

simulation & inference of transients

Carl Boettiger, Jorge Arroyo Esquivel, Junjie Jiang, Jody Reimer, Henry Scharf, Elizabeth Wolkovich, and Kai Zhu

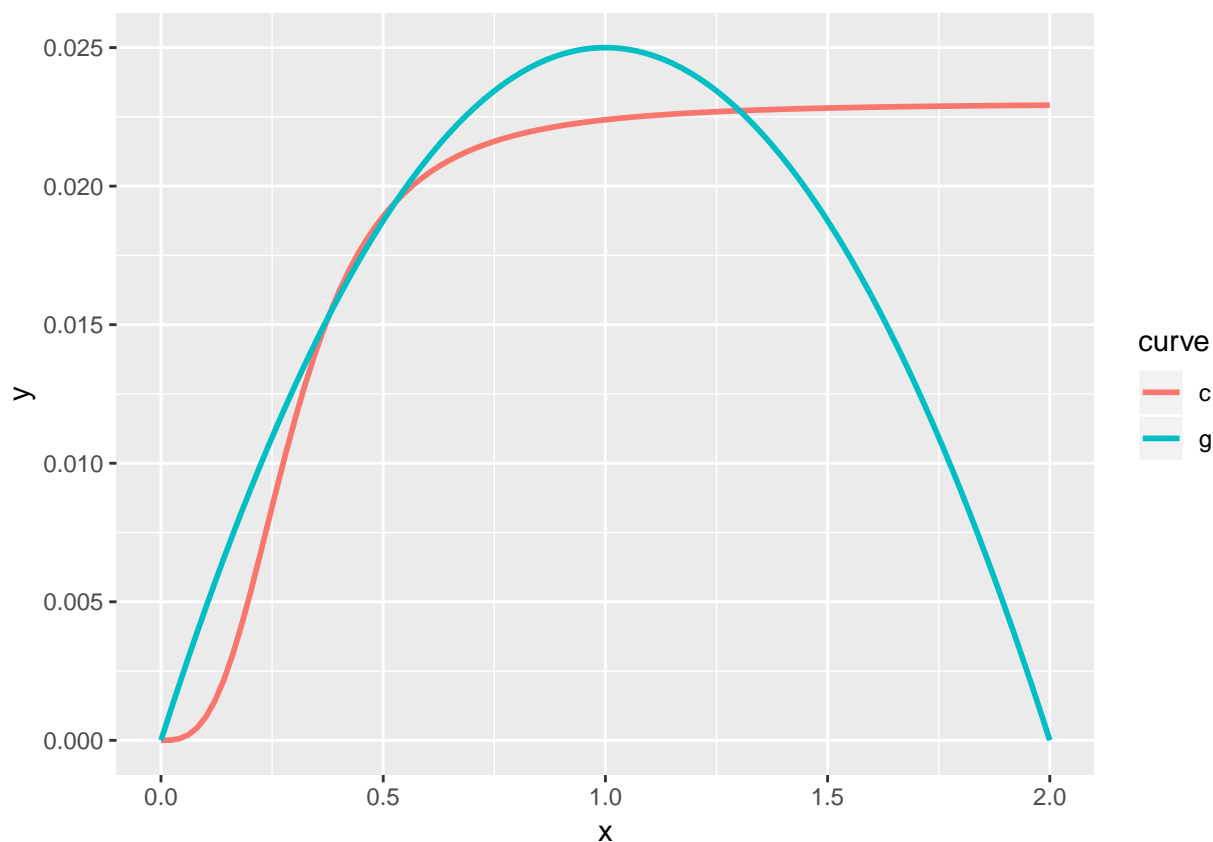
2019-09-08

```
library(tidyverse)
```

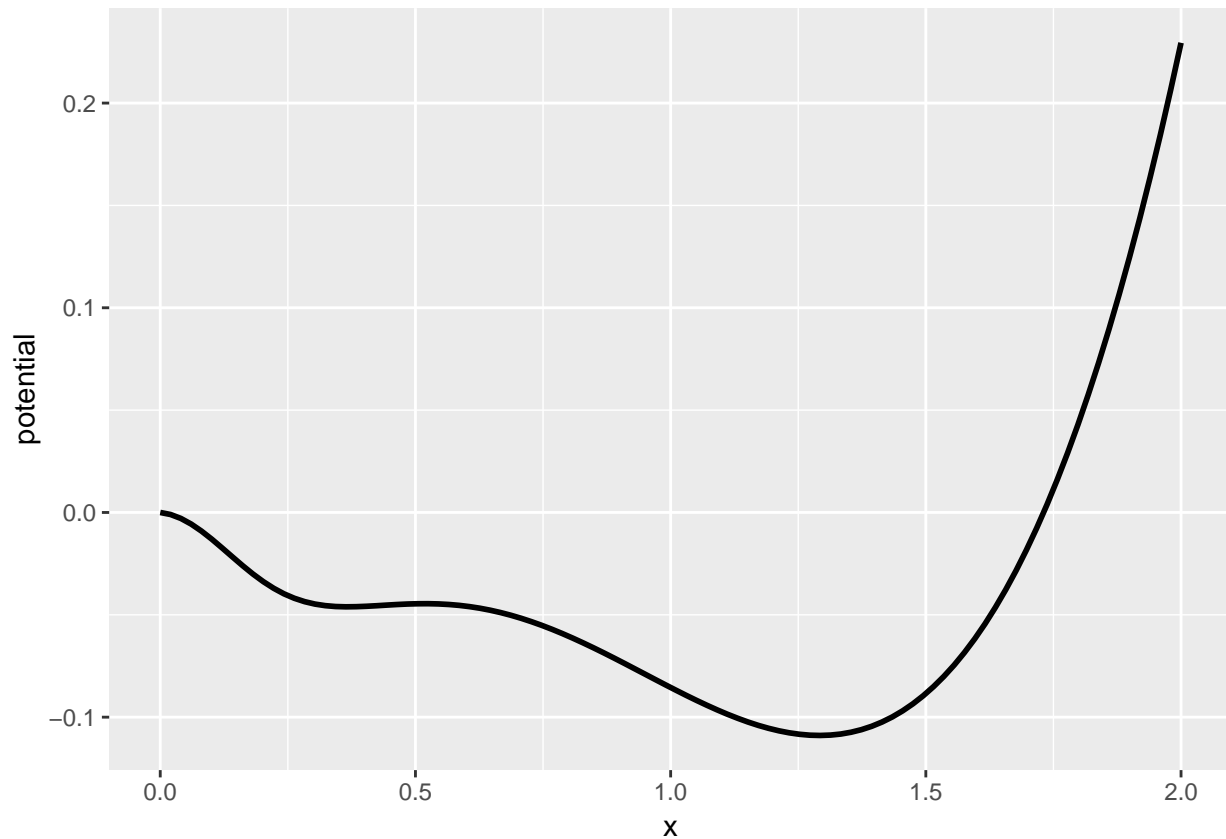
```
N <- 1e4  
r <- 0.05; K <- 2  
a <- 0.023; H <- 0.3; Q <- 3  
x0 <- 0.2; sigma <- 0.02  
growth <- function(x, r, K) x * r * (1 - x / K)  
consumption <- function(x, a, H, Q) a * x^Q / (x^Q + H^Q)
```

```
theory <-  
  tibble(x = seq(0,K, length.out = 100)) %>%  
  mutate(g = growth(x, r, K),  
         c = consumption(x, a, H, Q)) %>%  
  mutate(potential = - cumsum(g - c)) %>%  
  gather(curve, y, -x, -potential)
```

```
theory %>%  
  ggplot(aes(x, y, col = curve)) +  
  geom_line(lwd = 1)
```



```
theory %>%
  ggplot(aes(x, potential)) +
  geom_line(lwd = 1)
```



```
library(nimble)
```

```
## nimble version 0.8.0 is loaded.
## For more information on NIMBLE and a User Manual,
## please visit http://R-nimble.org.
##
## Attaching package: 'nimble'
## The following object is masked from 'package:stats':
##
##   simulate
## define truncated normal distribution ----
dtruncnorm <- nimbleFunction(
  run = function(x = double(0), mean = double(0),
                 sd = double(0), log = integer(0, default = 0)) {
    returnType(double(0))
    log_prob <- dnorm(x = x, mean = mean, sd = sd, log = 1) -
      pnorm(q = 0, mean = -mean, sd = sd, log = 1)
    if(log) return(log_prob)
    else return(exp(log_prob))
  })
rtruncnorm <- nimbleFunction(
  run = function(n = integer(0, default = 1), mean = double(0),
```

```

        sd = double(0)) {
  returnType(double(0))
  if(n != 1) print("rtruncnorm only allows n = 1; using n = 1.")
  draw <- rnorm(n = 1, mean = mean, sd = sd)
  while(draw < 0)
    draw <- rnorm(n = 1, mean = mean, sd = sd)
  return(draw)
})
# Define stochastic model in BUGS notation ----
may <- nimble::nimbleCode({
  log(r) ~ dnorm(mu_r, sd_r)
  log(K) ~ dnorm(mu_K, sd_K)
  log(a) ~ dnorm(mu_a, sd_a)
  log(H) ~ dnorm(mu_H, sd_H)
  log(Q) ~ dnorm(mu_Q, sd_Q)
  x0 ~ dtruncnorm(mu_x0, sd_x0)
  log(sigma) ~ dnorm(mu_sigma, sd_sigma)
  x[1] <- x0
  for(t in 1:(N - 1)){
    mu[t] <- x[t] + x[t] * r * (1 - x[t] / K) - a * x[t]^Q / (x[t]^Q + H^Q)
    x[t + 1] ~ dtruncnorm(mean = mu[t], sd = sigma)
  }
})
# constants ----
constants <- list(
  N = N,
  mu_r = log(r), sd_r = 1,
  mu_K = log(K), sd_K = 1,
  mu_a = log(a), sd_a = 1,
  mu_H = log(H), sd_H = 1,
  mu_Q = log(Q), sd_Q = 1,
  mu_x0 = x0, sd_x0 = 1,
  mu_sigma = log(sigma), sd_sigma = 1
)
## inits ----
inits <- list("log_r" = log(r), "log_K" = log(K),
             "log_a" = log(a), "log_H" = log(H), "log_Q" = log(Q),
             "log_sigma" = log(sigma), "x0" = x0)
# define and compile model ----
model <- nimbleModel(code = may, constants = constants, inits = inits)

