R Notebook to illustrate the Metropolis–Hastings algorithm

Contents

1 Binomial likelihood and Zellner's prior

Assume that we have an observation Y from a Binomial distribution with parameter $n = 20$ and success probability p:

$$Y \sim \text{Bin}(20, p)$$

Further assume that we observed $y = 12$ and what to make inference about the value of p using Zellner's prior, i.e. we want to use the following PDF as prior for p:

$$f(p) = cp^p(1-p)^{(1-p)}$$

where c is such that $\int_0^1 f(p)\,dp = 1$. We can determine this constant by numeric integration in R:

```
integrand <- function(p) p^p * (1-p)^(1-p)
res <- integrate(integrand, lower=0, upper=1)
res
```

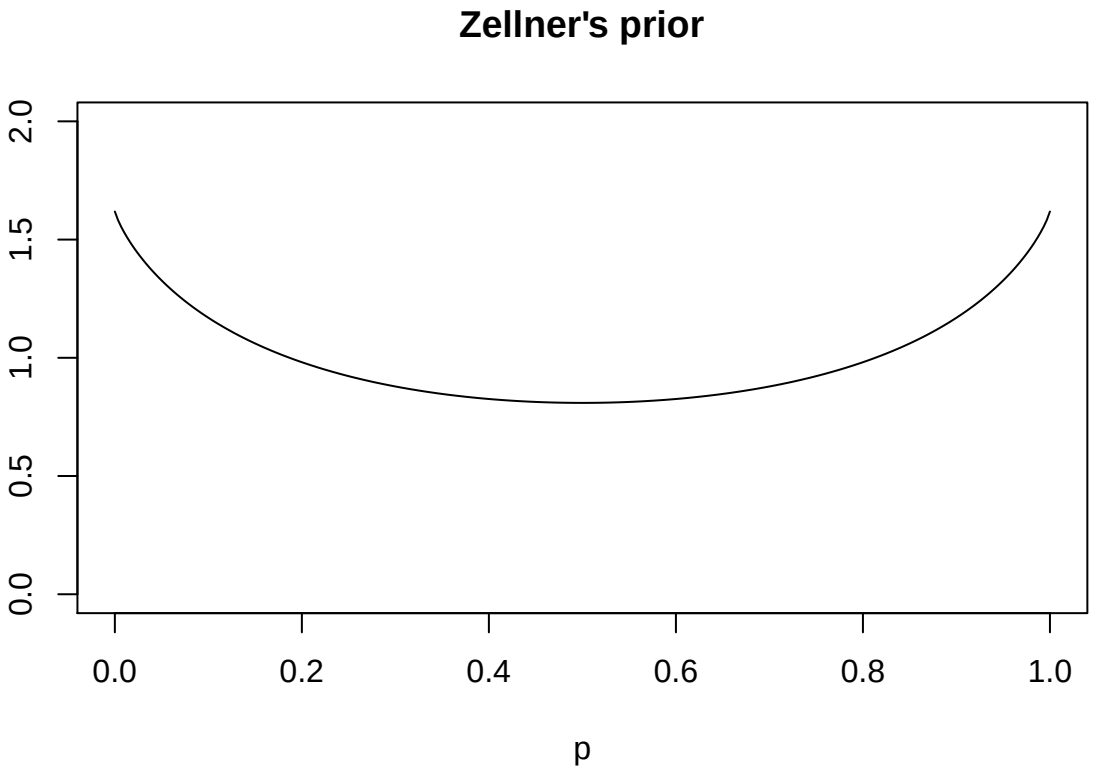
```
## 0.6178269 with absolute error < 5.5e-05
```

```
(c <- 1/res$val)
```

```
## [1] 1.618576
```

And plot the prior:

```
curve(c * x^x * (1 - x)^(1 - x), from = 0, to = 1, n = 301, ylim = c(0, 2), ylab = "",
    xlab = "p", main = "Zellner's prior")
```



Given that we have observed $y = 12$, the likelihood $L(p|y)$ is

$$L(p|y) = \binom{20}{12} p^{12}(1-p)^8$$

The posterior of p is

$$f(p|y) \propto L(p|y)f(p) = \binom{20}{12} p^{12}(1-p)^8 c p^p (1-p)^{(1-p)}$$

whence

$$f(p|y) \propto p^{12+p}(1-p)^{9-p}$$

2 Exploring the posterior using numeric integration

Again, we could find the normalising constant for this density by numerical integration:

```
integrand <- function(p) p^(12+p) * (1-p)^(9-p)
res1 <- integrate(integrand, lower=0, upper=1)
res1
```

