Simulating Markov Chains

Hemant Banke (MD2107)

2022-07-20

A Simple Markov Chain

Here we will simulate a discrete time, discrete state space Markov Chain using a random TPM. Goals will be as follows :
- Simulate MC using random TPM
- Check simulated TPM
- Check Stationary distribution

We shall consider a MC with state space $S = \{1, 2, \ldots, 10\}$.

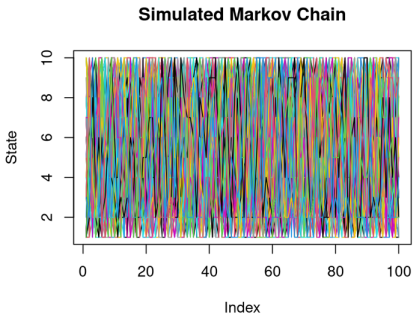
```
## States
S.no = 10
S = 1:S.no

## Generate random TPM
tpm = matrix(nrow = S.no, ncol = S.no)
# Get irreducible MP w.p. 1
for (i in 1:S.no){
  tpm_row = c(sort(sample(1:100, size = S.no-1, replace = TRUE)),
100)/100
  tpm_row = tpm_row - c(0, tpm_row[1:(S.no-1)])
  tpm[i,] = tpm_row
}

tpm
```

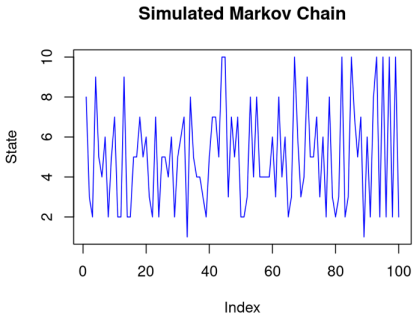
```
##       [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
##  [1,] 0.16 0.11 0.04 0.10 0.04 0.34 0.11 0.09 0.01  0.00
##  [2,] 0.07 0.17 0.27 0.02 0.20 0.05 0.02 0.04 0.09  0.07
##  [3,] 0.14 0.13 0.00 0.05 0.01 0.10 0.12 0.27 0.02  0.16
##  [4,] 0.06 0.14 0.25 0.12 0.03 0.15 0.03 0.06 0.11  0.05
##  [5,] 0.01 0.05 0.03 0.13 0.08 0.18 0.20 0.08 0.09  0.15
##  [6,] 0.05 0.44 0.08 0.01 0.07 0.01 0.12 0.06 0.11  0.05
##  [7,] 0.15 0.19 0.09 0.09 0.19 0.02 0.17 0.08 0.02  0.00
##  [8,] 0.04 0.01 0.20 0.35 0.06 0.01 0.11 0.00 0.10  0.12
##  [9,] 0.06 0.12 0.01 0.02 0.11 0.04 0.10 0.25 0.07  0.22
## [10,] 0.07 0.25 0.06 0.03 0.00 0.23 0.07 0.10 0.09  0.10
```

```
## Generate random initial distribution
pi.initial = c(sort(sample(1:100, size = S.no-1, replace = TRU
E)), 100)/100
pi.initial = pi.initial - c(0, pi.initial[1:(S.no-1)])
```

Generating 500 Markov Chains, each of length 1000 using the generated random TPM.

```
## Generate MC's
mc.length = 1000
mc.iter = 500
mc.data = matrix(1, nrow = mc.length, ncol = mc.iter)
mc.data[1,] = sample(1:S.no, size = mc.iter, replace = TRUE, prob
= pi.initial)
for (i in 2:mc.length){
  for (j in 1:mc.iter){
    mc.data[i, j] = sample(S, size = 1, prob = tpm[mc.data[i-1,
j],])
  }
}
# View(mc.data)


## Plotting MC (100 iterations) length : 100
plot(1, type = "n", main = "Simulated Markov Chain", xlab = "Inde
x",
     ylab = "State", xlim = c(1, 100), ylim = c(S[1], S[S.no]))
for (j in 1:100){
  lines(x = 1:100, y = mc.data[1:100,j], col = j)
}

## Plotting MC (1'st iteration) length : 100
plot(x = 1:100, y = mc.data[1:100,1], col = 'blue', type = "l",
     main = "Simulated Markov Chain", xlab = "Index", ylab = "Sta
te")
```





Estimating TPM

We can estimate the TPM as follows :
Let $\{X\}_{i=1}^{n}$ be the generated markov chain, and we repeat the process r times. Then we can estimate Probability of transition from state $'a'$ to state $'b'$ as :

$$\hat{P}(X_1 = b|X_0 = a) = \frac{\sum_{j=1}^{r}\sum_{i=2}^{n} 1\{X_n = b, X_{n-1} = a\}}{\sum_{j=1}^{r}\sum_{i=1}^{n} 1\{X_n = a\}}$$

We can further calculate the MSE of our estimate to check if the estimated TPM is close to the original TPM.

```
## Estimate TPM from simulated chain
tpm.est = matrix(0, nrow = S.no, ncol = S.no)

# Count no. of transitions from state a to state b
for (i in 2:mc.length){
  for (j in 1:mc.iter){
    tpm.est[mc.data[i-1, j], mc.data[i, j]] =
      tpm.est[mc.data[i-1, j], mc.data[i, j]] + 1
  }
}

# Divide no. of transitions by no. of times MC was in state a
for (i in 1:S.no){
  tpm.est[i,] = tpm.est[i,]/sum(mc.data == i)
}

# MSE of estimate
tpm.est.mse = sum((tpm.est - tpm)^2)
tpm.est.mse
```

```
## [1] 0.0001332803
```

Estimating Stationary Distribution

As the number of transitions of the Markov Chain increases, the distribution of states gets closer to its stationary distribution.

```
## Check stationary distribution
# Simulated Distribution
pi.est = vector(length = S.no)
for (i in 1:S.no){
  pi.est[i] = sum(mc.data[mc.length, ] == i)/mc.iter
}
pi.est
```

```
##  [1] 0.100 0.132 0.136 0.086 0.092 0.090 0.110 0.092 0.082 0.0
80
```

