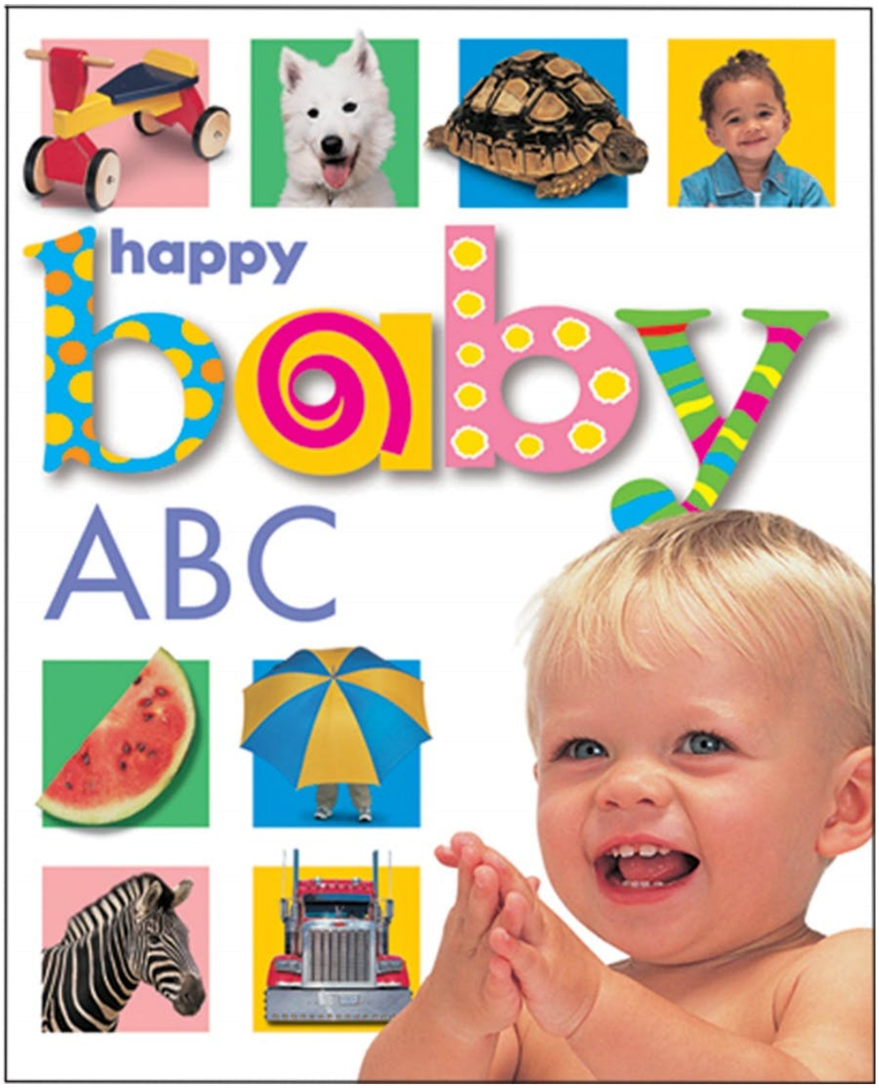
Introduction to Simulation for Biologists: Approximate Bayesian Computation (ABC) in `R`

Jun Ishigohoka

2024-06-10



Contents

Introduction

Likelihood function, which depicts the probability of observed data as a function of parameters of a statistical model, is essential in statistical inference. For many biological problems, it is challenging for us empiricists — and often even for theoreticians — to derive likelihood functions. **Approximate Bayesian Computation (ABC)** bypasses analytical evaluation of the likelihood function using simulation.

In ABC, the likelihood function is approximated by comparing simulated data (under a model with sampled parameters) with the observed data. In the simplest form of ABC with the rejection algorithm, a set of parameters are sampled from a prior distribution. Given a parameter θ, a dataset D_{sim} is simulated under a model M. Parameter θ under model M is accepted when

$$\rho(D_{sim}, D_{obs}) \leq \epsilon$$

where $\rho(D_{sim}, D_{obs})$ denotes the distance (e.g. Euclidean) between D_{sim} and D_{obs}, and $\epsilon \geq 0$ is tolerance. In simple words, this means that we reject a parameter θ under model M if the simulated data D_{sim} is too different from D_{obs}.

The probability of generating a dataset D_{sim} closer to the observed dataset D_{obs} than the distance measure of ϵ typically decreases as the dimensionality of the data increases (Think of throwing 100 coins otherwise identical but with 100 different colours). This is problematic for ABC because it reduces the computational efficiency. To deal with this issue, a set of summary statistics $S(D)$ with a lower dimensionality is used, instead of raw data D. In the Luria-Delbrück experiment, one can use mean and variance of resistant colonies per plate instead of frequency distribution of number of colonies. By substituting D_{sim} and D_{obs} with $S(D_{sim})$ and $S(D_{obs})$, the acceptance criterion of the ABC rejection algorithm becomes

$$\rho(S(D_{sim}), S(D_{obs})) \leq \epsilon$$

By applying this rejection algorithm to all simulated data under model M, we obtain a subset of sampled parameter values, the distribution of which can be regarded as an approximation to the posterior distribution.

