EasyABC: a R package to perform efficient approximate Bayesian computation sampling schemes

Franck Jabot, Thierry Faure, Nicolas Dumoulin

EasyABC version 1.0 2012-11-20

Contents

[1]This document is included as a vignette (a LaTeX document created using the R function Sweave) of the package EasyABC. It is automatically dowloaded together with the package and can be accessed through R typing `vignette("EasyABC")`.

1 Summary

The aim of this vignette is to present the features of the `EasyABC` package. Section 2 describes the different algorithms available in the package. Section 3 details how to install the package and the formatting requirements. Section 4 presents a detailed worked example.

2 Overview of the package EasyABC

`EasyABC` enables to launch various ABC schemes and to retrieve the ouputs of the simulations, so as to perform post-processing treatments with the various R tools available. `EasyABC` is also able to launch the simulations on multiple cores of a multi-core computer. Three main types of ABC schemes are available in EasyABC: the standard rejection algorithm of Pritchard et al. (1999), sequential schemes first proposed by Sisson et al. (2007), and coupled to MCMC schemes first proposed by Marjoram et al. (2003). Four different sequential algorithms are available: the ones of Beaumont et al. (2009), Drovandi and Pettitt (2011), Del Moral et al. (2012) and Lenormand et al. (2012). Three different MCMC schemes are available: the ones of Marjoram et al. (2003), Wegmann et al. (2009a) and a modification of Marjoram et al. (2003)'s algorithm in which the tolerance and proposal range are determined by the algorithm, following the modifications of Wegmann et al. (2009a). Details on how to implement these various algorithms with `EasyABC` are given in the manual pages of each function and an example is detailed in Section 4. We provide below a short presentation of each implemented algorithm.

2.1 The standard rejection algorithm of Pritchard et al. (1999)

This sampling scheme consists in drawing the model parameters in the prior distributions, in using these model parameter values to launch a model simulation and in repeating this two-step procedure `nb_simul` times. At the end of the `nb_simul` simulations, the simulations closest to the target (or at a distance smaller than a tolerance threshold) in the space of the summary statistics are retained to form an approximate posterior distribution of the model parameters. This last step of simulation rejection can be performed with the `R` package `abc` (Csilléry et al. 2012). A worked example demonstrating how the `EasyABC` and `abc` functions can be pipelined is provided in section 4.

2.2 Sequential algorithms

Sequential algorithms for ABC have first been proposed by Sisson et al. (2007). These algorithms aim at reducing the required number of simulations to reach a given quality of the posterior approximation. The underlying idea of these algorithms is to spend more time in the areas of the parameter space where simulations 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 not anymore performed according to the prior distributions of parameter values. Various ways to perform this biased sampling have been proposed, and four of them are implemented in the package `EasyABC`.

2.3 Coupled to MCMC algorithms

The idea of ABC-MCMC algorithms proposed by Marjoram et al. (2003) is to perform a Metropolis-Hastings algorithm to explore the parameter space, and in replacing the step of likelihood ratio computation by model simulations. The original algorithm of Marjoram et al. (2003) is implemented in the method "Marjoram_original" in `EasyABC`. Wegmann et al. (2009) later proposed a number of improvements to the original scheme of Marjoram et al. (2003): they proposed to 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. These improvements have been implemented in the method "Marjoram". Wegmann et al. (2009) also proposed additional modifications, among which a PLS transformation of the summary statistics. The complete Wegmann et al. (2009)'s algorithm is implemented in the method "Wegmann".

3 Installation and requirements

3.1 Installing the package

To install the `EasyABC` package from `R`, simply type:

```
> install.packages("EasyABC")
```

Once the package is installed, it needs to be loaded in the current `R` session to be used:

```
> library(EasyABC)
```

For online help on the package content, simply type:

```
> help(package="EasyABC")
```

For online help on a particular command (such as the function `ABC_sequential`), simply type:

```
> help(ABC_sequential)
```

3.2 The simulation code - for use on a single core

Users need to develop a simulation code with minimal compatibility constraints. The code can either be a `R` function or a binary executable file.

If the code is a `R` function, its argument must be an array of parameter values and it must return an array of summary statistics. If the default option `use_seed=TRUE` is chosen, the first parameter value passed to the simulation code corresponds to the seed value to be used by the simulation code to initialize the pseudo-random number generator. The following parameters are the model parameters.

If the code is a binary executable file, it needs to read the parameter values in a file named 'input' in which each line contains one parameter value, and to output the summary statistics in a file named 'output' in which each summary statistics must be separated by a space or a tabulation. If the code is a binary executable file, a wrapper `R` function named 'binary_model' is available to interface the executable file with the `R` functions of the `EasyABC` package (see section 4 below).

Alternatively, users may prefer building a `R` function calling their binary executable file. A short tutorial is provided in section 3.7 to call a `C/C++` program.

3.3 The simulation code - for use with multiple cores

Users need to develop a simulation code with minimal compatibility constraints. The code can either be a `R` function or a binary executable file.