```

defining model...

Registering the following user-provided distributions: dtruncnorm .

NIMBLE has registered dtruncnorm as a distribution based on its use in BUGS code. Note that if you m

building model...

setting data and initial values...

running calculate on model (any error reports that follow may simply reflect missing values in model

checking model sizes and dimensions... This model is not fully initialized. This is not an error. To

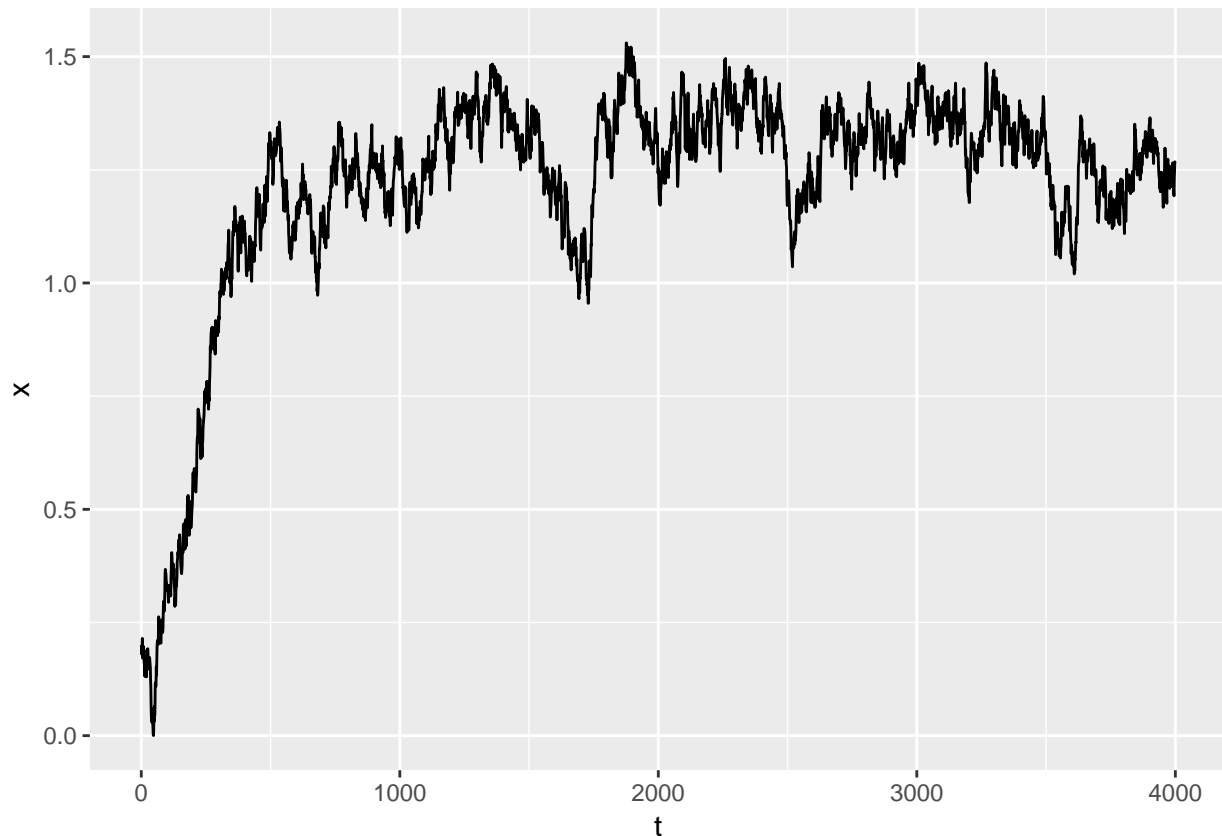
model building finished.

```
cmodel <- compileNimble(model)
```

```
## compiling... this may take a minute. Use 'showCompilerOutput = TRUE' to see C++ compilation details.
## compilation finished.
```

```
set.seed(1234)
simulate(cmodel, nodes = c('x', 'mu'))
df <- tibble(t = seq_along(cmodel$x), x = cmodel$x)
```

```
df %>% filter(t < 4000) %>% ggplot(aes(t, x)) + geom_line()
```



```
dnorm_mod <- function(x, mean, sd, log = T){
  dnorm(x = x, mean = mean, sd = sd, log = log) -
    log(1 - pnorm(q = 0, mean = mean, sd = sd))
}
get_ll <- function(x, r, K, a, H, Q, sigma){
  TIME <- length(x)
  mu <- x + x * r * (1 - x / K) - a * x^Q / (x^Q + H^Q)
  sum(dnorm_mod(x = x[-1], mean = mu[-TIME], sd = sigma, log = T))
}
##
grid_r <- seq(1e-2, 1e-1, l=5e1)
ll_r <- sapply(grid_r, function(r) get_ll(a = cmodel$a, x = cmodel$x, r = r, K = cmodel$K,
                                          H = cmodel$H, Q = cmodel$Q, sigma = cmodel$sigma))
##
grid_K <- seq(1, 5, l=5e1)
ll_K <- sapply(grid_K, function(K) get_ll(a = cmodel$a, x = cmodel$x, r = cmodel$r, K = K,
                                          H = cmodel$H, Q = cmodel$Q, sigma = cmodel$sigma))
##
grid_a <- seq(1e-3, 1e-1, l=5e1)
