Portfolio 2 (Extended) - Integrating R and C++

Sam Bowyer

2023-05-29

Typically, interfacing `R` with `C++` is done through `Rcpp`. In the final section of this extended portfolio we'll use `Rcpp`, however, first we'll look into how we can use the raw `C` API given by `R` to give us a better sense of what `Rcpp` is doing.

Simulation-based inference on the Ricker model

First we will be considering inference on the Ricker model, a simple model for population dynamics:

$$y_{t+1} = r y_t e^{-y_t}$$

where $y_t > 0$ is the population size at time t and $r > 0$ is the growth rate.

Below we present an implementation of the Ricker model in `R`, for an initial population size `y0` running for `n` time steps after an initial `nburn` time steps which are discarded.

```r
rickerSimulR <- function(n, nburn, r, y0 = 1){
  y <- numeric(n)
  yx <- y0

  # Burn in phase
  if(nburn > 0){
    for(ii in 1:nburn){
      yx <- r * yx * exp(-yx)
    }
  }

  # Simulating and storing
  for(ii in 1:n){
    yx <- r * yx * exp(-yx)
    y[ii] <- yx
  }

  return(y)
}
```

Question 1

We can write a version of the above model in `C` as follows:

```
cat ./rickerSimul.c
```

```
## #include <R.h>
## #include <Rinternals.h>
## #include <Rmath.h>
##
```

```
## SEXP rickerSimul(SEXP num, SEXP numburn, SEXP rate, SEXP initialPop){
##     double *xys;
##     int n, nburn;
##     double r, y0;
##     SEXP ys;
##
##     n = INTEGER(num)[0];
##     ys = PROTECT(allocVector(REALSXP, n));
##     xys = REAL(ys);
##
##     nburn = INTEGER(numburn)[0];
##     r = REAL(rate)[0];
##     y0 = REAL(initialPop)[0];
##
##     double yx = y0;
##
##     // Burn in phase
##     if(nburn > 0){
##       for(int i = 0; i < nburn; i++){
##         yx = r * yx * exp(-yx);
##       }
##     }
##
##     // Simulating and storing
##     for(int i=1; i < n; i++){
##       yx = r * yx * exp(-yx);
##       xys[i] = yx;
##     }
##
##     UNPROTECT(1);
##
##     return ys;
##   }
```

Then we compile it with in `R` with the following line

```
system("R CMD SHLIB rickerSimul.c")
```

This has created two files, a `.o` and a `.so` file.

```
ls rickerSimul.*
```

```
## rickerSimul.c
## rickerSimul.o
## rickerSimul.so
```

We then load the `.so` file into `R` and call it using `.Call`

```
dyn.load("rickerSimul.so")
is.loaded("rickerSimul")
```

```
## [1] TRUE
```

```
n = 25L
nburn=5L
r = 5
y0 = 4
c_output = .Call("rickerSimul", n, nburn, r, y0)
```

```
c_output
```

```
##  [1] 4.658982e-310  1.569201e+00  1.633629e+00  1.594584e+00  1.618446e+00
##  [6]  1.603932e+00  1.612787e+00  1.607394e+00  1.610682e+00  1.608679e+00
## [11]  1.609900e+00  1.609156e+00  1.609610e+00  1.609333e+00  1.609502e+00
## [16]  1.609399e+00  1.609462e+00  1.609423e+00  1.609447e+00  1.609433e+00
## [21]  1.609441e+00  1.609436e+00  1.609439e+00  1.609437e+00  1.609438e+00
```

And importantly we can see that this has produced the same results as the `R` version:

```
r_output = rickerSimulR(n, nburn, r, y0)
max(abs(c_output - r_output))
```

```
## [1] 1.569201
```

But importantly, the `C` version is much faster than the `R` version.

```
rickerSimulC_ <- function() .Call("rickerSimul", 100L, 20L, 10, 1)
rickerSimulR_ <- function() rickerSimulR(100L, 20L, 10, 1)

library(microbenchmark)
microbenchmark(rickerSimulC_(), rickerSimulR_(), times=10000)
```