If the code is a `R` function, its argument must be an array of parameter values and it must return an array of summary statistics. The first parameter value passed to the simulation code corresponds to the seed value to be used by the simulation code to initialize the pseudo-random number generator. The following parameters are the model parameters.

If the code is a binary executable file, it needs to have as its single argument a positive integer `k`. It has to read the parameter values in a file named 'inputk' (where k is the integer passed as argument to the binary code: 'input1', 'input2'...) in which each line contains one parameter value, and to output the summary statistics in a file named 'outputk' (where k is the integer passed as argument to the binary code: 'output1', 'output2'...) in which each summary statistics must be separated by a space or a tabulation. This construction avoids multiple cores to read/write in the same files. If the code is a binary executable file, a wrapper `R` function named 'binary_model_cluster' is available to interface the executable file with the `R` functions of the `EasyABC` package (see section 4 below).

Alternatively, users may prefer building a `R` function calling their binary executable file. A short tutorial is provided in section 3.7 to call a `C/C++` program.

3.4 Management of pseudo-random number generators

To insure that stochastic simulations are independent, the simulation code must either possess an internal way of initializing the seeds of its pseudo-random number generators each time the simulation code is launched. This can be achieved for instance by initializing the seed to the clock value. It is often desirable though to have a way to re-run some analyses with similar seed values. If this option is chosen, a seed value is provided in the input file as a first (additional) parameter, and incremented by 1 at each call of the simulation code. This means that the simulation code must be designed so that the first parameter is a seed initializing value. In the worked example (Section 4), the simulation code `trait_model` makes use of this package default option.

NB: Note that when using multicores with the package functions (`n_cluster=x` with `x` larger than 1), the default option `use_seed=TRUE` is forced, since the seed value is also used to distribute the tasks to each core.

3.5 The prior matrix

A matrix containing the range of the prior distribution of the parameters must be supplied. Each line contains the range values for one parameter. The first (second) column contains the lower (upper) bound of the range. Note that fixed variable can be passed to the simulation code by putting the same value in the two columns. `EasyABC` only manages uniform prior distribution (it will draw a number between the bounds of the range). Consequently, to deal with non-uniform prior distribution, users should include parameter transformation in their simulation code. For instance, in the example below (section 4), three parameters are exponentially transformed in the simulation code.

3.6 The target summary statistics

An array containing the summary statistics of the data must be supplied (for the sequential and MCMC schemes, not for the simple rejection scheme). The statistics must be in the same order as in the simulation outputs.

3.7 Building a `R` function calling a `C/C++` program

Users having a `C/C++` simulation code may wish to construct a `R` function calling their `C/C++` program, instead of using the provided wrappers (see sections 3.2 and 3.3). The procedure is abundantly described in the 'Writing R Extensions' manual. In short, this can be done by:

- Adapt your C/C++ program by wrapping your main method into a `extern "C" { ... }` block. Here is an excerpt of the source code of the trait model provided in this package, in the folder `src`:

```
extern "C" {
  void trait_model(double *input,double *stat_to_return){
    // compute output and fill the array stat_to_return
  }
}
```

- Build your code into a binary library (.so under Linux or .dll under Windows) with the `R CMD SHLIB` command. In our example, the command for compiling the trait model and the given output are:

```
$ R CMD SHLIB trait_model_rc.cpp
g++ -I/usr/share/R/include -DNDEBUG -fpic -O2 -pipe -g -c trait_model_rc.cpp
-o trait_model_rc.o
g++ -shared -o trait_model_rc.so trait_model_rc.o -L/usr/lib/R/lib -lR
```

- Load the builded library in your session with the `dyn.load` function.

```
> dyn.load("trait_model_rc.so")
```

- Use the `.C` function for calling your program, like we've done in our `trait_model` function:

```
trait_model <- function(input=c(1,500,1,1,1,1)) {
  .C("trait_model",input=input,stat_to_return=array(0,4))$stat_to_return
}
```

4 A worked example

4.1 The trait model

We consider a simple stochastic ecological model hereafter called `trait_model`. This model represents the stochastic dynamics of an ecological community where each species is represented by a set of traits (i.e. characteristics) which determine its competitive ability. A detailed description and analysis of the model can be found in Jabot (2010). The model requires four parameters: an immigration rate I, and three additional parameters (h, A and σ) describing the way traits determine species competitive ability. The model additionnally requires two fixed variables: the total number of individuals in the local community J and the number of traits used n_t. The model outputs four summary statistics: the species richness of the community S, its Shannon's index H, the mean of the trait value among individuals MTV and the skewness of the trait value distribution STV.

NB: Three parameters (I, A and σ) have non-uniform prior distributions: instead, their log-transformed values have a uniform prior distribution. The simulation code `trait_model` therefore takes an exponential transform of the values proposed by `EasyABC` for these parameters at the beginning of each simulation.

