Approximate Bayesian Computation Lab

Kai Liu & Kavitha N.

October 21, 2015

There are two packages are available in R to implement approximate bayesian computations – `abc` and `EasyABC`.

First, load the packages:

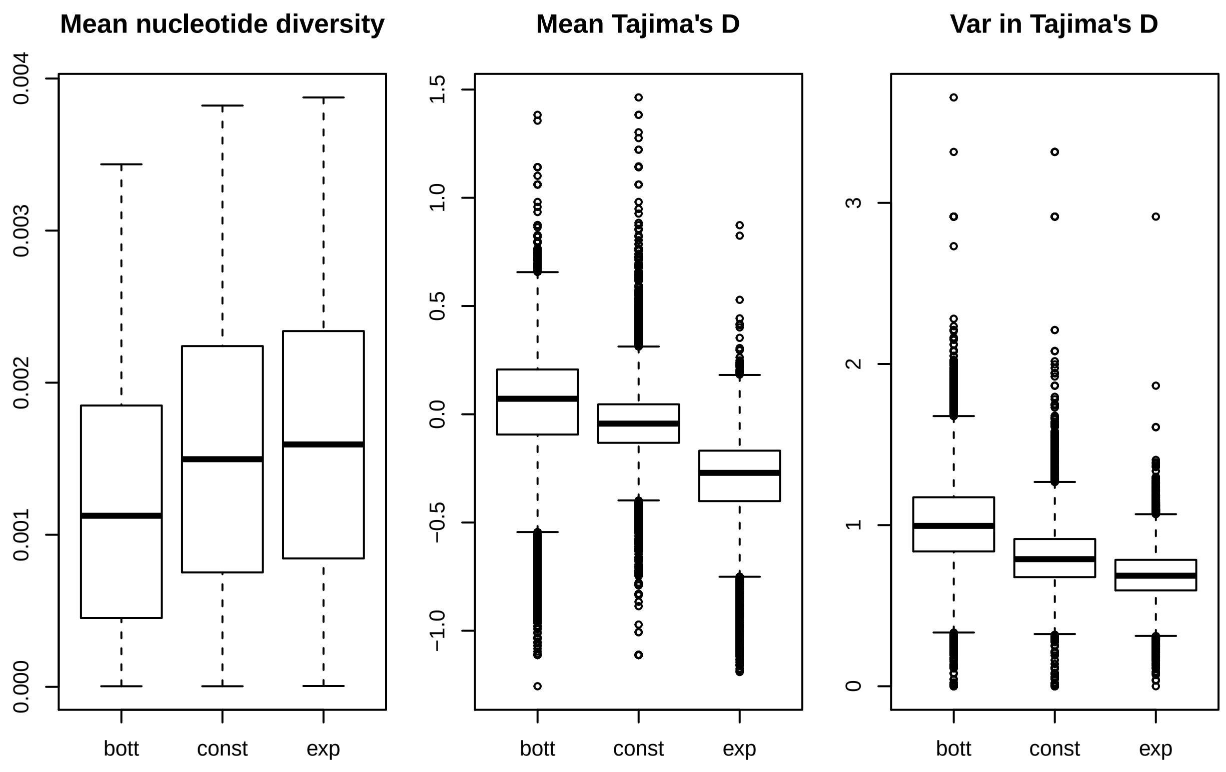
```
library("abc")
library("EasyABC")
```

abc

```
library("abc")
require(abc.data)
data(musigma2)
require(abc.data)
data(human)
stat.voight
```

```
##               pi TajD.m TajD.v
## hausa   0.00110  -0.20   0.55
## italian 0.00085   0.28   1.19
## chinese 0.00079   0.18   1.08
```

```
?stat.voight
 par(mfcol = c(1,3), mar=c(5,3,4,.5))
boxplot(stat.3pops.sim[,"pi"]~models, main="Mean nucleotide diversity")
boxplot(stat.3pops.sim[,"TajD.m"]~models, main="Mean Tajima's D")
boxplot(stat.3pops.sim[,"TajD.v"]~models, main="Var in Tajima's D")
```



```
cv.modsel <- cv4postpr(models, stat.3pops.sim, nval=5, tol=.01, method="mnlogistic")
```

```
## Warning: There are 3 models but only 2 for which simulations have been accepted.
## No regression is performed, method is set to rejection.
## Consider increasing the tolerance rate.TRUE
```

```
s <- summary(cv.modsel)
```

```
## Confusion matrix based on 5 samples for each model.
##
## $tol0.01
##       bott const exp
## bott     5     0   0
## const    1     3   1
## exp      0     0   5
##
##
## Mean model posterior probabilities (mnlogistic)
##
## $tol0.01
##         bott  const    exp
## bott  0.8273 0.1535 0.0192
## const 0.2508 0.5083 0.2409
## exp   0.0980 0.2280 0.6740
```

```
plot(cv.modsel, names.arg=c("Bottleneck", "Constant", "Exponential"))
modsel.ha <- postpr(stat.voight["hausa",],
                    models, stat.3pops.sim,
```

```r
                    tol=.05, method="mnlogistic")
modsel.it <- postpr(stat.voight["italian",],
                    models, stat.3pops.sim,
                    tol=.05, method="mnlogistic")
modsel.ch <- postpr(stat.voight["chinese",],
                    models, stat.3pops.sim,
                    tol=.05, method="mnlogistic")
summary(modsel.ha)
```

```
## Call:
## postpr(target = stat.voight["hausa", ], index = models, sumstat = stat.3pops.sim,
##     tol = 0.05, method = "mnlogistic")
## Data:
##  postpr.out$values (7500 posterior samples)
## Models a priori:
##  bott, const, exp
## Models a posteriori:
##  bott, const, exp
##
## Proportion of accepted simulations (rejection):
##   bott  const    exp
## 0.0199 0.3132 0.6669
##
## Bayes factors:
##          bott   const     exp
## bott   1.0000  0.0634  0.0298
## const 15.7651  1.0000  0.4696
## exp   33.5705  2.1294  1.0000
##
##
## Posterior model probabilities (mnlogistic):
##   bott  const    exp
## 0.0164 0.3591 0.6245
##
## Bayes factors:
##          bott   const     exp
## bott   1.0000  0.0456  0.0262
## const 21.9360  1.0000  0.5751
## exp   38.1446  1.7389  1.0000
```

```r
summary(modsel.it)
```

```
## Call:
## postpr(target = stat.voight["italian", ], index = models, sumstat = stat.3pops.sim,
##     tol = 0.05, method = "mnlogistic")
## Data:
##  postpr.out$values (7500 posterior samples)
## Models a priori:
##  bott, const, exp
## Models a posteriori:
##  bott, const, exp
##
```

```
## Proportion of accepted simulations (rejection):
##   bott  const    exp
## 0.8487 0.1509 0.0004
##
## Bayes factors:
##            bott     const       exp
## bott     1.0000    5.6228 2121.6667
## const    0.1778    1.0000  377.3333
## exp      0.0005    0.0027    1.0000
##
##
## Posterior model probabilities (mnlogistic):
##   bott  const    exp
## 0.9369 0.0628 0.0004
##
## Bayes factors:
##            bott     const       exp
## bott     1.0000   14.9246 2508.9790
## const    0.0670    1.0000  168.1105
## exp      0.0004    0.0059    1.0000
```

```r
summary(modsel.ch)
```

```
## Call:
## postpr(target = stat.voight["chinese", ], index = models, sumstat = stat.3pops.sim,
##     tol = 0.05, method = "mnlogistic")
## Data:
##  postpr.out$values (7500 posterior samples)
## Models a priori:
##  bott, const, exp
## Models a posteriori:
##  bott, const, exp
##
## Proportion of accepted simulations (rejection):
##   bott  const    exp
## 0.6837 0.3159 0.0004
##
## Bayes factors:
##            bott     const       exp
## bott     1.0000    2.1646 1709.3333
## const    0.4620    1.0000  789.6667
## exp      0.0006    0.0013    1.0000
##
##
## Posterior model probabilities (mnlogistic):
##   bott  const    exp
## 0.7610 0.2389 0.0001
##
## Bayes factors:
##             bott      const        exp
## bott      1.0000     3.1853 10840.7807
## const     0.3139     1.0000  3403.3749
## exp       0.0001     0.0003     1.0000
```

```r
res.gfit.bott=gfit(target=stat.voight["italian",],
                   sumstat=stat.3pops.sim[models=="bott",],
                   statistic=mean, nb.replicate=100)
plot(res.gfit.bott, main="Histogram under H0")

 res.gfit.exp=gfit(target=stat.voight["italian",],
                   sumstat=stat.3pops.sim[models=="exp",],
                   statistic=mean, nb.replicate=100)
 res.gfit.const=gfit(target=stat.voight["italian",],
                     sumstat=stat.3pops.sim[models=="const",],
                     statistic=mean, nb.replicate=100)
 summary(res.gfit.bott)
```

```
## $pvalue
## [1] 0.55
##
## $s.dist.sim
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   1.600   1.830   2.057   2.175   2.377   4.337
##
## $dist.obs
## [1] 1.997769
```