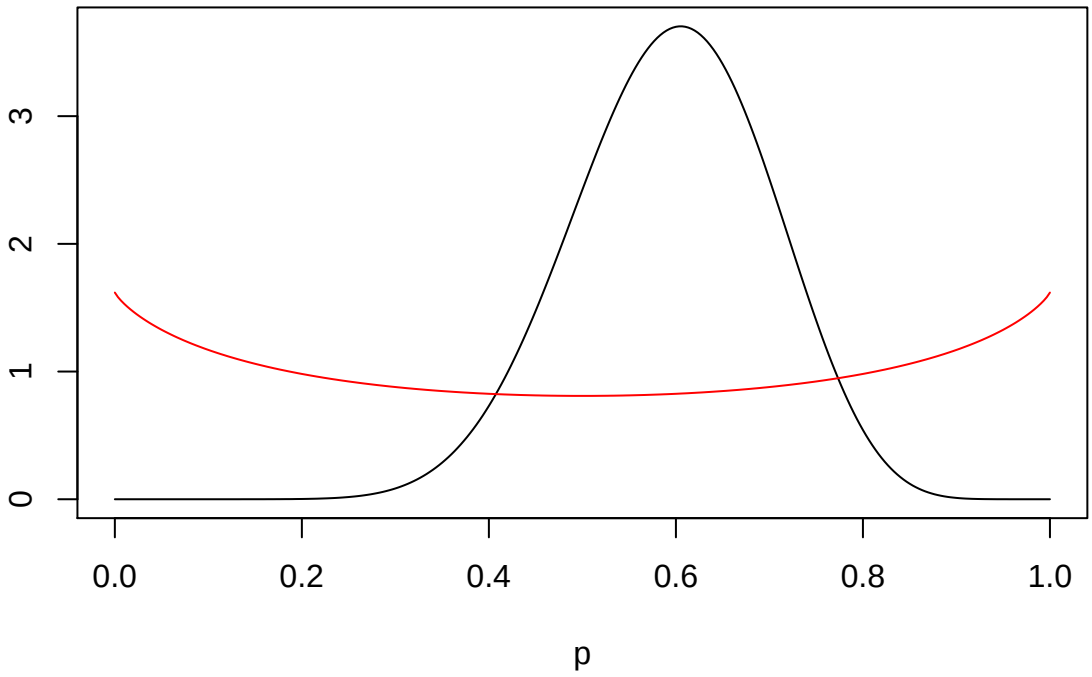
```
## 1.967116e-07 with absolute error < 2.1e-12
```

```
(c1 <- 1/res1$val)
```

```
## [1] 5083585
```

And plot the posterior and the prior:

```
curve(c1 * x^(12+x)*(1-x)^(9-x), from=0, to=1, n=301, ylab="", xlab="p",
      main="Zellner's prior and posterior for p")
curve(c * x^x * (1-x)^(1-x), from=0, to=1, n=301, col="red", add=TRUE)
```



We see that the posterior distribution $f(p|y)$ is quite different from the prior distribution $f(p)$, indicating that the posterior is mainly determined by the likelihood.

We could also use numerical integration to calculate the posterior mean and posterior variance of p. The following calculates $\mathbb{E}[p|y]$, the expectation of the posterior distribution:

```
integrand <- function(p) c1 * p * p^(12+p) * (1-p)^(9-p)
(resEp <- integrate(integrand, lower=0, upper=1))
```

```
## 0.5947345 with absolute error < 1.1e-05
```

```
Ep <- resEp$val
```

Now we calculate $\mathbb{E}[p^2|y]$, from which we can determine $\text{Var}[p|y] = \mathbb{E}[p^2|y] - (\mathbb{E}[p|y])^2$, the variance of the posterior distribution:

```
integrand <- function(p) c1 * p^2 * p^(12+p) * (1-p)^(9-p)
(resEp2 <- integrate(integrand, lower=0, upper=1))
```

```
## 0.3645911 with absolute error < 2.6e-07
```

```
Ep2 <- resEp2$val
(Varp <- Ep2 - Ep*Ep)
```

```
## [1] 0.01088195
```