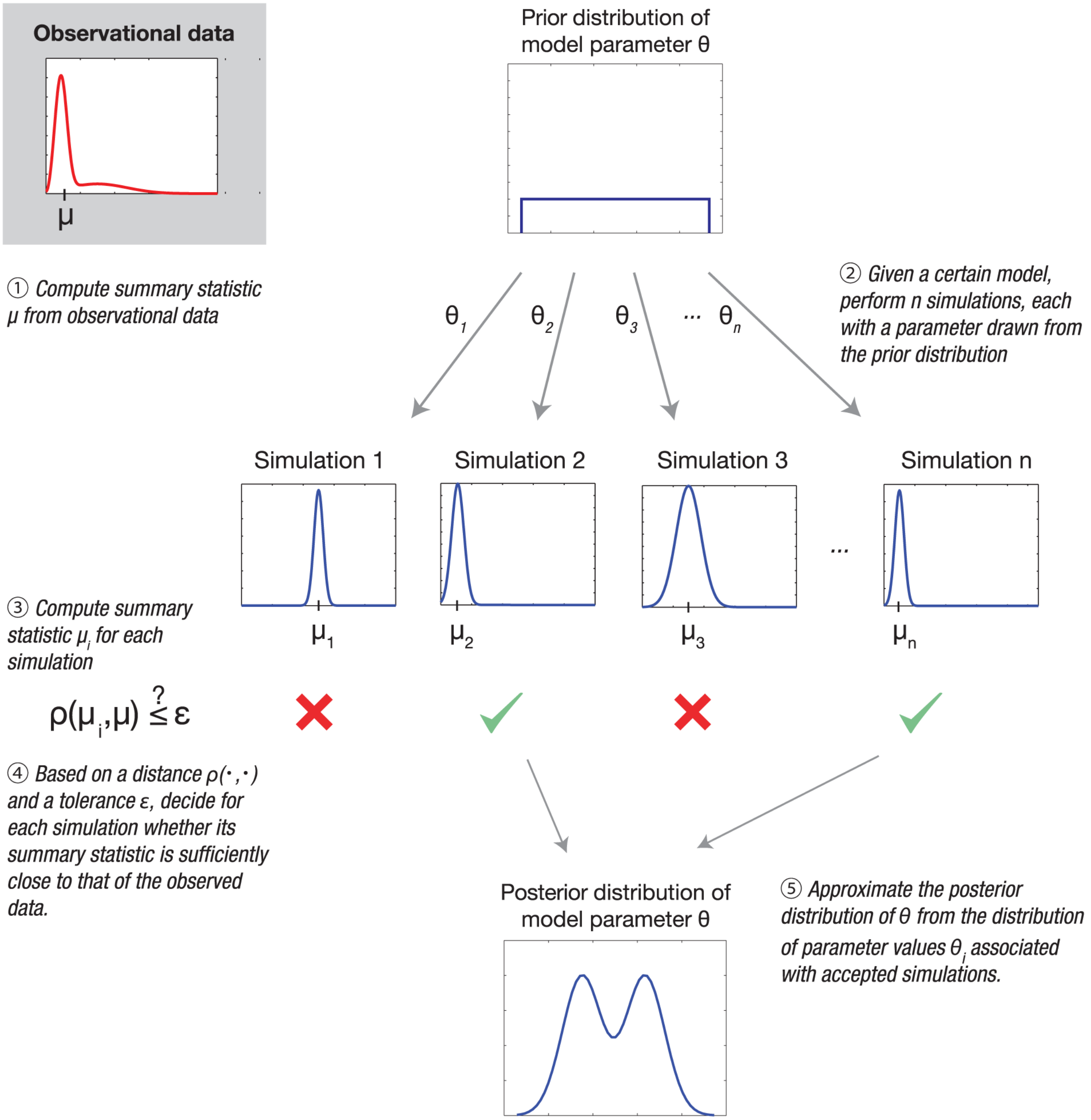

Observational data
μ
① Compute summary statistic μ from observational data
Prior distribution of model parameter θ
θ1
θ2
θ3
... θn
② Given a certain model, perform n simulations, each with a parameter drawn from the prior distribution
Simulation 1
Simulation 2
Simulation 3
Simulation n
...
μ1
μ2
μ3
μn
③ Compute summary statistic μi for each simulation
ρ(μi,μ) ≤? ε
④ Based on a distance ρ(·,·) and a tolerance ε, decide for each simulation whether its summary statistic is sufficiently close to that of the observed data.
Posterior distribution of model parameter θ
⑤ Approximate the posterior distribution of θ from the distribution of parameter values θi associated with accepted simulations.

Figure 1: Parameter estimation by ABC. Adapted from Sunnåker et al. (2013)

In addition to parameter inference described above, ABC can be used for model selection (Bertorelle, Benazzo, and Mona 2010; Csilléry, François, and Blum 2012). Posterior probability of a model M_i is approximated as

$$Pr(M_i|\rho(S(D_{sim}), S(D_{obs})) \leq \epsilon)$$

In simple words, the posterior probability of model M_i is approximated as the proportion of accepted simulations of model M_i out of accepted simulations of all models.

Here, we will apply ABC to the Luria-Delbrück experiment. The objectives are

1. to determine which of the induced or spontaneous mutation models is the case;
2. to estimate mutation rate under the model

In this exercise, we will apply ABC to the Luria-Delbrück experiment. First, we will determine which of models (induced and spontaneous mutation) fits better to data. Second, we will infer parameter (mutation rate). To this end, we will use `abc` package (Csilléry, François, and Blum 2012), which implements the rejection algorithm described above in functions. Optionally, you can try other methods than the rejection algorithm implemented in `abc` (multinomial logistic regression and neural networks for model selection; local linear regression and neural networks for parameter inference) and ABC random forest implemented in `abcrf` (Raynal et al. 2019).

Installation of required packages

We will use

- `abc`
- `abcrf`
- `fluctuateR`

`fluctuateR` contains a few functions defined during the first exercise. You can use your own functions instead if you want.

To install them, run:

```r
install.packages("abc")
install.packages("abcrf")
devtools::install_github("junishigohoka/fluctuateR")
```

To load them, run:

```r
library("abc")
library("abcrf")
library("fluctuateR")
```

Simulation of a Luria-Delbrück experiment

Here we will run a simulation, the output of which will be used as our observation. As biologists in 2024 we know that the spontaneous mutation model is the case, so we run this simulation using `simLD_spo`. The mutation rate is read from a compressed file `data/mu_truth.txt.gz`, which we will infer with ABC later.

To be a little bit more realistic, I suggest we change the parameter values of the experiment from the previous exercise.

