Report 2 — Interfacing R with C++ via Rcpp

Jake Spiteri

04/02/2020

Exercises on chaotic maps and kernel regression smoothing

Simulation-based inference

In general it is preferred to use the `Rcpp` package to interface R with C++, but in this exercise we will demonstrate the use of R's C API.

We will use a simple model for population dynamics, the Ricker model:

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

where $y_t > 0$ represents the size of the population at time t, and $r > 0$ is the population's growth rate. The model can capture a wide variety of chaotic behavior depending on the value chosen for r.

Below is a simple R function which will allow us to simulate a trajectory from the map.

```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 )
}
```

where

- `y0` is the initial population size,
- `n` is is the total number of time steps to run the simulation for,
- `nburn` is the number of simulations to be discarded before storing the `n` iterations. The period $t = 1, \dots, n_{burn}$ is the burn-in period.

Question 1

We will create a C version of the above code in order to efficiently simulate the population trajectory. The C code can be seen below.

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

SEXP rickerSimul(SEXP n_, SEXP nburn_, SEXP r_, SEXP y0_)
{
  // create C objects
  int n, nburn;
  double r, yx, *y;

  // create SEXP object to return to R
  SEXP y_;

  // initialize C objects
  // use macro functions to point to R objects
  // we extract the value of first element with [0]
  n = INTEGER(n_)[0];
  nburn = INTEGER(nburn_)[0];
  r = REAL(r_)[0];
  yx = REAL(y0_)[0];

  // allocate memory for output
  y_ = PROTECT(allocVector(REALSXP, n));
  y = REAL(y_);

  // burn in
  if(nburn>0)
  {
    for(int i=0; i<=nburn; i++)
    {
      yx = r * yx * exp(-yx);
    }
  }

  // simulate for n time periods
  y[0] = yx;
  for(int i=1; i<n; i++)
  {
    yx = r * yx * exp(-yx);
    y[i] = yx;
  }

  UNPROTECT(1);

  return y_;
}
```

The above C code is saved in the same directory as this document, with name `rickerSimul.c`. Before we can call the function we need to compile the C code. We can do this as follows

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

This outputs a shared object `rickerSimul.so` which we can load into R by doing the following

```r
dyn.load("rickerSimul.so")
```

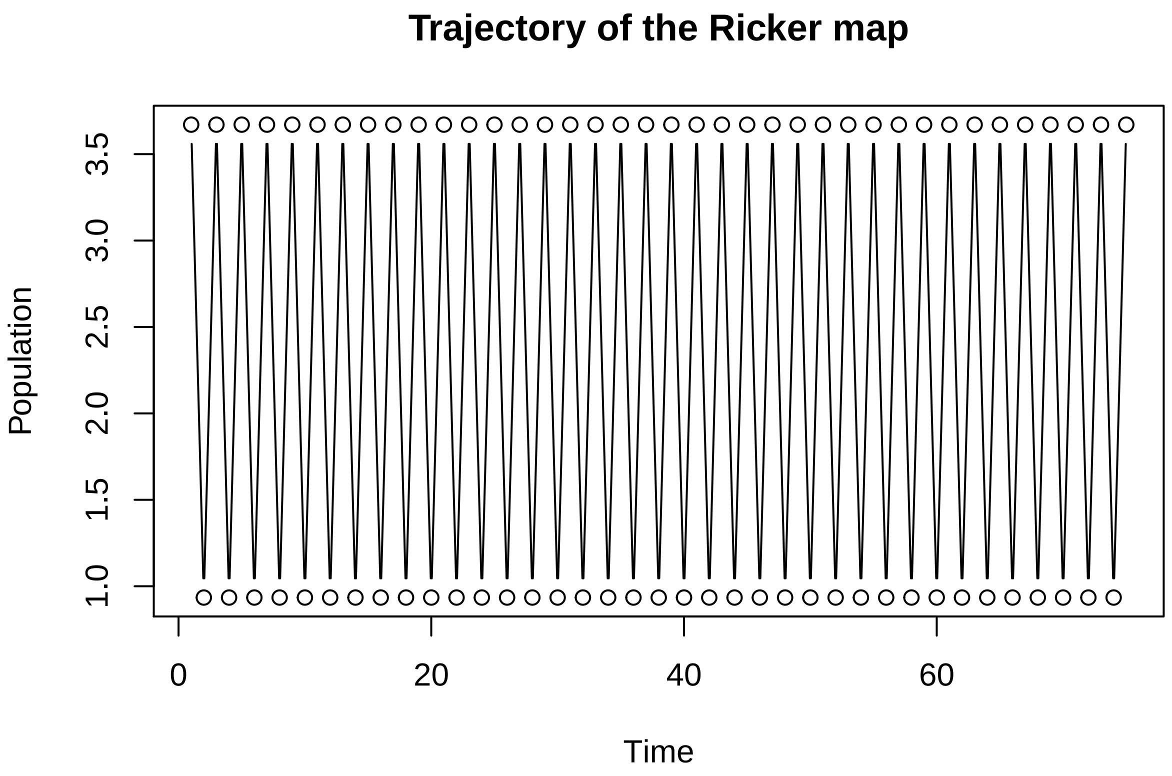
At any time we can check that the function is loaded

```r
is.loaded("rickerSimul")
```

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

We can now call the `rickerSimul` function using the `.Call` interface provided by R. `.Call` allows us to call C functions that take SEXP objects as inputs and provide a SEXP object as an output. We simulate a trajectory below and plot the output of the C function.