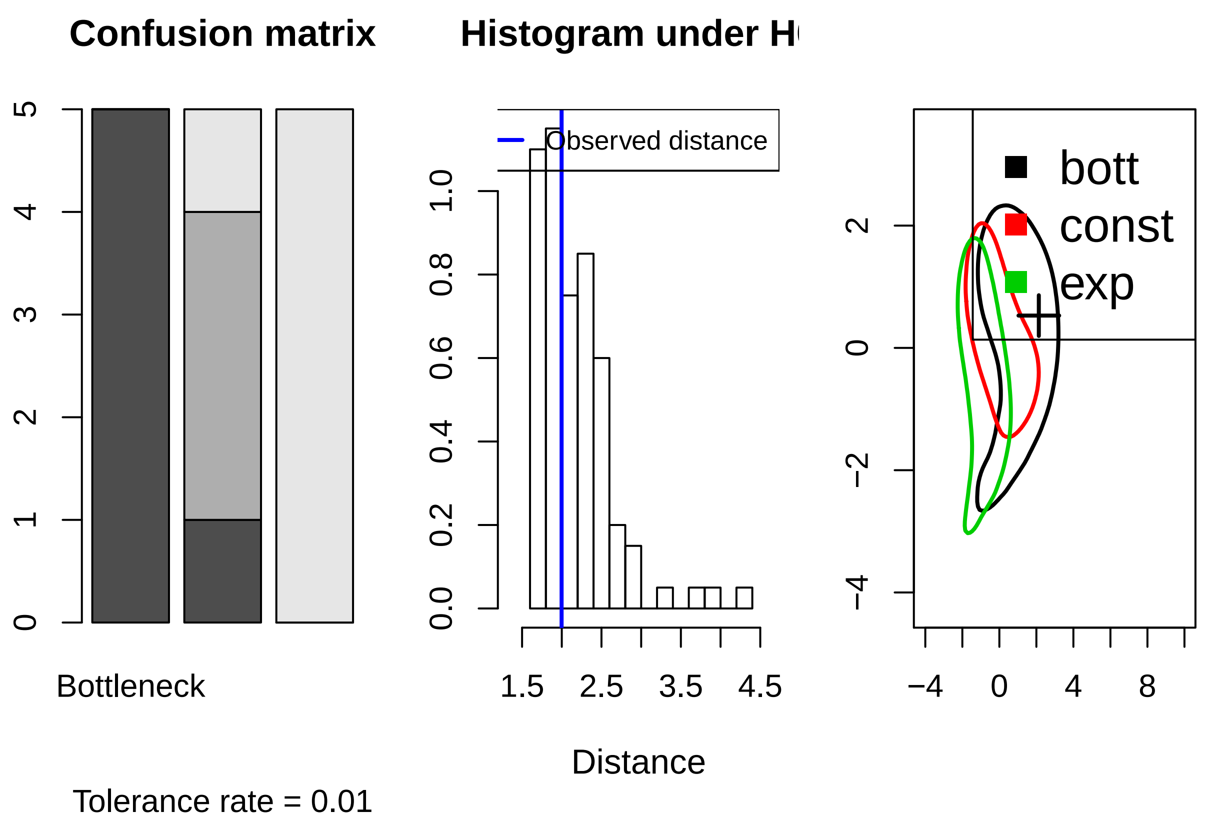
```r
 summary(res.gfit.exp)
```

```
## $pvalue
## [1] 0
##
## $s.dist.sim
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   1.589   1.784   2.116   2.202   2.469   4.820
##
## $dist.obs
## [1] 5.181088
```

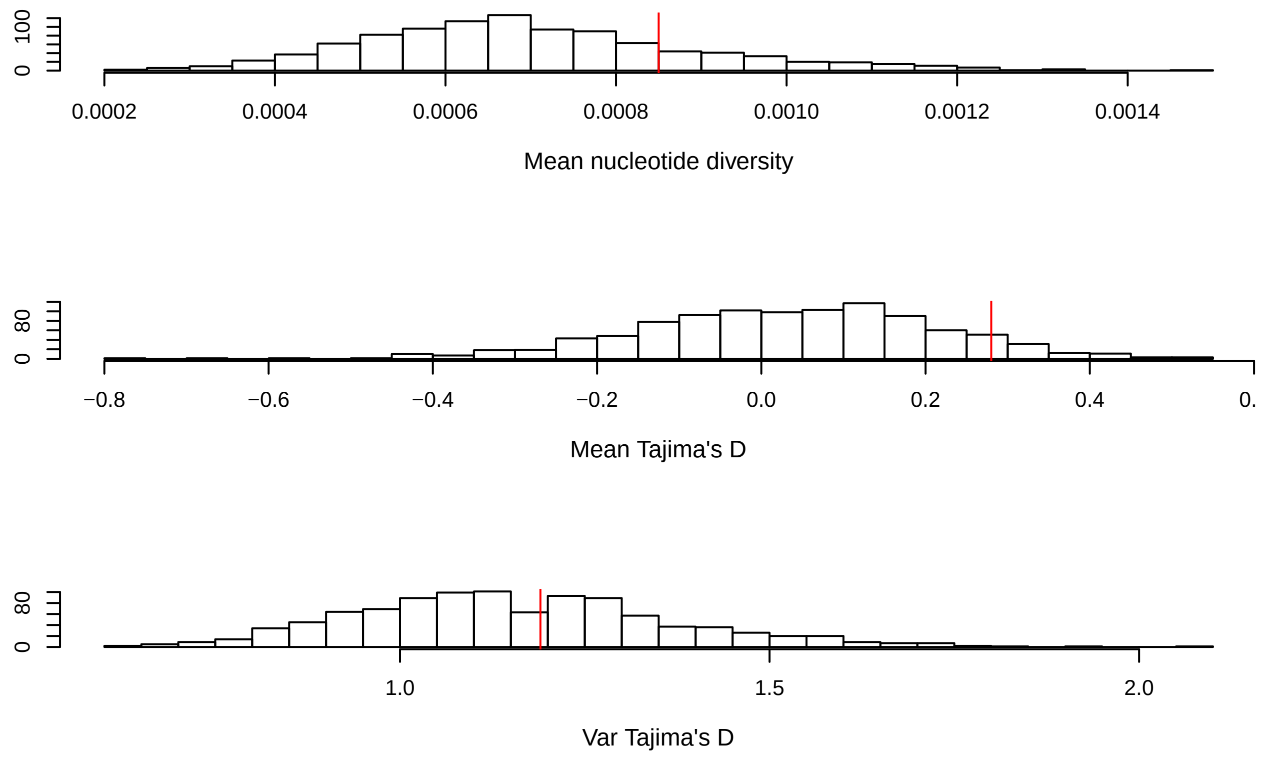
```r
 summary(res.gfit.const)
```

```
## $pvalue
## [1] 0.03
##
## $s.dist.sim
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   1.545   1.800   1.953   2.094   2.188   5.388
##
## $dist.obs
## [1] 3.610239
```

```r
 gfitpca(target=stat.voight["italian",],
         sumstat=stat.3pops.sim, index=models, cprob=.1)
```



```
 require(abc.data)
data(ppc)
mylabels <- c("Mean nucleotide diversity","Mean Tajima's D", "Var Tajima's D")
par(mfrow = c(3,1), mar=c(5,2,4,0))
for (i in c(1:3)){
  hist(post.bott[,i],breaks=40, xlab=mylabels[i], main="")
  abline(v = stat.voight["italian", i], col = 2)
}
```



```r
stat.italy.sim <- subset(stat.3pops.sim, subset=models=="bott")
head(stat.italy.sim)
```

```
##                   pi      TajD.m    TajD.v
## 110002 0.0010531112 0.057649360 1.3024486
## 210000 0.0012026666 0.342608757 1.1042096
## 310000 0.0008920002 0.247601809 0.9147642
## 410002 0.0015328889 0.470767633 1.5339313
## 51002  0.0008988888 0.008920568 1.0751214
## 61002  0.0019851108 0.208063980 0.8684532
```

```r
head(par.italy.sim)
```

```
##          Ne        a duration    start
## 1 14621.380 28.58894 5428.309 40421.04
## 2 13098.281 18.65774 5081.701 43255.28
## 3  7936.504 12.00553 3369.698 50781.08
## 4 17823.659 42.57813 7857.355 46397.16
## 5 12294.555 23.60331 6633.745 58208.44
## 6 25626.369 40.09893 7067.745 50385.51
```