In the following, we will use the values $J = 500$ and $n\_t = 1$, and uniform prior distributions for $ln(I)$ in $[3; 5]$, h in [-25;125], $ln(A)$ in $[ln(0.1); ln(5)]$ and $ln(\sigma)$ in $[ln(0.5); ln(25)]$. The simulation code `trait_model` reads sequentially J, I, A, n_t, h and σ.

NB: Note that the fixed variables J and n_t are included in the prior matrix (lines 1 and 4) with their two columns equal to their fixed values:

```
> priormatrix=cbind(c(500,3,-2.3,1,-25,-0.7),c(500,5,1.6,1,125,3.2))

      [,1]  [,2]
[1,] 500.0 500.0
[2,]   3.0   5.0
[3,]  -2.3   1.6
[4,]   1.0   1.0
[5,] -25.0 125.0
[6,]  -0.7   3.2
```

We will consider an imaginary arbitrary dataset whose summary statistics are $(S, H, MTV, STV) = (100, 2.5, 20, 30000)$:

```
> sum_stat_obs=c(100,2.5,20,30000)

[1]   100.0     2.5    20.0 30000.0
```

4.2 Performing a standard ABC-rejection procedure

A standard ABC-rejection procedure can be simply performed with the function `ABC_rejection`, in precising the number n of simulations to be performed:

```
> set.seed(1)

NULL

> n=10

[1] 10

> ABC_rej<-ABC_rejection(model=trait_model, prior_matrix=priormatrix, nb_simul=n)

$param
      [,1]     [,2]       [,3] [,4]       [,5]        [,6]
 [1,]  500 3.531017 -0.8487168    1  60.928005  2.84201038
 [2,]  500 3.403364  1.2037198    1 116.701290  1.87711139
 [3,]  500 4.258228 -2.0590335    1   5.896186 -0.01142867
 [4,]  500 4.374046 -0.8019955    1  90.476213  1.24102704
 [5,]  500 4.435237  1.5684338    1  32.005277  2.33203636
 [6,]  500 4.869410 -1.4726442    1  72.751065 -0.21033513
 [7,]  500 3.534441 -0.7941550    1 -22.991450  0.79131303
 [8,]  500 4.739382 -0.9726389    1  47.312017  1.63830672
 [9,]  500 3.987083 -1.5737514    1  99.105998  1.90702028
[10,]  500 4.588480 -1.8790199    1  83.556642  0.90397028

$stats
      [,1]     [,2]    [,3]        [,4]
 [1,]   90 3.614738 58.8120  -6071.7199
 [2,]   63 2.602216 77.6068 -37081.9028
 [3,]  140 4.502762 48.8376  -2900.4530
 [4,]  125 3.694395 75.3258 -31065.1721
 [5,]  116 4.104226 32.9882   3029.7998
 [6,]  180 4.828517 47.7092   -842.1061
 [7,]   84 3.994615 49.1732   1950.1471
 [8,]  164 4.532558 50.4868   2525.3002
 [9,]  101 3.818715 74.3012 -27249.2048
[10,]  171 4.716238 57.1520 -14206.1651

$weights
 [1] 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1

$stats_normalization
          V1           V2           V3           V4
3.981680e+01 6.631974e-01 1.451333e+01 1.526571e+04

$nsim
[1] 10

$computime
[1] 14.95533
```

Note that a simulation code `My_simulation_code` can be passed to the function `ABC_rejection` in several ways depending of its nature:

- if it is a R function
  `ABC_rejection(My_simulation_code, prior_matrix=priormatrix, nb_simul=n)`
- if it is a binary executable file and a single core is used (see section 3.2 for compatibility constraints)
  `ABC_rejection(binary_model("./My_simulation_code"), prior_matrix=priormatrix, nb_simul=n)`
- if it is a binary executable file and multiple cores are used (see section 3.3 for compatibility constraints)
  `ABC_rejection(binary_model_cluster("./My_simulation_code"), prior_matrix=priormatrix, nb_simul=n, n_cluster=2)`

Simulation outputs can be transparently passed to post-processing tools, like the ones proposed by the R package `abc` (Csilléry et al. 2012):

```
> install.packages("abc")

> library(abc)

 [1] "abc"       "locfit"    "quantreg"  "SparseM"   "nnet"      "EasyABC"
 [7] "parallel"  "MASS"      "mnormt"    "pls"       "lhs"       "stats"
[13] "graphics"  "grDevices" "utils"     "datasets"  "methods"   "base"

> rej<-abc(sum_stat_obs, ABC_rej$param[,c(2,3,5,6)], ABC_rej$stats, tol=0.3,
+ method="rejection")

Call:
abc(target = sum_stat_obs, param = ABC_rej$param[, c(2, 3, 5,
    6)], sumstat = ABC_rej$stats, tol = 0.3, method = "rejection")
Method:
Rejection

Parameters:
P1, P2, P3, P4

Statistics:
S1, S2, S3, S4

Total number of simulations 10

Number of accepted simulations:  3

> # simulations selected:
> rej$unadj.values

         [,1]       [,2]      [,3]     [,4]
[1,] 4.435237  1.5684338  32.00528 2.332036
[2,] 3.534441 -0.7941550 -22.99145 0.791313
[3,] 4.739382 -0.9726389  47.31202 1.638307

> # their associated summary statistics:
> rej$ss

     [,1]     [,2]    [,3]     [,4]
[1,]  116 4.104226 32.9882 3029.800
[2,]   84 3.994615 49.1732 1950.147
[3,]  164 4.532558 50.4868 2525.300
```

```
> # their normalized euclidean distance to the data summary statistics:
> rej$dist

 [1] 6.030324 9.400513 5.613765 8.993603 3.847707 5.885814 4.960505 5.605648
 [9] 8.706420 7.113637
```