```r
n <- as.integer(75)
nburn <- as.integer(20)
plot(.Call("rickerSimul", n, nburn, 10, 1), type = "b",
     main = "Trajectory of the Ricker map",
     xlab = "Time", ylab = "Population")
```



The code seems to work — it provides a similar output to the original R code. Let's use the `microbenchmark` package to compare the speed of the two implementations.

```r
library(microbenchmark)
simC <- function() .Call("rickerSimul", n, nburn, 10, 1)
simR <- function() rickerSimulR(n, nburn, 10, 1)
microbenchmark(simC(), simR(), times=5e4)
```

```
## Unit: microseconds
##    expr   min    lq     mean median    uq      max neval
##  simC() 2.483 2.851 3.465483  3.052 3.293 5651.222 50000
##  simR() 11.005 12.509 14.905611 13.387 14.375 22674.887 50000
```

At the time of running this code, the C implementation was on average (mean) three times faster than the R implementation.

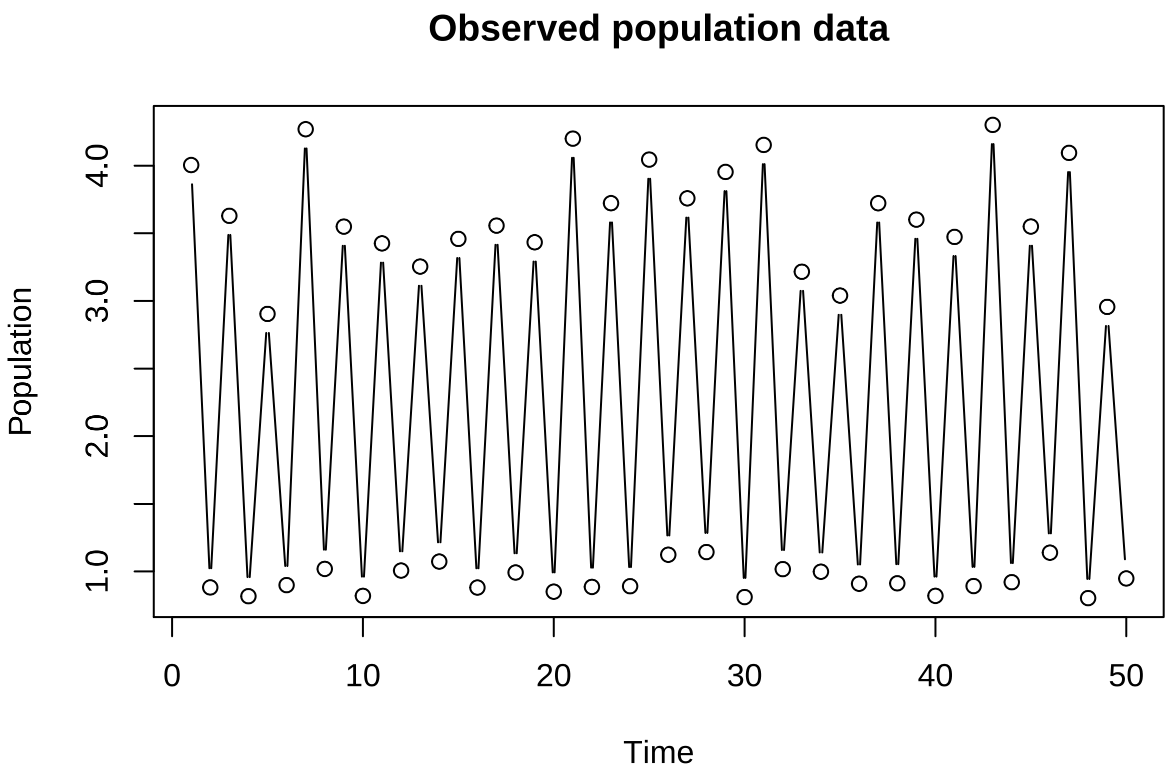
Question 2

We now observe some noisy data from the map. We observe

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

Specifically, we assume the observed data has been generated as below.

```r
nburn <- as.integer(100)
n <- as.integer(50)

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',
     main = "Observed population data",
     xlab = "Time", ylab = "Population")
```



We will create a C function which computes the likelihood of the observed data. The C function can be seen below.

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

SEXP rickerLLK(SEXP yobs_, SEXP ysim_, SEXP sig_)
{
  // create C objects
  double *yobs, *ysim, sig, *lik;
  int n;

  // create SEXP output
  SEXP lik_;

  // initialize C objects
  yobs = REAL(yobs_);
  ysim = REAL(ysim_);
  sig = REAL(sig_)[0];
  n = length(yobs_);

  // allocate memory to output
  lik_ = PROTECT(allocVector(REALSXP, 1));
  lik = REAL(lik_);

  // compute likelihood
  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 lik_;
}
```

The above code is saves in a C file called `rickerLLK.c`, and is compiled and loaded below.