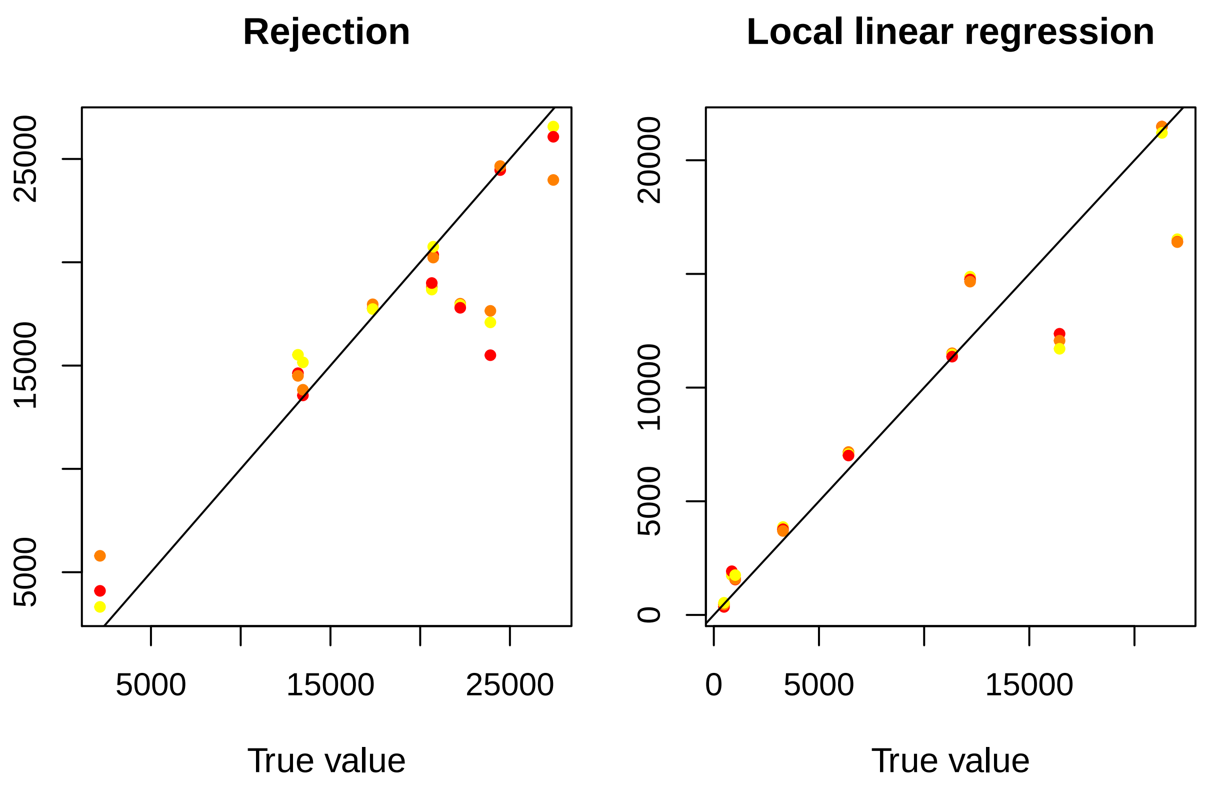
```r
 cv.res.rej <- cv4abc(data.frame(Na=par.italy.sim[,"Ne"]),
                      stat.italy.sim, nval=10,
                      tols=c(.005,.01, 0.05), method="rejection")
 cv.res.reg <- cv4abc(data.frame(Na=par.italy.sim[,"Ne"]),
                      stat.italy.sim, nval=10,
                      tols=c(.005,.01, 0.05), method="loclinear")
summary(cv.res.rej)
```

```
## Prediction error based on a cross-validation sample of 10

##                Na
## 0.005 0.1195803
## 0.01  0.1413346
## 0.05  0.2326595
```

```
summary(cv.res.reg)
```

```
## Prediction error based on a cross-validation sample of 10

##                 Na
## 0.005 0.08021701
## 0.01  0.08442814
## 0.05  0.08914801
```

```
 par(mfrow=c(1,2), mar=c(5,3,4,.5), cex=.8)
 plot(cv.res.rej, caption="Rejection")
plot(cv.res.reg, caption="Local linear regression")
```



```
 res <- abc(target=stat.voight["italian",],
            param=data.frame(Na=par.italy.sim[, "Ne"]),
            sumstat=stat.italy.sim, tol=0.05,
            transf=c("log"), method="neuralnet")
```

```
## 12345678910
## 12345678910
```