First, I do not want to waste my working hours waiting for cells to grow. I also do not want to leave the lab late at night or come to the lab early in the morning. Because I work 8 hours a day, overnight is $24 - 8 = 16$ hours. Assuming the cells divide 3 times every hour, the number of generations overnight is $3 \times 16 = 48$. In addition, because I am lazy, I do not want to count cells before plating. So I would rather take a fixed proportion of the medium of the tube $(1/1,000,000,000)$ by serial dilution. Let's run an experiment and store the mean and variance of the number of resistant colonies in a named vector `d_obs`.

```r
T <- 48 # Number of generations
n_0 <- 100 # Initial number of cells in the tube
n_sample <- (n_0 * (2^T))/1e9 # Number of cells to plate
r <- 50 # Number of plates per experiment (A/B)


set.seed(1234)
d_obs <- simLD_spo(n_gens = T,
          mut_rate = as.numeric(readLines("data/mu_truth.txt.gz")),
          ncells_init = 100,
          n_sample = n_sample,
          n_plates = r
)
d_obs
```

```
##     mean_a     mean_b      var_a      var_b
##   28.80000   40.58000   27.63265 1575.84041
```

Simulation for inference

In ABC, we need to sample parameters and simulate data for models. In our Luria-Delbrück experiment, we have two models: induced and spontaneous mutation. We will simulate each model 10,000 times with mutation rates sampled from a log uniform distribution between 10^{-10} and 10^{-5}. We store sampled mutation rates in a vector `mu_sim`.

```
nsims <- 10000
mu_sim <- 10^runif(nsims, -10, -5)
head(mu_sim)
```

```
## [1] 6.539262e-09 2.281259e-08 1.990270e-10 2.274115e-09 8.646574e-06 1.351128e-07
```

First, let's simulate the Luria-Delbrück experiment for these mutation rates under the spontaneous mutation model. The simulated data will be in a data frame `d_sim_spo`. In the code block below, I simulate the data if simulated data has not been written in `data/d_sim_spo.rds` and store it in `data/d_sim_spo.rds`.

```
if(file.exists("data/d_sim_spo.rds")){
        d_sim_spo <- readRDS("data/d_sim_spo.rds")
}else{
        d_sim_spo <- as.data.frame(t(sapply(mu_sim,
               function(x){
                       simLD_spo(T, x, 100, n_0 * (2^T)/1e9, r)
               }
        )))
        saveRDS(d_sim_spo, "data/d_sim_spo.rds")
}



head(d_sim_spo)
```

```
##     mean_a   mean_b        var_a        var_b
## 1     8.62     6.70 5.832245e+00 1.037755e+01
## 2    24.66    23.20 2.622898e+01 1.266939e+02
## 3     0.10     0.10 9.183673e-02 9.183673e-02
## 4     2.30     2.14 1.765306e+00 2.163673e+00
## 5 13099.66 11173.52 1.152745e+04 1.118872e+07
## 6   137.26   145.34 1.071759e+02 1.489902e+03
```

Second, let's simulate the Luria-Delbrück experiment under the induced mutation model. The simulated data will be in a data frame `d_sim_ind`.

```
if(file.exists("data/d_sim_ind.rds")){
        d_sim_ind <- readRDS("data/d_sim_ind.rds")
}else{
        d_sim_ind <- as.data.frame(t(sapply(mu_sim,
               function(x){
```

```r
                       simLD_ind(n_plates = r, n_sample = as.integer(n_sample), mut_rate = x)
               }
        )))
        saveRDS(d_sim_ind, "data/d_sim_ind.rds")
}



head(d_sim_ind)
```

```
##   mean_a mean_b       var_a        var_b
## 1   0.16   0.18   0.1371429   0.15061224
## 2   0.50   0.70   0.5816327   0.62244898
## 3   0.00   0.00   0.0000000   0.00000000
## 4   0.00   0.10   0.0000000   0.09183673
## 5 245.12 243.28 136.9648980 202.94040816
## 6   4.32   4.52   3.6506122   4.37714286
```

Let's concatenate the data frames into a single data frame `d_sim`.

```r
d_sim <- as.data.frame(rbind(d_sim_spo, d_sim_ind))
head(d_sim)
```

```
##     mean_a   mean_b        var_a        var_b
## 1     8.62     6.70 5.832245e+00 1.037755e+01
## 2    24.66    23.20 2.622898e+01 1.266939e+02
## 3     0.10     0.10 9.183673e-02 9.183673e-02
## 4     2.30     2.14 1.765306e+00 2.163673e+00
## 5 13099.66 11173.52 1.152745e+04 1.118872e+07
## 6   137.26   145.34 1.071759e+02 1.489902e+03
```

```r
tail(d_sim)
```

```
##       mean_a mean_b        var_a       var_b
## 19995   1.44   1.60   2.21061224  1.42857143
## 19996   0.02   0.00   0.02000000  0.00000000
## 19997  88.30  87.58 104.78571429 82.37102041
## 19998   0.04   0.04   0.03918367  0.03918367
## 19999   0.00   0.00   0.00000000  0.00000000
## 20000   2.88   3.20   3.53632653  3.26530612
```

The models of in total 20,000 simulations are recorded in a vector `models`.

```r
models <- rep(c("spo", "ind"), each = nsims)
head(models)
```

```
## [1] "spo" "spo" "spo" "spo" "spo" "spo"
```

```r
tail(models)
```

```
## [1] "ind" "ind" "ind" "ind" "ind" "ind"
```

ABC using `abc`

This tutorial is based on the official vignette of `abc`, tailored for our Luria-Delbrück experiment. Interested readers are encouraged to read https://cran.r-project.org/web/packages/abc/vignettes/abcvignette.pdf as well as the original paper (Csilléry, François, and Blum 2012).