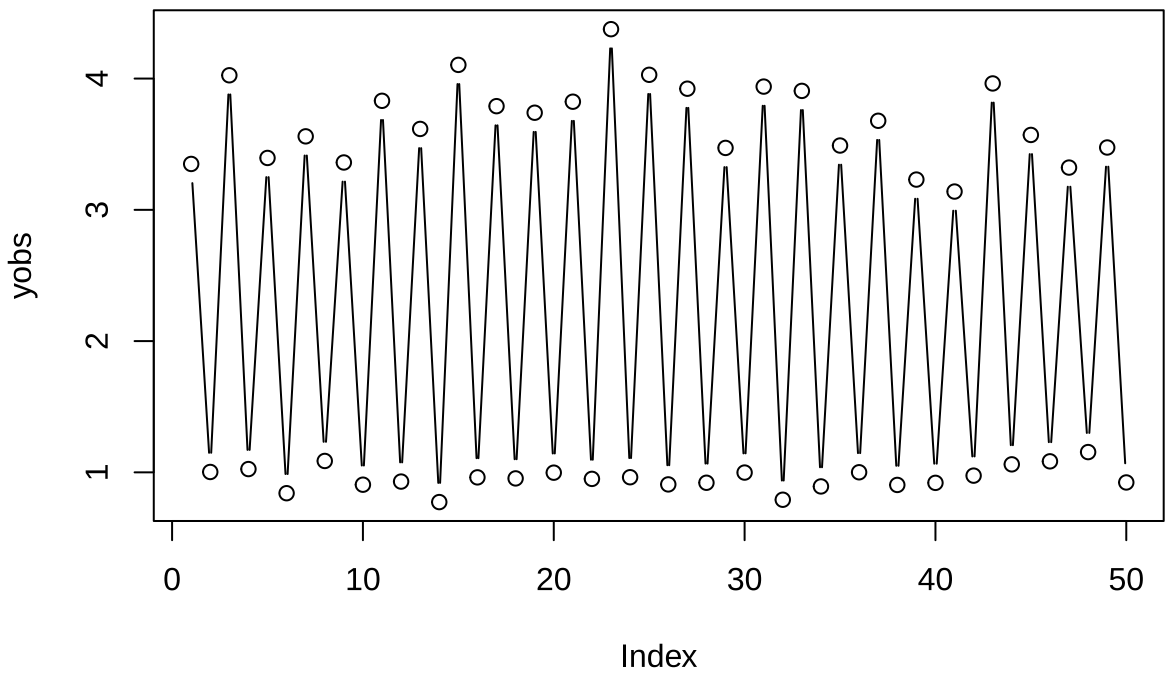
```
## Unit: microseconds
##             expr   min    lq     mean median     uq     max neval
##  rickerSimulC_() 1.276 1.322 1.685081  1.374 1.6155 552.512 10000
##  rickerSimulR_() 5.123 5.442 6.009669  5.590 6.2580 352.197 10000
```

Question 2

Now suppose we have noisy observations from the Rocker model:

$$z_t = y_t e^{\epsilon_t} \text{ where } \epsilon_t \sim N(0, \sigma^2)$$

```
nburn <- 100L
n <- 50L

y0_true <- 1
sig_true <- 0.1
r_true <- 10

Ntrue <- rickerSimulR(n = n, nburn = nburn, r = r_true, y0 = y0_true)
yobs <- Ntrue * exp(rnorm(n, 0, sig_true))

plot(yobs, type = 'b')
```



We then write the following function in `C` to calculate the log likelihood of the data (this function is in the file `rickerLLK.c`):

```
cat rickerLLK.c
```

```
## #include <R.h>
## #include <Rinternals.h>
## #include <Rmath.h>
##
## SEXP rickerLLK(SEXP observed, SEXP simulated, SEXP sigma){
##     double *yobs, *ysim, sig, *lik;
##     int n;
##
##     SEXP LLK;
##
##     yobs = REAL(observed);
##     ysim = REAL(simulated);
##     sig  = REAL(sigma)[0];
##     n = length(observed);
##
##     LLK = PROTECT(allocVector(REALSXP, 1));
##
##     lik = REAL(LLK);
##     lik[0] = 0;
##
##     for (int i = 1; i < n; i++){
##         lik[0] = lik[0] + dnorm(log(yobs[i]/ysim[i]), 0, sig, 1);
##     }
##
##     UNPROTECT(1);
##     return LLK;
## }
```

```
system("R CMD SHLIB rickerLLK.c")
dyn.load("rickerLLK.so")
is.loaded("rickerLLK")
```

```
## [1] TRUE
```