The Actual Stationary Distribution of the markov chain can be found by solving the following systems of linear equation :
Let π be the row vector representing stationary distribution and P be the TPM, then

$$\sum_{i=1}^{n} \pi_i = 1 \; ; \; \pi * P = \pi$$

We can further calculate the MSE of our estimate to check if the estimated distribution is close to the actual distribution.

```
# consider first S.no-1 equations of (P - I)'*pi' = 0 and 1'*pi'
= 1
pi.actual = solve(rbind(t(tpm - diag(nrow = S.no))[1:S.no-1, ], r
ep(1, S.no)),
                  c(rep(0, S.no - 1), 1))
pi.actual
```

```
##  [1] 0.08156156 0.16522002 0.11544215 0.08765891 0.08709571 0.
10376712
##  [7] 0.09985384 0.09842504 0.07178150 0.08919415
```

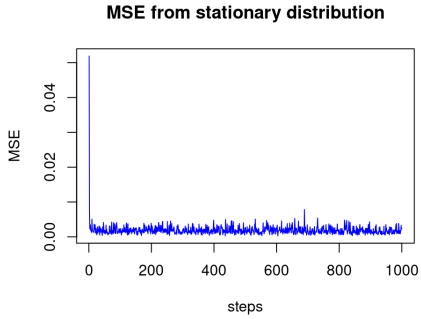
```
# MSE
pi.est.mse = sum((pi.est - pi.actual)^2)
pi.est.mse
```

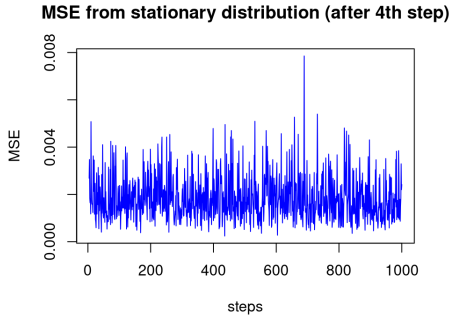
```
## [1] 0.002415685
```

Convergence to stationary distribution

Here we observe the MSE of estimated stationary distribution calculated at each transition step of the markov chain.
We noticed here the convergence is rapid, and only after 4-5 steps the MSE stabilizes to a very low value.

```
## Convergence to stationary distribution
# Plotting MSE
plot(x = 1:mc.length, y = sapply(1:mc.length, function(i) {
      pi.row = vector(length = S.no)
      for (j in 1:S.no){
        pi.row[j] = sum(mc.data[i, ] == j)/mc.iter
      }
      return (sum((pi.row - pi.actual)^2))
    }),
     col = "blue", main = "MSE from stationary distribution",
     xlab = "steps", ylab = "MSE", type = "l")


# Plotting MSE (after few initial steps)
plot(x = 4:mc.length, y = sapply(4:mc.length, function(i) {
  pi.row = vector(length = S.no)
  for (j in 1:S.no){
    pi.row[j] = sum(mc.data[i, ] == j)/mc.iter
  }
  return (sum((pi.row - pi.actual)^2))
}),
col = "blue", main = "MSE from stationary distribution (after 4th
step)",
xlab = "steps", ylab = "MSE", type = "l")
```





Alternating TPM's

Here we will simulate a discrete time, discrete state space Markov Chain with the following properties :
- First Transition happens with TPM P_1
- Second Transition happens with TPM P_2
- following this every odd transition occurs with TPM P_1 and every even transition occurs with TPM P_2.

We shall consider a MC with state space $S = \{1, 2, \ldots, 6\}$. To visually observe behavior of such models we will consider P_1 as a reducible TPM with equivalence classes $\{1, 2, 3\}$ and $\{4, 5, 6\}$. P_2 will be a random irreducible TPM.

```
## States
S.no = 6
S = 1:S.no

## Generate random TPM
P1 = matrix(nrow = S.no, ncol = S.no)
# Get reducible MP
for (i in 1:S.no){
  tpm_row = c(sort(sample(1:100, size = floor(S.no/2)-1, replace
= TRUE)), 100)/100
  tpm_row = tpm_row - c(0, tpm_row[1:(floor(S.no/2)-1) ])
  if(i <= floor(S.no/2)){
    P1[i, ] = c(tpm_row, rep(0, floor(S.no/2)))
  }
  else{
    P1[i, ] = c(rep(0, floor(S.no/2)), tpm_row)
  }
}


## Generate irreducible second TPM
P2 = matrix(nrow = S.no, ncol = S.no)
for (i in 1:S.no){
  tpm_row = c(sort(sample(1:100, size = S.no-1, replace = TRUE)),
100)/100
  tpm_row = tpm_row - c(0, tpm_row[1:(S.no-1)])
  P2[i,] = tpm_row
}

P1
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,] 0.01 0.28 0.71 0.00 0.00 0.00
## [2,] 0.56 0.43 0.01 0.00 0.00 0.00
## [3,] 0.71 0.18 0.11 0.00 0.00 0.00
## [4,] 0.00 0.00 0.00 0.06 0.11 0.83
## [5,] 0.00 0.00 0.00 0.14 0.52 0.34
## [6,] 0.00 0.00 0.00 0.69 0.26 0.05
```

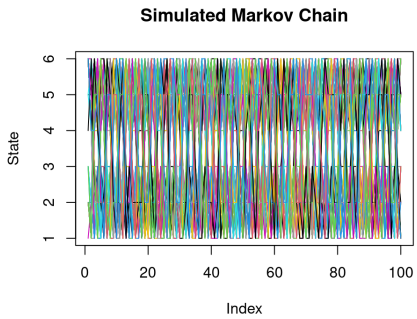
```
P2
```

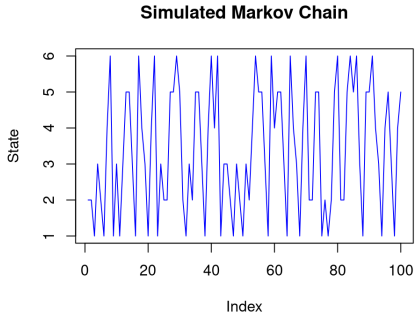
```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,] 0.06 0.04 0.04 0.34 0.23 0.29
## [2,] 0.20 0.08 0.03 0.11 0.43 0.15
## [3,] 0.17 0.65 0.06 0.02 0.06 0.04
## [4,] 0.04 0.29 0.36 0.04 0.12 0.15
## [5,] 0.01 0.16 0.26 0.07 0.07 0.43
## [6,] 0.25 0.35 0.03 0.20 0.16 0.01
```

```
## Generate random initial distribution
pi.initial = c(sort(sample(1:100, size = S.no-1, replace = TRU
E)), 100)/100
pi.initial = pi.initial - c(0, pi.initial[1:(S.no-1)])
```