Cross-validation for model selection

To evaluate if ABC can, at all, distinguish between the two models, we perform a cross-validation for model selection using `abc::cv4postpr`, which implements a leave-one-out cross-validation. Here, we randomly take summary statistics of one simulation replicate as a pseudo-observation, and its parameter is estimated with the rejecting algorithm using all other simulations. This is repeated `nval = 100` times, and numbers of classifications are summarised in a confusion matrix.

```
cv_modsel_rej <- cv4postpr(index = models,
                       sumstat = d_sim,
                       method = "rejection",
                       nval = 100,
                       tols = 0.05
)
summary(cv_modsel_rej)
```

```
## Confusion matrix based on 100 samples for each model.
##
## $tol0.05
##     ind spo
## ind 100   0
## spo  32  68
##
##
## Mean model posterior probabilities (rejection)
##
## $tol0.05
##        ind    spo
## ind 0.8288 0.1712
## spo 0.2361 0.7639
```

In the confusion matrix, the j-th column of the i-th row is the number of times that the i-th model were classified as the j-th model. Our confusion matrix shows that if the truth is the induced model, then you can tell that it is induced. If the truth is the spontaneous model, then you can tell that it is induced 68 times out of 100. When the classified model is spontaneous, you can be sure that it is spontaneous. When the classified model is induced, the truth could still be the spontaneous model 32 times out of 132.

Model selection

Now, let's perform model selection with ABC using `abc::postpr`.

```
modsel_abc <- postpr(target = as.data.frame(t(d_obs)),
       index = models,
       sumstat = d_sim,
       tol = 0.05,
       method="rejection")

summary(modsel_abc)
```

```
## Call:
## postpr(target = as.data.frame(t(d_obs)), index = models, sumstat = d_sim,
##     tol = 0.05, method = "rejection")
## Data:
##  postpr.out$values (1000 posterior samples)
## Models a priori:
##  ind, spo
## Models a posteriori:
```

```
##  ind, spo
##
## Proportion of accepted simulations (rejection):
## ind spo
##   0   1
##
## Bayes factors:
##     ind spo
## ind       0
## spo Inf   1
```

The summary shows that the posterior probability of the spontaneous model is 1 (!!!).

Goodness-of-fit

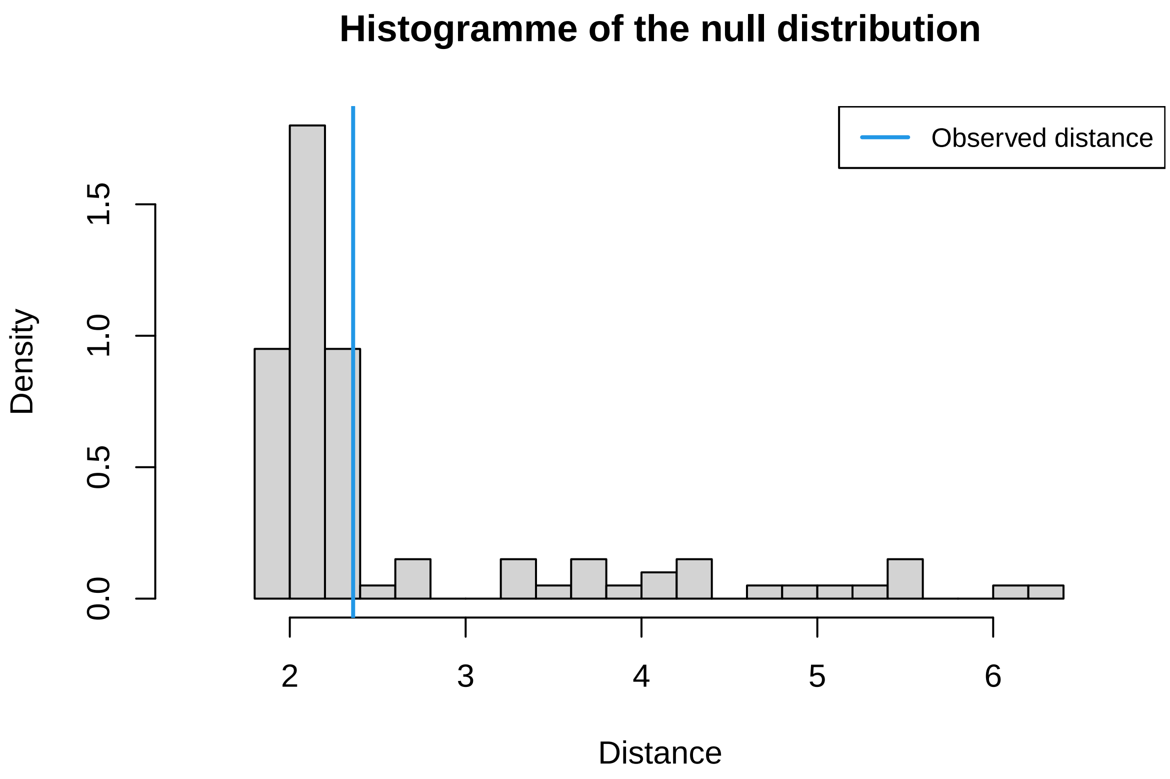
Before turning to parameter inference, it is important to check that the preferred model provides a good fit to the data. The null distribution is computed for the distance between true parameter value and the mean parameter values for accepted replicates with `abc` for `nb.replicate = 100` randomly sampled simulation replicates. The observed distance (i.e. the distance based on the output of `abc` using observed data) is compared with the null distribution. If the model provides a good fit to the data, then the observed distance should be within the null distribution (say, $p > 0.05$).

```
gfit_abc <- gfit(target=log10(as.data.frame(t(d_obs)) + 1),
             sumstat = log10(d_sim + 1),
             tol = 0.05,
             statistic=mean,
             nb.replicate=100)

summary(gfit_abc)
```

```
## $pvalue
## [1] 0.26
##
## $s.dist.sim
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   1.891   2.063   2.182   2.642   2.597   6.322
##
## $dist.obs
## [1] 2.360076
```

```
plot(gfit_abc)
```