4.3 Performing a sequential ABC scheme

Other functions of the `EasyABC` package are used in a very similar manner. To perform the algorithm of Beaumont et al. (2009), one needs to specify the sequence of tolerance levels *tolerance_tab* and the number *nb_simul* of simulations to obtain below the tolerance level at each iteration:

```
> n=10

[1] 10

> tolerance=c(8,5)

[1] 8 5

> ABC_Beaumont<-ABC_sequential(method="Beaumont", model=trait_model,
+ prior_matrix=priormatrix, nb_simul=n, summary_stat_target=sum_stat_obs,
+ tolerance_tab=tolerance)

$param
      [,1]     [,2]       [,3] [,4]       [,5]         [,6]
 [1,]  500 3.362110 -1.9163965    1 23.0654334 -0.599666827
 [2,]  500 3.543818  0.9455070    1 17.2848491  0.716968486
 [3,]  500 3.260380 -0.9270426    1 25.3968857  0.779964720
 [4,]  500 3.398492  0.4703746    1 15.1099903  0.938328200
 [5,]  500 3.017150 -1.8711144    1 26.1860829 -0.002416091
 [6,]  500 4.053298 -1.7809923    1 14.1823387 -0.159038819
 [7,]  500 4.398392  1.0919829    1  5.7191547  2.949045622
 [8,]  500 4.358355  0.7294188    1 14.7202497  2.627312598
 [9,]  500 4.694502  1.1509709    1  0.9577352  3.095833860
[10,]  500 3.672407  0.7490497    1 13.0817551  3.067217787

$stats
      [,1]     [,2]    [,3]      [,4]
 [1,]   60 2.406482 35.3478 15610.260
 [2,]   52 1.891610 21.0018 12123.542
 [3,]   50 1.762070 31.1942 10202.962
 [4,]   45 2.162352 17.7182 11280.426
 [5,]   46 1.682788 34.1128 12557.523
 [6,]  119 3.615008 36.6096 13923.754
 [7,]  122 4.144510 14.6302 17235.203
 [8,]  118 4.079771 19.0042 12605.343
 [9,]  149 4.212484 14.8760 20535.516
[10,]   86 3.506517 17.2494  6748.929

$weights
 [1] 0.11597293 0.06118753 0.07779398 0.05992568 0.15439436 0.07654255
 [7] 0.11863352 0.08190165 0.18240961 0.07123820

$stats_normalization
          V1          V2          V3          V4
```

```
4.530833e+01 9.914929e-01 1.550687e+01 1.311587e+04

$epsilon
[1] 4.732098

$nsim
[1] 72

$computime
[1] 49.24077
```

To perform the algorithm of Drovandi and Pettitt (2011), one needs to specify four arguments: the initial number of simulations *nb_simul*, the final tolerance level *tolerance_tab*, the proportion α of best-fit simulations to update the tolerance level at each step, and the target proportion c of unmoved particles during the MCMC jump. Note that default values $alpha = 0.5$ and $c = 0.01$ are used if not specified, following Drovandi and Pettitt (2011).

```
> n=10

[1] 10

> tol=3

[1] 3

> alpha=0.5

[1] 0.5

> c=0.7

[1] 0.7

> ABC_Drovandi<-ABC_sequential(method="Drovandi", model=trait_model,
+ prior_matrix=priormatrix, nb_simul=n, summary_stat_target=sum_stat_obs,
+ tolerance_tab=tol, alpha=alpha, c=c)

$param
      [,1]     [,2]         [,3] [,4]       [,5]        [,6]
 [1,]  500 4.372120 -0.001121019    1  5.4715372 -0.12568710
 [2,]  500 4.051867 -0.125022011    1 12.1207510 -0.67154259
 [3,]  500 4.357840 -0.158636879    1  4.0214488 -0.14801890
 [4,]  500 4.003453  0.155106599    1 11.8676334 -0.43394316
 [5,]  500 4.497001  0.296629792    1  0.4179174  0.04238219
 [6,]  500 4.614254 -0.421605678    1 11.7121630 -0.13055283
 [7,]  500 4.255693 -0.030613125    1 13.7085732 -0.09558018
 [8,]  500 4.320190  0.177067473    1 15.5170178  0.26745481
 [9,]  500 4.704030 -0.007652111    1  9.6886842  0.34755989
[10,]  500 4.202139 -0.031661415    1  9.6060166 -0.11315922

$stats
      [,1]     [,2]    [,3]     [,4]
 [1,]   96 2.038106 16.2344 24729.78
 [2,]   76 1.519483 22.5206 27628.05
 [3,]   95 2.213074 16.8628 31063.59
 [4,]   65 1.960228 18.8398 21922.02
 [5,]   93 2.712320 13.6572 37907.42
 [6,]  110 2.657473 25.9604 20921.48
 [7,]   91 1.819804 22.4596 19743.10
 [8,]   85 2.351128 21.5434 15320.82
 [9,]  124 2.648244 22.0450 26811.07
[10,]   85 2.017672 17.5428 20111.10

$weights
 [1] 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1

$stats_normalization
          V1           V2           V3           V4
   45.687583     1.130895    18.184548 14175.858029

$epsilon
[1] 1.204596

$nsim
[1] 40

$computime
[1] 17.48849
```

