, niter)
  p = rep(0, niter)
  f[1] = fstartval
  p[1] = pstartval
  for(i in 2:niter){
    currentf = f[i-1]
    currentp = p[i-1]

    # Component f
    newf = currentf + rnorm(1, 0, fproposalsd) # proposed f
    # Rejection criterion for f
    A_f = exp(log_prior(newf) + log_lik_fp(currentp, newf, nAA, nAa, naa) -
                log_prior(currentf) -
                log_lik_fp(currentp, currentf, nAA, nAa, naa))
    # Acceptance/Rejection step
    if(runif(1) < A_f){
      f[i] = newf
    }else{
      f[i] = currentf
    }

    # Component p
    newp = currentp + rnorm(1, 0, pproposalsd) # proposed p
    # Rejection criterion for p
    A_p = exp(log_prior(newp) + log_lik_fp(newp, f[i], nAA, nAa, naa) -
                log_prior(currentp) -
                log_lik_fp(currentp, f[i], nAA, nAa, naa))

    # Acceptance/Rejection step
    if(runif(1) < A_p){
      p[i] = newp
    }else{
      p[i] = currentp
    }
  }
  return(list(f = f,p = p)) # return a list with two elements named f and p
}
```

I run this function with the same parameter values, starting points and proposed standard deviations as the previous function.

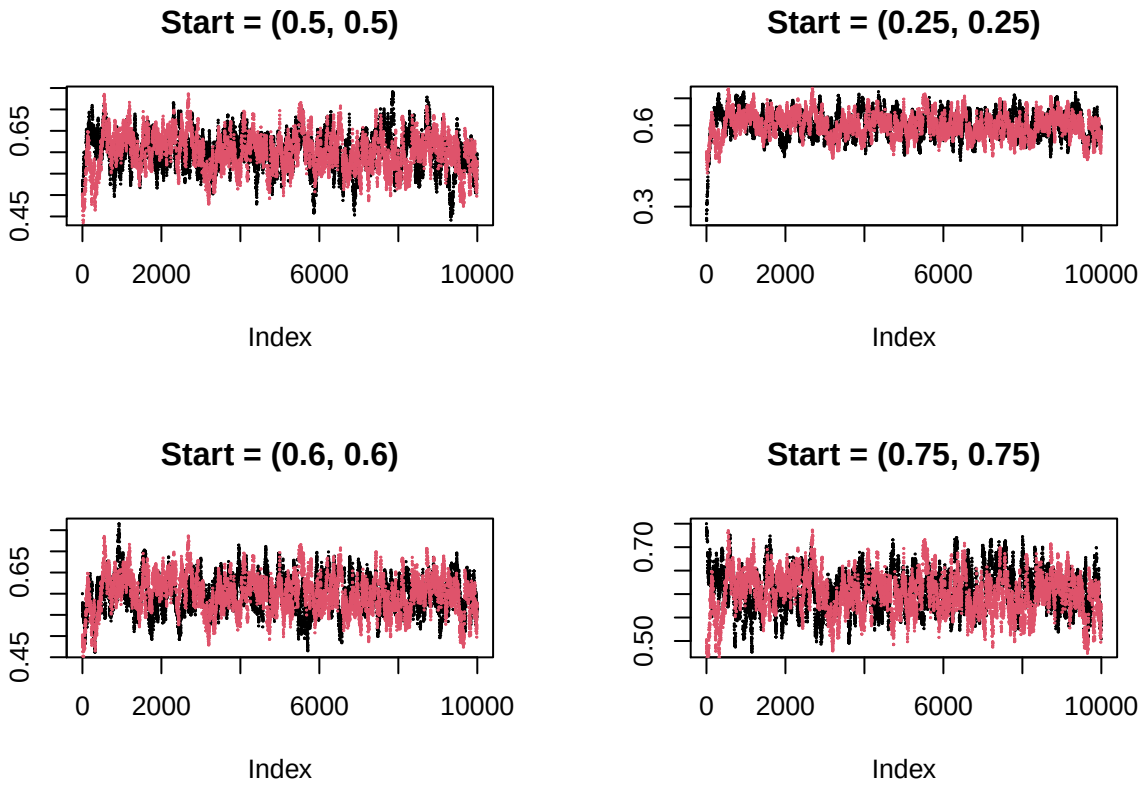
```r
# Run the sampler
set.seed(1234) # for reproducibility
z1_cw = fpsampler_cw(50, 21, 29, 10000, 0.5, 0.5, 0.01, 0.01)
z2_cw = fpsampler_cw(50, 21, 29, 10000, 0.25, 0.25, 0.01, 0.01)
z3_cw = fpsampler_cw(50, 21, 29, 10000, 0.6, 0.6, 0.01, 0.01)
z4_cw = fpsampler_cw(50, 21, 29, 10000, 0.75, 0.75, 0.01, 0.01)