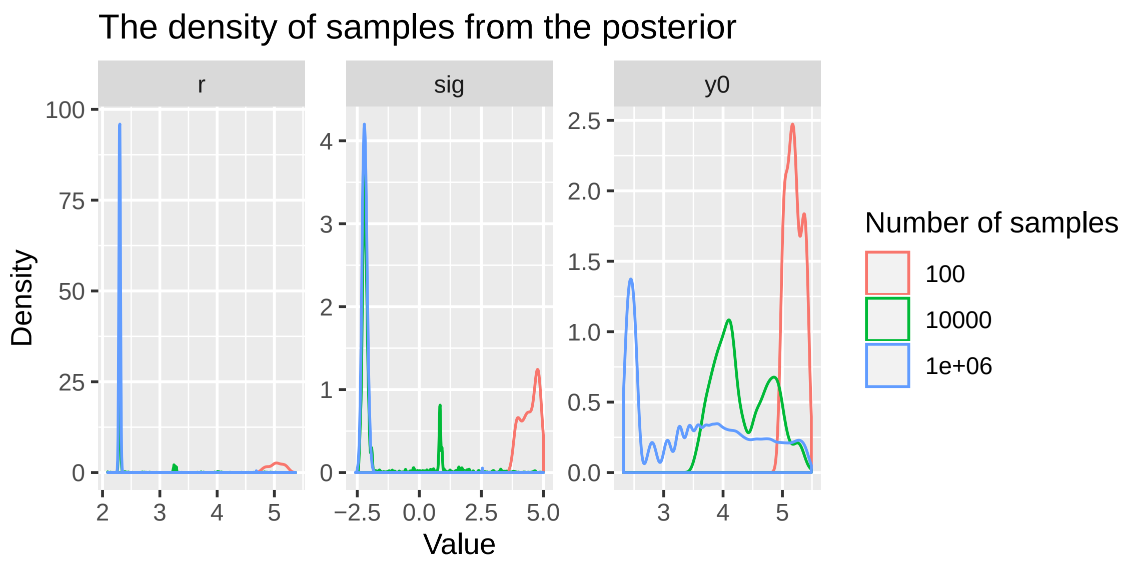
```r
system("R CMD SHLIB rickerLLK.c")
dyn.load("rickerLLK.so")
```

We wrap the C likelihood into a R function which takes as input the logarithm of the parameters `r, sig`, `y0`, and `yobs, nburn`. The function will simulate data using the provided parameters and then compute the likelihood of the observed data. Note that we work with the log-variables to enforce positivity when sampling new variables.

```r
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 )
}
```

We will use this function in a Metropolis-Hastings algorithm to sample from the posterior distribution of $\log(r), \log(\sigma), \log(y_0)$.

```
metrop_wrapper <- function(par) myLikR(par[1], par[2], par[3], yobs, nburn)

df <- data.frame()
for(n in c(1e2, 1e4, 1e6)){
  samples <- metrop(metrop_wrapper,
               initial = c(5, 5, 5),
               nbatch = n, scale = 0.05)
  df <- rbind(df,
               data.frame("r" = samples$batch[,1], "sig" = samples$batch[,2],
                          "y0" = samples$batch[,3], "nsamples" = n))
}

df %>%
  pivot_longer(names_to = "Variable", cols = c("r", "sig", "y0")) %>%
  mutate(Variable = factor(Variable), "Number of samples" = factor(nsamples)) %>%
  ggplot() +
  geom_density(aes(value, col = `Number of samples`)) +
  facet_wrap(~Variable, scales = "free") +
  labs(title = "The density of samples from the posterior",
       x = "Value", y = "Density")
```