Finally, we can calculate the standard deviation of the posterior:

```
(Sdp <- sqrt(Varp))
```

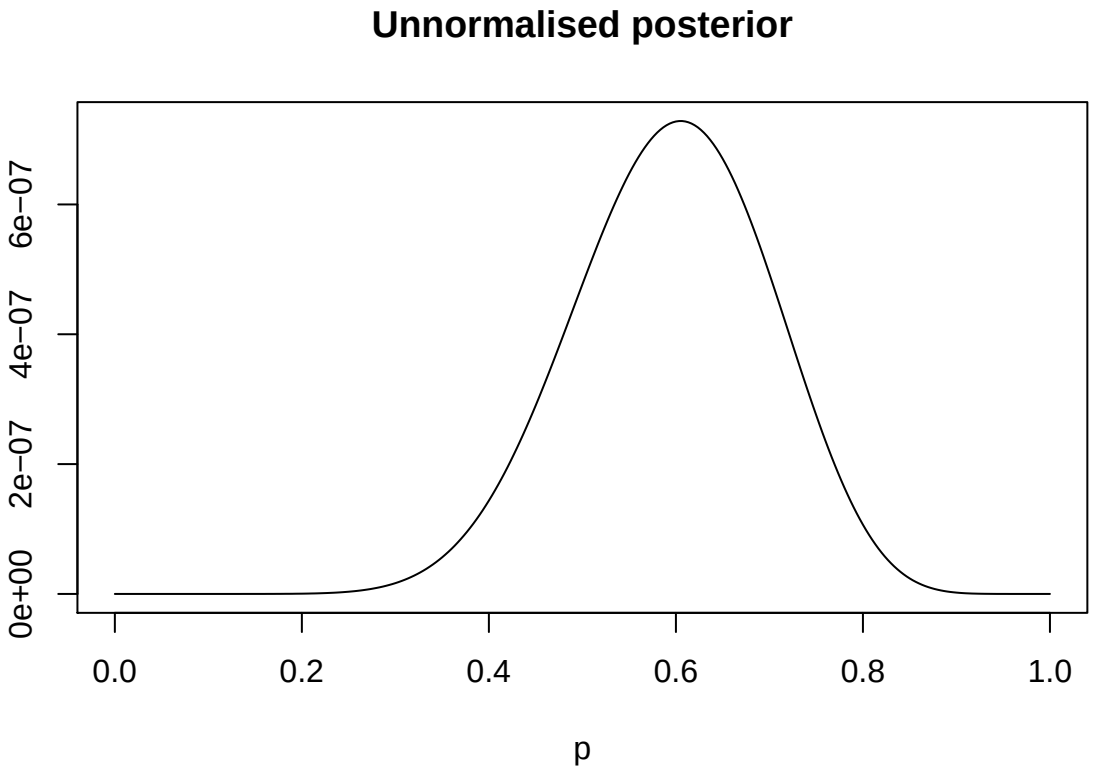
```
## [1] 0.1043166
```

3 Exploring the posterior using simulation

While in this example we can calculate the normalising constant for the posterior using numeric integration, and also calculate some moments of the posterior in the same manner, this is not possible in more complicated settings. In such settings we may have to use simulation techniques. Here we illustrate how to implement the Metropolis–Hastings algorithm for our running example to show how such simulation techniques can be implemented.