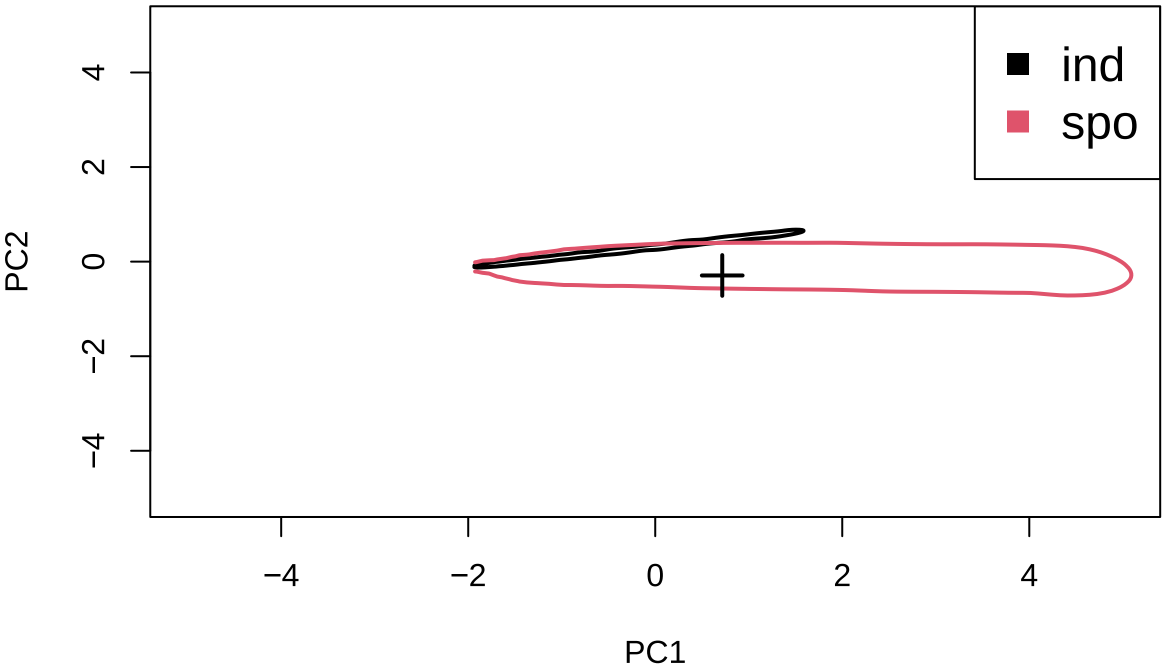


The observed distance is between mean and median of null distribution, so the model fits very well to the data.

Another way to investigate the goodness-of-fit is PCA. We can visualise the PCA envelope for the considered models and plot the observed data onto it using `abc::gfitpca`.

```
gfitpca(target = log10(as.data.frame(t(d_obs)) + 1),
        sumstat = log10(d_sim + 1),
        index = models,
        cprob = 0.1,
        xlim = c(-5, 5)
)
```

```
## Warning in lfproc(x, y, weights = weights, cens = cens, base = base, geth = geth, : procv: parameters
## Warning in lfproc(x, y, weights = weights, cens = cens, base = base, geth = geth, : procv: parameters
## Warning in lfproc(x, y, weights = weights, cens = cens, base = base, geth = geth, : procv: parameters
```



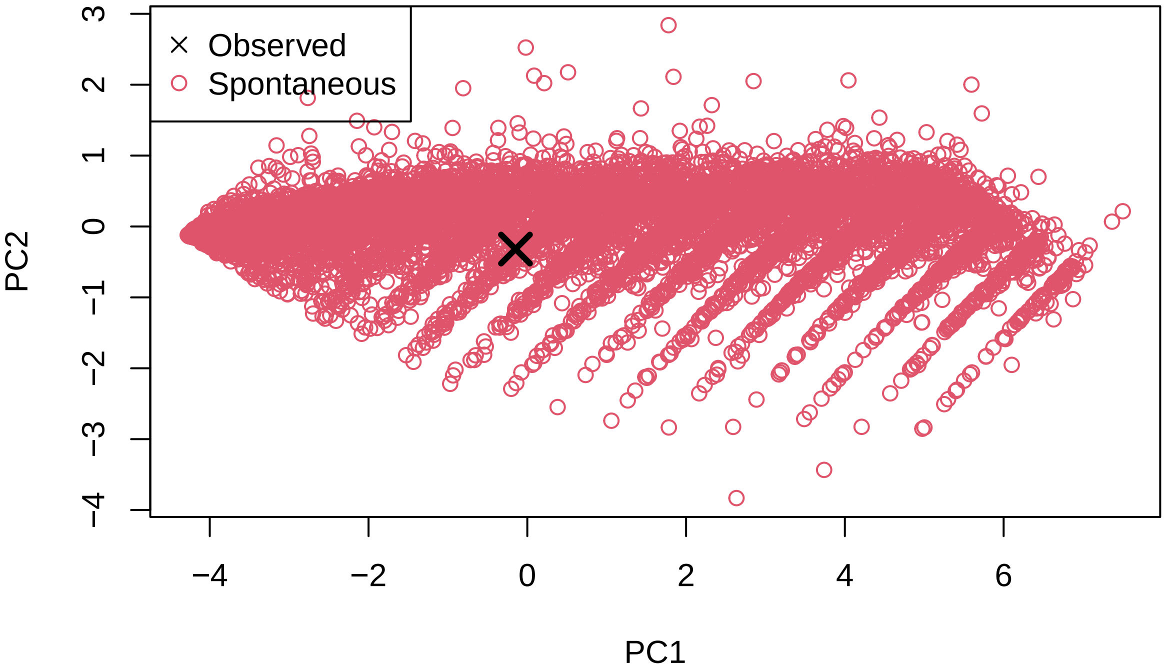
Alternatively, we could also use a base R function `prcomp`.

```
d_sim <- as.data.frame(rbind(d_sim_spo, d_sim_ind))


pca_both <- prcomp(log10(rbind(d_obs, d_sim_spo, d_sim_ind) + 1))
pca_spo <- prcomp(log10(rbind(d_obs, d_sim_spo) + 1))



plot(pca_spo$x[2:(nsims),1], pca_spo$x[2:(nsims),2], col = 2,
     xlab = "PC1",
     ylab = "PC2"
)
points(pca_spo$x[1,1], pca_spo$x[1,2], pch = c(4,1), cex = 2, lwd =3)
legend("topleft",
       legend = c("Observed", "Spontaneous"),
       col = c(1, 2),
       pch = c(4, 1),
       pt.cex = c(1, 1)
)
```



In either way, the spontaneous mutation model fits well with the observed data.