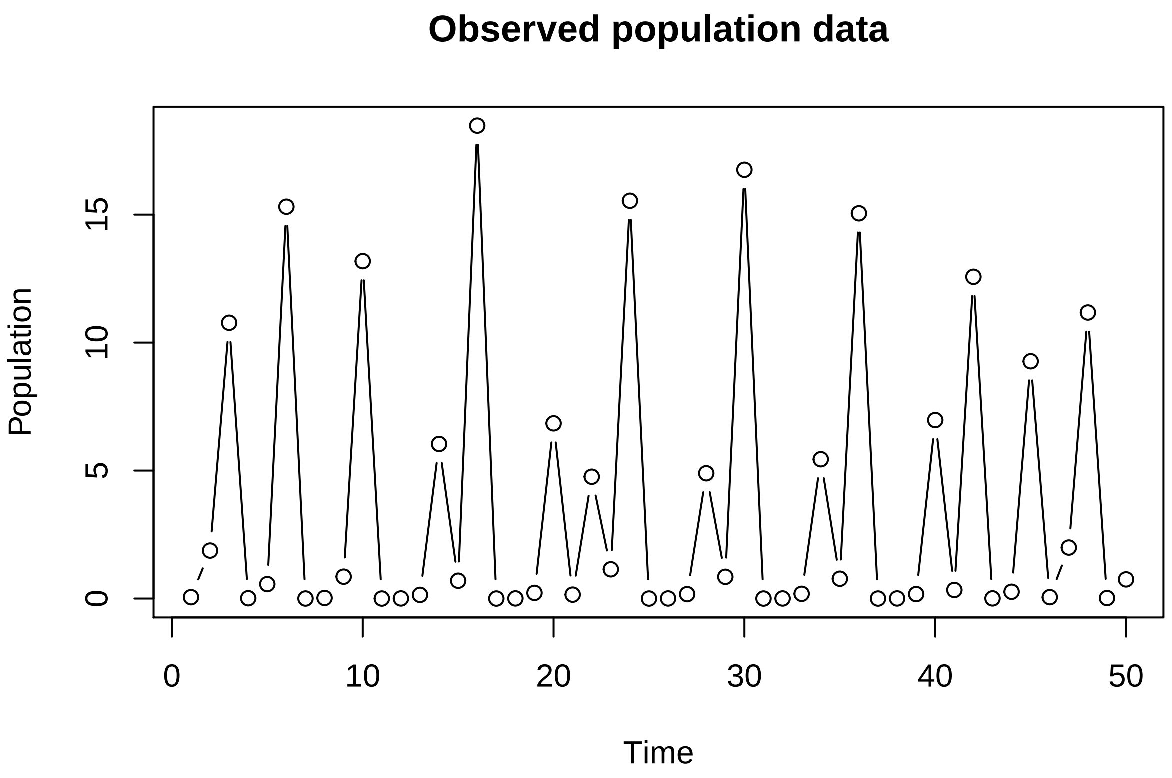


We see that as the number of samples increases the markov chains for $\log(r)$ and $\log(\sigma)$ converge to the true values, as demonstrated by the density of the posteriors. The posterior distribution for $\log(y_0)$ is quite dispersed and doesn't concentrate on a particular value. This is due to the nature of the Ricker map: the behavior of the chaotic system which we aim to capture is determined by r and σ, not necessarily y_0; the initial population value does not have a large impact on the population for a large t.

Question 3

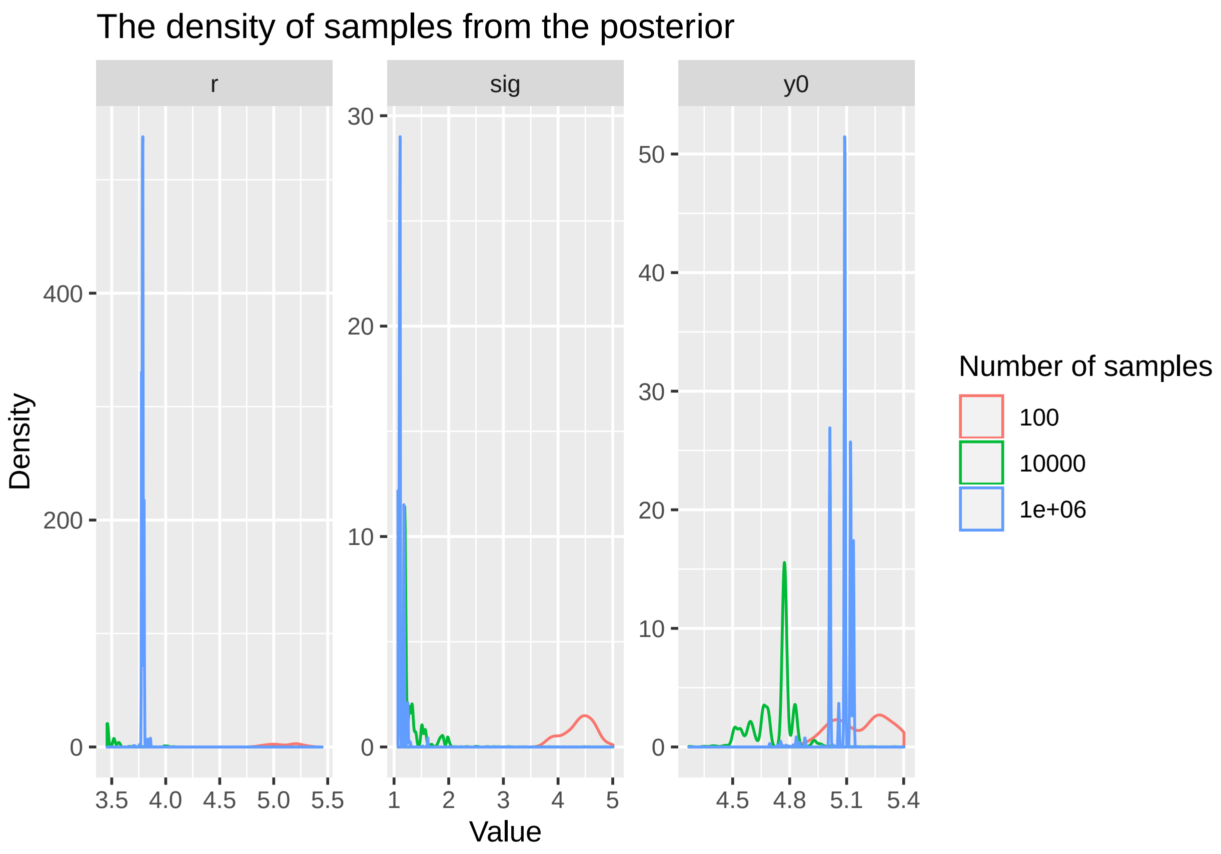
Suppose that the data has instead been simulated as follows

```r
r_true <- 44
n = as.integer(50)

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

plot(yobs, type = 'b',
     main = "Observed population data",
     xlab = "Time", ylab = "Population")
```