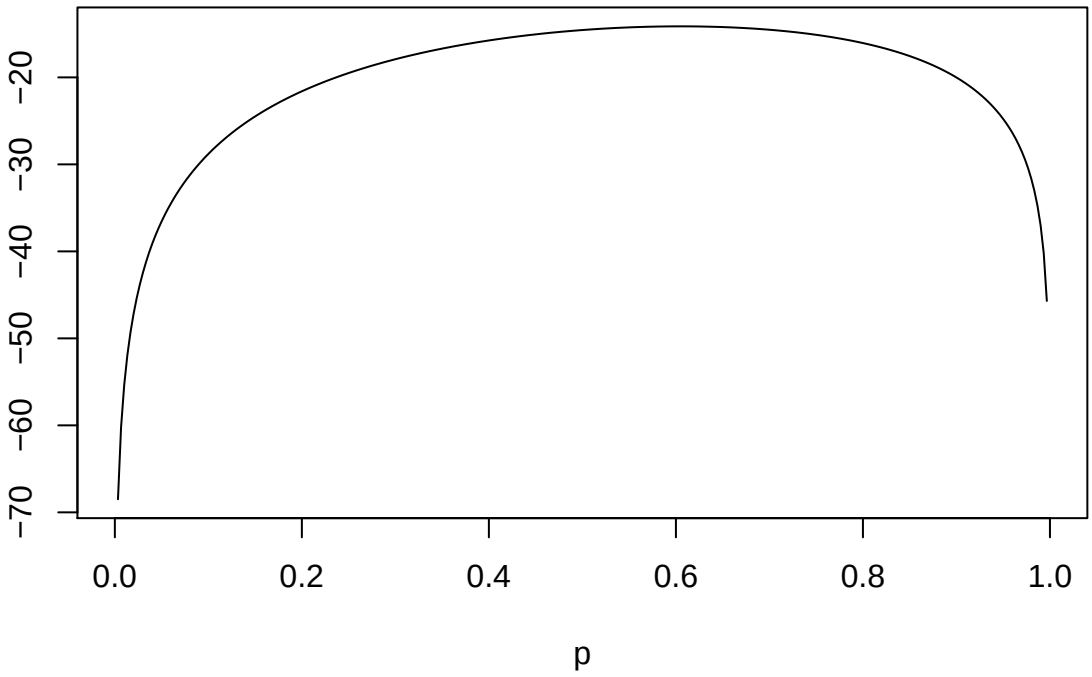
First, you should realise that posterior distributions, if the normalising constant is not known, can take quite small values:

```
curve(x^(12 + x) * (1 - x)^(9 - x), from = 0, to = 1, n = 301, ylab = "", xlab = "p",
    main = "Unnormalised posterior")
```



Thus, it is preferable to work with them on a logarithmic scale. The following function implements the log-posterior distribution (ignoring the normalising constant):

```
log.posterior <- function(p) (12+p)*log(p) + (9-p)*log(1-p)
curve(log.posterior(x), from=0, to=1, n=301, ylab="", xlab="p", main="log posterior")
```



Assume we want to generate 10,000 observations from this posterior. We start at $p^{(0)} = 0.5$ and use as proposal distribution $q(x|y)$ the uniform distribution on [0,1]. Note

- it is very easy to simulate from this proposal distribution.
- this proposal distribution does not depend on the current point $p^{(t-1)}$ at which the Markov chain is. I.e., the proposal $x^{(t)}$ is always sampled from $q\left(\cdot|p^{(t-1)}\right) = q\left(\cdot\right)$ for all t.
- in fact, $q(x|p) = 1$ for all $p, x \in [0, 1]$. This simplifies the calculation of the acceptance probability a lot since $\frac{q\left(p^{(t-1)}|x^{(t)}\right)}{q\left(x^{(t)}|p^{(t-1)}\right)} = 1$ and, of course, $\log(1) = 0$.

Thus, we can implement the Metropolis-Hastings algorithm as follows:

```
B <- 10000            ## number of realisations we want to have
chain <- rep(0, B+1)  ## vector to hold realisations
chain[1] <- 0.5       ## initial value
num.accept <- 0       ## keep track on how often we accept proposals
for(i in 1:B){
  ptm1 <- chain[i]      ## current point
  xt <- runif(1)        ## proposal
  lapt <- log.posterior(xt)-log.posterior(ptm1)   ## acceptance probability on the log scale
  if( runif(1) <= exp(lapt) ){
    chain[i+1] <- xt    ## accept proposal if runif(1) is less or equal to the acceptance probility
    num.accept <- num.accept + 1 ## proposal was accepted
  }else
    chain[i+1] <- ptm1  ## reject proposal
}
```