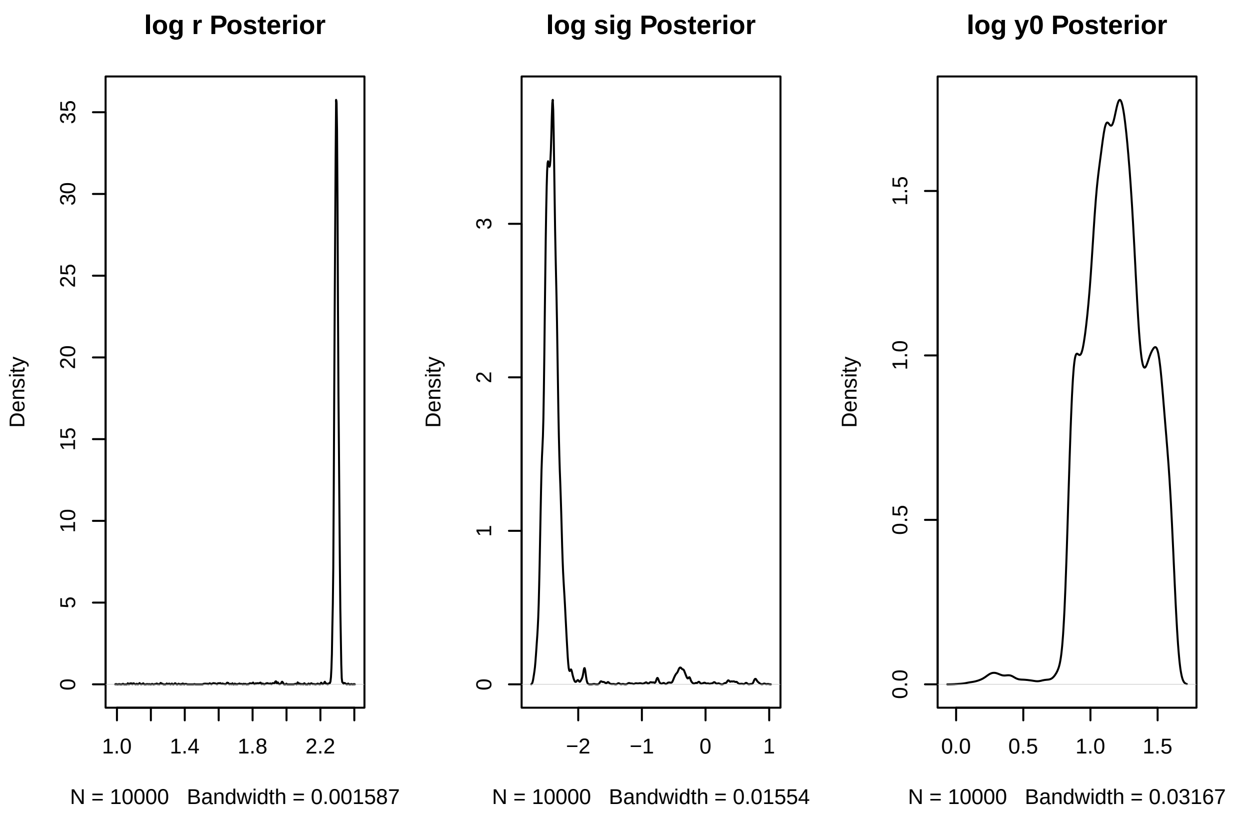
Next we wrap the likelihood calculation in an R function that takes in the logarithm of `r`, `sig` and `y0` as well as `yobs` and `nburn`:

```
myLikR <- function(logr, logsig, logy0, yobs, nburn){
  n <- length(yobs)
  r <- exp(logr)
  sig <- exp(logsig)
  y0 <- exp(logy0)

  ysim <- .Call("rickerSimul", n, nburn, r, y0)

  llk <- .Call("rickerLLK", yobs, ysim, sig)

  return( llk )
}

myLikR(log(r_true), log(sig_true), log(y0_true), yobs, nburn)
```

```
## [1] -4482.756
```

With this, we can then sample from the posterior distribution of $\log(r), log(\sigma), \log(y_0)$ by providing `myLikR` to a Metropolis-Hastings algorithm (using the `metrop` function):

```
library(mcmc)
par(mfrow=c(1,3))
samples = metrop(function(params) myLikR(params[1], params[2], params[3], yobs, nburn),
                 initial = c(1,1,1),
                 nbatch = 10000,
                 scale = 0.05)
plot(density(samples$batch[,1]), main="log r Posterior")
plot(density(samples$batch[,2]), main="log sig Posterior")
plot(density(samples$batch[,3]), main="log y0 Posterior")
```



Recall the true log values of the parameters:

```
log(r_true); log(sig_true); log(y0_true)
```

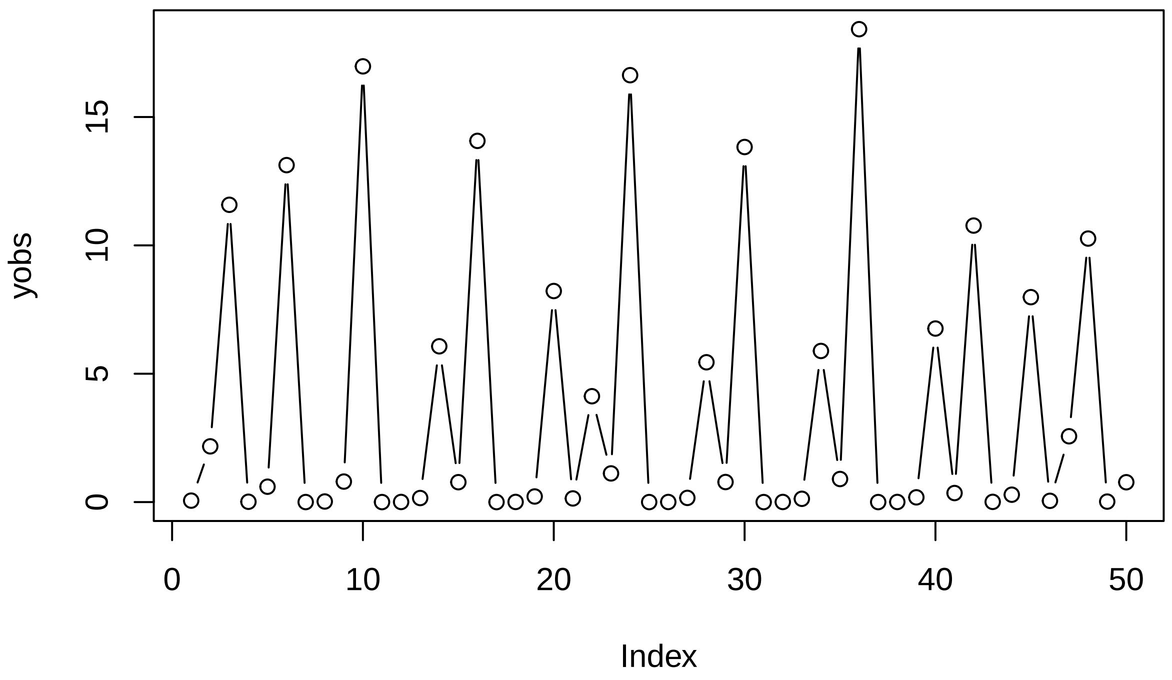
```
## [1] 2.302585

## [1] -2.302585

## [1] 0
```

We see that we've got sharp posteriors around the correct values for `r` and `sig`, however, a much broader posterior for `y0` that isn't very close to the true value of 0. This is because although `r` and `sig` have a great impact on every observation, as the simulation goes on for many samples, the impact of the initial state, `y0`, drastically decreases, meaning it is harder to infer from the data.