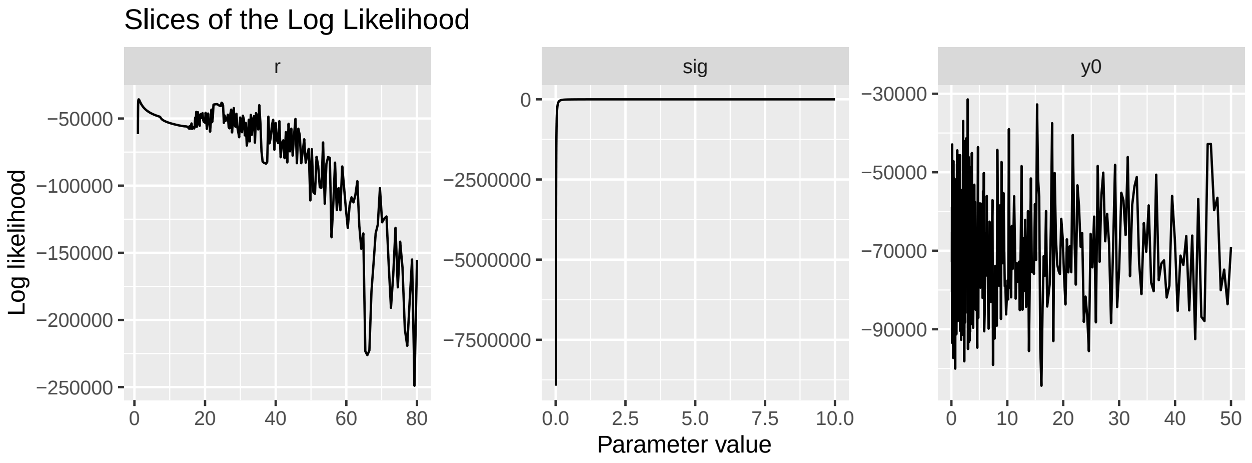
We try to use a Metropolis-Hastings algorithm again to sample from the posterior under the new data.

```r
metrop_wrapper <- function(par) myLikR(par[1], par[2], par[3], yobs, nburn)

df <- data.frame()
for(n in c(1e2, 1e4, 1e6)){
  samples <- metrop(metrop_wrapper,
              initial = c(5, 5, 5),
              nbatch = n, scale = 0.05)
  df <- rbind(df,
              data.frame("r" = samples$batch[,1], "sig" = samples$batch[,2],
                         "y0" = samples$batch[,3], "nsamples" = n))
}

df %>%
  pivot_longer(names_to = "Variable", cols = c("r", "sig", "y0")) %>%
  mutate(Variable = factor(Variable), "Number of samples" = factor(nsamples)) %>%
  ggplot() +
  geom_density(aes(value, col = `Number of samples`)) +
  facet_wrap(~Variable, scales = "free") +
  labs(title = "The density of samples from the posterior",
       x = "Value", y = "Density")
```



We see that the Metropolis-Hastings algorithm is failing to mix well. This can easily be seen by viewing the traces of the chains. To investigate why this might be the case, we plot the likelihood with respect to different model parameters — we keep two parameters fixed to their true values and plot the likelihood.

```
par_seq1 <- seq(log(1),log(80),length.out=500)
par_seq2 <- seq(log(0.01),log(10),length.out=500)
par_seq3 <- seq(log(0.1),log(50),length.out=500)

df <- data.frame("r" = sapply(par_seq1, function(x) myLikR(x, log(sig_true), log(y0_true), yobs, nburn)
                 "sig" = sapply(par_seq2, function(x) myLikR(log(r_true), x, log(y0_true), yobs, nburn)
                 "y0" = sapply(par_seq3, function(x) myLikR(log(r_true),  log(sig_true), x, yobs, nburn

df %>%
  pivot_longer(everything()) %>%
  group_by(name) %>%
  mutate(seq = if_else(name == "r", exp(par_seq1), if_else(name == "sig", exp(par_seq2), exp(par_seq3))
  ggplot() +
  geom_line(aes(x=seq, y=value)) +
  facet_wrap(~name, scales = "free") +
  labs(title = "Slices of the Log Likelihood",
       x = "Parameter value", y="Log likelihood")
```



We see that for a value of r greater than 14 the system becomes chaotic and the trajectories change rapidly for small changes in r. Because of this, the likelihood is no longer smooth.