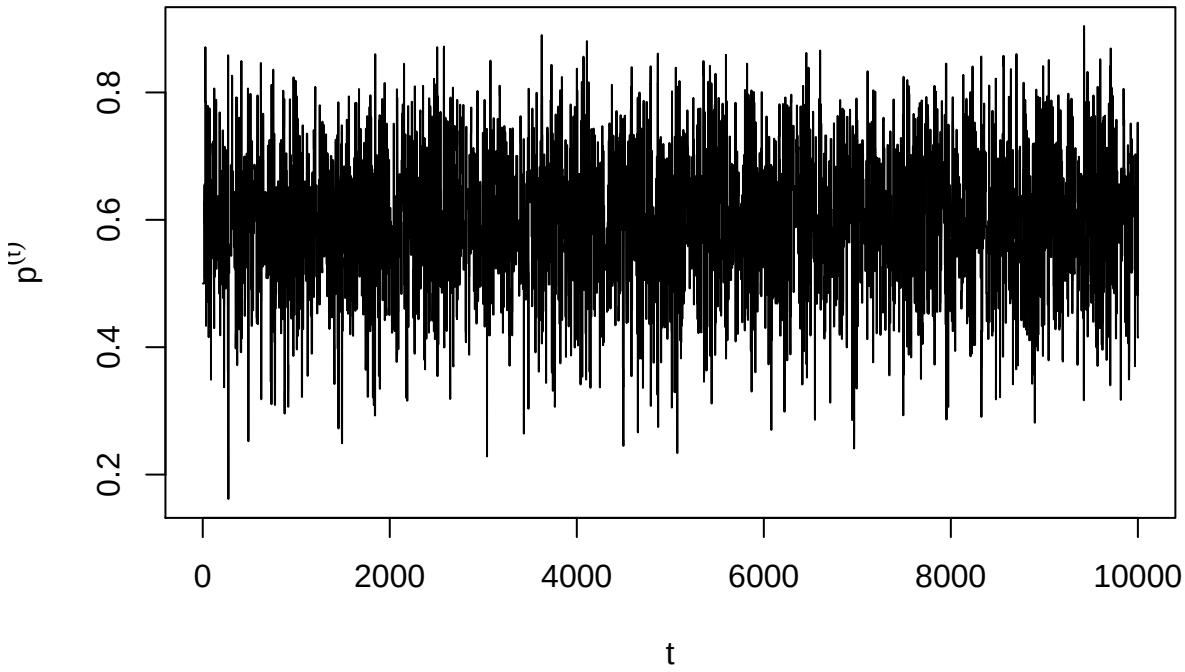
How often do we accept proposals:

```
num.accept/B
```

```
## [1] 0.3428
```

This looks a bit small, how does the chain look like:

```
plot(chain, type="l", xlab="t", ylab=expression(p^{(t)}))
```



Which does not look too bad. We can now estimate the mean and the standard deviation of the posterior distribution using these simulated values:

```
mean(chain)
```

```
## [1] 0.5957708
```

```
sd(chain)
```

```
## [1] 0.1049569
```

Which compares quite well with the theoretical results from the previous section.

Additionally, it is now quite easy to find out an interval in which p falls with 95% probability (according to the posterior):

```
quantile(chain, c(0.025, 0.975))
```

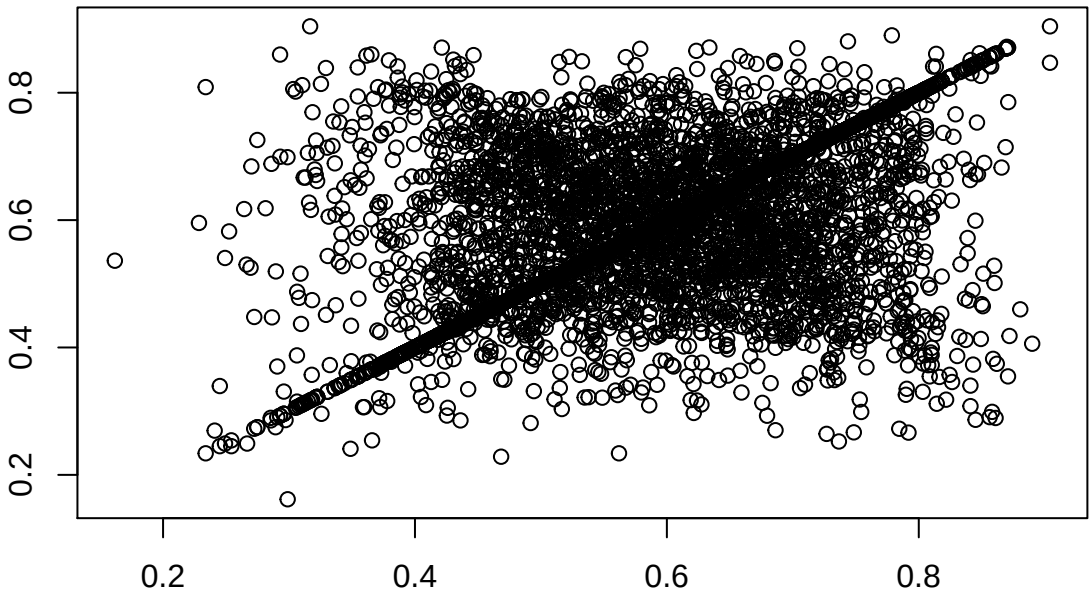
```
##      2.5%     97.5%
## 0.3861977 0.7927257
```

That is, according to our simulated realisations from the posterior distribution $P[0.39 \leq p \leq 0.79|y] = 95\%$.