```

```

ll_a <- sapply(grid_a, function(a) get_ll(a = a, x = cmodel$x, r = cmodel$r, K = cmodel$K,
                                          H = cmodel$H, Q = cmodel$Q, sigma = cmodel$sigma))

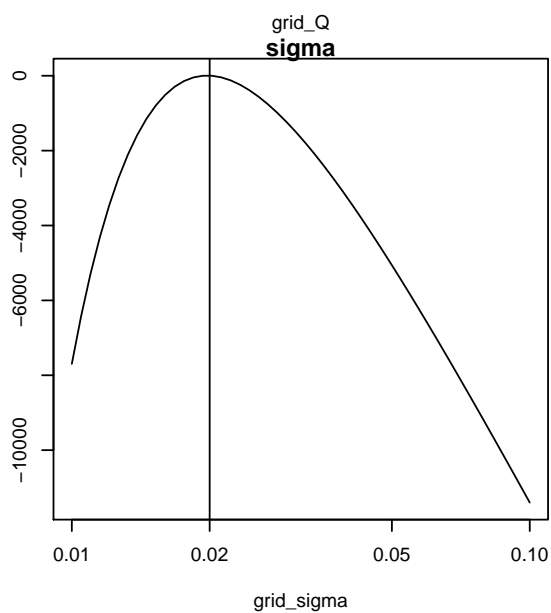
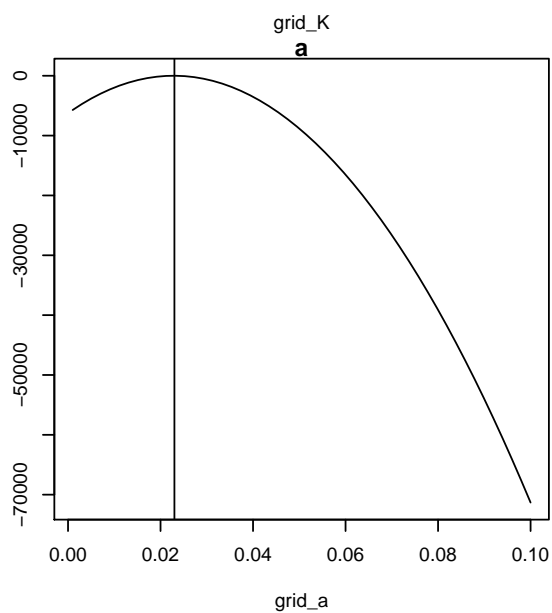
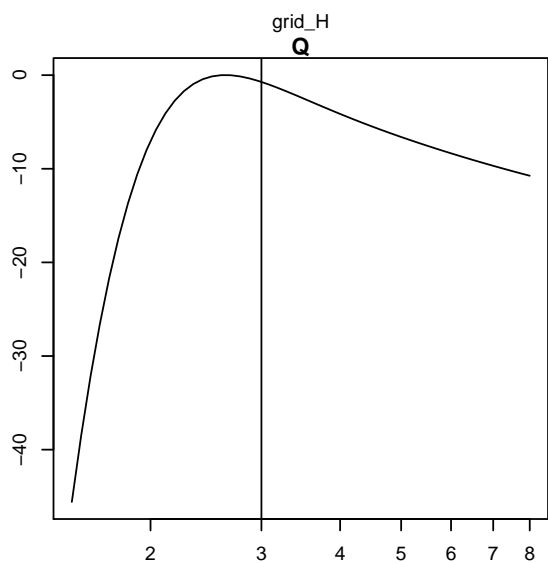
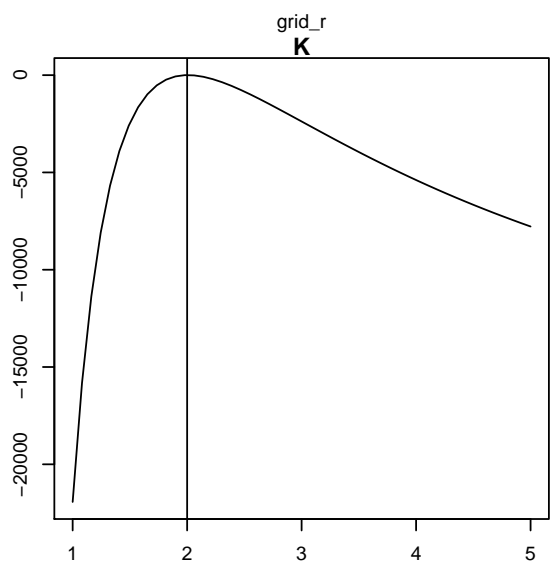
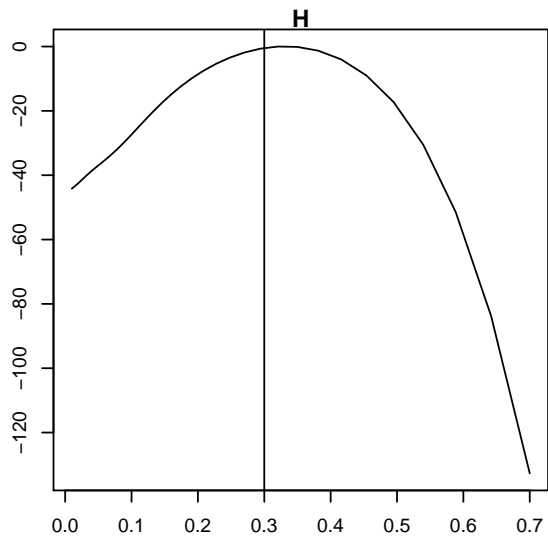
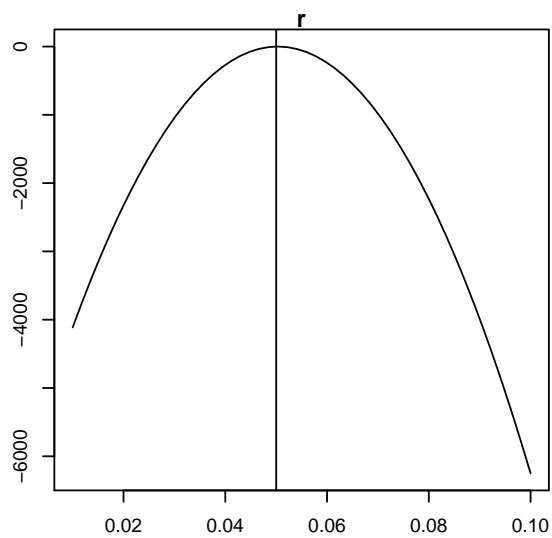
##
grid_H <- exp(seq(log(1e-2), log(0.7), l=5e1))
ll_H <- sapply(grid_H, function(H) get_ll(a = cmodel$a, x = cmodel$x, r = cmodel$r, K = cmodel$K,
                                          H = H, Q = cmodel$Q, sigma = cmodel$sigma))

##
grid_Q <- exp(seq(log(1.5), log(8), l=5e1))
ll_Q <- sapply(grid_Q, function(Q) get_ll(a = cmodel$a, x = cmodel$x, r = cmodel$r, K = cmodel$K,
                                          H = cmodel$H, Q = Q, sigma = cmodel$sigma))

##
grid_sigma <- exp(seq(log(1e-2), log(1e-1), l=5e1))
ll_sigma <- sapply(grid_sigma, function(sigma)
  get_ll(a = cmodel$a, x = cmodel$x, r = cmodel$r, K = cmodel$K,
        H = cmodel$H, Q = cmodel$Q, sigma = sigma))

##
layout(matrix(1:6, 3, 2)); par(mar = c(4.1, 2, 1, 1))
plot(grid_r, (ll_r - max(ll_r)), type = "l", main = "r")
abline(v = cmodel$r)
plot(grid_K, (ll_K - max(ll_K)), type = "l", main = "K")
abline(v = cmodel$K)
plot(grid_a, (ll_a - max(ll_a)), type = "l", main = "a")
abline(v = cmodel$a)
plot(grid_H, (ll_H - max(ll_H)), type = "l", main = "H")
abline(v = cmodel$H)
plot(grid_Q, (ll_Q - max(ll_Q)), type = "l", main = "Q", log = "x")
abline(v = cmodel$Q)
plot(grid_sigma, (ll_sigma - max(ll_sigma)), log = "x", type = "l", main = "sigma")
abline(v = cmodel$sigma)