Question 4

To avoid the non-smooth likelihood given admitted by the model, we work with the synthetic likelihood. The synthetic likelihood is based on a set of summary statistics of the raw data, so it captures the general behavior of the ricker map without trying to model the exact chaotic behavior produced by a single trajectory. In the following we assume that we know the true value of σ used to generate the observed data, and we do not care about the exact value of y_0 — we simply assume it is a random variable distributed as $y_0 \sim U[1, 10]$.

We will consider the sample mean s_1 and sample standard deviation s_2 for our summary statistics of the observed data. We assume that these statistics are independently normally distributed such that

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

Then we can easily write down $p(s_1, s_2 \mid r)$ as the product of the two densities above. To sample from the posterior we need to estimate this likelihood by simulating `nsim` trajectories from the model to estimate $\mu_1, \mu_2, \tau_1, \tau_2$ for a given r.

The code used to estimate the synthetic likelihood has been written below using Rcpp. We have also rewritten the C function used to simulate the trajectories, using Rcpp.

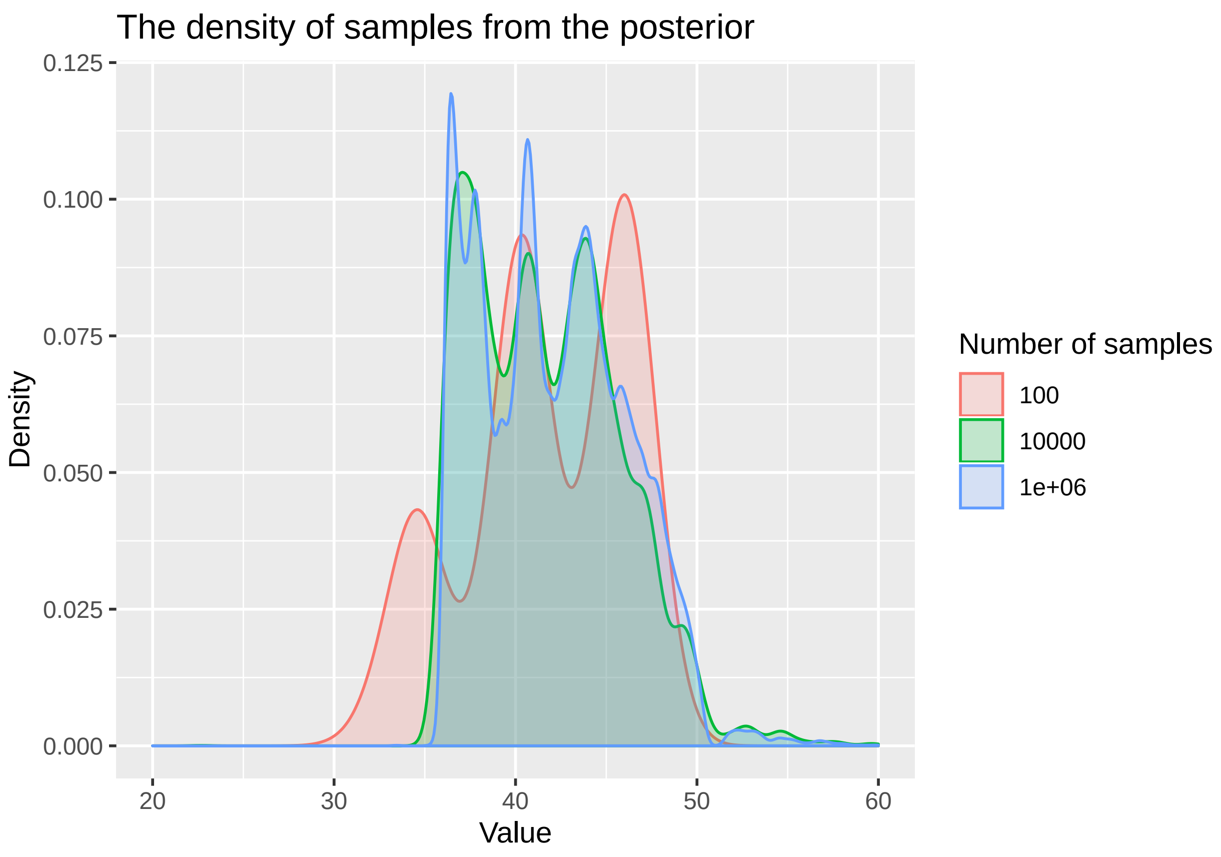
Rcpp

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
NumericVector rickerSimul(const int n, const int nburn, const double r, const double y0)
{
  double yx;
  NumericVector y(n);

  // burn in
  yx = y0;
  if(nburn>0)
  {
    for(int i=0; i<=nburn; i++)
    {
      yx = r * yx * exp(-yx);
    }
  }

  // simulate for n time periods
  y[0] = yx;
  for(int i=1; i<n; i++)
  {
    yx = r * yx * exp(-yx);
    y[i] = yx;
  }

  return y;
}

// [[Rcpp::export]]
NumericVector synllk(const double logr, const int nsim, const NumericVector yobs)
{
  double r;
  NumericVector ysim;
  NumericVector s1(nsim), s2(nsim);
  NumericVector y0(nsim);
  NumericVector out;