Generating 500 Markov Chains, each of length 500 using the generated random TPMs.

```
mc.length = 500
mc.iter = 500
mc.data = matrix(1, nrow = mc.length, ncol = mc.iter)
mc.data[1,] = sample(1:S.no, size = mc.iter, replace = TRUE, prob
= pi.initial)
for (i in 2:mc.length){
  for (j in 1:mc.iter){
    if (i%%2 == 0)
      mc.data[i, j] = sample(S, size = 1, prob = P1[mc.data[i-1,
j],])
    else
      mc.data[i, j] = sample(S, size = 1, prob = P2[mc.data[i-1,
j],])
  }
}
# View(mc.data)


## Plotting MC (100 iterations) length : 100
plot(1, type = "n", main = "Simulated Markov Chain", xlab = "Inde
x",
     ylab = "State", xlim = c(1, 100), ylim = c(S[1], S[S.no]))
for (j in 1:100){
  lines(x = 1:100, y = mc.data[1:100,j], col = j)
}

## Plotting MC (1'st iteration) length : 100
plot(x = 1:100, y = mc.data[1:100,1], col = 'blue', type = "l",
     main = "Simulated Markov Chain", xlab = "Index", ylab = "Sta
te")
```





Estimating TPMs

We can estimate TPM of the combined chain as done above and calculate its MSE against an intuitive estimator of combined TPM $\frac{1}{2}(P_1 + P_2)$.

```
## Estimate TPM from simulated chain
tpm.est = matrix(0, nrow = S.no, ncol = S.no)
# Count no. of transitions from state a to state b
for (i in 2:mc.length){
  for (j in 1:mc.iter){
    tpm.est[mc.data[i-1, j], mc.data[i, j]] =
      tpm.est[mc.data[i-1, j], mc.data[i, j]] + 1
  }
}
# Divide no. of transitions by no. of times MC was in state a
for (i in 1:S.no){
  tpm.est[i,] = tpm.est[i,]/sum(mc.data == i)
}

# MSE of estimate
tpm.est.mse = sum((tpm.est - (P1+P2)/2)^2)
tpm.est.mse
```

```
## [1] 0.02429618
```

Estimating P_1 and P_2

```
## Estimating P1 and P2
# Divide the MC into multiple chains
p1.est = matrix(0, nrow = S.no, ncol = S.no)
p2.est = matrix(0, nrow = S.no, ncol = S.no)
# Count no. of transitions from state a to state b
for (i in 2:mc.length){
  for (j in 1:mc.iter){
    if (i%%2 == 0)
      p1.est[mc.data[i-1, j], mc.data[i, j]] =
        p1.est[mc.data[i-1, j], mc.data[i, j]] + 1
    else
      p2.est[mc.data[i-1, j], mc.data[i, j]] =
        p2.est[mc.data[i-1, j], mc.data[i, j]] + 1
  }
}
# Divide no. of transitions by no. of times MC was in state a
for (i in 1:S.no){
  p1.est[i,] = p1.est[i,]/sum(mc.data[seq(from = 1, to = mc.lengt
h, by = 2),] == i)
  p2.est[i,] = p2.est[i,]/sum(mc.data[seq(from = 2, to = mc.lengt
h, by = 2),] == i)
}

# MSE of estimate
p1.est.mse = sum((p1.est - P1)^2)
p2.est.mse = sum((p2.est - P2)^2)
p1.est.mse
```

```
## [1] 9.989674e-05
```

```
p2.est.mse
```

```
## [1] 0.0002110357
```

Estimating Stationary Distribution

As the number of transitions of the Markov Chain increases, the distribution of states gets closer to its stationary distribution.

```
## Check stationary distribution
# Simulated Distribution
pi.est = vector(length = S.no)
for (i in 1:S.no){
  pi.est[i] = sum(mc.data[mc.length, ] == i)/mc.iter
}
pi.est
```

```
## [1] 0.232 0.132 0.094 0.218 0.160 0.164
```

We now calculate the actual stationary distribution of P_1, P_2 and then calculate MSE of simulated stationary distribution from the obtained actual distributions,

```
# Actual Stationary Distribution of P_2
pi2.actual = solve(rbind(t(P2 - diag(nrow = S.no))[1:(S.no-1), ],
rep(1, S.no)),
                   c(rep(0, S.no - 1), 1))
pi2.actual
```

```
## [1] 0.1292849 0.2373810 0.1228556 0.1283562 0.1979110 0.184211
3
```

```
# MSE
pi2.est.mse = sum((pi.est - pi2.actual)^2)
pi2.est.mse
```

```
## [1] 0.03236995
```

```
# Actual Stationary Distribution of P_1
P11 = P1[1:(S.no/2), 1:(S.no/2)]
P12 = P1[(S.no/2 + 1):S.no, (S.no/2 + 1):S.no]
pi11.actual = solve(rbind(t(P11 - diag(nrow = S.no/2))[1:(S.no/2
- 1), ], rep(1, S.no/2)),
                    c(rep(0, S.no/2 - 1), 1))
pi12.actual = solve(rbind(t(P12 - diag(nrow = S.no/2))[1:(S.no/2
- 1), ], rep(1, S.no/2)),
                     c(rep(0, S.no/2 - 1), 1))
pi11.actual = c(pi11.actual, rep(0, S.no/2))
pi12.actual = c(rep(0, S.no/2), pi12.actual)
pi11.actual
```

```
## [1] 0.3918605 0.2922481 0.3158915 0.0000000 0.0000000 0.000000
0
```

```
pi12.actual
```

```
## [1] 0.0000000 0.0000000 0.0000000 0.3271336 0.2850405 0.387825
9
```

```
# MSE
pi11.est.mse = sum((pi.est - pi11.actual)^2)
pi12.est.mse = sum((pi.est - pi12.actual)^2)
pi11.est.mse
```

```
## [1] 0.2004906
```

```
pi12.est.mse
```

```
## [1] 0.1577273
```

Noise in Morkov Chains

Here we will simulate a discrete time, discrete state space Markov Chain with the following properties :
- Transitions happen through TPM P with probability $1-\epsilon$
- Transitions happen through TPM Q with probability ϵ