Cross-validation

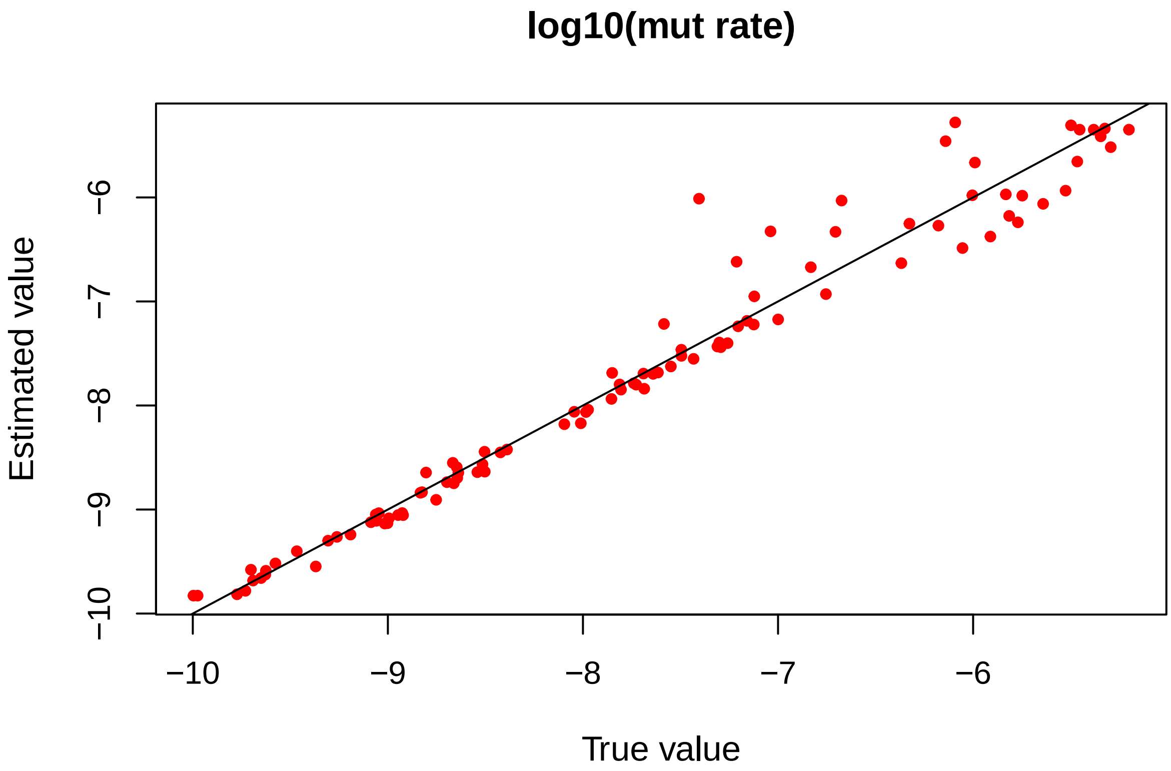
As in model selection, we should check if parameter inference of ABC (implemented in `abc::abc`) can estimate the parameter.

```
cv_parinf_rej <- cv4abc(param = log10(mu_sim),
                        sumstat = d_sim_spo,
                        nval=100,
                        tols=0.05,
                        method="rejection")
summary(cv_parinf_rej)
```

```
## Prediction error based on a cross-validation sample of 100
```

```
##              P1
## 0.05 0.03665123
```

```
plot(cv_parinf_rej, caption = "log10(mut rate)")
```



The log-transformed mutation rate can be estimated well with `abc::abc`.