```

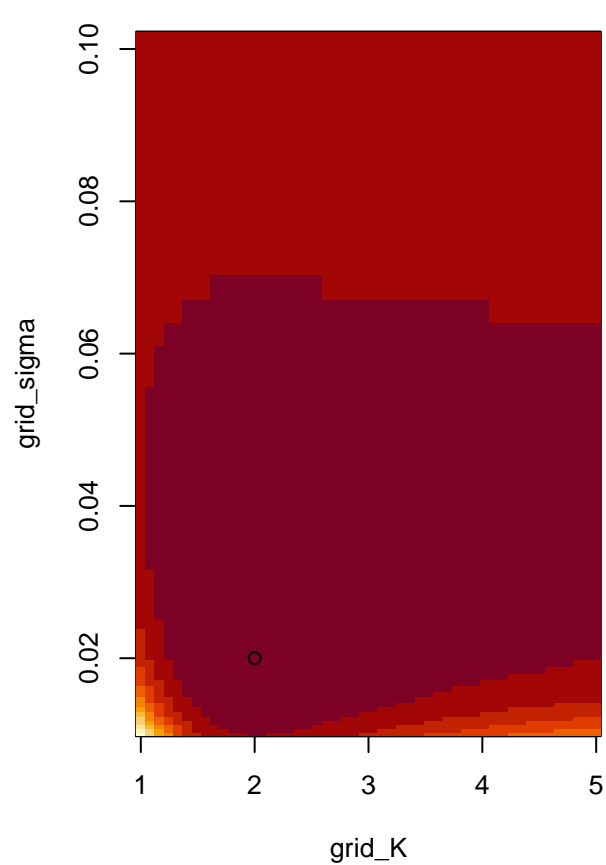
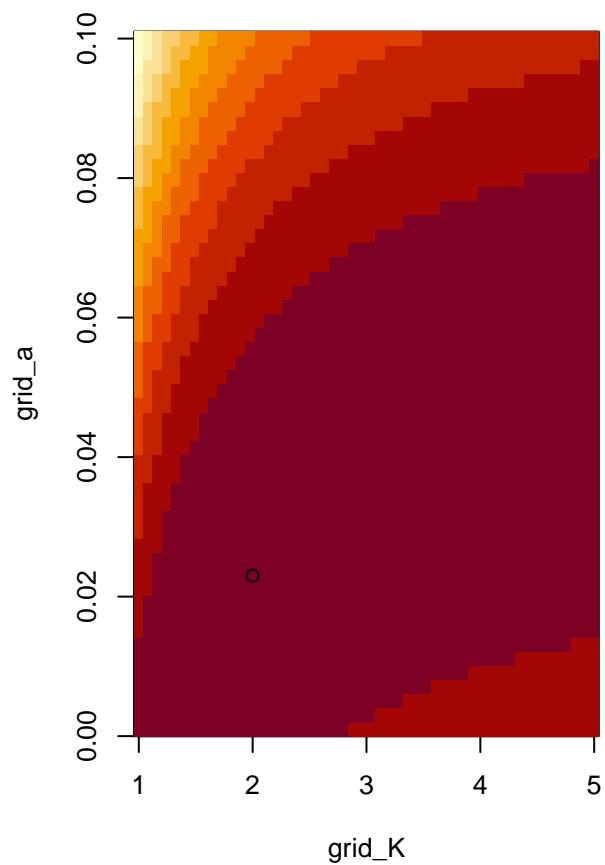
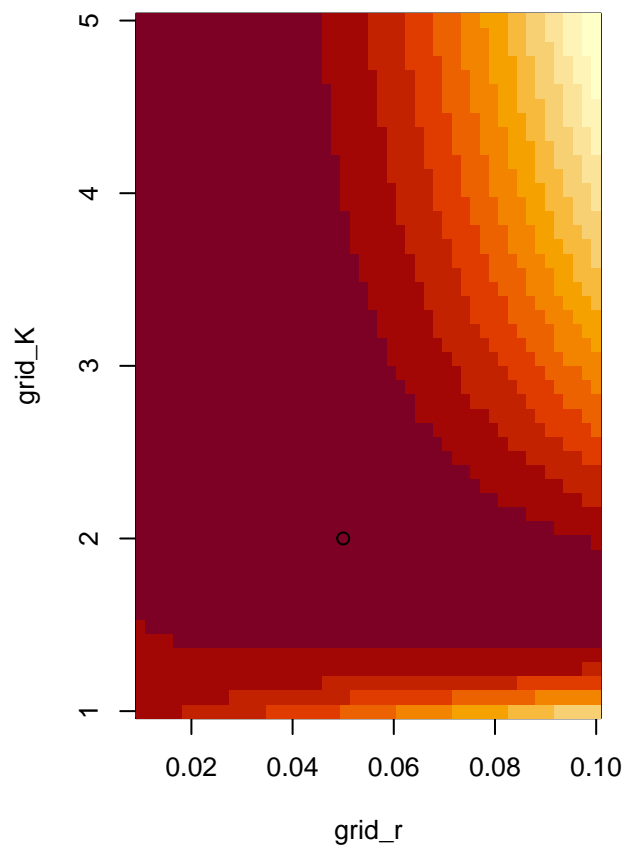
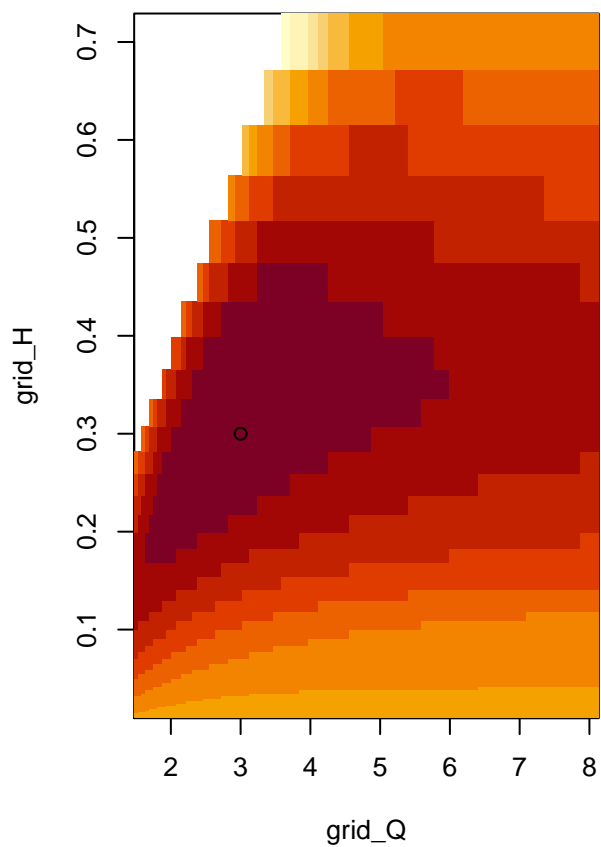


```

ll_QH <- apply(expand.grid(grid_Q, grid_H), 1, function(pair)
  if(pair[2] > (0.6 / 3) * pair[1] + 0.18 - (0.6 / 3)) NA else
  get_ll(a = cmodel$a, x = cmodel$x, r = cmodel$r, K = cmodel$K,
    H = pair[2], Q = pair[1], sigma = cmodel$sigma))
ll_Ka <- apply(expand.grid(grid_K, grid_a), 1, function(pair)
  get_ll(a = pair[2], x = cmodel$x, r = cmodel$r, K = pair[1],
    H = cmodel$H, Q = cmodel$Q, sigma = cmodel$sigma))
ll_rK <- apply(expand.grid(grid_r, grid_K), 1, function(pair)
  get_ll(x = cmodel$x, r = pair[1], K = pair[2], a = cmodel$a,
    H = cmodel$H, Q = cmodel$Q, sigma = cmodel$sigma))
ll_Ksigma <- apply(expand.grid(grid_K, grid_sigma), 1, function(pair)
  get_ll(x = cmodel$x, r = cmodel$r, K = pair[1], a = cmodel$a,
    H = cmodel$H, Q = cmodel$Q, sigma = pair[2]))

layout(matrix(1:4, 2, 2)); par(mar = c(4, 4, 1, 1))
image(grid_Q, grid_H, matrix((ll_QH - max(ll_QH, na.rm = T)), length(grid_Q), length(grid_H)))
points(cmodel$Q, cmodel$H)
##
image(grid_K, grid_a, matrix((ll_Ka - max(ll_Ka)), length(grid_K), length(grid_a)))
points(cmodel$K, cmodel$a)
##
image(grid_r, grid_K, matrix((ll_rK - max(ll_rK)), length(grid_r), length(grid_K)))
points(cmodel$r, cmodel$K)
##
image(grid_K, grid_sigma, matrix((ll_Ksigma - max(ll_Ksigma)), length(grid_a), length(grid_sigma)))
points(cmodel$K, cmodel$sigma)