We shall consider a MC with state space $S = \{1, 2, \ldots, 6\}$. To visually observe behavior of such models we will consider P as a reducible TPM with equivalence classes $\{1, 2, 3\}$ and $\{4, 5, 6\}$. Q will be a random irreducible TPM.

```
## States
S.no = 6
S = 1:S.no

## Generate random TPM
P = matrix(nrow = S.no, ncol = S.no)
# Get reducible MP
for (i in 1:S.no){
  tpm_row = c(sort(sample(1:100, size = floor(S.no/2)-1, replace
= TRUE)), 100)/100
  tpm_row = tpm_row - c(0, tpm_row[1:(floor(S.no/2)-1) ])
  if(i <= floor(S.no/2)){
    P[i, ] = c(tpm_row, rep(0, floor(S.no/2)))
  }
  else{
    P[i, ] = c(rep(0, floor(S.no/2)), tpm_row)
  }
}


## Generate random Noise TPM
Q = matrix(nrow = S.no, ncol = S.no)
# Get irreducible MP w.p. 1
for (i in 1:S.no){
  tpm_row = c(sort(sample(1:100, size = S.no-1, replace = TRUE)),
100)/100
  tpm_row = tpm_row - c(0, tpm_row[1:(S.no-1)])
  Q[i,] = tpm_row
}

P
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,] 0.56 0.29 0.15 0.00 0.00 0.00
## [2,] 0.06 0.04 0.90 0.00 0.00 0.00
## [3,] 0.34 0.41 0.25 0.00 0.00 0.00
## [4,] 0.00 0.00 0.00 0.44 0.05 0.51
## [5,] 0.00 0.00 0.00 0.17 0.65 0.18
## [6,] 0.00 0.00 0.00 0.45 0.38 0.17
```

```
Q
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,] 0.22 0.37 0.12 0.28 0.01 0.00
## [2,] 0.26 0.08 0.40 0.03 0.14 0.09
## [3,] 0.25 0.53 0.08 0.07 0.07 0.00
## [4,] 0.15 0.53 0.03 0.19 0.09 0.01
## [5,] 0.02 0.02 0.11 0.51 0.27 0.07
## [6,] 0.21 0.18 0.37 0.02 0.01 0.21
```

```
## Generate random initial distribution
pi.initial = c(sort(sample(1:100, size = S.no-1, replace = TRU
E)), 100)/100
pi.initial = pi.initial - c(0, pi.initial[1:(S.no-1)])
```

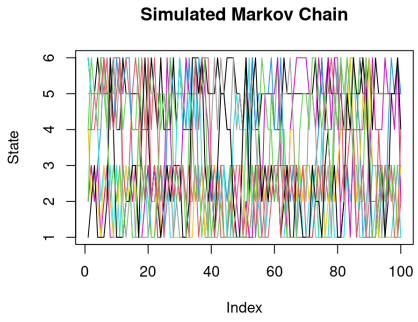
We will consider the following sequence of noise proabilities :

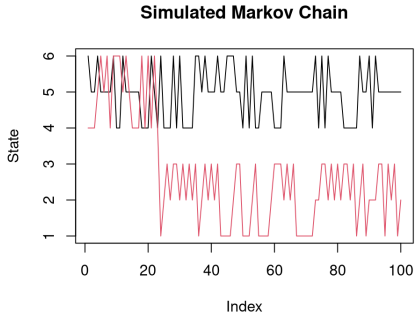
```
## Probability of noise
epsilon = c(0, seq(0.05, 0.5, length = 10))
epsilon
```

```
##  [1] 0.00 0.05 0.10 0.15 0.20 0.25 0.30 0.35 0.40 0.45 0.50
```

Generating 500 Markov Chains, each of length 500 using the generated random TPMs.

```
mc.length = 500
mc.iter = 500
mc.data = list()
for (k in 1:length(epsilon)){
  data = matrix(1, nrow = mc.length, ncol = mc.iter)
  data[1,] = sample(1:S.no, size = mc.iter, replace = TRUE, prob
= pi.initial)
  for (i in 2:mc.length){
    for (j in 1:mc.iter){
      noise = runif(1)
      if (noise < epsilon[k])
        data[i, j] = sample(S, size = 1, prob = Q[data[i-1, j],])
      else
        data[i, j] = sample(S, size = 1, prob = P[data[i-1, j],])
    }
  }
  mc.data[[k]] = data
}
# View(mc.data)


## Plotting MC (1'st iteration) length : 100
plot(1, type = "n", main = "Simulated Markov Chain", xlab = "Inde
x",
     ylab = "State", xlim = c(1, 100), ylim = c(S[1], S[S.no]))
for (k in 1:length(epsilon)){
  lines(x = 1:100, y = mc.data[[k]][1:100, 1], col = k)
}

# epsilon = 0, 0.05
plot(1, type = "n", main = "Simulated Markov Chain", xlab = "Inde
x",
     ylab = "State", xlim = c(1, 100), ylim = c(S[1], S[S.no]))
for (k in c(1,2)){
  lines(x = 1:100, y = mc.data[[k]][1:100, 1], col = k)
}
```





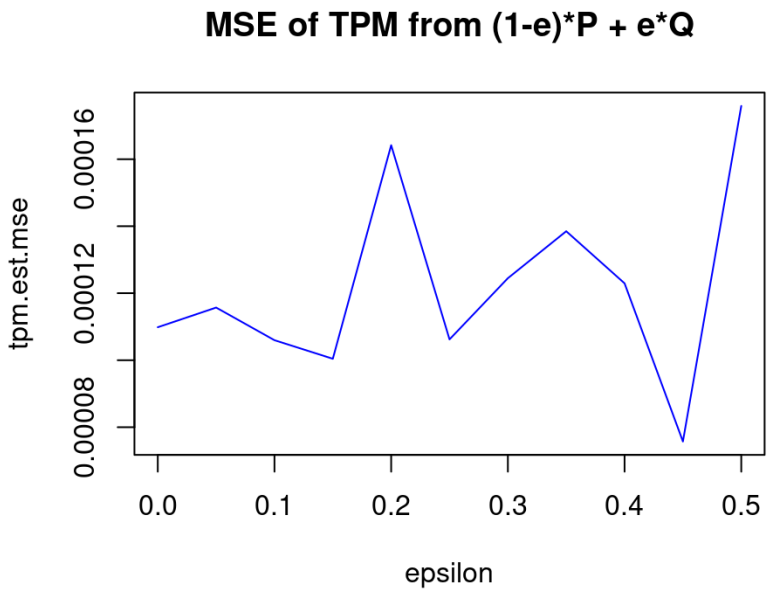
We can observe the red line (corresponding to $\epsilon = 0.05$) jumps between the equivalance classes, demonstrating presence of noise.