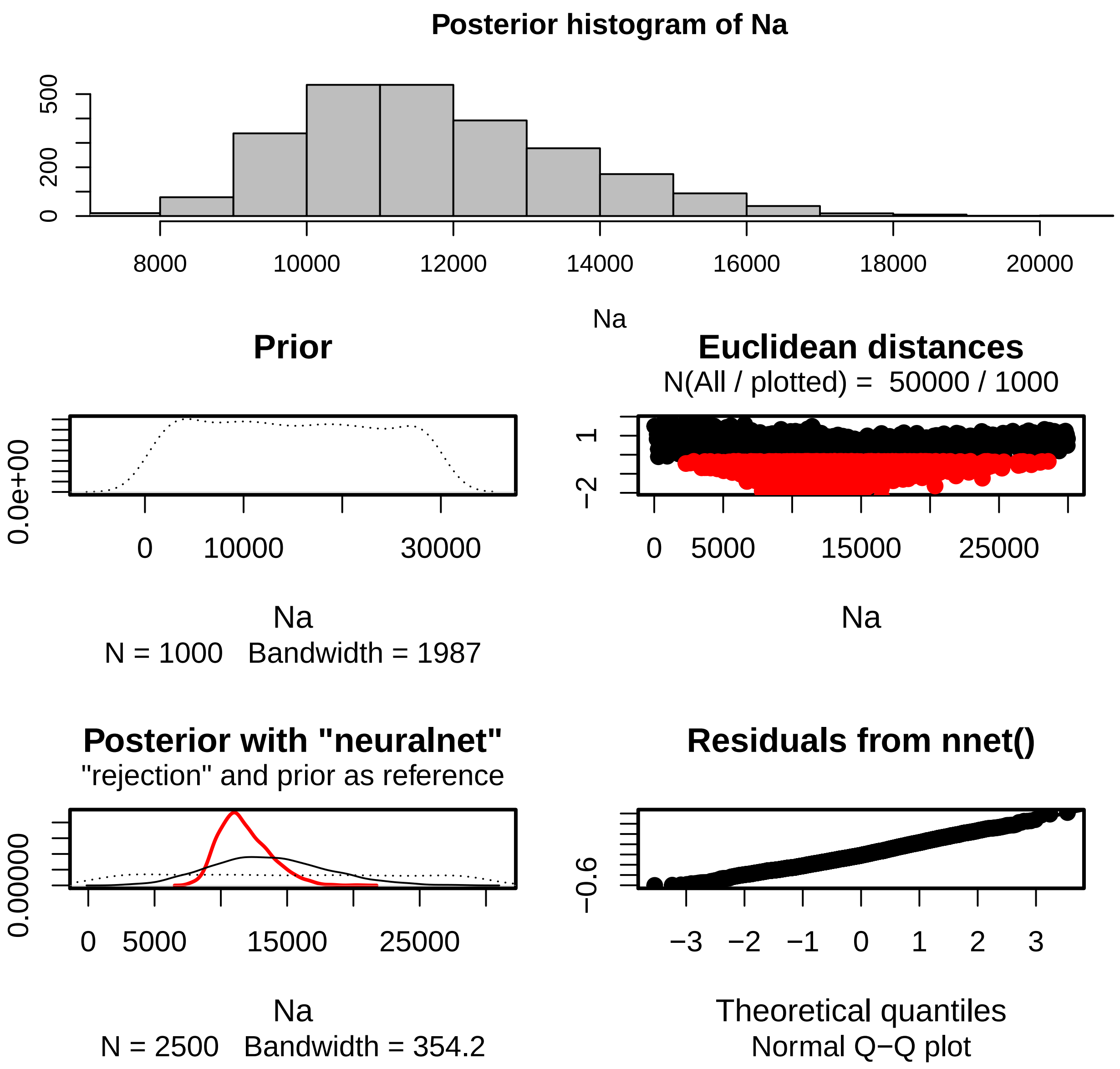
```r
res
```

```
## Call:
## abc(target = stat.voight["italian", ], param = data.frame(Na = par.italy.sim[,
##     "Ne"]), sumstat = stat.italy.sim, tol = 0.05, method = "neuralnet",
##     transf = c("log"))
## Method:
## Non-linear regression via neural networks
## with correction for heteroscedasticity
##
## Parameters:
## Na
##
## Statistics:
## pi, TajD.m, TajD.v
##
## Total number of simulations 50000
##
## Number of accepted simulations:  2500
```

```r
summary(res)
```

```
## Call:
## abc(target = stat.voight["italian", ], param = data.frame(Na = par.italy.sim[,
##     "Ne"]), sumstat = stat.italy.sim, tol = 0.05, method = "neuralnet",
##     transf = c("log"))
## Data:
##  abc.out$adj.values (2500 posterior samples)
## Weights:
##  abc.out$weights
##
##                                 Na
## Min.:                     7572.546
## Weighted 2.5 % Perc.:     8748.055
## Weighted Median:         11445.056
## Weighted Mean:           11744.610
## Weighted Mode:           11043.490
## Weighted 97.5 % Perc.:   15991.502
## Max.:                    20688.275
```

```r
par(mfrow=c(2,1),cex=.8)
hist(res)
plot(res, param=par.italy.sim[, "Ne"])
```



EasyABC

Model

Let's consider a stochastic individual-based model to demonstrate how `EasyABC` can be used. This model is drawn from Jabot (2010), representing the stochastic dynamics of an ecological community.

Each species in the community are given by a local competitive ability as determined by a filtering function of one quantitative trait t: $F(t) = 1 + A \exp\left(\frac{-(t-h)^2}{2\sigma^2}\right)$. At each time step, one individual drawn at random dies in a local community of size J. It is replaced either by an immigrant from the regional pool with probability $\frac{I}{I+J-1}$ or by the descendant of a local individual. Parameter I measures the amount of immigration from the regional pool into the local community. The probability that the replacing individual is of species i is proportional to the abundance of this species in the local community multiplied by its local competitive ability F_i. Here, the parameters of interest are I, h, A, σ and the local community size J is fixed at 500. The summary statistics are species richness of the community, Shannon's index, the mean of the trait value among individuals and the skewness of the trait value distribution. The model is a built-in model in this package.

ABC schemes

There are 4 types of schemes available in `EasyABC`: standard rejection algorithm, sequential schemes, coupled to MCMC sequential schemes and a Simulated Annealing algorithm, implemented by `ABC_rejection()`, `ABC_sequential()`, `ABC_mcmc()` and `SABC()`, respectively.

All these functions require a model used to generate data and return a set of summary statistics, prior distributions, summary statistics from the observed data.

In our example, we have summary statistics of the observed data

```
sum_stat_obs <- c(100,2.5,20,30000)
```

and assume prior distributions

```
trait_prior <- list(c("unif",3,5),
                    c("unif",-2.3,1.6),
                    c("unif",-25,125),
                    c("unif",-.7,3.2))
```

First, let's look at the abc rejection algorithm.

```
set.seed(9)
ABC_rej <- ABC_rejection(model=trait_model, prior=trait_prior,nb_simul=100,
                          summary_stat_target=sum_stat_obs,tol=.1,use_seed=T)
ABC_rej
```

```
## $param
##           [,1]       [,2]       [,3]        [,4]
##  [1,] 3.443203 -2.2054878   6.067853  0.14136084
##  [2,] 4.025082  0.7987833  -1.574728  0.04390927
##  [3,] 4.298096  0.9562522   5.929186 -0.22679304
##  [4,] 4.920236  0.5652761  18.788669  0.22884513
##  [5,] 4.221460  1.0181108  -7.818319  2.77775992
##  [6,] 3.555287 -1.9756224   4.499501  2.25331504
##  [7,] 4.578879 -0.4494255   9.807345 -0.18367680
##  [8,] 4.468684  0.2788487   8.046597  0.36940514
##  [9,] 4.733991  1.0632640  11.948136  0.97694971
## [10,] 4.286998  1.5845963 -24.536332  2.69545417
##
## $stats
##       [,1]     [,2]    [,3]     [,4]
##  [1,]   68 2.864373 33.0070 24250.34
##  [2,]   77 1.523352 12.4892 36114.16
##  [3,]   72 1.916359 13.5280 23673.86
##  [4,]  123 3.441161 29.2958 22165.77
##  [5,]   92 2.695370  7.4828 20098.59
##  [6,]   76 3.181871 23.6530 30104.14
##  [7,]  125 2.694780 25.1790 23808.99
##  [8,]  113 2.777789 21.1614 27930.82
##  [9,]  116 3.218911 19.9760 21025.62
## [10,]   95 2.231885 14.4554 32660.25
##
## $weights
##  [1] 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1
##
## $stats_normalization
## [1]    42.828665     1.177572    23.618121 15176.184404
##
## $nsim
## [1] 100
##
## $nrec
## [1] 10
##
## $computime
## [1] 26.50279
```

Here, the `tol` is the percentage of simulations that are nearest the observed summary statistics. The model must be a R function, taking a vector of model parameter values as arguments and return a vector of summary statistics. The available prior distribution are uniform, normal, lognormal and exponential.

ABC rejection algorithm is computationally inefficient.