  // initialize C objects
  r = exp(logr);
  y0 = runif(nsim, 0, 10);

  for(int i=0; i<nsim; i++)
  {
    ysim = rickerSimul(50, 100, r, y0[i]) * exp(rnorm(50, 0, 0.1));
    s1[i] = mean(ysim);
    s2[i] = sd(ysim);
  }

  out = R::dnorm(mean(yobs), mean(s1), sd(s1), true) + R::dnorm(sd(yobs), mean(s2), sd(s2), true);

  return out;
}
```

The above code is saves in a file called `rickerRcpp.cpp` and is loaded below using `sourceCpp`. We use this code and a Metropolis-Hastings algorithm to generate samples from the posterior.

```r
sourceCpp("rickerRcpp.cpp")

df <- data.frame()
for(n in c(1e2, 1e4, 1e6)){
  samples <- metrop(function(r) synllk(r, 100, yobs), log(10), nbatch=n)
  df <- rbind(df,
              data.frame("r" = samples$batch[,1], "nsamples" = n))
}

df %>%
  pivot_longer(names_to = "Variable", cols = "r") %>%
  mutate(Variable = factor(Variable), "Number of samples" = factor(nsamples)) %>%
  ggplot() +
  geom_density(aes(exp(value), fill = `Number of samples`, col = `Number of samples`), alpha=0.2) +
  labs(title = "The density of samples from the posterior",
       x = "Value", y = "Density") +
  xlim(20,60)
```

```
## Warning: Removed 409 rows containing non-finite values (stat_density).
```

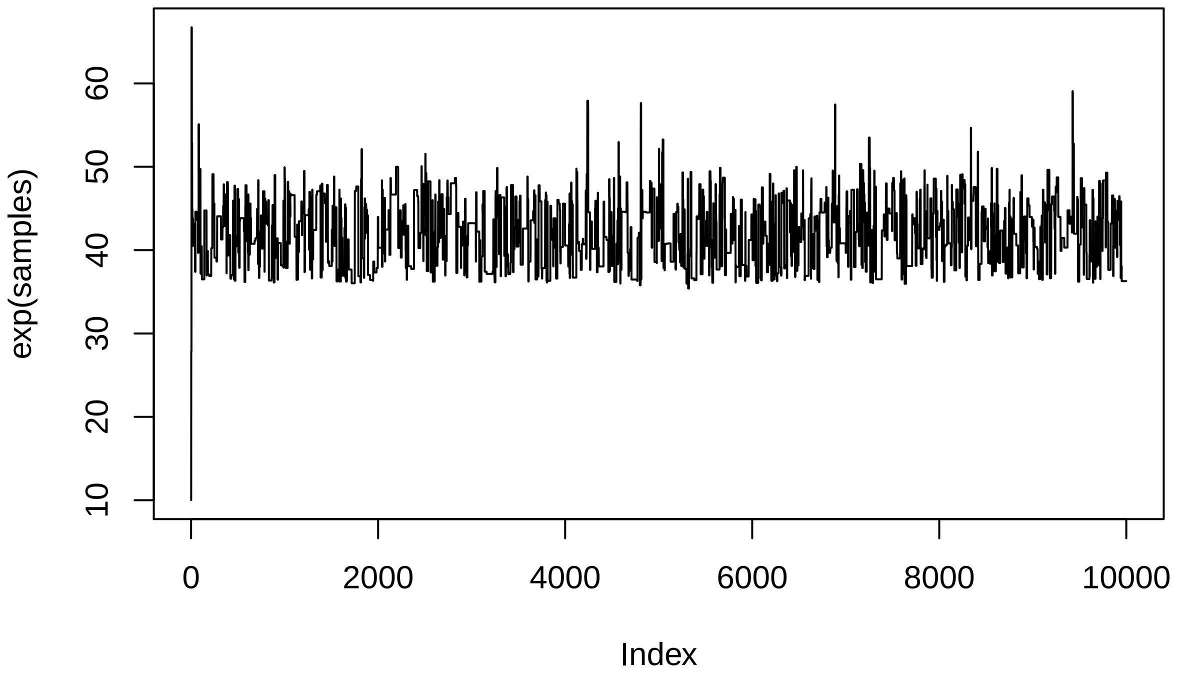


We see that the posterior focuses around the true value 44 for a large number of samples. They do not fully converge to the true value as we are only *estimating* the true likelihood. We could improve performance by using more summary statistics.

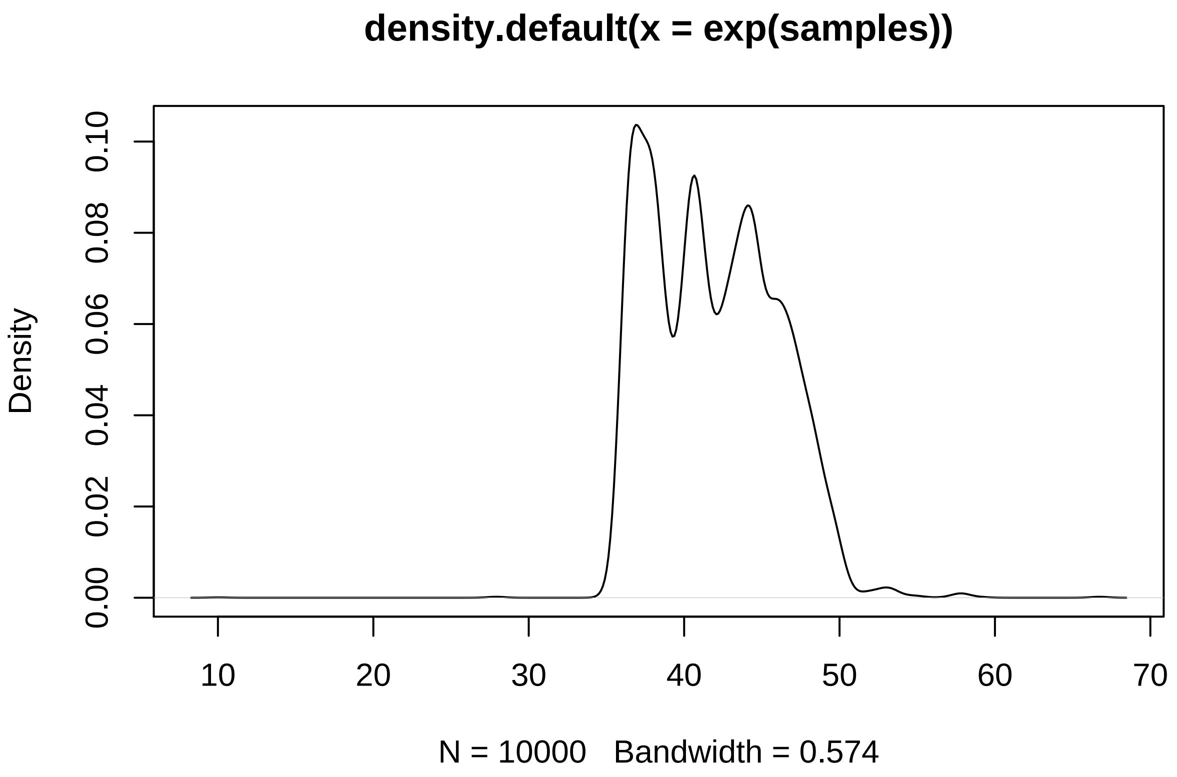
We could also improve performance by writing our own Metropolis-Hastings algorithm in Rcpp. Below is an implementation of Metropolis-Hastings which is saves as `metropolis.cpp`.