Question 3

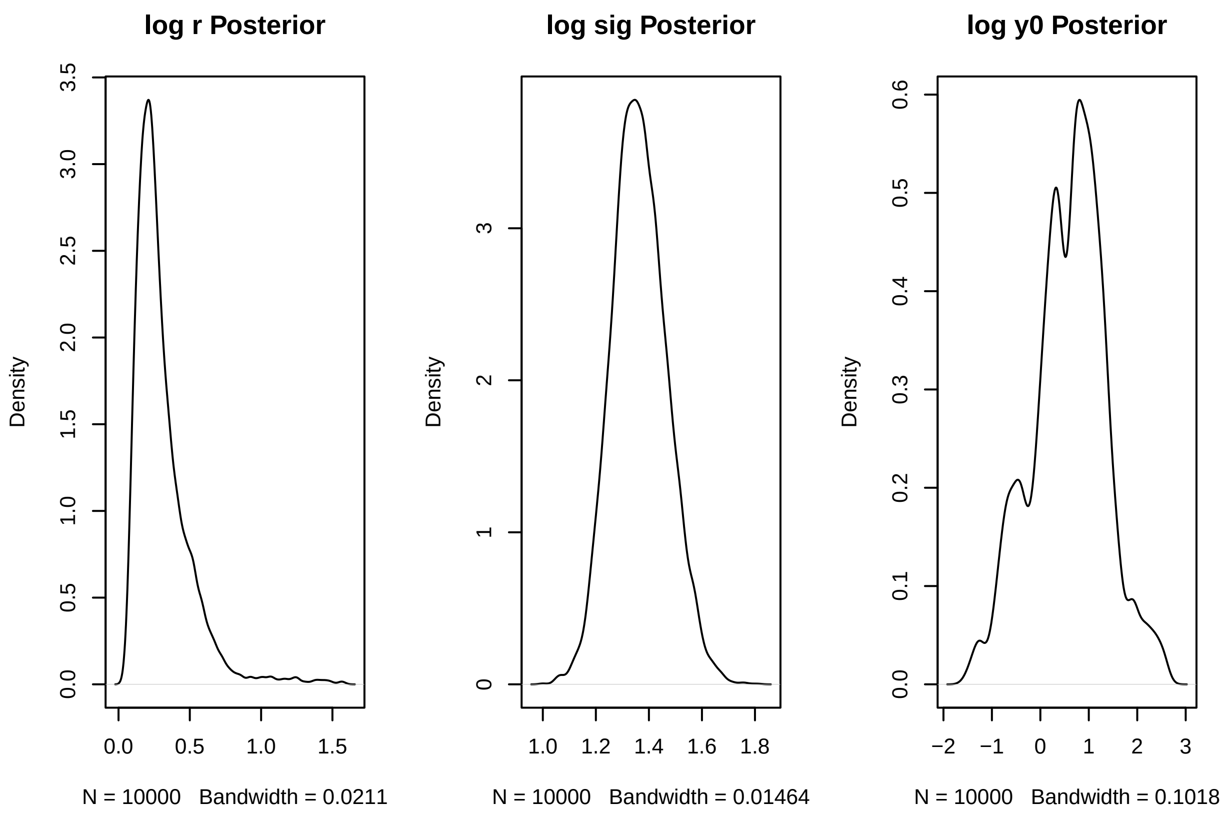
Now assume that the data has been simulated as follows:

```
r_true <- 44

Ntrue <- rickerSimulR(n = n, nburn = nburn, r = r_true, y0 = y0_true)
yobs <- Ntrue * exp(rnorm(n, 0, sig_true))

plot(yobs, type = 'b')
```