# Plot paths for f
par(mfrow = c(2, 2))
plot(z1_cw$f, cex = 0.1, main = "Start = (0.5, 0.5)", ylab = "")
points(z1$f, cex = 0.1, col = 2)
plot(z2_cw$f, cex = 0.1, main = "Start = (0.25, 0.25)", ylab = "")
points(z2$f, cex = 0.1, col = 2)
plot(z3_cw$f, cex = 0.1, main = "Start = (0.6, 0.6)", ylab = "")
points(z3$f, cex = 0.1, col = 2)
plot(z4_cw$f, cex = 0.1, main = "Start = (0.75, 0.75)", ylab = "")
points(z4$f, cex = 0.1, col = 2)
```



I plot the original function's results in red alongside the component-wise MH output in black. For f, I feel the black lines are more variable, which is consistent with what we have learned about the component-wise Metropolis-Hastings algorithm: it will accept larger steps for individual parameters.

```r
# Plot paths for p
par(mfrow = c(2, 2))
plot(z1_cw$p, cex = 0.1, main = "Start = (0.5, 0.5)", ylab = "")
points(z1$p, cex = 0.1, col = 2)
plot(z2_cw$p, cex = 0.1, main = "Start = (0.25, 0.25)", ylab = "")
points(z1$p, cex = 0.1, col = 2)
plot(z3_cw$p, cex = 0.1, main = "Start = (0.6, 0.6)", ylab = "")
```

```
points(z1$p, cex = 0.1, col = 2)
plot(z4_cw$p, cex = 0.1, main = "Start = (0.75, 0.75)", ylab = "")
points(z1$p, cex = 0.1, col = 2)
```



I do the same for p. The greater variation is less visible for p, perhaps due to the fact that the proposed standard deviation is already quite small. I check the posterior means and the credible intervals with the same burn-in of 2000.

```
# Posterior means
post_mean = c(mean(z1_cw$f[-c(1:2000)]), mean(z2_cw$f[-c(1:2000)]),
              mean(z3_cw$f[-c(1:2000)]), mean(z4_cw$f[-c(1:2000)]))
# 90% Credible intervals for f
as.data.frame(cbind(post_mean, rbind(quantile(z1_cw$f[-c(1:2000)], probs = c(0.05, 0.95)), quantile(z2_
```

post_mean	5%	95%
0.5577053	0.4378494	0.6794184
0.5526367	0.3900593	0.7034191
0.5661456	0.4352588	0.7004411
0.5521405	0.3965512	0.6803618

I find that the point estimates (i.e., the posterior means) are much more consistent across the different trials for the component-wise method. For f, the posterior mean seems to be roughly 0.55. The credible intervals are also less volatile, though not without variation across the trials.

```
# Posterior means
post_mean = c(mean(z1_cw$p[-c(1:2000)]), mean(z2_cw$p[-c(1:2000)]),
              mean(z3_cw$p[-c(1:2000)]), mean(z4_cw$p[-c(1:2000)]))
# 90% Credible intervals for p
as.data.frame(cbind(post_mean, rbind(quantile(z1_cw$p[-c(1:2000)], probs = c(0.05, 0.95)), quantile(z2_
```

post_mean	5%	95%
0.5993879	0.5265788	0.6718736
0.6029697	0.5370583	0.6710256
0.5993627	0.5347779	0.6601495
0.6094949	0.5437715	0.6776559

I find similar results for p as well. The point estimates are much more convergent across the trials. The credible intervals are more or less equivalent to the original vector-wise approach, which was already quite consistent across the runs with different starting points.