```
#include <Rcpp.h>
using namespace Rcpp;


// [[Rcpp::export]]
List metropolis(Function lik, double initial, int n, double scale)
{
  NumericVector par(n);
  double proposal;
  double acceptance_rate;
  NumericVector alpha(n);
  par[0] = initial;

  for(int i=1; i<n; i++)
  {
    proposal = R::rnorm(par[i-1], sqrt(scale));
    alpha[i] = as<double>(lik(proposal))/as<double>(lik(par[i-1]));
    if(R::runif(0,1) < alpha[i]){
      par[i] = proposal;
      acceptance_rate++;
    }
    else {par[i] = par[i-1];}
  }

  acceptance_rate = acceptance_rate/n;
  return List::create(
    Named("samples") = par,
    Named("acceptance rate") = acceptance_rate);
}
```

```r
sourceCpp("metropolis.cpp")
samples <- metropolis(function(r) exp(synllk(r, 100, yobs)), log(10), 10000, 1)$samples
plot(exp(samples), type = "l")
```



```r
plot(density(exp(samples)))
```



We can test the speed of our Rcpp function `metropolis` against that offered by the `mcmc` package (`metrop`).

```
library(microbenchmark)
metropolis_C <- function() metropolis(function(r) synllk(r, 100, yobs), log(40), 10, 1)
metropolis_R <- function() metrop(function(r) synllk(r, 100, yobs), log(40), 10)
microbenchmark(metropolis_C(), metropolis_R(), times=100, unit="relative")
```

```
## Unit: relative
##            expr      min       lq     mean   median       uq      max neval
##  metropolis_C() 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000   100
##  metropolis_R() 7.961368 6.075677 5.758869 5.859735 5.323612 4.838558   100
```

For a small number of samples, our function `metropolis` is approximately eight times faster. However, for a larger chain length the `metrop` function is nearly twice as fast as seen below.

```
library(microbenchmark)
metropolis_C <- function() metropolis(function(r) synllk(r, 100, yobs), log(40), 10000, 1)
metropolis_R <- function() metrop(function(r) synllk(r, 100, yobs), log(40), 10000)
microbenchmark(metropolis_C(), metropolis_R(), times=4, unit="relative")
```

```
## Unit: relative
##            expr      min       lq     mean   median      uq      max neval
##  metropolis_C() 1.534227 1.528275 1.898223 1.896769 2.26162 2.254834     4
##  metropolis_R() 1.000000 1.000000 1.000000 1.000000 1.00000 1.000000     4
```