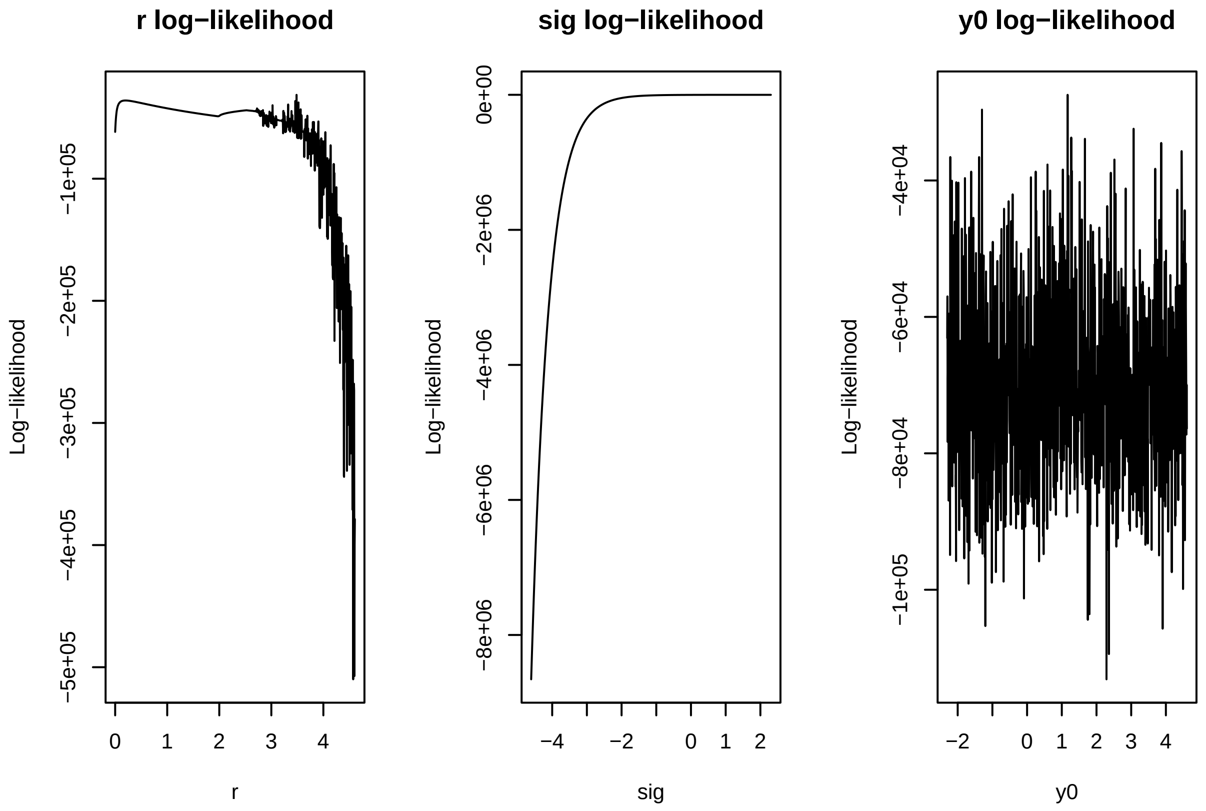
Attempting to run the MH algorithm as before, we notice that the chain is failing to mix properly, resulting in broader, less accurate posteriors:

```
par(mfrow=c(1,3))
samples = metrop(function(params) myLikR(params[1], params[2], params[3], yobs, nburn),
                 initial = c(1,1,1),
                 nbatch = 10000,
                 scale = 0.05)
plot(density(samples$batch[,1]), main="log r Posterior")
plot(density(samples$batch[,2]), main="log sig Posterior")
plot(density(samples$batch[,3]), main="log y0 Posterior")
```



To investigate this further, we'll look at the slices of likelihood with respect to each parameter (keeping the other two parameters fixed at a time to their true values).

```
par(mfrow=c(1,3))

r_seq = seq(log(1),log(100), length.out=1000)
sig_seq = seq(log(0.01),log(10), length.out=1000)
y0_seq = seq(log(0.1),log(100), length.out=1000)

plot(r_seq, sapply(r_seq, function(x) myLikR(x, log(sig_true), log(y0_true), yobs, nburn)),
     main="r log-likelihood", xlab="r", ylab="Log-likelihood", type='l')
plot(sig_seq, sapply(sig_seq, function(x) myLikR(log(r_true), x, log(y0_true), yobs, nburn)),
     main="sig log-likelihood", xlab="sig", ylab="Log-likelihood", type='l')
plot(y0_seq, sapply(y0_seq, function(x) myLikR(log(r_true), log(sig_true), x, yobs, nburn)),
     main="y0 log-likelihood", xlab="y0", ylab="Log-likelihood", type='l')
```



We see above that the log-likelihood for `r` gets very unpleasant for $r > \exp(3)$, meaning that the MH algorithm isn't able to mix well with the chaotic behaviour of the model for small changes in `r` and `y0`.

Question 4

To deal with this, we will now assume that we know the true value of σ and that $y_0 \sim \text{Unif}(1, 10)$ (but in particular we don't care about the value of y_0). We can come up with another wat to express the likelihood of our data by assuming that the sample mean s_1 and standard deviation s_2 are independently normally distributed:

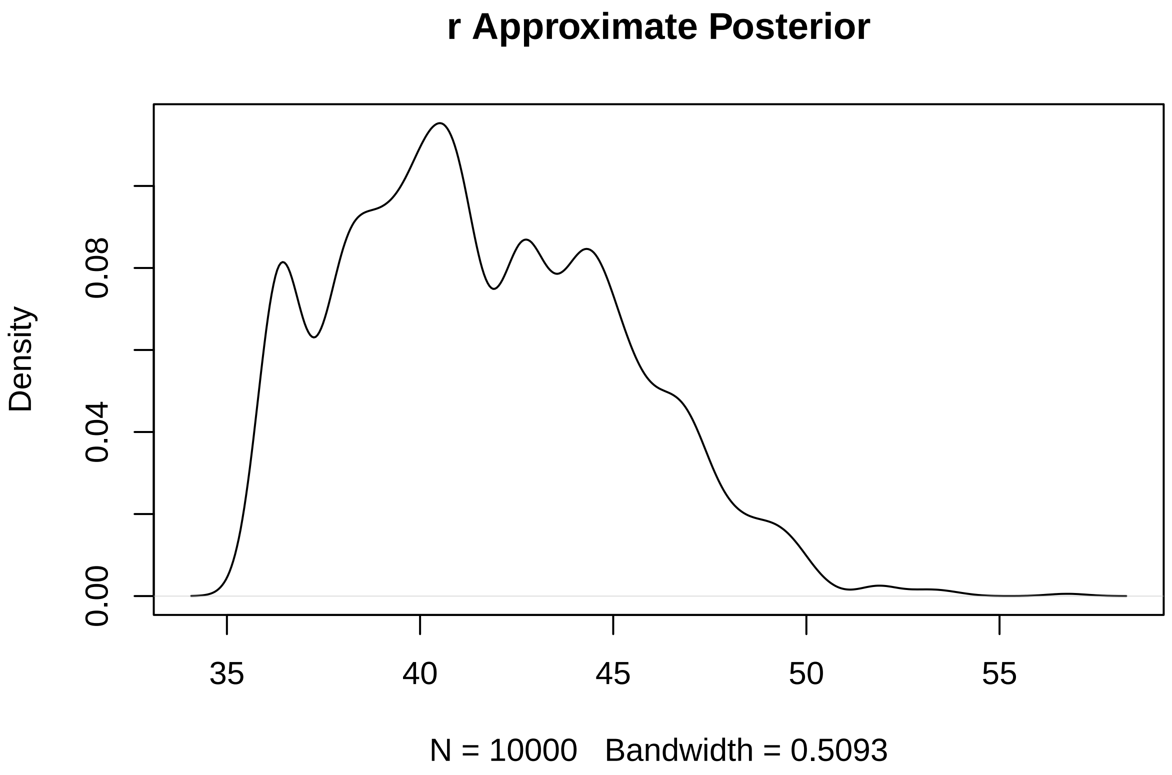
$$s_1 \sim \mathcal{N}(\mu_1, \tau_1^2), \quad s_2 \sim \mathcal{N}(\mu_2, \tau_2^2).$$

Then the desired likelihood $p(s_1, s_2|r)$ is simply the product of these two normal densities, where $\mu_1, \mu_2, \tau_1, \tau_2$ are functions of r. We can sample the corresponding posterior by simulation (`nsim` times for a given value of `logr`) via the following function:

```r
synllk <- function(logr, nsim){

  r <- exp(logr)
  s1 <- s2 <- numeric(nsim)
  y0 <- runif(nsim, 1, 10)

  # Note: sigma is assumed to be known!
  for(ii in 1:nsim){
    ysim <- rickerSimulR(n = n, nburn = nburn, r = r, y0 = y0[ii]) * exp(rnorm(n, 0, sig_true))
    s1[ii] <- mean(ysim)
    s2[ii] <- sd(ysim)
  }

  out <- dnorm(mean(yobs), mean(s1), sd(s1), log = TRUE) +
         dnorm(sd(yobs), mean(s2), sd(s2), log = TRUE)

  return( out )
}
```

Then using the same Metropolis-Hastings approach as before we can sampling from an estimate of the posterior:

```
samples = metrop(function(r) synllk(r, 100), initial=log(40), nbatch=10000)
plot(density(exp(samples$batch[,1])), main="r Approximate Posterior")
```



This does a much better job at estimating the true value of $r = 44$ (though this being an approximation of the posterior, it is somewhat broad and multimodal). However, the implementation of this synthetic likelihood in `R` is very slow, and so below we provide an implementation of `synllk` and `rickerSimul` using `Rcpp`, stored in a file `rickerRcpp.cpp`.