Estimating TPMs

We can estimate TPM of the combined chain as done above and calculate its MSE against the actual TPM of combined MC, $(1-\epsilon) * P + \epsilon * Q$. Provided ϵ is known.

$$\begin{aligned} P(X_{n+1} = x_{n+1} | X_n = x_n) &= (1-\epsilon) * P(X_{n+1} = x_{n+1} | X_n = x_n, TPM = P) \\ &+ \epsilon * P(X_{n+1} = x_{n+1} | X_n = x_n, TPM = Q) \\ &= (1-\epsilon) * p_{x_n, x_{n+1}} + \epsilon * q_{x_n, x_{n+1}} \end{aligned}$$

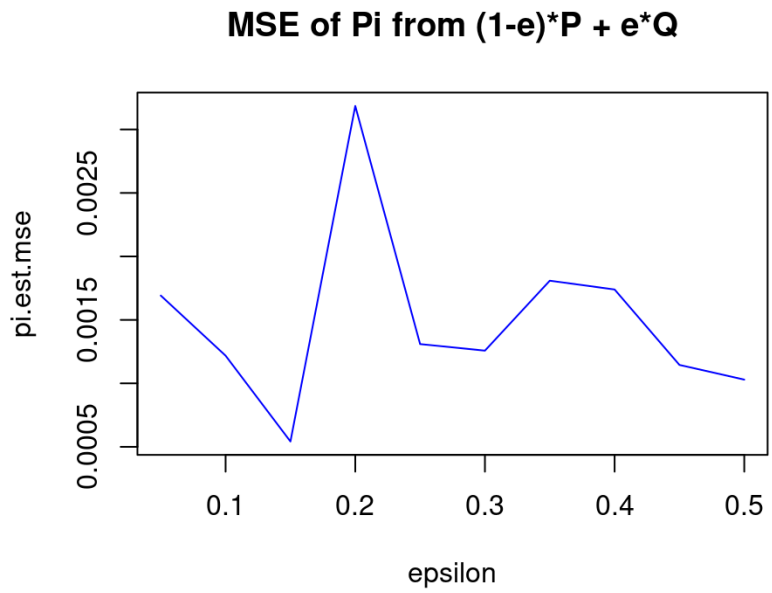
```
## Estimate TPM from simulated chain
tpm.est = list()
tpm.est.mse = c()

for(k in 1:length(epsilon)){
  est = matrix(0, nrow = S.no, ncol = S.no)
  # Count no. of transitions from state a to state b
  for (i in 2:mc.length){
    for (j in 1:mc.iter){
      est[mc.data[[k]][i-1, j], mc.data[[k]][i, j]] = est[mc.dat
a[[k]][i-1, j], mc.data[[k]][i, j]] + 1
    }
  }
  # Divide no. of transitions by no. of times MC was in state a
  for (i in 1:S.no){
    est[i,] = est[i,]/sum(mc.data[[k]] == i)
  }

  tpm.est[[k]] = est
  tpm.actual = (1-epsilon[k])*P + epsilon[k]*Q
  tpm.est.mse = append(tpm.est.mse, sum((tpm.est[[k]] - tpm.actua
l)^2))
}

# Plotting MSE of estimate
plot(x = epsilon, y = tpm.est.mse, type = 'l', col = "blue",
     main = "MSE of TPM from (1-e)*P + e*Q")
```



Estimating Stationary Distribution

We can now calculate the MSE of simulated stationary distribution against the stationary distribution obtained from actual TPM $(1-\epsilon) * P + \epsilon * Q$.

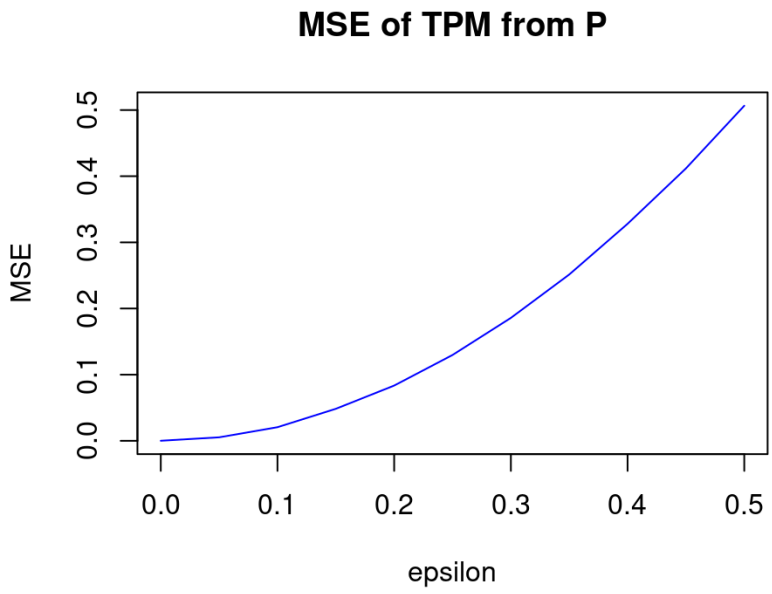
```
## Check stationary distribution
pi.est = list()
pi.actual = list()
pi.est.mse = c()

for (k in 2:length(epsilon)){

  # Simulated Stationary Distribution
  pi.est[[k]] = vector(length = S.no)
  for (i in 1:S.no){
    pi.est[[k]][i] = sum(mc.data[[k]][mc.length, ] == i)/mc.iter
  }

  # Actual Stationary Distribution
  tpm.actual = (1-epsilon[k])*P + epsilon[k]*Q
  pi.actual[[k]] = solve(rbind(t(tpm.actual - diag(nrow = S.no))[
1:S.no-1, ], rep(1, S.no)),
                         c(rep(0, S.no - 1), 1))

  # MSE
  pi.est.mse = append(pi.est.mse, sum((pi.est[[k]] - pi.actual[[
k]])^2))
}


# Plotting MSE of estimate
plot(x = epsilon[2:length(epsilon)], y = pi.est.mse, type = 'l',
col = "blue",
     main = "MSE of Pi from (1-e)*P + e*Q", xlab = "epsilon")
```