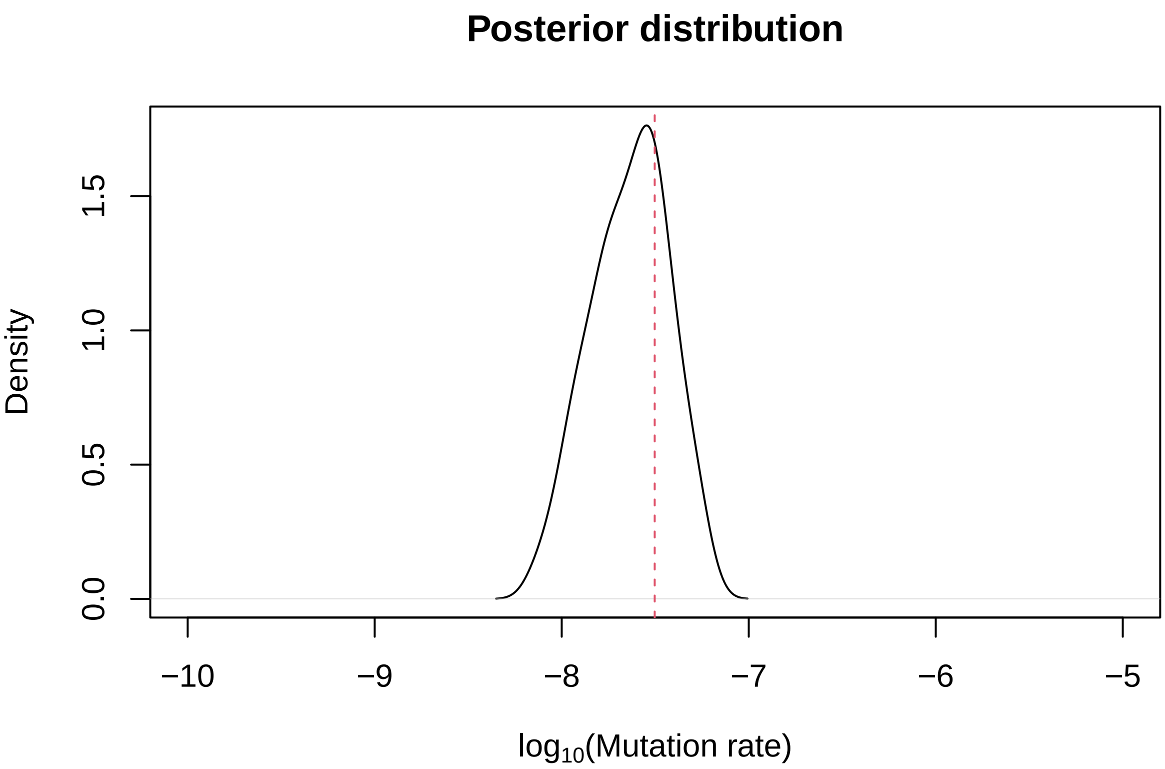
Parameter inference

Let's estimate the mutation rate.

```
parinf_abc <- abc(target = as.data.frame(t(d_obs)),
                  param = log10(mu_sim),
                  sumstat = d_sim_spo,
                  tol=0.01,
                  method="rejection")
```

```
## Warning in abc(target = as.data.frame(t(d_obs)), param = log10(mu_sim), : No parameter names are giv
```

```
plot(density(parinf_abc$unadj.value),
     xlim = c(-10, -5),
     xlab = expression(paste(log[10], "(Mutation rate)")),
     main = "Posterior distribution"
)
abline(v = log10(as.numeric(readLines("data/mu_truth.txt.gz"))),
       col = 2,
       lty = 2
)
```



The vertical line is the true mutation rate, so ABC with the simple rejection algorithm can tell the (order of) mutation rate very well.

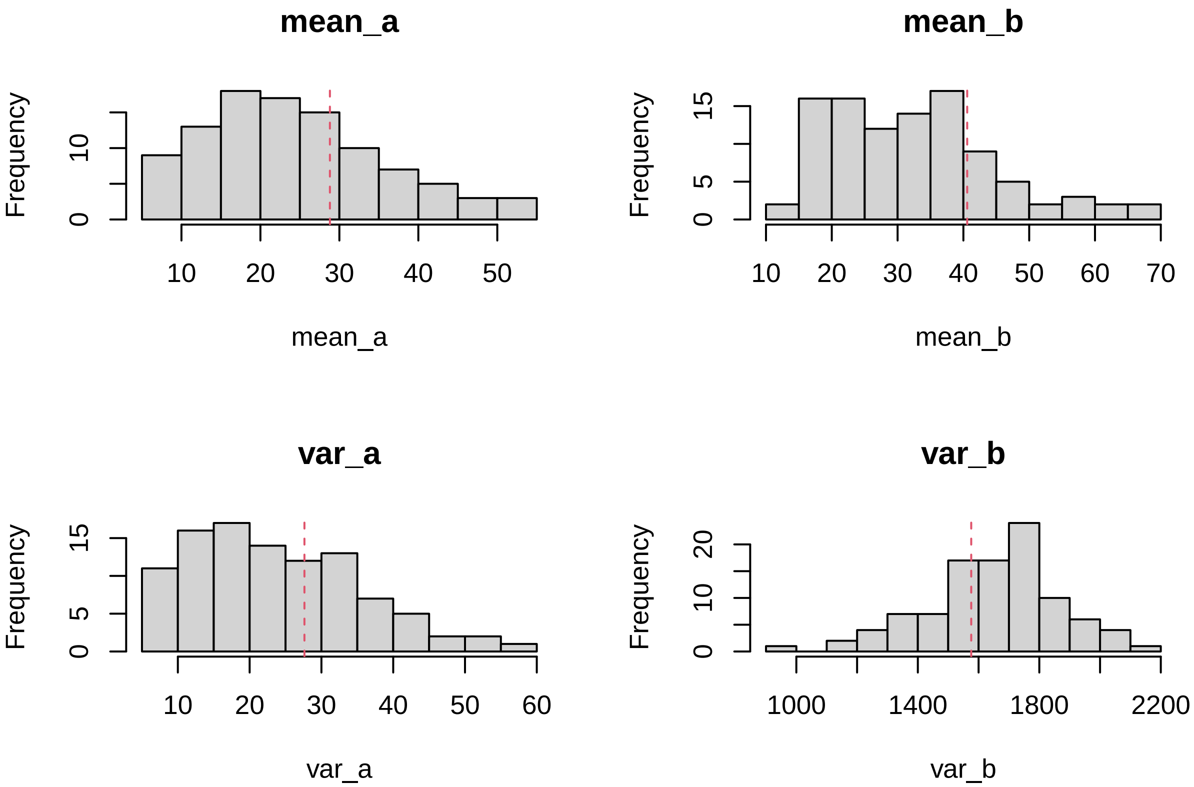
```r
10^quantile(parinf_abc$unadj.value, c(0.025, 0.5, 0.975))
```

```
##         2.5%          50%        97.5% 
## 9.397760e-09 2.413971e-08 5.292799e-08
```

Posterior predictive check

Lastly, let’s check if the observed summary statistics are well represented by the summary statistics of the accepted simulations. Note this is a bit of cheating: one should actually simulate *a posteori* using parameters sampled from the posterior.

```r
par(mfrow=c(2,2))
for(i in 1:4){
        hist(parinf_abc$ss[,i],
             main = colnames(parinf_abc$ss)[i],
             xlab = colnames(parinf_abc$ss)[i]
        )
        abline(v=d_obs[i], col = 2, lty = 2)
}
```



The observed summary statistics (vertical lines) are in the middle of those of accepted simulations.