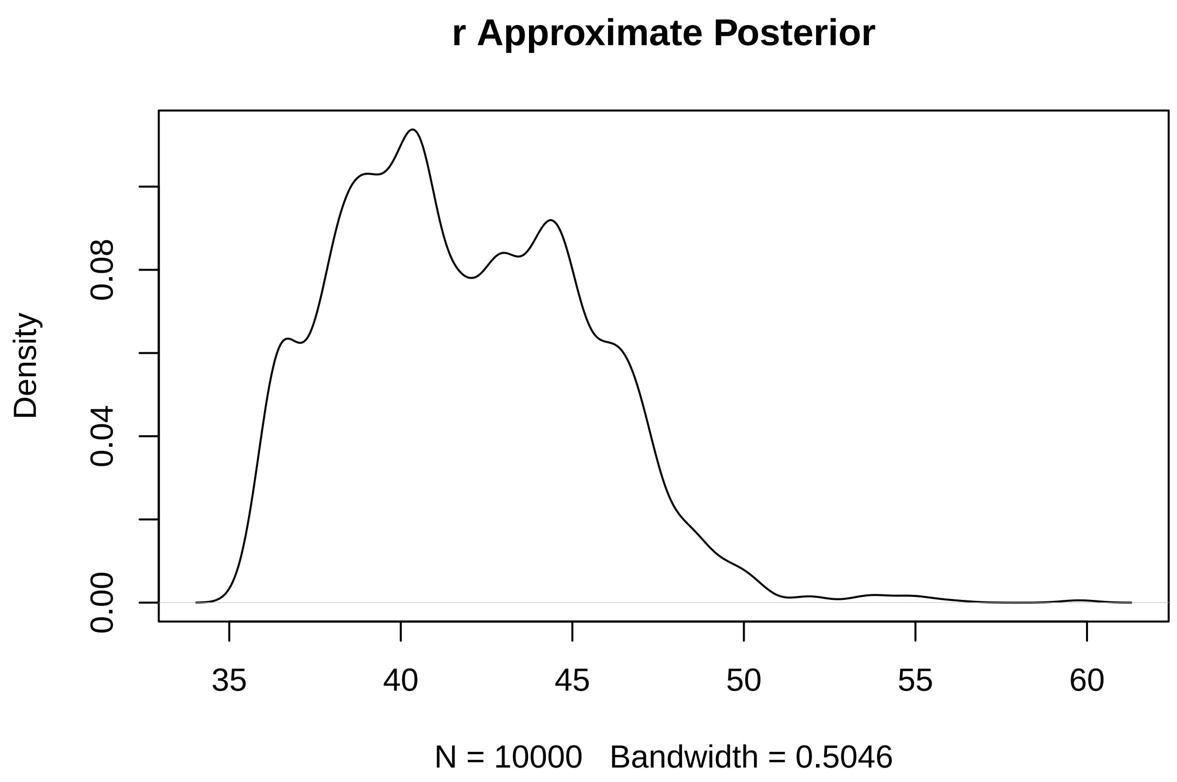
```
cat rickerRcpp.cpp
```

```
## #include <Rcpp.h>
## using namespace Rcpp;
##
## // [[Rcpp::export]]
## NumericVector rickerSimul_Rcpp(const int n, const int nburn, const double r, const double y0){
##     // vector to return
##     NumericVector y(n);
##
##     double yx = y0;
##
##     // burn-in
##     if(nburn > 0){
##         for(int i=0; i<=nburn; i++){
```

```
##         yx = r * yx * exp(-yx);
##         }
##     }
##
##     // run simulation and store values
##     y[0] = yx;
##     for(int i=1; i<n; i++){
##         yx = r * yx * exp(-yx);
##         y[i] = yx;
##     }
##
##     return y;
## }
##
## // [[Rcpp::export]]
## NumericVector synllk_Rcpp(const double logr, const int nsim, const NumericVector yobs){
##     NumericVector ysim;
##     NumericVector s1(nsim), s2(nsim);
##     NumericVector y0(nsim);
##     NumericVector out;
##
##     double r = exp(logr);
##
##     y0 = runif(nsim, 0, 10);
##     for(int i=0; i<nsim; i++){
##         // assume we know sd = 0.1
##         ysim = rickerSimul_Rcpp(50, 100, r, y0[i]) * exp(rnorm(50, 0, 0.1));
##         s1[i] = mean(ysim);
##         s2[i] = sd(ysim);
##     }
##
##     // add log-likelihoods of our two summary statistics
##     out = R::dnorm(mean(yobs), mean(s1), sd(s1), true) + R::dnorm(sd(yobs), mean(s2), sd(s2), true);
##
##     return out;
## }
```

Then we can load this in `R` with the `sourceCpp` function:

```
library(Rcpp)
sourceCpp("rickerRcpp.cpp")
samples = metrop(function(r) synllk_Rcpp(r, 100, yobs), initial=log(40), nbatch=10000)
plot(density(exp(samples$batch[,1])), main="r Approximate Posterior")
```



As we can see, this produces similar results to before and, importantly, the `Rcpp` version of `synllk` is much faster than the pure `R` version:

```
synlkk_Rcpp_ <- function() synllk_Rcpp(r_true, 100, yobs)
synllk_R_ <- function() synllk(r_true, 100)
microbenchmark(synlkk_Rcpp_(), synllk_R_(), times=10000)
```

```
## Unit: microseconds
##            expr      min       lq      mean    median        uq       max neval
##  synlkk_Rcpp_()  186.476  200.056  222.4572  207.9165  227.5625  2832.971 10000
##     synllk_R_() 1453.666 1591.685 1751.5768 1640.6925 1706.5085 37708.934 10000
```