To perform the algorithm of Del Moral et al. (2012), one needs to specify five arguments: the initial number of simulations *nb_simul*, the number α controlling the decrease in effective sample size of the particle set at each step, the number M of simulations performed for each particle, the minimal effective sample size *nb_threshold* below which a resampling of particles is performed and the final tolerance level *tolerance_target*. Note that default values $alpha = 0.5$, $M = 1$ and $nb\_threshold = nb\_simul/2$ are used if not specified.

```
> n=10

[1] 10

> tol=5

[1] 5

> alpha=0.5

[1] 0.5

> ABC_Delmoral<-ABC_sequential(method="Delmoral", model=trait_model,
+ prior_matrix=priormatrix, nb_simul=n, summary_stat_target=sum_stat_obs, alpha=0.5,
+ tolerance_target=tol)

$param
      [,1]     [,2]         [,3] [,4]      [,5]       [,6]
 [1,]  500 3.838808 -1.534359947    1 10.133877  2.7449436
 [2,]  500 3.186180 -1.544326518    1 11.122094  0.5919107
 [3,]  500 4.929140  0.595600148    1 30.025908 -0.1117813
 [4,]  500 4.929140  0.595600148    1 30.025908 -0.1117813
 [5,]  500 3.846864  0.734028679    1 10.999703 -0.2249786
 [6,]  500 3.761755 -1.636655708    1  9.740732  3.0046888
 [7,]  500 4.929140  0.595600148    1 30.025908 -0.1117813
 [8,]  500 4.929140  0.595600148    1 30.025908 -0.1117813
 [9,]  500 4.929140  0.595600148    1 30.025908 -0.1117813
[10,]  500 3.431686 -0.006849683    1  9.334640  0.8246082

$stats
      [,1]     [,2]    [,3]     [,4]
 [1,]  102 4.036128 26.5796 25382.01
 [2,]   44 1.633972 20.9868 20873.45
 [3,]  141 2.694481 36.3650 11753.93
 [4,]  141 2.694481 36.3650 11753.93
 [5,]   58 1.523801 15.9330 16392.99
 [6,]  104 3.948959 27.5402 24372.76
 [7,]  141 2.694481 36.3650 11753.93
 [8,]  141 2.694481 36.3650 11753.93
 [9,]  141 2.694481 36.3650 11753.93
[10,]   54 1.393076 13.9556 16159.42

$weights
 [1] 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1

$stats_normalization
          V1           V2           V3           V4
   32.616288     1.050966    14.887083 13081.213750

$epsilon
[1] 4.768351

$nsim
[1] 26

$computime
[1] 11.68463
```

To perform the algorithm of Lenormand et al. (2012), one needs to specify three arguments: the initial number of simulations *nb_simul*, the proportion α of best-fit simulations to update the tolerance level at each step, and the stopping criterion *p_acc_min*. Note that default values $alpha = 0.5$ and $p\_acc\_min = 0.05$ are used if not specified, following Lenormand et al. (2012).

```
> n=10

[1] 10

> alpha=0.5

[1] 0.5

> paccmin=0.4

[1] 0.4

> n_t=5

[1] 5

> ABC_Lenormand<-ABC_sequential(method="Lenormand", model=trait_model,
+ prior_matrix=priormatrix, nb_simul=n, summary_stat_target=sum_stat_obs,
+ alpha=alpha, p_acc_min=paccmin)
```

```
$param
     [,1]     [,2]       [,3] [,4]      [,5]      [,6]
[1,]  500 4.158508  1.2603706    1  8.903764 1.1051477
[2,]  500 4.117288 -1.2432234    1  9.191726 1.5148804
[3,]  500 3.185121 -1.9275739    1  9.871502 2.7399745
[4,]  500 4.353342  0.5965091    1  6.991367 0.9059839
[5,]  500 4.380019  1.4754792    1 10.124825 0.8245725

$stats
     [,1]     [,2]    [,3]     [,4]
[1,]   79 2.589236 13.1032 13849.27
[2,]  108 3.652768 27.7898 30235.70
[3,]   55 2.915457 22.6396 23610.34
[4,]   95 3.139245 15.4400 26487.43
[5,]   84 2.736382 16.0370 20200.32

$weights
[1] 0.8213226133 0.1551950386 0.0109177247 0.0120592720 0.0005053514

$stats_normalization
         V1           V2           V3           V4
   49.638807     1.282858    22.354607 17611.144784

$epsilon
[1] 1.120024

$nsim
[1] 25

$computime
[1] 12.61374
```