Effect of small ϵ

As practically ϵ will be an unknown small quantity, we should see how it affects our estimation.

```
## Effect of small epsilon in TPM estimation
plot(x = epsilon, y = sapply(1:length(epsilon), function(k) sum((
tpm.est[[k]] - P)^2)),
     main = "MSE of TPM from P", ylab = "MSE", type = 'l', col =
'blue')
```



```
## Effect of small epsilon in stationary distribution  estimation
# plot(x = epsilon, y = sapply(1:length(epsilon), function(k) su
m((pi.est[[k]] - pi.actual[[1]])^2)),
#      main = "MSE of Stationary Dist. from P", ylab = "MSE", typ
e = 'l', col = 'blue')
```

Hidden Markov Chain

We simulate a Hidden Markov Chain with state space $S_h = \{1, 2\}$. The hidden TPM (P) is generated such that probability in the diagonals is the highest.

```
S.h.no = 2
S.h = 1:S.h.no

## Generate hidden TPM with max probability in diagonals
tpm.h = matrix(nrow = S.h.no, ncol = S.h.no)
for (i in 1:S.h.no){
  tpm_row = c(sort(sample(1:100, size = S.h.no-1, replace = TRU
E)), 100)/100
  tpm_row = tpm_row - c(0, tpm_row[1:(S.h.no-1)])
  tpm_row[c(i, which(tpm_row == max(tpm_row)))] = c(max(tpm_row),
tpm_row[i])
  tpm.h[i,] = tpm_row
}
tpm.h
```

```
##      [,1] [,2]
## [1,] 0.94 0.06
## [2,] 0.25 0.75
```

The Observable Markov Chain has the state space $S = \{1, 2, 3\}$.
If $\{Y\}_{i=1}^{n}$ is the hidden markov chain and $\{X\}_{i=1}^{n}$ is the observable markov chain, then we define emission probabilities as $P(X = a|Y = b)$, i.e. conditional probability of observing state $'a'$ when the hidden MC is in state $'b'$. Hence, the emission probability matrix (E) will be a 2×3 matrix with rows representing hidden states and columns representing observable states.
We will consider a random emission probability matrix.

```
## Observable MC
S.no = 3
S = 1:S.no

# Emission Probabilities
emis = matrix(nrow = S.h.no, ncol = S.no)
for (i in 1:S.h.no){
  tpm_row = c(sort(sample(1:100, size = S.no-1, replace = TRUE)),
100)/100
  tpm_row = tpm_row - c(0, tpm_row[1:(S.no-1)])
  emis[i,] = tpm_row
}
emis
```

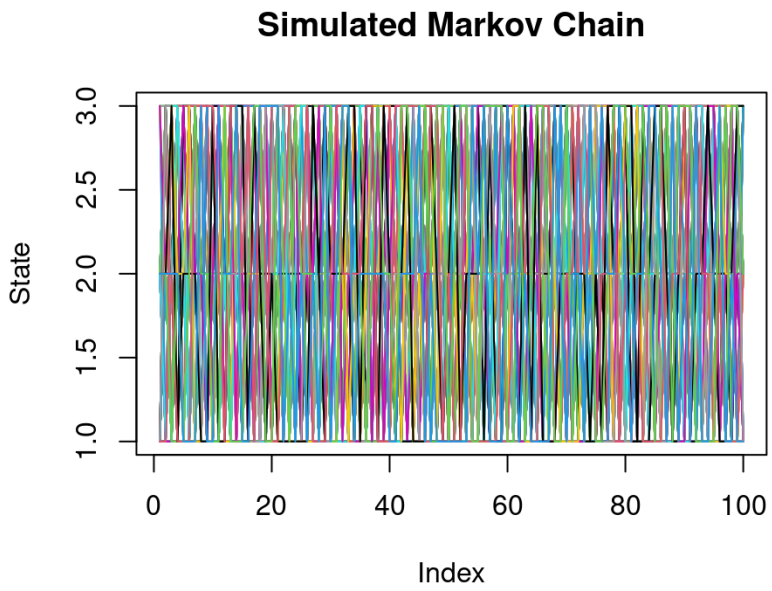
```
##      [,1] [,2] [,3]
## [1,] 0.33 0.32 0.35
## [2,] 0.50 0.45 0.05
```

```
## Generate random initial distribution
pi.initial = c(sort(sample(1:100, size = S.h.no-1, replace = TRU
E)), 100)/100
pi.initial = pi.initial - c(0, pi.initial[1:(S.h.no-1)])
```

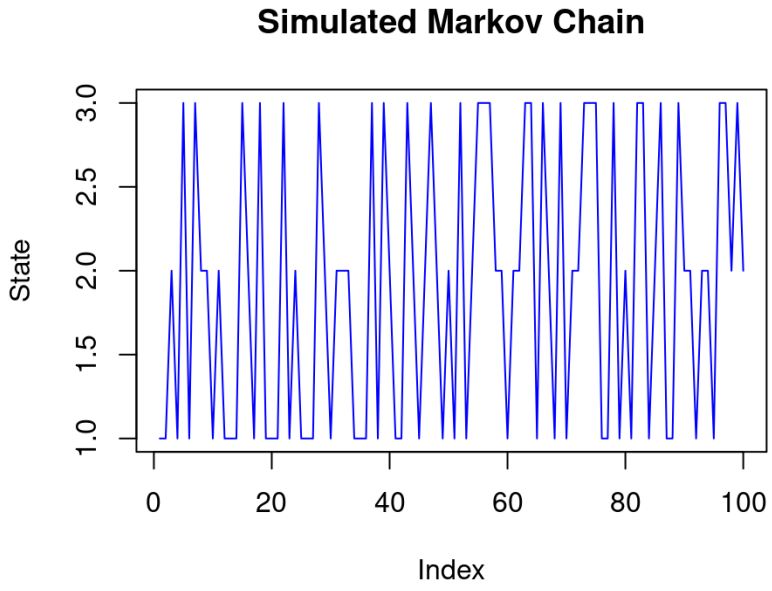
```
## Generate MC's
mc.length = 1000
mc.iter = 500

# Generating Hidden MC
hidden.data = matrix(1, nrow = mc.length, ncol = mc.iter)
hidden.data[1,] = sample(S.h, size = mc.iter, replace = TRUE, pro
b = pi.initial)
for (i in 2:mc.length){
  for (j in 1:mc.iter){
    hidden.data[i, j] = sample(S.h, size = 1, prob = tpm.h[hidde
n.data[i-1, j],])
  }
}
View(head(hidden.data))
```

```
# Generating Observable MC
obs.data = matrix(1, nrow = mc.length, ncol = mc.iter)
for (i in 1:mc.length){
  for (j in 1:mc.iter){
    obs.data[i, j] = sample(S, size = 1, prob = emis[hidden.data[
i, j],])
  }
}
View(head(obs.data))
```

```
## Plotting MC (100 iterations) length : 100
plot(1, type = "n", main = "Simulated Markov Chain", xlab = "Inde
x",
     ylab = "State", xlim = c(1, 100), ylim = c(S[1], S[S.no]))
for (j in 1:100){
  lines(x = 1:100, y = obs.data[1:100,j], col = j)
}
```




```
## Plotting MC (1'st iteration) length : 100
plot(x = 1:100, y = obs.data[1:100,1], col = 'blue', type = "l",
     main = "Simulated Markov Chain", xlab = "Index", ylab = "Sta
te")
```