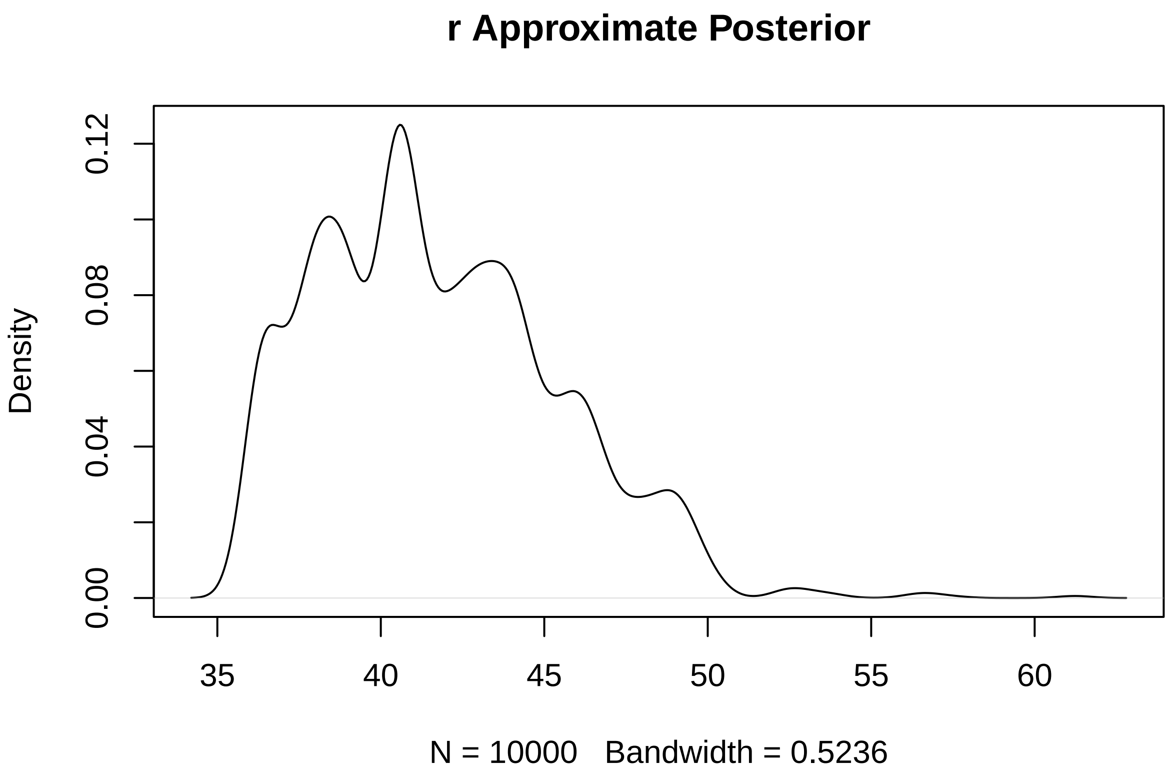
Question 5

To speed this up even further, we can write our own Metropolis-Hastings algorithm in `Rcpp` too and observe once again the same general results (note that the results won't be identical because of the stochasticity of simulations inside `synllk`).

```
cat mh.cpp
```

```
## #include <Rcpp.h>
## using namespace Rcpp;
##
## // [[Rcpp::export]]
## NumericVector metrop_Rcpp(Function lik, double init, int n, double scale){
##     double proposal;
##
##     NumericVector samples(n);
##     NumericVector alpha(n);
##
##     samples[0] = init;
##
##     for(int i=1; i<n; i++){
##         proposal = R::rnorm(samples[i-1], sqrt(scale));
##
##         alpha[i] = as<double>(lik(proposal))/as<double>(lik(samples[i-1]));
##
##         if(R::runif(0,1) < alpha[i]){
##             samples[i] = proposal;
##         }
##         else {
##             samples[i] = samples[i-1];
##         }
##     }
##
##     return samples;
## }
```

```r
sourceCpp("mh.cpp")
samples = metrop_Rcpp(function(r) exp(synllk_Rcpp(r, 100, yobs)), log(40), 10000, 1)
plot(density(exp(samples)), main="r Approximate Posterior")
```



But now we have the benefit of a significant speed-up in obtaining these results: running everything in R is around 5 times slower than the full Rcpp implementation, which itself is about 40% faster than using metrop with an Rcpp implementation of synllk.

```r
MH_R <- function() metrop(function(r) synllk(r, 100), initial=log(40), nbatch=50, scale=0.1)
MH_mix <- function() metrop(function(r) synllk_Rcpp(r, 100, yobs), initial=log(40), nbatch=50,
                            scale=0.1)
# Note below we exponentiate the likelihood function due to differences in the M-H implementation
MH_Rcpp <- function() metrop_Rcpp(function(r) exp(synllk_Rcpp(r, 100, yobs)), log(40), 50, 0.1)
microbenchmark(MH_R(), MH_mix(), MH_Rcpp(), times=100)
```

```
## Unit: milliseconds
##       expr       min        lq      mean    median        uq       max neval
##     MH_R() 115.46187 119.19204 121.40465 121.14991 122.51691 153.77629   100
##   MH_mix()  43.07378  46.44682  47.52050  47.53224  48.69843  53.08890   100
##  MH_Rcpp()  28.14810  28.76566  30.13895  29.88991  31.25612  33.43923   100
```

However, note that the above benchmark only ran the Metropolis-Hastings algorithms for 50 iterations. When we run this benchmark for a larger number of samples (below we use 500), the `R` implementation of `metrop` is faster than our `Rcpp` function `metrop_Rcpp`.

```r
MH_R <- function() metrop(function(r) synllk(r, 100), initial=log(40), nbatch=500, scale=0.1)
MH_mix <- function() metrop(function(r) synllk_Rcpp(r, 100, yobs), initial=log(40), nbatch=500,
                            scale=0.1)
# Note below we exponentiate the likelihood function due to differences in the M-H implementation
MH_Rcpp <- function() metrop_Rcpp(function(r) exp(synllk_Rcpp(r, 100, yobs)), log(40), 500, 0.1)
microbenchmark(MH_R(), MH_mix(), MH_Rcpp(), times=5)
```

```
## Unit: milliseconds
##       expr      min       lq     mean   median       uq      max neval
##     MH_R() 908.4620 909.7799 915.6838 909.8278 910.7625 939.5870     5
##   MH_mix() 175.5941 176.2796 177.9067 177.3838 180.0604 180.2158     5
##  MH_Rcpp() 302.2229 302.8542 304.3112 303.4769 305.8246 307.1774     5
```