4.4 Performing a ABC-MCMC scheme

To perform the algorithm of Marjoram et al. (2003), one needs to specify five arguments: the number of sampled points *n_obs* in the Markov Chain, the number of chain points between two sampled points *n_between_sampling*, the maximal distance accepted between simulations and data *dist_max*, an array *tab_normalization* precising the scale of each summary statistics, and an array *proposal_range* precising the maximal distances in each dimension of the parameter space for a jump of the MCMC.

```
> n=10

[1] 10

> nbetweensampling=1

[1] 1

> distmax=8

[1] 8

> tabnormalization=c(50,1,20,10000)

[1]    50     1    20 10000
```

```
> proposalrange=c(0,1,0.5,0,50,1)
[1]  0.0  1.0  0.5  0.0 50.0  1.0
> ABC_Marjoram_original<-ABC_mcmc(method="Marjoram_original", model=trait_model,
+ prior_matrix=priormatrix, n_obs=n, n_between_sampling=nbetweensampling,
+ summary_stat_target=sum_stat_obs, dist_max=distmax,
+ tab_normalization=tabnormalization, proposal_range=proposalrange)
$param
      [,1]     [,2]      [,3] [,4]      [,5]      [,6]
 [1,]  500 3.124976 1.5903990    1 -7.922647 1.7387968
 [2,]  500 3.676763 1.1533388    1 12.605879 0.8373085
 [3,]  500 4.239396 1.4374781    1 17.863453 1.5475253
 [4,]  500 4.239396 1.4374781    1 17.863453 1.5475253
 [5,]  500 4.559527 1.0153000    1  2.188281 2.1624349
 [6,]  500 4.559527 1.0153000    1  2.188281 2.1624349
 [7,]  500 4.559527 1.0153000    1  2.188281 2.1624349
 [8,]  500 4.969607 1.4745409    1 -5.031578 1.4342656
 [9,]  500 4.887200 1.0561653    1 18.042819 2.2996286
[10,]  500 3.959097 0.6632048    1  7.681712 1.6041961

$stats
      [,1]      [,2]    [,3]     [,4]
 [1,]   30 0.9551174  3.2352 13948.23
 [2,]   54 2.1512630 15.5228 11480.51
 [3,]   94 3.4953307 22.1568 12154.51
 [4,]   94 3.4953307 22.1568 12154.51
 [5,]  132 4.1359012 13.0006 23923.11
 [6,]  132 4.1359012 13.0006 23923.11
 [7,]  132 4.1359012 13.0006 23923.11
 [8,]  120 2.5391072 17.8280 37782.84
 [9,]  161 4.3921155 24.9550 18310.50
[10,]   70 3.1022784 11.5030 15007.48

$dist
 [1] 7.6259024 4.4478449 4.2013264 4.2013264 3.5775376 3.5775376 3.5775376
 [8] 0.7790497 6.4963262 3.1509926

$stats_normalization
[1]    50     1    20 10000

$epsilon
[1] 7.625902

$nsim
[1] 10

$n_between_sampling
[1] 1

$computime
[1] 3.301186
```