```




```
model$setData(x = cmodel$x)
cmodel <- compileNimble(model)
```

```
## compiling... this may take a minute. Use 'showCompilerOutput = TRUE' to see C++ compilation details.
```

```
## compilation finished.
```

```
## block sampler
system.time({
  mcmcConf <- configureMCMC(model)
  mcmcConf$getSamplers()
  mcmcConf$removeSamplers(c("x0"))
  mcmcConf$addSampler(target = c("log_r", "log_K", "log_a",
                                "log_H", "log_Q", "log_sigma"),
                      type = 'RW_block')
  mcmcConf$getSamplers()
})
```

```
## Note: Assigning an RW_block sampler to nodes with very different scales can result in low MCMC efficiency.
```

```
##   user  system elapsed
##  8.364   0.075   8.505
```

```
system.time({
  mcmc <- buildMCMC(mcmcConf)
  Cmcmc <- compileNimble(mcmc, project = model)
})
```

```
## compiling... this may take a minute. Use 'showCompilerOutput = TRUE' to see C++ compilation details.
```

```
## compilation finished.
```

```
##   user  system elapsed
## 19.179   0.539  19.950
```

```
##
n_iterations <- 1e4
system.time({
  Cmcmc$run(n_iterations, nburnin = n_iterations / 2)
})
```

```
## |-----|-----|-----|-----|
## |-----|-----|-----|-----|
```

```
##   user  system elapsed
## 159.215   0.863 161.344
```

```
samples <- as.matrix(Cmcmc$mvSamples)
```