Estimating properties of observable MC

Estimating TPM :

If we treat $\{X\}$ as a usual markov chain with state space S, then we can estimate it's TPM as follows,

```
## Estimate TPM from simulated chain
tpm.est = matrix(0, nrow = S.no, ncol = S.no)
# Count no. of transitions from state a to state b
for (i in 2:mc.length){
  for (j in 1:mc.iter){
    tpm.est[obs.data[i-1, j], obs.data[i, j]] =
      tpm.est[obs.data[i-1, j], obs.data[i, j]] + 1
  }
}
# Divide no. of transitions by no. of times MC was in state a
for (i in 1:S.no){
  tpm.est[i,] = tpm.est[i,]/sum(obs.data == i)
}
```

TPM of observable MC for discrete S :

$$P(X_{n+1} = b|X_n = a) = \sum_{y \in S_h} P(X_{n+1} = b|X_n = a, Y_{n+1} = y)P(Y_{n+1} = y)$$
$$= \sum_{y \in S_h} P(X_{n+1} = b|Y_{n+1} = y)P(Y_{n+1} = y)$$
$$= \sum_{y \in S_h} E_{y,b} * P(Y_{n+1} = y)$$
$$= \sum_{y \in S_h} E_{y,b} * (\pi_0 * P^{n+1})_y$$

where π_0 is the initial distribution

particularly,

$$P(X_1 = b|X_0 = a) = \sum_{y \in S_h} E_{y,b} * (\pi_0 * P)_y$$
$$= \sum_{y \in S_h} E_{y,b} * \pi_0 * P^{(y)}$$

Estimating Stationary Distribution :

```
## Check stationary distribution
pi.est = vector(length = S.no)
for (i in 1:S.no){
  pi.est[i] = sum(obs.data[mc.length, ] == i)/mc.iter
}
pi.est
```

```
## [1] 0.368 0.352 0.280
```

But does the HMM even approach any stationary distribution?

$$\lim_{n\to\infty} P(X_n = x) = \lim_{n\to\infty} \sum_{y\in S_h} P(X_n = x|Y_n = y)P(Y_n = y)$$
$$= \lim_{n\to\infty} \sum_{y\in S_h} E_{y,x} * P(Y_n = y)$$
as S is finite, we can interchange lim and sum
$$= \sum_{y\in S_h} E_{y,x} * \pi^{(y)}$$

Estimating Hidden TPM

If we assume the emission probabilities are known (**Note :** Here we are making a much stronger assumption that state space of the hidden markov chain is known and further we know the conditional probabilities for observed states. In some problems such assumption can be valid; but the main goal of the Hidden MC setup is to explore the unknown factors affecting the observations, this can not be done through this assumption).

$$\text{We have, } P(X_n = x|Y_n = y) = \frac{P(X_n = x, Y_n = y)}{P(Y_n = y)} = E_{y,x}$$
$$\implies P(Y_n = y|X_n = x) = E_{y,x} * \frac{P(Y_n = y)}{P(X_n = x)}$$
$$\text{Now, } argmax_{y\in S_h} P(Y_n = y|X_n = x) = argmax_{y\in S_h} E_{y,x} * P(Y_n = y)$$
$$= argmax_{y\in S_h} E_{y,x} * (\pi_0 * P^n)_y$$
$$= argmax_{y\in S_h} E_{y,x} * \pi_0 * (P^n)^{(y)}$$

A Maximum Likelihood approach would require finding y in the hidden State space which maximizes the above quantity, but π_0 and P are unknown.

Estimating the hidden sequence

Viterbi Algorithm

The algorithm aims at iteratively finding the most probable path of hidden sequence that generates the observable sequence.
The most likely hidden sequence $\{y_1, \ldots, y_T\}$ that produces the observed sequence $\{x_1, \ldots, x_T\}$ is given by recursively solving,

$$V_{1,k} = P(X_1 = x_1 \cap Y_1 = k) = P(X_1 = x_1|Y_1 = k) * \pi_0^{(k)}$$
$$V_{t,k} = \max_{y\in S_h} P(X_t = x_t|Y_t = k) * P_{y,k} * V_{t-1,y}$$

$V_{t,k}$ is the joint probability of the most probable path with $'k'$ as the last state of the hidden MC, i.e.
$\max_{y_1,\ldots,y_{t-1}\in S_h} P(X_1 = x_1, \ldots, X_t = x_t, Y_1 = y_1, \ldots, Y_{t-1} = y_{t-1}, Y_t = k)$. This can be shown using Markov property as follows,

$$V_{t,k} = \max_{y_1,\ldots,y_{t-1}\in S_h} P(X_1 = x_1, \ldots, X_t = x_t, Y_1 = y_1, \ldots, Y_{t-1} = y_{t-1}, Y_t = k)$$
$$= \max_{y_1,\ldots,y_{t-1}\in S_h} P(X_t = x_t|X_1 = x_1, \ldots, X_{t-1} = x_{t-1}, Y_1 = y_1, \ldots, Y_{t-1} = y_{t-1}, Y_t = k)$$
$$* P(X_1 = x_1, \ldots, X_{t-1} = x_{t-1}, Y_1 = y_1, \ldots, Y_{t-1} = y_{t-1}, Y_t = k)$$
$$= \max_{y_1,\ldots,y_{t-1}\in S_h} P(X_t = x_t|Y_t = k) * P(Y_t = k|X_1 = x_1, \ldots, X_{t-1} = x_{t-1}, Y_1 = y_1, \ldots, Y_{t-1} =$$
$$* P(X_1 = x_1, \ldots, X_{t-1} = x_{t-1}, Y_1 = y_1, \ldots, Y_{t-1} = y_{t-1})$$
$$= \max_{y_{t-1}\in S_h} P(X_t = x_t|Y_t = k) * P(Y_t = k|Y_{t-1} = y_{t-1}) * V_{t-1,y_{t-1}}$$