3.1 Metropolis–Hastings creates correlated samples

An issue with MCMC samples is that the generated sequence $p^{(0)}, p^{(1)}, \ldots$ are correlated in general. Some chains tend to get stuck in certain regions of the parameter space, rather than exploring the whole space. If a chain can get stuck easily, then $p^{(t)}$ may be highly positively correlated with $p^{(t+1)}$. This is illustrated in the following figures, the first plots the pairs $\left(p^{(t)}, p^{(t+1)}\right)$ for $t = 0, 1, \ldots, 9999$:

```
ind <- 1:B
plot(chain[ind], chain[ind+1], xlab="", ylab="", main="Correlation at lag 1")
```



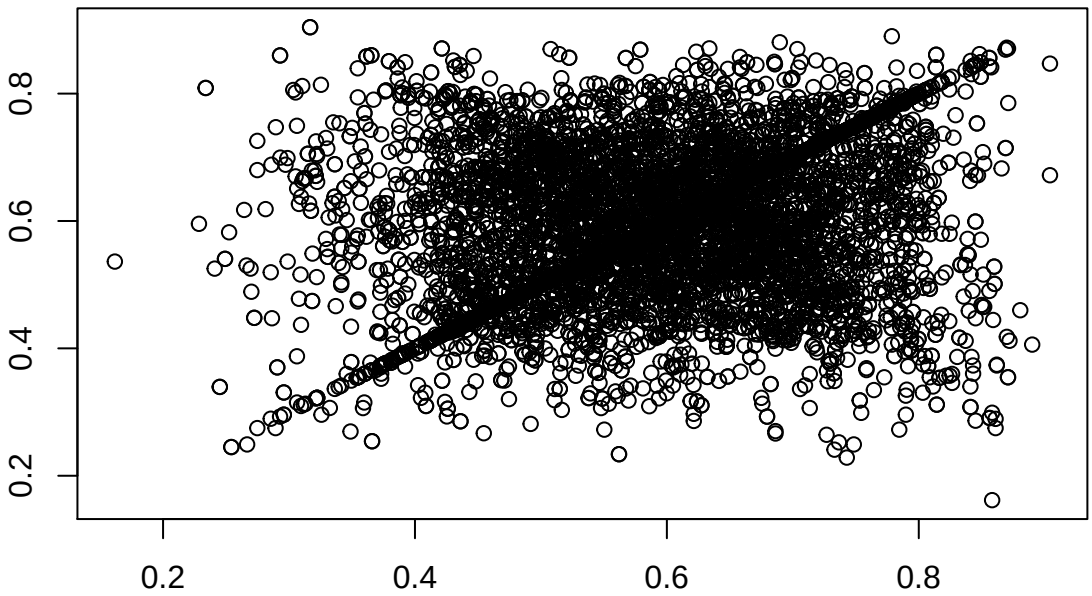
The cases when a proposal was rejected are clearly visible. The sample correlation for this lag is:

```
cor(chain[ind], chain[ind+1])
```

```
## [1] 0.5530362
```

At lag 2, i.e. plotting the pairs $\left(p^{(t)}, p^{(t+2)}\right)$ for $t = 0, 1, \ldots, 9998$, we have:

```
ind <- 1:(B-1)
plot(chain[ind], chain[ind+2], xlab="", ylab="", main="Correlation at lag 2")
```