The idea of ABC-MCMC is to perform a Metropolis-Hastings algorithm to explore the parameter space, and in replacing the step of likelihood ratio computation by simulations of the model.

```
ABC_Marjoram_original<-ABC_mcmc(method = "Marjoram_original",
                                model = trait_model,
                                prior = trait_prior,
                                summary_stat_target = sum_stat_obs,
                                n_rec=10, use_seed=T,dist_max=0.2)
```

```
## [1] "Warning: summary statistics are normalized by default through a division by the target summary s
## [1] "Consider providing normalization constants for each summary statistics in the option 'tab_normal
## [1] "Warning: default values for proposal distributions are used - they may not be appropriate to you
## [1] "Consider providing proposal range constants for each parameter in the option 'proposal_range' or
```

```
ABC_Marjoram_original
```

```
## $param
##           [,1]        [,2]        [,3]     [,4]
##  [1,] 4.383332 -0.16236918  -5.6601492 1.675962
##  [2,] 4.416247 -0.06241183  -2.9649732 1.905677
##  [3,] 4.430384 -0.04903160  -3.3304248 1.865512
##  [4,] 4.336837  0.16232622  -0.9491732 1.885433
##  [5,] 4.299518  0.30252674  -4.9198927 1.927855
##  [6,] 4.191820  0.15925548 -11.4197175 2.061380
##  [7,] 4.248792  0.27186477  -9.5412117 2.073467
##  [8,] 4.194222  0.34671421  -9.2869659 2.006412
##  [9,] 4.228673  0.22722033 -11.9358224 2.047472
## [10,] 4.268778  0.07838035  -2.4485084 2.028016
##
```

```
## $stats
##        [,1]     [,2]    [,3]     [,4]
##  [1,]   97 2.757901 18.8940 33610.95
##  [2,]  108 2.851848 14.9334 35390.56
##  [3,]  114 2.822641 18.2786 30459.18
##  [4,]  110 3.332327 15.5828 32564.53
##  [5,]   87 2.335249 13.4136 33305.45
##  [6,]   98 2.674719 19.6144 33715.83
##  [7,]  105 2.764839 17.6616 40449.27
##  [8,]   86 2.590971 15.3664 36427.63
##  [9,]  103 3.032221 15.7604 25609.98
## [10,]   98 2.417096 12.5614 33046.63
##
## $dist
##  [1] 0.02908786 0.12267047 0.04389789 0.17692955 0.14183454 0.02099750
##  [7] 0.14871175 0.12050466 0.11257056 0.15014487
##
## $stats_normalization
## [1]   100.0     2.5    20.0 30000.0
##
## $epsilon
## [1] 0.1769295
##
## $nsim
## [1] 155
##
## $n_between_sampling
## [1] 10
##
## $computime
## [1] 27.73544
```

Wegmann et al.(2009) proposed a number of improvements by perform a calibration step so that the algorithm automatically determines the tolerance threshold, the scaling of the summary statistics and the scaling of the jumps in the parameter space during the MCMC.

```
ABC_Marjoram<-ABC_mcmc(method = "Marjoram", model=trait_model,
                       prior=trait_prior,summary_stat_target=sum_stat_obs,
                       n_rec=10,n_calibration=10,tolerance_quantile=0.2,
                       use_seed=T)
ABC_Marjoram
```

```
## $param
##           [,1]       [,2]       [,3]     [,4]
##  [1,] 3.378062 -1.7295164  -8.328470 2.587793
##  [2,] 3.388977 -2.2445931  34.746151 2.593468
##  [3,] 3.296217 -0.3380868  15.196437 2.662757
##  [4,] 3.319484 -1.4571644  30.873390 2.666327
##  [5,] 3.324878 -0.5121046 -23.044456 2.634135
##  [6,] 3.347526 -0.8307147 -22.179891 2.591154
##  [7,] 3.322663  0.3217378  15.053223 2.591246
##  [8,] 3.269246 -0.2207575  15.201788 2.652557
##  [9,] 3.323276 -1.2544138  10.621443 2.651864
## [10,] 3.361711 -0.6504210  -4.748879 2.623117
##
## $stats
##       [,1]     [,2]    [,3]      [,4]
##  [1,]   84 3.372386 28.5362 28239.124
##  [2,]   76 3.606984 46.0246 11027.533
##  [3,]   58 2.893990 17.4430  5804.960
##  [4,]   63 3.183574 34.7146  4503.444
##  [5,]   58 2.218784 15.8190 38445.179
##  [6,]   78 3.166390 30.3062 23865.587
##  [7,]   60 3.080823 18.4824  4414.969
##  [8,]   60 2.765726 18.2994  5966.159
##  [9,]   65 3.194459 22.1990 19199.037
## [10,]   69 2.109477  9.0198 25084.798
##
## $dist
##  [1] 0.7307141 4.0042011 3.5413470 4.2819393 1.1630293 0.8620983 3.8897098
##  [8] 3.3772062 1.3585925 0.8569033
##
## $stats_normalization
## [1]    48.392378     1.286419    22.240825 14776.526143
##
## $epsilon
## [1] 4.281939
##
## $nsim
## [1] 101
##
## $n_between_sampling
## [1] 10
##
## $computime
## [1] 15.73947
```

Wegmann et al.(2009) also proposed additional modification, among which a partial least squares transformation of the summary statistics.

```
ABC_Wegmann <-ABC_mcmc(method="Wegmann",model=trait_model,
                       prior=trait_prior,summary_stat_target=sum_stat_obs,
                       n_rec=10,n_calibration=10,
                       tolerance_quantile=.2,use_seed=T)
ABC_Wegmann
```

```
## $param
##           [,1]     [,2]        [,3]         [,4]
##  [1,] 4.209921 1.233677  12.7644485 -0.481933300
##  [2,] 4.174215 1.242621   3.5367025 -0.168057917
##  [3,] 4.263536 1.221604  13.3597382  0.188878292
##  [4,] 4.308689 1.253652  16.7983421  0.206274890
##  [5,] 4.200412 1.231540  18.0144199  0.737463930
##  [6,] 4.344696 1.254900  12.6583406  0.004363433
##  [7,] 4.355712 1.328136 -20.4904812  2.201962192
##  [8,] 4.318964 1.342670 -19.3281488  2.472649974
##  [9,] 4.242251 1.290874 -18.7126639  2.582764072
## [10,] 4.373894 1.250817   0.4516707  2.073106242
##
## $stats
##       [,1]     [,2]    [,3]     [,4]
##  [1,]   74 2.253642 17.8136 14133.14
##  [2,]   74 2.246055 10.6340 25538.67
##  [3,]   76 2.254629 18.5316 12833.84
##  [4,]   85 2.209858 21.3838 12327.82
##  [5,]   79 2.609538 21.9860 10224.99
##  [6,]   84 2.284550 18.7412 17013.74
##  [7,]  105 3.001523 21.9832 30644.62
##  [8,]   85 2.092985 10.7044 29325.41
##  [9,]   96 2.275941 13.2132 31486.88
## [10,]   99 3.053656  8.3546 22158.12
##
## $dist
##  [1] 0.8450342 0.8209698 0.7984677 0.6203454 0.7485710 0.4135168 0.5377672
##  [8] 0.7823378 0.2004877 0.7973408
##
## $epsilon
## [1] 0.8450342
##
## $nsim
## [1] 101
##
## $n_between_sampling
## [1] 10
##
## $min_stats
## [1]     50.000000      1.953164     17.813600 -31380.799750
##
## $max_stats
## [1]   172.000000     4.797662    75.737000 41221.689574
##
## $lambda
## [1] -0.6060606 -0.6060606  3.0303030 -0.6060606
##
## $geometric_mean
## [1] 1.446475 1.440196 1.490122 1.392293
##
## $boxcox_mean
## [1] 0.5780640 0.5650132 0.4470526 0.4953800
##
## $boxcox_sd
## [1] 0.3297220 0.3491347 0.3402070 0.2777210
##
## $pls_transform
##            [,1]       [,2]        [,3]        [,4]
## [1,] -0.1510477 -0.1856123 -0.73066738  0.73780972
## [2,]  0.7388286  0.7032718  0.11284534 -0.04552618
## [3,]  0.5048261 -0.8849161  0.04888984 -0.12125866
## [4,]  0.1351737 -0.2372978  0.71182054  0.64709300
##
```

```
## $n_component
## [1] 4
##
## $computime
## [1] 13.50904
```