The hidden sequence is found by storing the y used while solving $V_{t,k}$. During implementation, to deal with probabilities quickly vanishing to 0, we use log probabilities instead.

```
## Viterbi Algorithm
viterbi = function(obs_path){
  h_path = vector(length = mc.length)
  V_prob = matrix(0, nrow = S.h.no, ncol = mc.length)
  V_path = matrix(0, nrow = S.h.no, ncol = mc.length)

  V_prob[,1] = log(pi.initial) + log(emis[, obs_path[1]])

  for (j in 2:mc.length){
    for (i in 1:S.h.no){
      k = S.h[i]
      y = S.h[which(log(tpm.h[, k]) + V_prob[, j-1] == max(log(tp
m.h[, k]) + V_prob[, j-1]))][1]
      V_prob[i, j] = log(emis[k, obs_path[j]]) + log(tpm.h[y, k])
+ V_prob[y, j-1]
      V_path[i, j] = y
    }
  }

  h_path[mc.length] = S.h[which(V_prob[, mc.length] == max(V_pro
b[, mc.length]))][1]
  for (j in seq(mc.length, 2, by = -1)){
    h_path[j-1] = V_path[h_path[j], j]
  }

  return (list("prob" = V_prob, "path" = h_path))
}
```

Estimating the hidden states for all iterations, we get the following Accuracy of estimate :

```
hidden.data.vest = matrix(1, nrow = mc.length, ncol = mc.iter)
for (j in 1:mc.iter){
  hidden.data.vest[,j] = viterbi(obs.data[,j])[["path"]]
}

# Accuracy
1 - sum(abs(hidden.data - hidden.data.vest))/(mc.length*mc.iter)
```

```
## [1] 0.806114
```

Forward Algorithm

Forward Algorithm aims at finding the most likely state at each time stamp, given the previous history of observations. This need not be the most likely sequence of hidden states (for which Viterbi algorithm is required). It involves computing the joint probability $P(Y_t = y_t, X_{1:t} = x_{1:t})$.

$$\text{Let, } \alpha_t(y_t) = P(Y_t = y_t, X_{1:t} = x_{1:t}) = \sum_{y_{t-1} \in S_h} P(Y_t = y_t, Y_{t-1} = y_{t-1}, X_{1:t} = x_{1:t})$$

$$= \sum_{y_{t-1} \in S_h} P(X_t = x_t | Y_t = y_t, Y_{t-1} = y_{t-1}, X_{1:t-1} = x_{1:t-1})$$

$$* P(Y_t = y_t | Y_{t-1} = y_{t-1}, X_{1:t-1} = x_{1:t-1})$$

$$* P(Y_{t-1} = y_{t-1}, X_{1:t-1} = x_{1:t-1})$$

$$\implies \alpha_t(y_t) = P(X_t = x_t | Y_t = y_t) * \sum_{y_{t-1} \in S_h} P(Y_t = y_t | Y_{t-1} = y_{t-1}) * \alpha_{t-1}(y_{t-1})$$

For $t = 1$ we set, $\alpha_1(y_1) = P(X_1 = x_1 | Y_1 = y_1) * \pi_1^{(y_1)}$.
We obtain the next most likely hidden step as follows :

$$\hat{y}_t = argmax_{y_t \in S_h} P(Y_t = y_t | X_{1:t} = x_{1:t}) = argmax_{y_t \in S_h} \alpha_t(y_t)$$

To deal with probabilities vanishing to 0, we can re-normalize $\alpha_t(y_t)$ to $\hat{\alpha}_t(y_t)$ at each step such that $\sum_{y_t \in S_h} \hat{\alpha}_t(y_t) = 1$.

$$\text{Let, } G_t = \sum_{y_t \in S_h} \alpha_t(y_t) \text{ and } \hat{\alpha}_1(y_1) = \frac{\alpha_1(y_1)}{G_1}$$

we now iteratively calculate,

$$\hat{\alpha}_t(y_t) = \frac{1}{G_t} P(X_t = x_t | Y_t = y_t) * \sum_{y_{t-1} \in S_h} P(Y_t = y_t | Y_{t-1} = y_{t-1}) * \hat{\alpha}_{t-1}(y_{t-1})$$

Now, $\hat{y}_t = argmax_{y_t \in S_h} \hat{\alpha}_t(y_t) = argmax_{y_t \in S_h} \frac{\prod_{1 \le k \le t-1} G_k}{G_t} \alpha_t(y_t) = argmax_{y_t \in S_h} \alpha_t(y_t)$

```
## Forward Algorithm
forward = function(obs_path){
  h_path = vector(length = mc.length)
  A_prob = matrix(0, nrow = S.h.no, ncol = mc.length)

  A_prob[, 1] = pi.initial * emis[, obs_path[1]]
  A_prob[, 1] = A_prob[, 1] / sum(A_prob[, 1])
  h_path[1] = S.h[which(A_prob[, 1] == max(A_prob[, 1]))]

  for (j in 2:mc.length){
    for (i in 1:S.h.no){
      A_prob[i, j] = emis[i, obs_path[j]] * sum(tpm.h[, i] * A_pr
ob[, j-1])
    }
    A_prob[, j] = A_prob[, j] / sum(A_prob[, j])
    h_path[j] = S.h[which(A_prob[, j] == max(A_prob[, j]))][1]
  }

  return (list("prob" = A_prob, "path" = h_path))
}
```

Estimating the hidden states for all iterations, we get the following Accuracy of estimate :

```
hidden.data.fest = matrix(1, nrow = mc.length, ncol = mc.iter)
for (j in 1:mc.iter){
  hidden.data.fest[,j] = forward(obs.data[,j])[["path"]]
}

# Accuracy
1 - sum(abs(hidden.data - hidden.data.fest))/(mc.length*mc.iter)
```

```
## [1] 0.804878
```