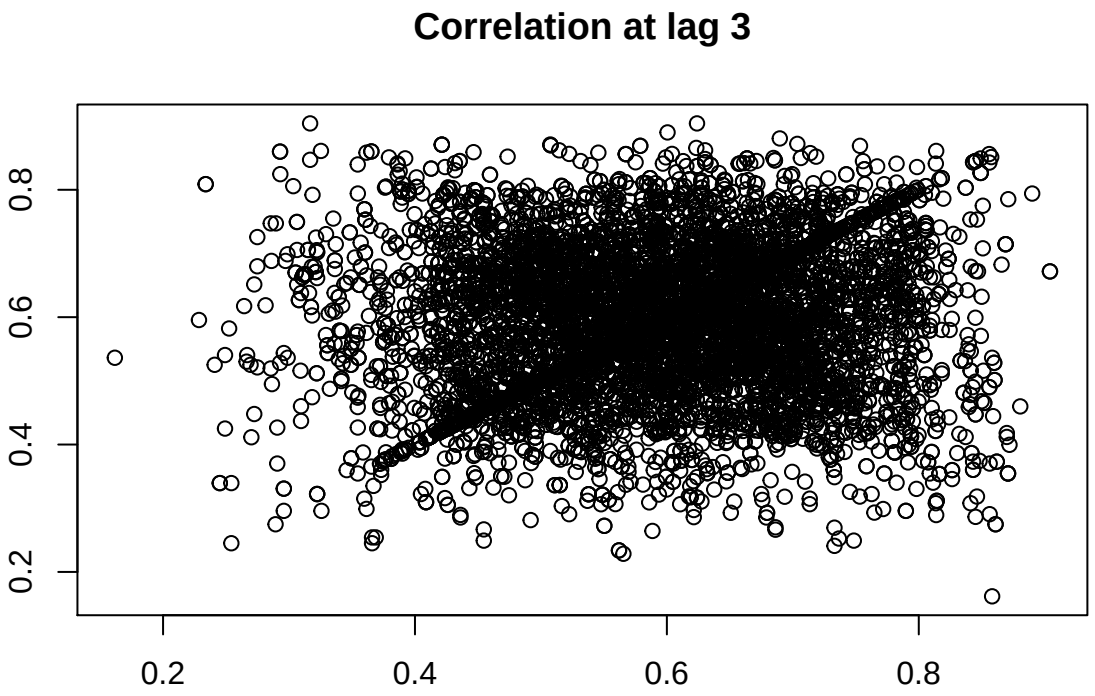


```
cor(chain[ind], chain[ind+2])
```

```
## [1] 0.3048248
```

At lag 3, i.e. plotting the pairs $\left(p^{(t)}, p^{(t+3)}\right)$ for $t = 0, 1, \ldots, 9997$, we have:

```
ind <- 1:(B-2)
plot(chain[ind], chain[ind+3], xlab="", ylab="", main="Correlation at lag 3")
```

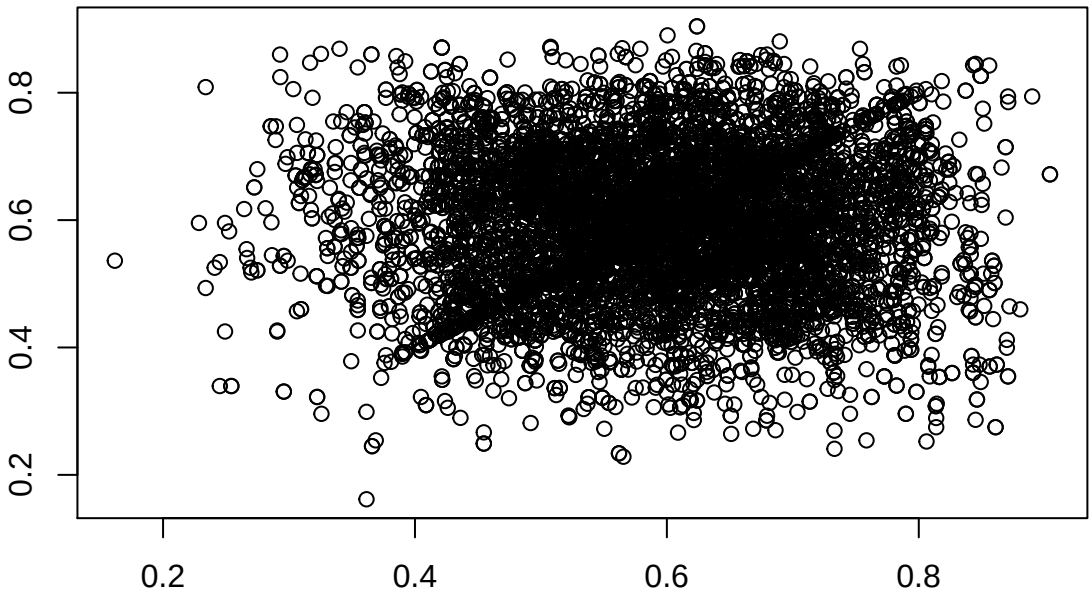



```
cor(chain[ind], chain[ind+3])
```

```
## [1] 0.1589444
```