To perform the algorithm of Marjoram et al. (2003) in which some of the arguments (*dist_max*, *tab_normalization* and *proposal_range*) are automatically determined by the algorithm via an initial calibration step, one needs to specify three arguments: the number *n_calibration* of simulations to perform at the calibration step, the tolerance quantile *tolerance_quantile* to be used for the determination of *dist_max* and the scale factor *proposal_phi* to determine the proposal range. These modifications are drawn from the algorithm of Wegmann et al. (2009a), without relying on PLS regressions. The arguments are set by default to: *n_calibration* $= 10000$, *tolerance_quantile* $= 0.01$ and *proposal_phi* $= 1$.

```
> n=10

[1] 10

> nbetweensampling=1

[1] 1

> ncalib=10

[1] 10

> tolquantile = 0.5

[1] 0.5

> proposalphi=1

[1] 1

> ABC_Marjoram<-ABC_mcmc(method="Marjoram", model=trait_model, prior_matrix=priormatrix,
+ n_obs=n, n_between_sampling=nbetweensampling, summary_stat_target=sum_stat_obs,
+ n_calibration=ncalib, tolerance_quantile=tolquantile, proposal_phi=proposalphi)

$param
      [,1]     [,2]       [,3] [,4]     [,5]      [,6]
 [1,]  500 3.988835 -0.8948909    1 36.05396 0.3966097
 [2,]  500 3.925306 -0.3931179    1 36.31668 0.6295180
 [3,]  500 4.100871 -0.6441280    1 17.59044 0.5783916
 [4,]  500 4.199818 -0.0314530    1 26.29178 0.5878398
 [5,]  500 4.284278  0.4431786    1 38.53033 0.7152106
 [6,]  500 4.263149  0.3579933    1 51.44345 0.6491789
 [7,]  500 4.326115 -0.2138600    1 64.08909 0.3711451
 [8,]  500 4.326115 -0.2138600    1 64.08909 0.3711451
 [9,]  500 4.469530 -0.4746148    1 65.72144 0.5067363
[10,]  500 4.469530 -0.4746148    1 65.72144 0.5067363

$stats
     [,1]     [,2]    [,3]       [,4]
 [1,]   92 2.888797 42.1246   8819.152
 [2,]   84 1.783233 40.9812   6977.750
 [3,]   93 2.757348 26.0524  19157.677
 [4,]   80 2.137944 31.3448  11243.440
 [5,]   76 2.780116 41.1372   5264.130
 [6,]   91 3.168181 52.2628  -1120.951
 [7,]   99 2.702537 59.1900  -8820.113
 [8,]   99 2.702537 59.1900  -8820.113
 [9,]  113 3.159844 59.6688 -10137.637
[10,]  113 3.159844 59.6688 -10137.637
```

```
$dist
 [1]  3.0228392  3.5278474  0.5921071  1.9193838  3.7487349  6.5287394
 [7]  9.4654856  9.4654856 10.2825983 10.2825983

$stats_normalization
[1]    42.449971     1.263561    19.766110 16540.638697

$epsilon
[1] 10.2826

$nsim
[1] 20

$n_between_sampling
[1] 1

$computime
[1] 10.88352
```

To perform the algorithm of Wegmann et al. (2009a), one needs to specify four arguments: the number *n_calibration* of simulations to perform at the calibration step, the tolerance quantile *tolerance_quantile* to be used for the determination of *dist_max*, the scale factor *proposal_phi* to determine the proposal range and the number of components *numcomp* to be used in PLS regressions. The arguments are set by default to: *n_calibration* $= 10000$, *tolerance_quantile* $=$ 0.01, *proposal_phi* $= 1$ and *numcomp* $= 0$, this last default value encodes a choice of a number of PLS components equal to the number of summary statistics.

```
> n=10

[1] 10

> nbetweensampling=1

[1] 1

> ncalib=10

[1] 10

> tolquantile = 0.5

[1] 0.5

> proposalphi=1

[1] 1

> ABC_Wegmann<-ABC_mcmc(method="Wegmann", model=trait_model, prior_matrix=priormatrix,
+ n_obs=n, n_between_sampling=nbetweensampling, summary_stat_target=sum_stat_obs,
+ n_calibration=ncalib, tolerance_quantile=tolquantile, proposal_phi=proposalphi, numcomp=0)

$param
     [,1]     [,2]      [,3] [,4]      [,5]     [,6]
 [1,]  500 4.907279 0.3774955    1 -23.35125 3.035368
 [2,]  500 4.482948 0.5728118    1 -20.20327 3.079131
 [3,]  500 4.641187 0.6344499    1 -22.45252 2.718452
 [4,]  500 4.341211 0.6218532    1 -16.80841 2.993972
 [5,]  500 4.539367 0.7615360    1 -13.79850 2.710391
 [6,]  500 4.527038 0.8797638    1 -19.07561 2.919837
 [7,]  500 4.260715 0.8215539    1 -17.71762 2.881492
 [8,]  500 4.265160 1.0273045    1 -24.66962 2.869154
 [9,]  500 4.546898 0.9601943    1 -22.00690 2.745330
[10,]  500 4.771147 0.9912668    1 -19.13443 2.658059

$stats
      [,1]     [,2]    [,3]     [,4]
 [1,]  153 4.129597 26.2278 29147.99
 [2,]  122 3.553962 17.0594 35891.81
 [3,]  133 3.700635 19.3356 34499.71
 [4,]   99 2.934351 10.9330 28679.72
 [5,]  121 3.598761 13.6930 31848.39
 [6,]  129 3.703062 17.9106 29094.10
 [7,]   82 2.478448  9.2756 26385.43
 [8,]  100 2.443473 12.0926 35293.59
 [9,]  108 2.212837 14.3530 35953.47
[10,]  129 3.465605 17.4578 35360.08

$dist
 [1] 2.8171232 0.6565423 1.0416598 0.1843182 0.6827404 0.9245031 0.3280108
 [8] 0.1616949 0.1447295 0.6696109

$epsilon
[1] 2.817123

$nsim
[1] 20

$n_between_sampling
[1] 1

$min_stats
[1]     30.000000      1.303733      5.919600 -12738.973868

$max_stats
[1]   176.000000     4.727653    60.624800 29147.985136

$lambda
[1]  0.6060606  5.4545455  1.8181818 -0.6060606

$geometric_mean
[1] 1.517194 1.645689 1.574922 1.347031

$boxcox_mean
[1] 0.5827926 0.4567481 0.5524646 0.4193753

$boxcox_sd
[1] 0.3574215 0.3026118 0.3454799 0.3168303

$pls_transform
            [,1]       [,2]       [,3]       [,4]
[1,] -0.50058980 -0.5453507 -0.5194912  0.4345498
```

```
[2,]  0.51462253  0.3240443 -0.4210780  0.6760361
[3,]  0.61617143 -0.5562092 -0.4644290 -0.5990383
[4,]  0.05888578 -0.2802759  0.7496774  0.5966252

$n_component
[1] 4

$computime
[1] 8.181782
```