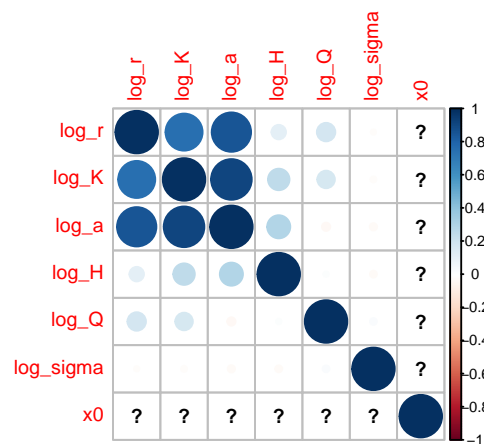
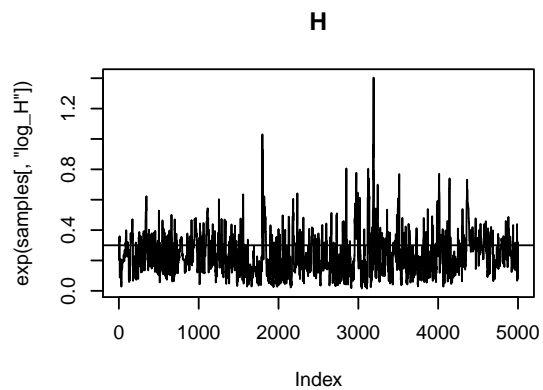
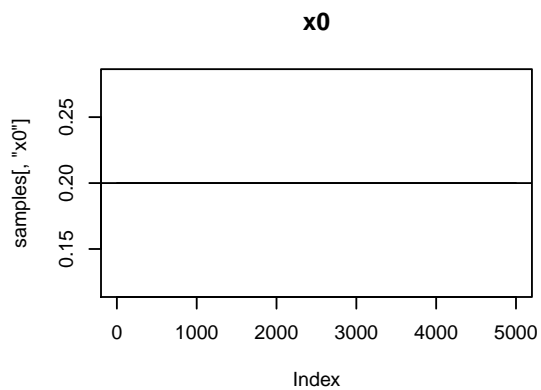
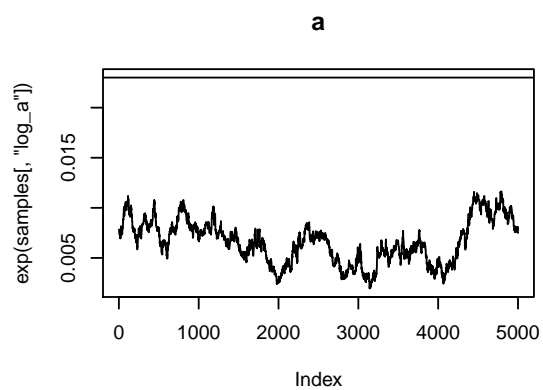
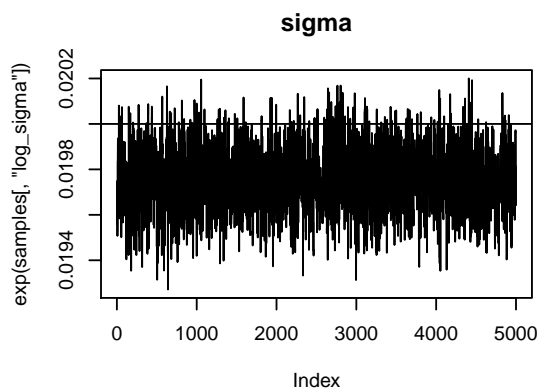
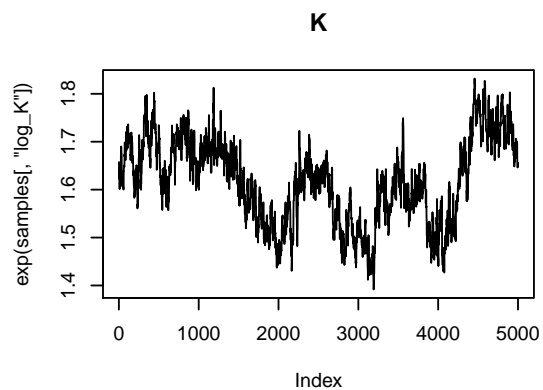
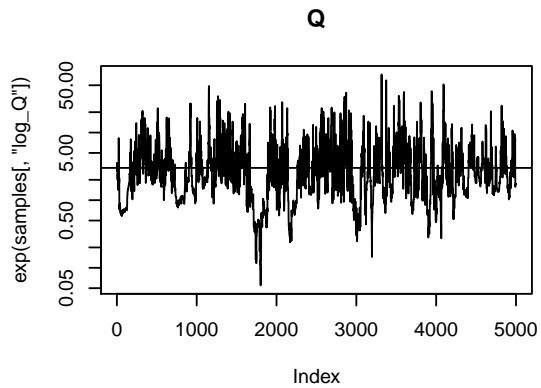
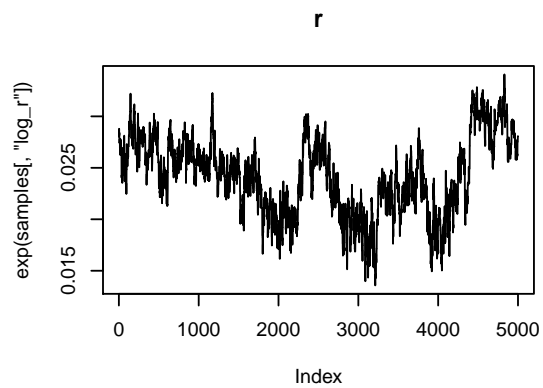
```
layout(matrix(1:8, 4, 2))
plot(exp(samples[, 'log_r']), type = "l", main = "r")
abline(h = r)
plot(exp(samples[, 'log_K']), type = "l", main = "K")
abline(h = K)
plot(exp(samples[, 'log_a']), type = "l", main = "a",
     ylim = range(exp(samples[, 'log_a']), a))
abline(h = a)
plot(exp(samples[, 'log_H']), type = "l", main = "H")
abline(h = H)
plot(exp(samples[, 'log_Q']), type = "l", main = "Q", log = "y")
```

```

abline(h = Q)
plot(exp(samples[, 'log_sigma']), type = "l", main = "sigma")
abline(h = sigma)
plot(samples[, 'x0'], type = "l", main = "x0")
abline(h = x0)
corrplot::corrplot(cor(samples[, c("log_r", "log_K", "log_a", "log_H", "log_Q", "log_sigma", "x0")]))

## Warning in cor(samples[, c("log_r", "log_K", "log_a", "log_H", "log_Q", :
## the standard deviation is zero

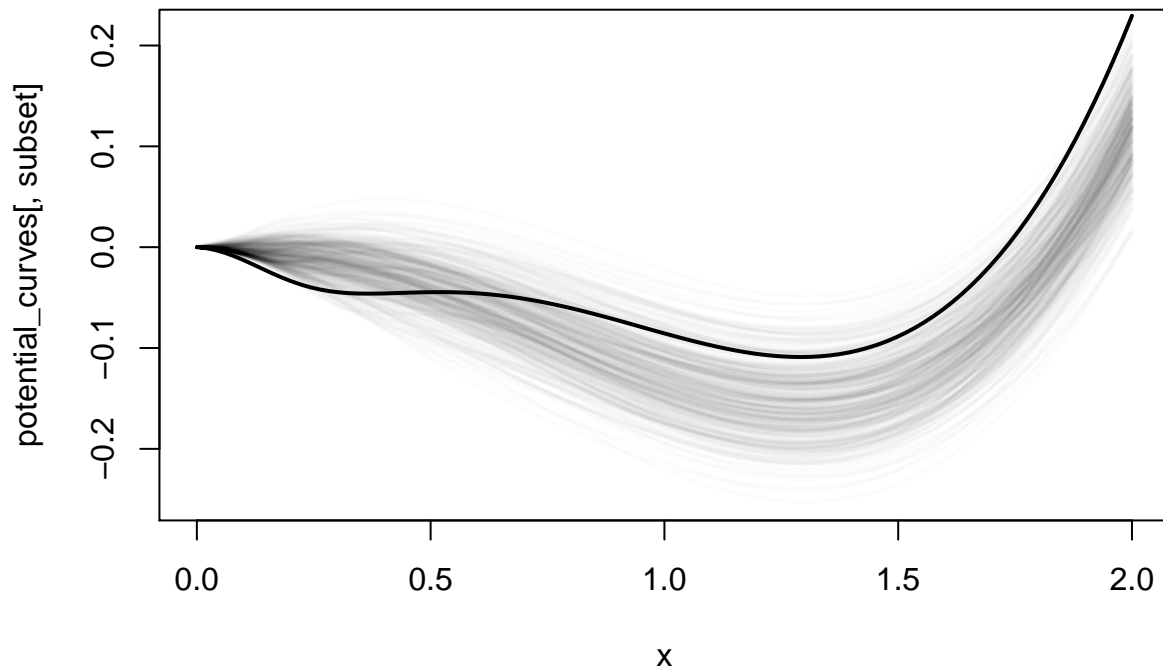
```



```

x <- seq(0, 2, l=1e2)
potential <- function(x = seq(0, 2, l = 1e2), a, r, H, Q, K){
  - cumsum(growth(x = x, r = r, K = K) -
           consumption(x = x, a = a, H = H, Q = Q))
}
potential_curves <- apply(samples, 1, function(row){
  potential(x = x, a = exp(row['log_a']), r = exp(row['log_r']),
           H = exp(row['log_H']), Q = exp(row['log_Q']), K = exp(row['log_K']))
})
subset <- sample(1:nrow(samples), min(400, nrow(samples)))
matplot(x, potential_curves[, subset], type = "l", lty = 1,
        col = scales::alpha(1, 1e-2), lwd = 2)
lines(x, potential(x = x, a = a, r = r, H = H, Q = Q, K = K), lwd = 2)