References

Bertorelle, G., A. Benazzo, and S. Mona. 2010. "ABC as a Flexible Framework to Estimate Demography over Space and Time: Some Cons, Many Pros." *Molecular Ecology* 19 (13): 2609–25. https://doi.org/10.1111/j.1365-294X.2010.04690.x.

Csilléry, Katalin, Olivier François, and Michael G. B. Blum. 2012. "Abc: An R Package for Approximate Bayesian Computation (ABC)." *Methods in Ecology and Evolution* 3 (3): 475–79. https://doi.org/10.1111/j.2041-210X.2011.00179.x.

Raynal, Louis, Jean-Michel Marin, Pierre Pudlo, Mathieu Ribatet, Christian P Robert, and Arnaud Estoup. 2019. "ABC Random Forests for Bayesian Parameter Inference." *Bioinformatics* 35 (10): 1720–28. https://doi.org/10.1093/bioinformatics/bty867.

Sunnåker, Mikael, Alberto Giovanni Busetto, Elina Numminen, Jukka Corander, Matthieu Foll, and Christophe Dessimoz. 2013. "Approximate Bayesian Computation." *PLOS Computational Biology* 9 (1): e1002803. https://doi.org/10.1371/journal.pcbi.1002803.

Appendix

ABC random forest using abcrf

Model selection

```
model_rf <- abcrf(formula = as.factor(models)~.,
                  data = d_sim,
                  ntree = 1000,
                  paral = TRUE,
                  ncores = 8
)
```

```
modsel_rf <- predict(object = model_rf,
                     obs = as.data.frame(t(d_obs)),
                     training = d_sim,
                     ntree = 1000,
                     paral = TRUE,
                     paral.predict = TRUE
)

modsel_rf
```

```
##   selected model votes model1 votes model2 post.proba
## 1            spo            0         1000          1
```

Parameter inference

```
model <- regAbcrf(formula = log10(mu_sim) ~ .,
                  data = d_sim_spo,
                  ntree = 1000,
                  paral = TRUE
)
```

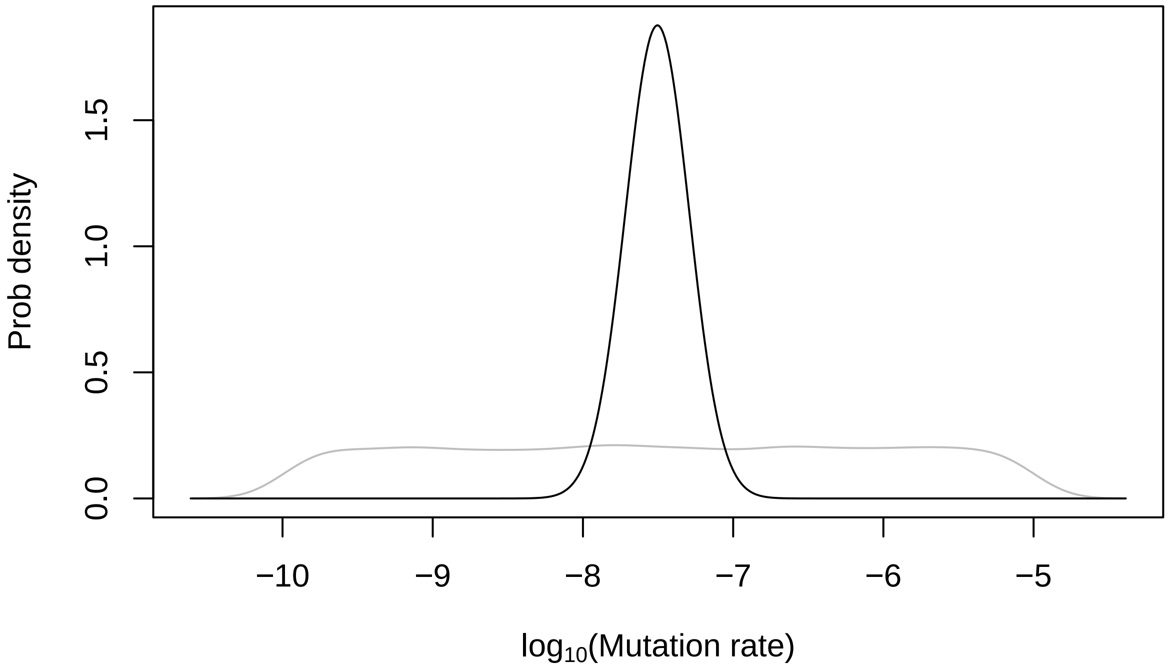
```
posterior <- predict(object = model,
                     obs = as.data.frame(t(d_obs)),
                     training = d_sim_spo,
                     paral = TRUE,
                     rf.writes = T
)

print(posterior)
```

```
##      expectation   median variance (post.MSE.mean) variance.cdf quantile=0.025 quantile=0.975 post.N
## [1,]   -7.505277 -7.49556             0.0008119156  0.003314681      -7.625142      -7.407507    0.0
```

```
densityPlot(object = model, obs = as.data.frame(t(d_obs)), training = d_sim_spo, paral = TRUE,
            xlab = expression(paste(log[10], "(Mutation rate)")),
            ylab = "Prob density",
            main = ""
)
```

```
## Warning in density.default(resp, weights = weights.std[, i], ...): Selecting bandwidth *not* using '
## Warning in density.default(resp, weights = weights.std[, i], ...): Selecting bandwidth *not* using '
```



```r
smr <- c(10^posterior$med, 10^posterior$quantiles)
names(smr) <- c("median", "q2.5", "q97.5")
print(smr)
```

```
##       median         q2.5        q97.5
## 3.194771e-08 2.370600e-08 3.912845e-08
```

Truth

```r
print(as.numeric(readLines("data/mu_truth.txt.gz")))
```

```
## [1] 3.141593e-08
```