Sequential algorithms aim at reducing the required number of simulations to reach a given quality of the posterior approximation. The underlying idea is to spend more time in the areas of the parameter space where simulation are frequently close to the target. Sequential algorithms consist in a first step of standard rejection ABC, followed by a number of steps where the sampling of the parameter space is the accepted parameter values in the previous iteration. There are 4 algorithms to perform sequential sampling schemes for ABC. Sequential sampling schemes have been shown to be more efficient than standard rejection-based procedures.

```
ABC_Beaumont <- ABC_sequential(method="Beaumont", model=trait_model,
                               prior=trait_prior,nb_simul=10,
                               summary_stat_target=sum_stat_obs,
                               tolerance=c(8,5),use_seed=T)
ABC_Beaumont
```

```
## $param
##           [,1]        [,2]        [,3]       [,4]
##  [1,] 4.299262  0.12015573  38.5238683  1.9881154
##  [2,] 4.061712 -0.62469651  20.7773326 -0.6426014
##  [3,] 3.619656  1.04074345  15.9697344  2.5456264
##  [4,] 4.389533  0.09666333   0.7694632  3.0817717
##  [5,] 3.302943 -1.27485742   5.9162374  0.5013323
##  [6,] 4.612064 -1.25432280   2.8421821  0.3301076
##  [7,] 3.750857 -0.07969019  27.5253870  2.9098108
##  [8,] 3.580274  1.00183825  12.8615826 -0.1498945
##  [9,] 4.499885  0.55746938   5.9708086  1.2501842
## [10,] 3.511829  0.64274075 -12.8482987  3.0037294
##
## $stats
##       [,1]     [,2]    [,3]      [,4]
##  [1,]  122 3.674038 39.9130  5695.205
##  [2,]   71 1.845863 27.5496 11207.178
##  [3,]   68 3.252695 17.9694  5255.497
##  [4,]  112 3.610280 13.3898 20357.742
##  [5,]   55 2.223056 17.2100 37283.456
##  [6,]  135 4.131359 35.0618 12348.235
##  [7,]   85 3.316323 33.3130  9736.714
##  [8,]   44 1.281906 16.1190 10344.302
##  [9,]  125 3.782996 18.6002 30876.990
## [10,]   51 1.483659  6.3764 17980.531
##
## $weights
##  [1] 0.06040362 0.08902218 0.06329142 0.08926412 0.20725345 0.24577232
##  [7] 0.08240631 0.04505123 0.04814566 0.06938971
##
## $stats_normalization
## [1]    39.693828     1.192364    26.677242 15619.312401
##
```

```
## $epsilon
## [1] 4.638769
##
## $nsim
## [1] 54
##
## $computime
## [1] 13.89043
```

This method is in fact the ABC population Monte Carlo algorithm.

```
ABC_Drovandi<-ABC_sequential(method="Drovandi", model=trait_model, prior=trait_prior,nb_simul=10, summa
ABC_Drovandi
```

```
## $param
##           [,1]       [,2]      [,3]       [,4]
##  [1,] 3.996740  0.2183355 21.689254  0.1142476
##  [2,] 3.849755 -1.5236000 29.656569  2.2684783
##  [3,] 4.202743 -1.2720383 13.243602  2.0376912
##  [4,] 3.657800 -1.2491982 21.441571  0.8655118
##  [5,] 3.992047 -1.1924274 12.740257  1.3427021
##  [6,] 4.173088 -0.5650540  5.558544  0.5022413
##  [7,] 4.448621  1.3849954  5.664796 -0.5399465
##  [8,] 4.153626 -0.4645068  3.391937 -0.2081809
##  [9,] 4.122602 -1.2659357  7.030051  1.0109211
## [10,] 4.070756 -1.1000352 19.681056  1.7214372
##
## $stats
##       [,1]     [,2]    [,3]      [,4]
##  [1,]   70 1.680671 25.6690  9953.917
##  [2,]   86 3.524698 38.5772 14303.679
##  [3,]  118 3.996445 28.8900 21507.770
##  [4,]   65 2.628859 33.4510 16178.740
##  [5,]   95 3.121249 26.2554 19786.994
##  [6,]   86 2.332023 20.6724 35240.299
##  [7,]   79 1.606054 12.9314 21441.183
##  [8,]   72 1.904785 14.4630 23580.204
##  [9,]  109 3.707804 24.9806 29036.791
## [10,]  104 3.361931 29.8226 15878.723
##
## $weights
##  [1] 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1
##
## $stats_normalization
## [1]    49.059014     1.089909    21.863793 18170.559775
##
## $epsilon
## [1] 2.433515
##
## $nsim
## [1] 40
##
## $computime
## [1] 8.95475
```

```
ABC_Delmoral<-ABC_sequential(method="Delmoral",model=trait_model,
                             prior=trait_prior,
                             nb_simul =10, summary_stat_target=sum_stat_obs,
                             alpha=.5,
                             tolerance=3,use_seed=T)
ABC_Delmoral
```

```
## $param
##           [,1]        [,2]      [,3]       [,4]
##  [1,] 4.263383  0.22562145 10.951391  0.9329045
##  [2,] 4.526854 -0.16410216 -5.330469  2.4506843
##  [3,] 4.526854 -0.16410216 -5.330469  2.4506843
##  [4,] 4.414175 -0.29338968  5.903345 -0.5052528
##  [5,] 4.715864  0.45954989 14.831182 -0.2061818
##  [6,] 4.624273 -0.33039897 -1.082010  0.4841457
##  [7,] 4.131724  0.07295841  6.746657  0.1368278
##  [8,] 4.624273 -0.33039897 -1.082010  0.4841457
##  [9,] 4.588103  0.77922915  9.555796  1.3102844
## [10,] 4.263383  0.22562145 10.951391  0.9329045
##
## $stats
##       [,1]     [,2]    [,3]     [,4]
##  [1,]   82 2.811874 17.2432 17333.33
##  [2,]  126 3.225556 20.6270 40871.36
##  [3,]  126 3.225556 20.6270 40871.36
##  [4,]  105 2.805392 24.0092 38086.19
##  [5,]  113 2.689082 25.0292 24129.15
##  [6,]  117 3.187478 24.1980 36854.13
##  [7,]   98 2.659451 18.2890 32086.67
##  [8,]  117 3.187478 24.1980 36854.13
##  [9,]  110 3.407226 18.0314 22891.35
## [10,]   82 2.811874 17.2432 17333.33
##
## $weights
##  [1] 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1
##
## $stats_normalization
## [1]   54.2070106    0.9870618   16.1313149 9382.0131066
##
## $epsilon
## [1] 2.114581
##
## $nsim
## [1] 82
##
## $computime
## [1] 14.20736
```

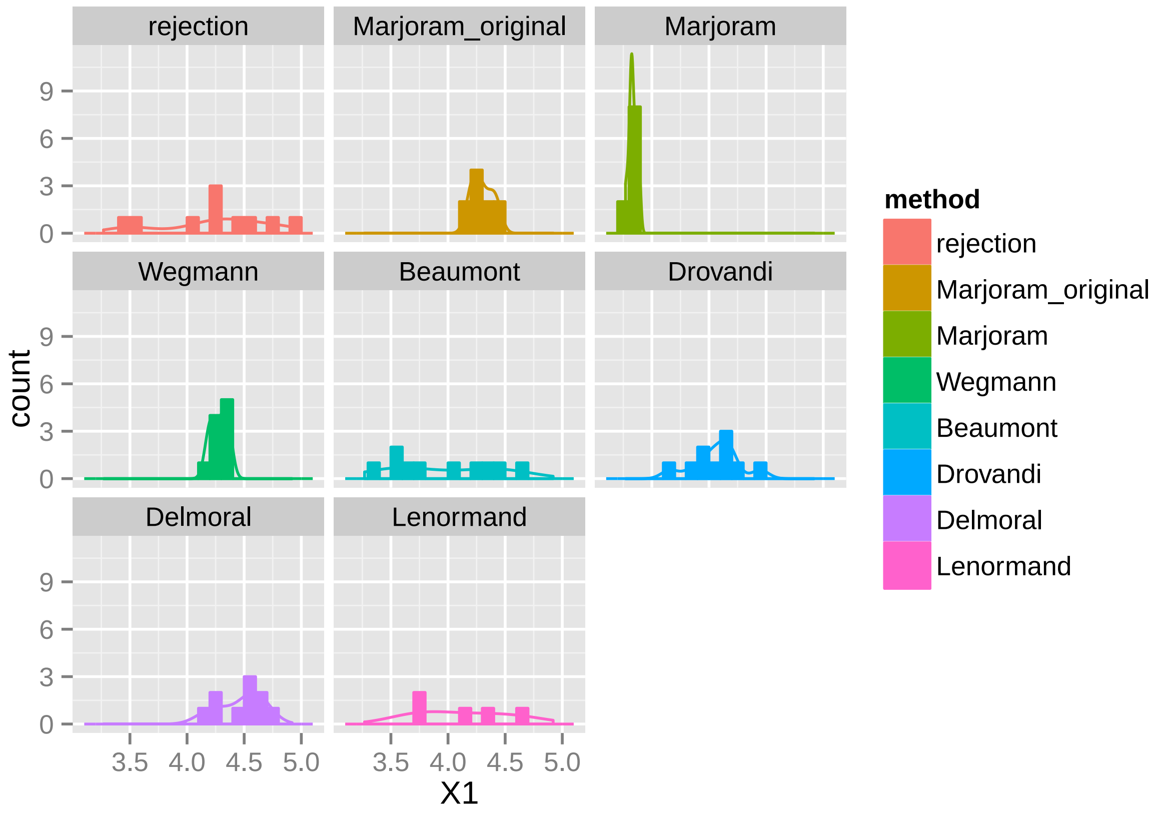
This is an adaptive sequential Monte Carlo method.

```
ABC_Lenormand <- ABC_sequential(method="Lenormand",model=trait_model,
                                prior=trait_prior,nb_simul=10,
                                summary_stat_target=sum_stat_obs,
                                p_acc_min=.4,
                                use_seed=T)
ABC_Lenormand
```