```

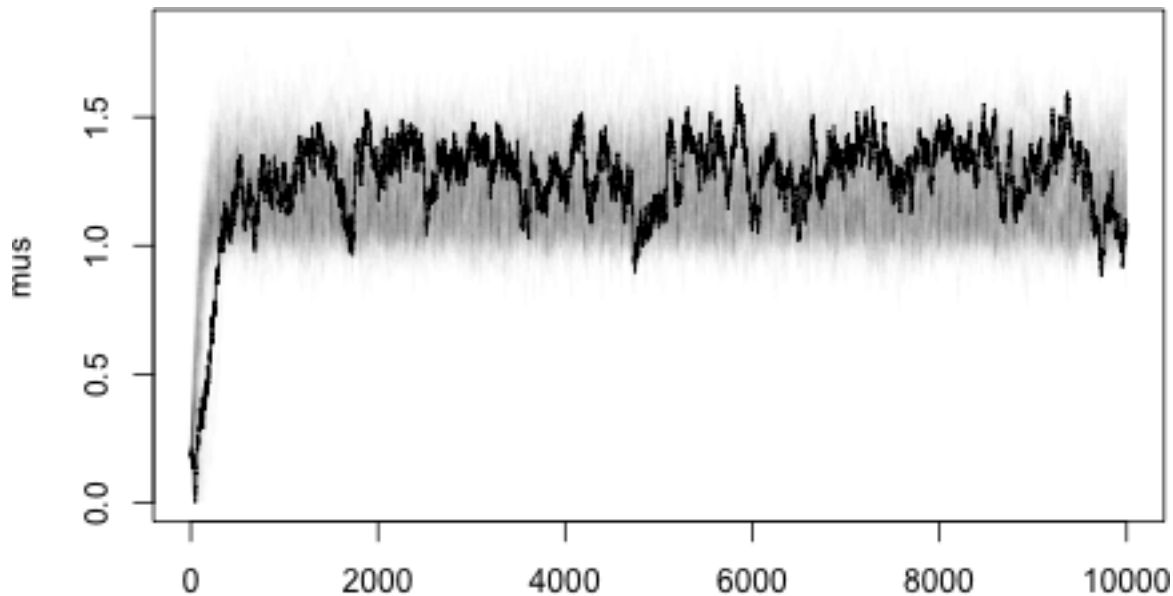


```

library(parallel)
cl <- makeForkCluster(2)
mus <- parApply(cl, samples[sample(1:nrow(samples), 100), ], 1,
  function(row){
    x <- rep(NA, N); x[1] <- row['x0']
    mu <- rep(NA, N - 1)
    for(t in 1:(N - 1)){
      mu[t] <-
        x[t] + x[t] * exp(row['log_r']) *
          (1 - x[t] / exp(row['log_K'])) -
        exp(row['log_a']) *
          x[t]^exp(row['log_Q']) / (x[t]^(row['log_Q']) +
            exp(row['log_H'])^(row['log_Q']))
      x[t + 1] <- rtruncnorm(mean = mu[t], sd = exp(row['log_sigma']))
    }
    return(mu)
  })
stopCluster(cl)

```

```
matplot(mus, type = "l", col = scales::alpha(1, 0.01), lty = 1)
lines(cmodel$mu)
```



Notice a , r , and K are highly correlated. Taking out two of them stabilizes somewhat, but dependence in the posterior distributions arises between other variables.

```
## block sampler
system.time({
  mcmcConf <- configureMCMC(model)
  mcmcConf$getSamplers()
  mcmcConf$removeSamplers(c("log_a", "log_K", "x0"))
  mcmcConf$addSampler(target = c("log_r", "log_H", "log_Q", "log_sigma"),
                      type = 'RW_block')
  mcmcConf$getSamplers()
})
```

Note: Assigning an RW_block sampler to nodes with very different scales can result in low MCMC efficiency.

```
## user system elapsed
## 8.077 0.046 8.157
```

```
system.time({
  mcmc <- buildMCMC(mcmcConf)
  Cmcmc <- compileNimble(mcmc, project = model, resetFunctions = T)
})
```

compiling... this may take a minute. Use 'showCompilerOutput = TRUE' to see C++ compilation details.