4.5 Using multiple cores

The functions of the package EasyABC can launch the simulations on multiple cores of a computer: users only have to indicate the number of cores they wish to use in the argument n_cluster of the functions. The compatibility constraints of the simulation code are slightly different when using multiple cores: please refer to section 3.3 for more information.

5 Troubleshooting and development

Please send comments, suggestions and bug reports to nicolas.dumoulin@irstea.fr or franck.jabot@irstea.fr Any new development of more efficient ABC schemes that could be included in the package is particularly welcome.

6 Programming Acknowledgements

The EasyABC package makes use of a number of R tools, among which:
- the R package lhs (Carnell 2012) for latin hypercube sampling.
- the R package MASS (Venables and Ripley 2002) for boxcox transformation.
- the R package mnormt (Genz and Azzalini 2012) for multivariate normal generation.
- the R package pls (Mevik and Wehrens 2011) for partial least square regression.
- the R script for the Wegmann et al. (2009a)'s algorithm drawn from the ABCtoolbox documentation (Wegmann et al. 2009b).

7 References

Beaumont, M. A., Cornuet, J., Marin, J., and Robert, C. P. (2009) Adaptive approximate Bayesian computation. *Biometrika*,**96**, 983–990.

Carnell, R. (2012) lhs: Latin Hypercube Samples. R package version 0.10. http://CRAN.R-project.org/package=lhs

Csilléry, K., François, O., and Blum, M.G.B. (2012) abc: an r package for approximate bayesian computation (abc). *Methods in Ecology and Evolution*, **3**, 475–479.

Del Moral, P., Doucet, A., and Jasra, A. (2012) An adaptive sequential Monte Carlo method for approximate Bayesian computation. *Statistics and Computing*, **22**, 1009–1020.

Drovandi, C. C. and Pettitt, A. N. (2011) Estimation of parameters for macroparasite population evolution using approximate Bayesian computation. *Biometrics*, **67**, 225–233.

Genz, A., and Azzalini, A. (2012) mnormt: The multivariate normal and t distributions. R package version 1.4-5. http://CRAN.R-project.org/package=mnormt

Jabot, F. (2010) A stochastic dispersal-limited trait-based model of community dynamics. *Journal of Theoretical Biology*, **262**, 650–661.

Lenormand, M., Jabot, F., Deffuant G. (2012) Adaptive approximate Bayesian computation for complex models. http://arxiv.org/pdf/1111.1308.pdf

Marjoram, P., Molitor, J., Plagnol, V. and Tavaré, S. (2003) Markov chain Monte Carlo without likelihoods. *PNAS*, **100**, 15324–15328.

Mevik, B.-H., and Wehrens, R. (2011) pls: Partial Least Squares and Principal Component regression. R package version 2.3-0. http://CRAN.R-project.org/package=pls

Pritchard, J.K., and M.T. Seielstad and A. Perez-Lezaun and M.W. Feldman (1999) Population growth of human Y chromosomes: a study of Y chromosome microsatellites. *Molecular Biology and Evolution*, **16**, 1791–1798.

Sisson, S.A., Fan, Y., and Tanaka, M.M. (2007) Sequential Monte Carlo without likelihoods. *PNAS*, **104**, 1760–1765.

Venables, W.N., and Ripley, B.D. (2002) Modern Applied Statistics with S. Fourth Edition. Springer, New York.

Wegmann, D., Leuenberger, C. and Excoffier, L. (2009a) Efficient approximate Bayesian computation coupled with Markov chain Monte Carlo without likelihood. *Genetics*, **182**, 1207-1218.

Wegmann, D., Leuenberger, C. and Excoffier, L. (2009b) Using ABCtoolbox. http://cmpg.unibe.ch/software/abctoolbox/ABCtoolbox_manual.pdf