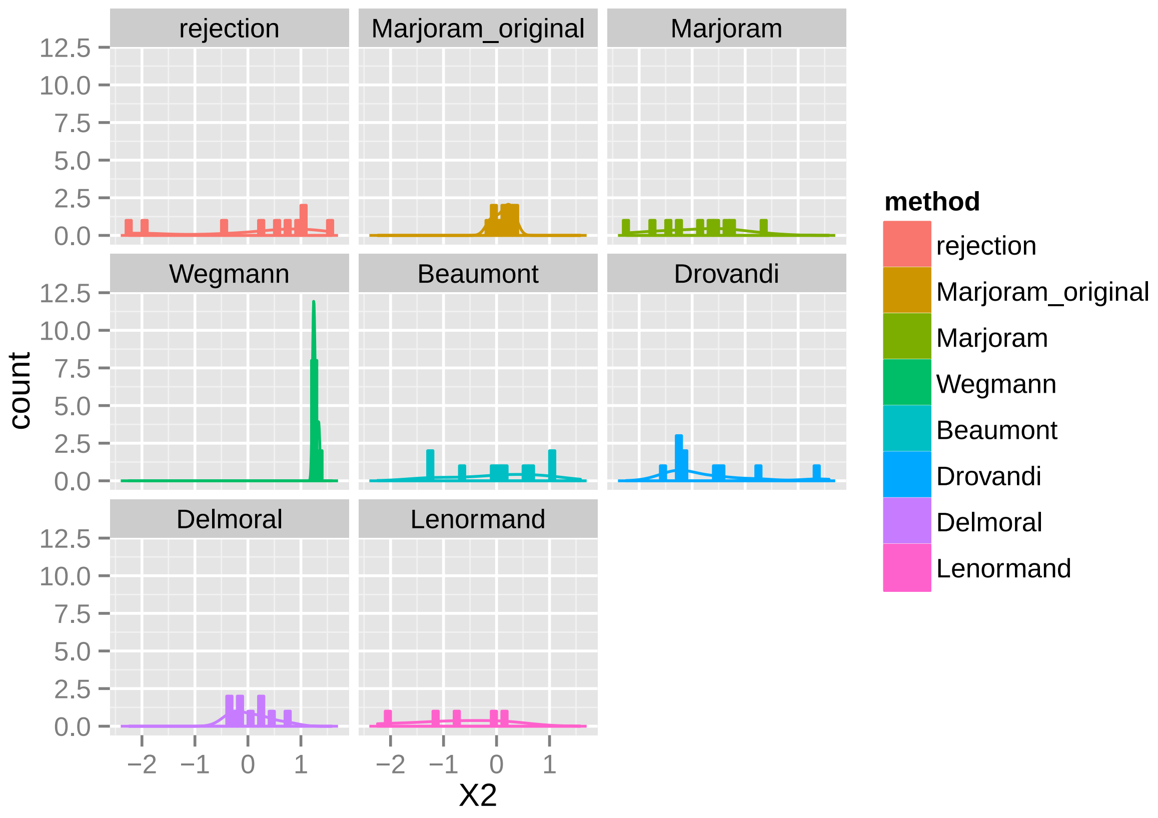
```
## $param
##          [,1]        [,2]      [,3]        [,4]
## [1,] 3.752129 -0.01213612 19.375807  1.29454198
## [2,] 4.395494  0.17715052 20.530796  2.93127900
## [3,] 4.669707 -0.77908001  6.567264  2.50392614
## [4,] 4.100702 -1.10392194 -4.684361  0.92223964
## [5,] 3.726607 -2.08464326 -7.877757 -0.01300076
##
## $stats
##      [,1]     [,2]    [,3]      [,4]
## [1,]   65 1.816746 23.1036 10135.919
## [2,]  106 3.863595 25.5474  7572.488
## [3,]  153 4.610654 36.8194 17057.023
## [4,]  129 4.310676 43.5926  5416.547
## [5,]   88 3.885550 47.6620  1486.324
##
## $weights
## [1] 0.31654615 0.31654615 0.31654615 0.02479799 0.02556358
##
## $stats_normalization
## [1]    37.744904     1.131197    19.151655 11396.766741
##
## $epsilon
## [1] 9.947083
##
## $nsim
## [1] 15
##
## $computime
## [1] 4.141729
```

```
data<-rbind(data.frame(ABC_rej$par,method="rejection"),
      data.frame(ABC_Marjoram_original$par, method="Marjoram_original"),
      data.frame(ABC_Marjoram$par,method="Marjoram"),
      data.frame(ABC_Wegmann$par,method="Wegmann"),
      data.frame(ABC_Beaumont$par,method="Beaumont"),
      data.frame(ABC_Drovandi$par,method="Drovandi"),
      data.frame(ABC_Delmoral$par,method="Delmoral"),
      data.frame(ABC_Lenormand$par,method="Lenormand")
      )
library(ggplot2)
g1<- ggplot(data) +
  geom_histogram(aes(x=X1,colour=method,fill=method),binwidth=0.1) +
  geom_density(aes(x=X1,colour=method)) +
  facet_wrap(~method)
g2 <- ggplot(data) +
  geom_histogram(aes(x=X2,colour=method,fill=method),binwidth=0.1) +
  geom_density(aes(x=X2,colour=method)) +
  facet_wrap(~method)
```