compilation finished.

```
## user system elapsed
## 17.981 0.432 18.533
```

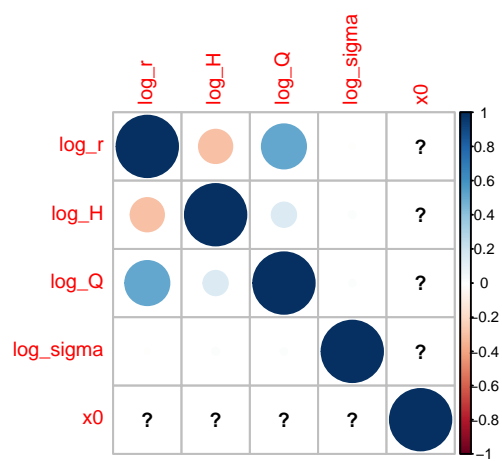
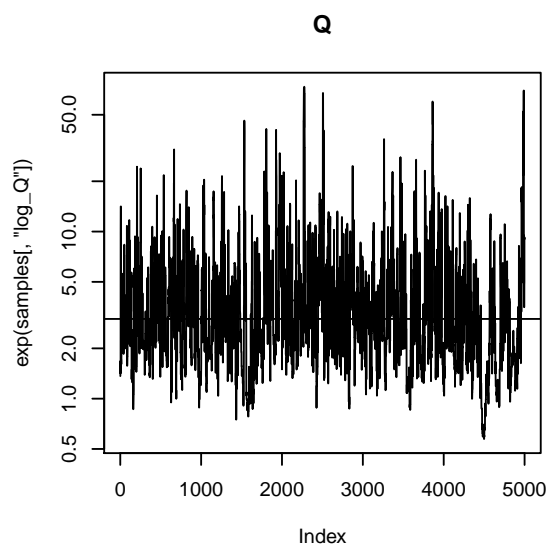
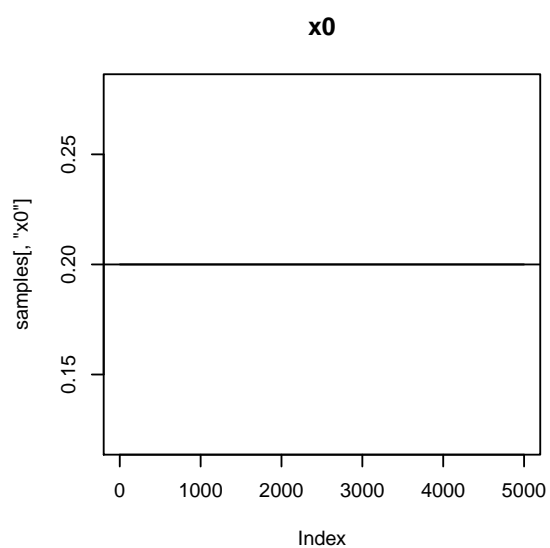
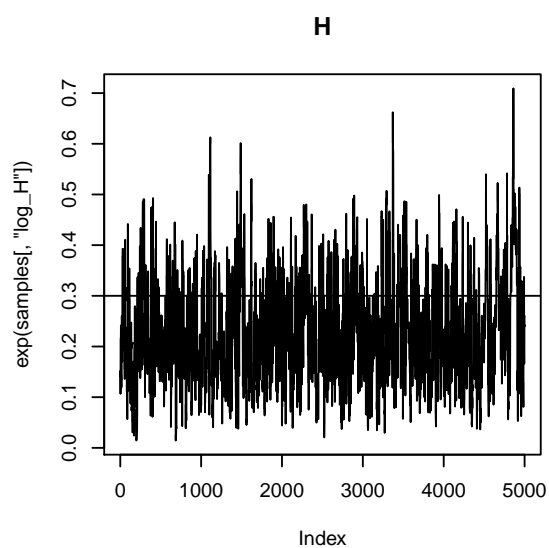
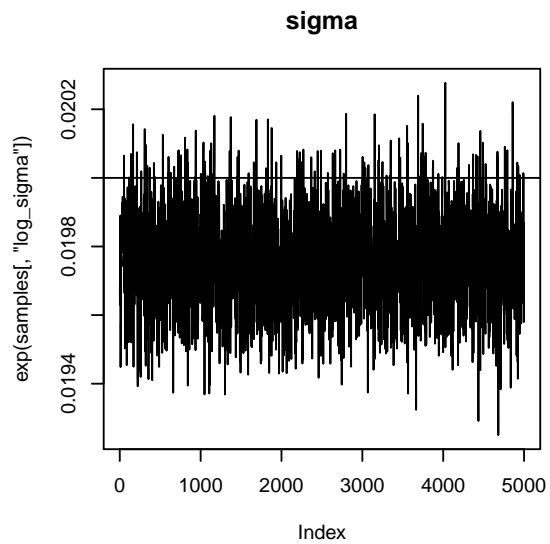
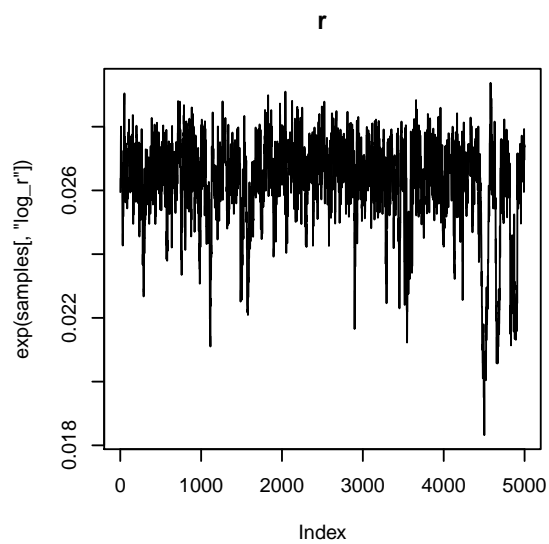
```
##
n_iterations <- 1e4
system.time({
  Cmcmc$run(n_iterations, nburnin = n_iterations / 2)
})
```

```
## |-----|-----|-----|-----|
## |-----|
##      user  system elapsed
## 108.063    0.408 109.393

samples <- as.matrix(Cmcmc$mvSamples)

layout(matrix(1:6, 3, 2))
plot(exp(samples[, 'log_r']), type = "l", main = "r")
abline(h = r)
plot(exp(samples[, 'log_H']), type = "l", main = "H")
abline(h = H)
plot(exp(samples[, 'log_Q']), type = "l", main = "Q", log = "y")
abline(h = Q)
plot(exp(samples[, 'log_sigma']), type = "l", main = "sigma")
abline(h = sigma)
plot(samples[, 'x0'], type = "l", main = "x0")
abline(h = x0)
corrplot::corrplot(cor(samples[, c("log_r", "log_H", "log_Q", "log_sigma", "x0")]))

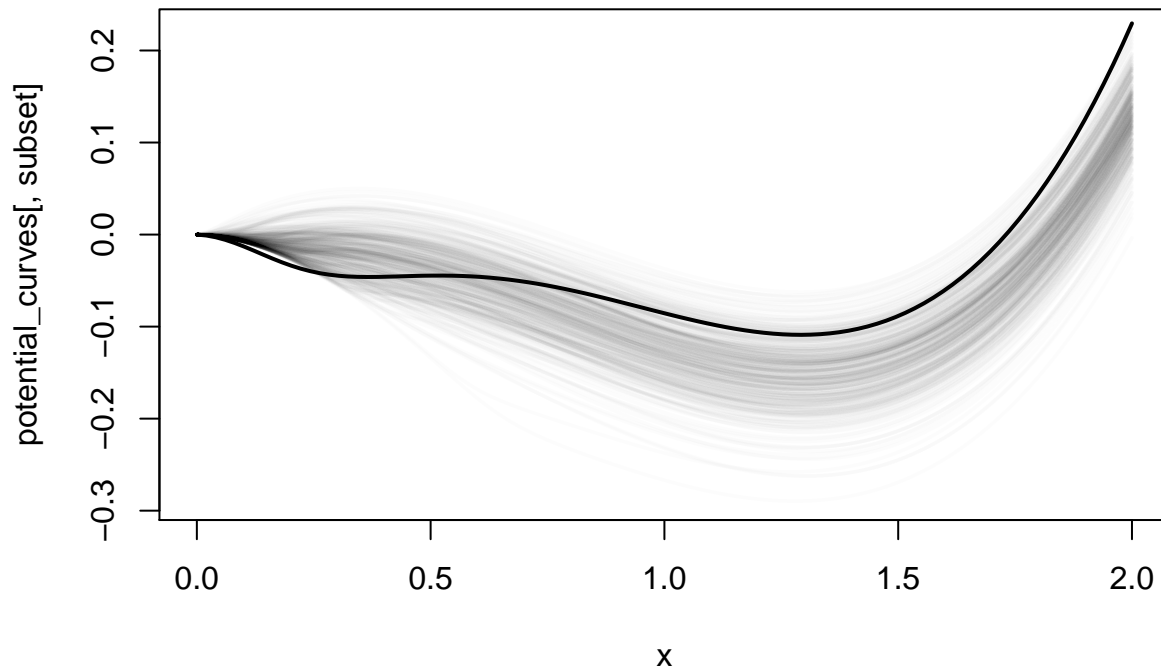
## Warning in cor(samples[, c("log_r", "log_H", "log_Q", "log_sigma", "x0")]):
## the standard deviation is zero
```



```

x <- seq(0, 2, l=1e2)
potential_curves <- apply(samples, 1, function(row){
  potential(x = x, a = exp(row['log_a']), r = exp(row['log_r']),
    H = exp(row['log_H']), Q = exp(row['log_Q']), K = exp(row['log_K']))
})
matplot(x, potential_curves[, subset], type = "l", lty = 1,
  col = scales::alpha(1, 1e-2), lwd = 2)
lines(x, potential(x = x, a = a, r = r, H = H, Q = Q, K = K), lwd = 2)

```



```

library(parallel)
cl <- makeForkCluster(2)
mus <- parApply(cl, samples[sample(1:nrow(samples), 100), ], 1,
  function(row){
    x <- rep(NA, N); x[1] <- row['x0']
    mu <- rep(NA, N - 1)
    for(t in 1:(N - 1)){
      mu[t] <-
        x[t] + x[t] * exp(row['log_r']) *
          (1 - x[t] / exp(row['log_K'])) -
        exp(row['log_a']) *
          x[t]^exp(row['log_Q']) / (x[t]^(row['log_Q']) +
            exp(row['log_H'])^(row['log_Q']))
      x[t + 1] <- rtruncnorm(mean = mu[t], sd = exp(row['log_sigma']))
    }
    return(mu)
  })
stopCluster(cl)
matplot(mus, type = "l", col = scales::alpha(1, 0.01), lty = 1)
lines(cmodel$mu)

```