At lag 4, i.e. plotting the pairs $\left(p^{(t)}, p^{(t+4)}\right)$ for $t = 0, 1, \ldots, 9996$, we have:

```
ind <- 1:(B-3)
plot(chain[ind], chain[ind+4], xlab="", ylab="", main="Correlation at lag 4")
```



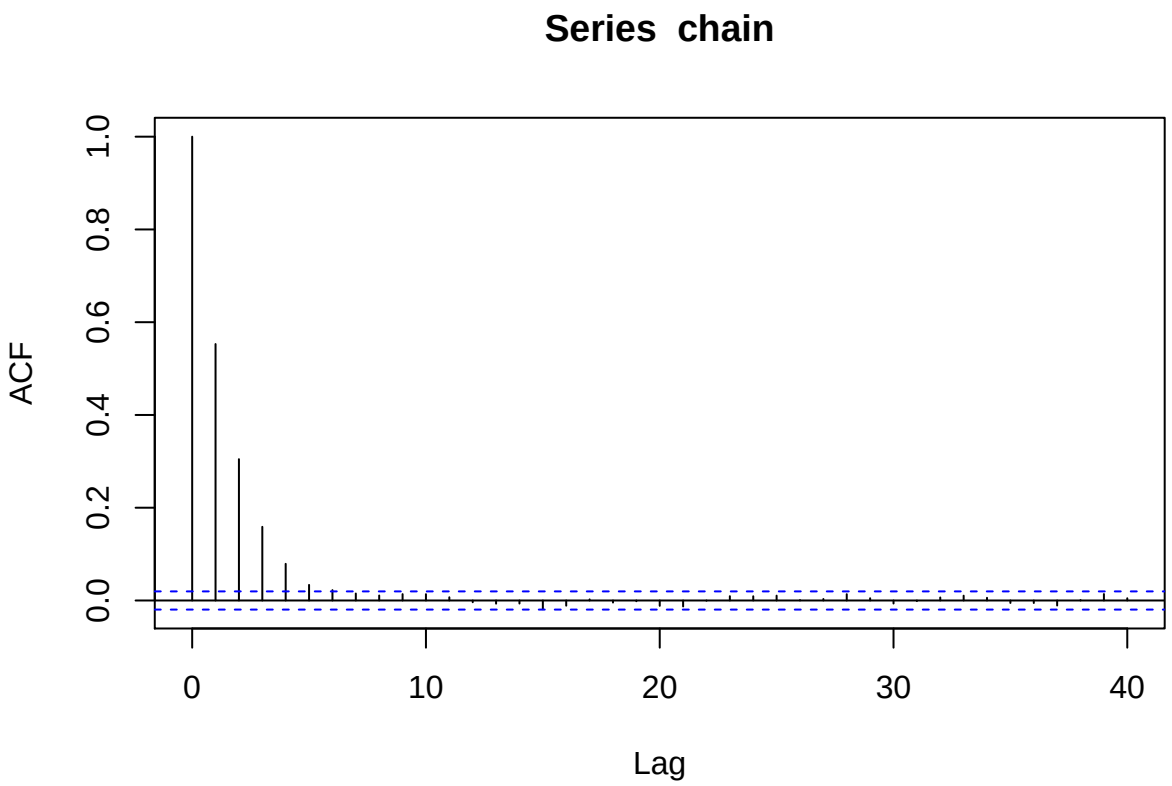

```
cor(chain[ind], chain[ind+4])
```

```
## [1] 0.07911605
```

In general, the *auto-correlation at lag k* is the correlation between $p^{(t)}$ and the value k steps later, $p^{(t+k)}$. It is desirable for the auto-correlation at lag k to approach 0 rapidly as k increases. High auto-correlation generally means high variances for Monte Carlo approximations.

Given that the calculation and graphing of auto-correlations at various lags is a standard task in *time series* analysis, it is not surprising that `R` has a command to calculate and plot all auto-correlations in one go:

```
acf(chain)
```


Series chain
ACF
0.0
0.2
0.4
0.6
0.8
1.0
0
10
20
30
40
Lag