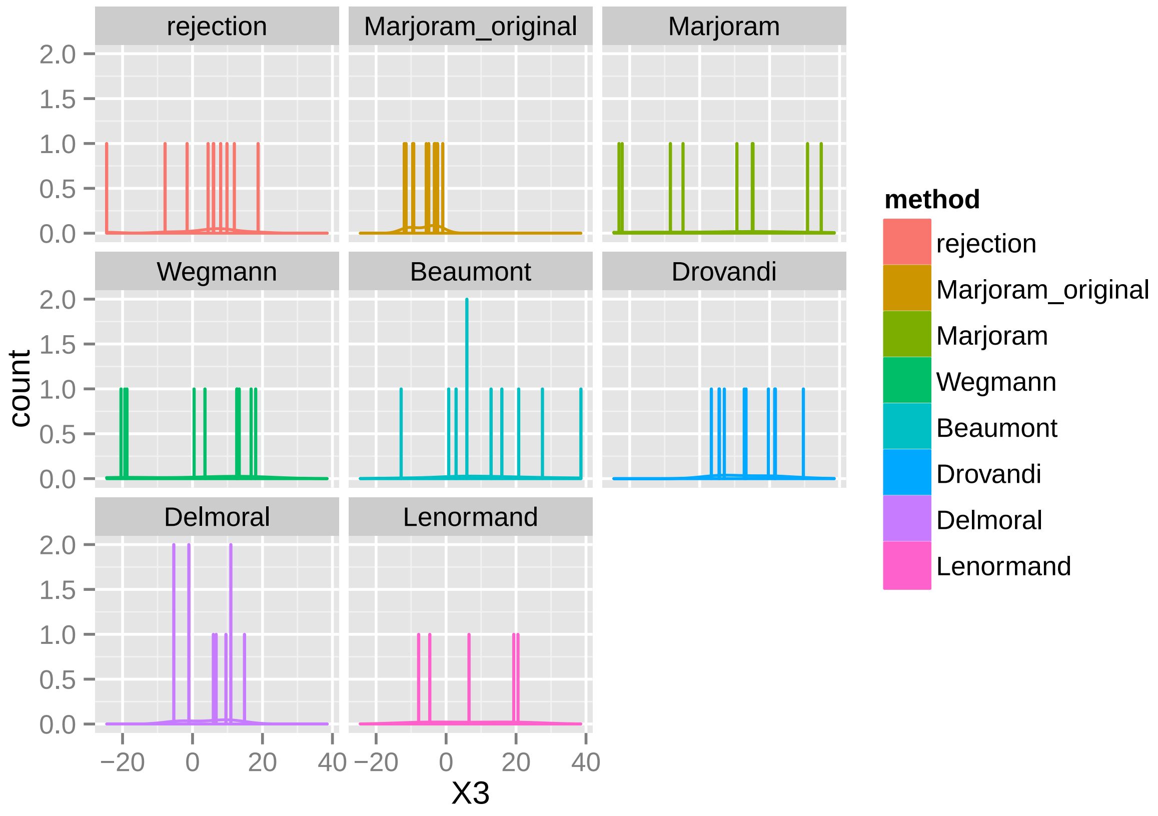
```
g3 <- ggplot(data) +
  geom_histogram(aes(x=X3,colour=method,fill=method),binwidth=0.1) +
  geom_density(aes(x=X3,colour=method)) +
  facet_wrap(~method)
g4 <- ggplot(data) +
  geom_histogram(aes(x=X4,colour=method,fill=method),binwidth=0.1) +
  geom_density(aes(x=X4,colour=method)) +
  facet_wrap(~method)
g1
```



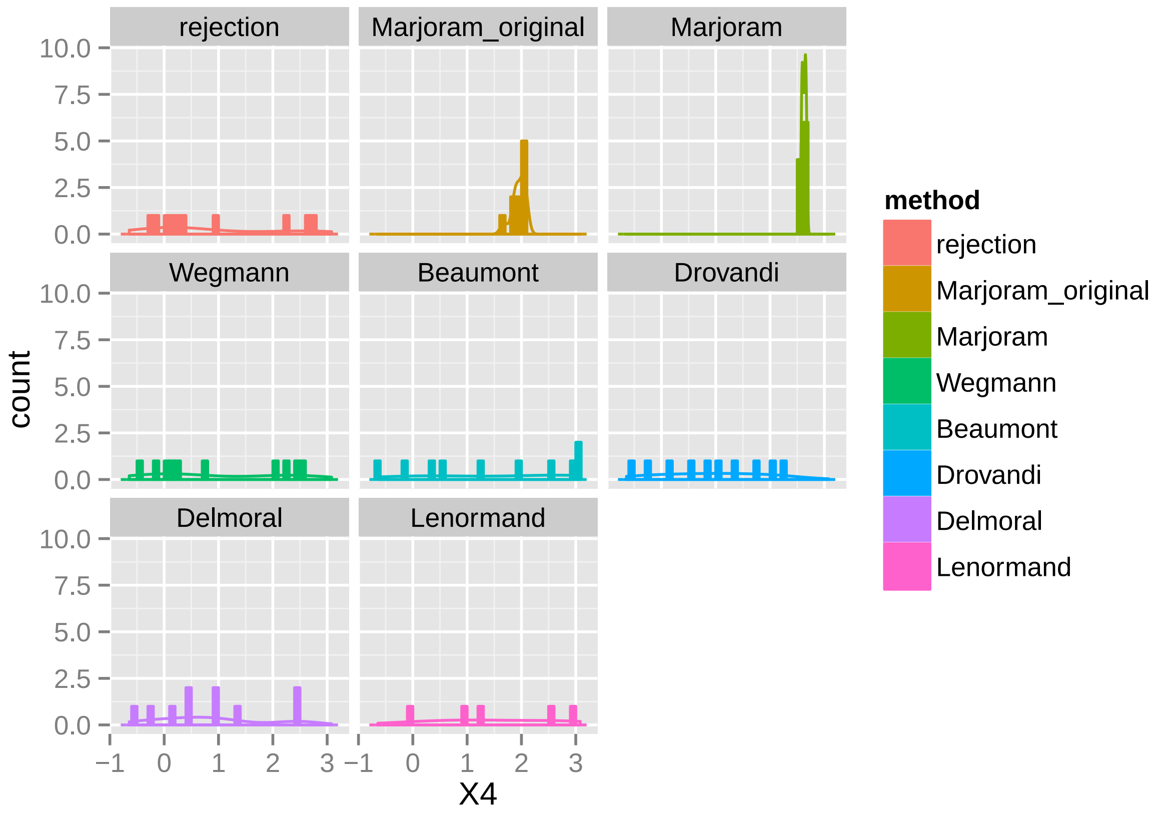
```
g2
```


rejection
Marjoram_original
Marjoram
Wegmann
Beaumont
Drovandi
Delmoral
Lenormand
12.5
10.0
7.5
5.0
2.5
0.0
count
−2
−1
0
1
X2
method
rejection
Marjoram_original
Marjoram
Wegmann
Beaumont
Drovandi
Delmoral
Lenormand

```
g3
```


rejection
Marjoram_original
Marjoram
Wegmann
Beaumont
Drovandi
Delmoral
Lenormand
2.0
1.5
1.0
0.5
0.0
count
−20
0
20
40
X3
method
rejection
Marjoram_original
Marjoram
Wegmann
Beaumont
Drovandi
Delmoral
Lenormand

```
g4
```



Exercise

1. Try the socks example by the two packages

```
sim_sock <- function(nb.mu,nb.sd,beta.a,beta.b){
# n_socks is positive and discrete, we can use Possion
#  (problemic: mean and variance is same)
# or use negative binomal
# suppose we have a family of 4 and each person changes socks
#  around 5 times a week, so we would have 20
# pairs of socks, so your mean is 20*2 = 40. our sd could be 15
prior_mu <- nb.mu
prior_sd <- nb.sd
prior_size <- prior_mu^2/(prior_sd^2-prior_mu)
n_socks <-rnbinom(1,mu=prior_mu,size=prior_size)
# proprotion of socks that are pair is Beta with a=2, b=2
prop_pairs <- rbeta(1,shape1=beta.a,shape2=beta.b)
n_pairs <- round(n_socks/2*prop_pairs)
n_odd <- n_socks-n_pairs*2
n_picked <- 11
socks <- rep(seq_len(n_pairs+n_odd),rep(c(2,1),c(n_pairs,n_odd)))
picked_socks<-sample(socks,size=min(n_picked,n_socks))
sock_counts <- table(picked_socks)
c(unique=sum(sock_counts==1),pairs=sum(sock_counts==2),nsocks=n_socks,npairs=n_pairs,nodd=n_odd,
  proppairs=prop_pairs)
}
simdata = data.frame(t(replicate(100000,sim_sock(40,15,2,2))))
```

2. Play the functions with toy model:

```
toy_model <- function(x) {2*x+5+rnorm(1,0,0.1)}
toy_prior <- list(c("unif",0,1))
```

3. Suppose a state-space model is given by $N(t+1) \sim Normal(N(t)+b, \sigma^2_{proc})$, $N_{obs}(t) \sim Normal(N(t), \sigma^2_{obs})$. The parameters of interest are $b, \sigma^2_{proc}, \sigma^2_{obs}, N(0)$. Suppose your true parameter values are $b = 3, \sigma^2_{proc} = 1, \sigma^2_{obs} = 1.2, N(0) = 100$. Simulate a data set as your observed data and obtain the summary statistics mean and standard deviation. Then use the model and different prior distributions to see how different abc schemes work.