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

Franck Jabot, Thierry Faure, Nicolas Dumoulin, Carlo Albert

EasyABC version 1.4.99, 2015-06-11

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. Sections 4 and 5 present two detailed worked examples.

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). Four different MCMC schemes are available: the ones of Marjoram et al. (2003), Wegmann et al. (2009a), 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) and a Simulated Annealing algorithm (SABC) suggested in Albert et al. (2014). Details on how to implement these various algorithms with `EasyABC` are given in the manual pages of each function and two examples are detailed in Sections 4 and 5. 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 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.

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 simulations of the model. 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".

2.4 Simulated annealing

Inspired by Simulated Annealing algorithms used for optimization, the SABC algorithm from Albert et al. (2014) uses MCMC chains to propagate an ensemble of particles in the product space of parameters and model outputs and continuously lowers the tolerance between model outputs and the data so that the parameter marginal converges to the posterior. The tolerance is lowered adaptively so as to minimize entropy production, which serves as a measure for computational waste. In `EasyABC`, SABC is implemented in the function `SABC`.

3 Installation and requirements

3.1 Installing the package

A version of R greater than or equal to 2.15.0 is required. The package has been tested on Windows 32 and Linux, but not on Mac. 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 a vector of parameter values and it must return a vector of summary statistics. If the 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 5 below).

Alternatively, users may prefer building a `R` function calling their binary executable file. A short tutorial is provided in section 3.9 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 a vector of parameter values and it must return a vector 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. This means that the option `use_seed` must be turned to `TRUE` when using `EasyABC` with multiple cores.

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 5 below).

Alternatively, users may prefer building a `R` function calling their binary executable file. A short tutorial is provided in section 3.9 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 5), the simulation code `trait_model` makes use of this package option, and in the first example (Section 4), the way this option can be used with a simple `R` function is demonstrated.

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

3.5 Encoding the prior distributions

A list encoding the prior distributions used for each model parameter must be supplied by the user. Each element of the list corresponds to a model parameter and can be defined in two ways:

1. By using predefined prior distributions. In this case, the list element must be a vector whose first argument determines the type of prior distribution followed by the argument of the distribution function, possible values are:

   - "unif" for a uniform distribution on a segment, followed by two numbers the minimum and maximum values of the uniform distribution
   - "normal" for a normal distribution, followed by two numbers the mean and standard deviation of the normal distribution
   - "lognormal" for a lognormal distribution, followed by two numbers: the mean and standard deviation on the log scale of the lognormal distribution
   - "exponential" for an exponential distribution, followed by one number: the rate of the exponential distribution

   ```
   > my_prior=list(c("unif",0,1),c("normal",1,2))
   [[1]]
   [1] "unif" "0"    "1"

   [[2]]
   [1] "normal" "1"      "2"
   ```

NB: Note that a fixed variable can be passed to the simulation code by choosing for this fixed variable a uniform prior distribution and a trivial range (with equal lower and upper bounds). The `EasyABC` methods will not work properly if these fixed variables are passed with other types of prior distributions (like a normal distribution with a standard deviation equal to zero).

2. By providing the user-defined sampling and density function. In this case, each list element must be itself a list of two elements: the sampling function and the density function. For example, a uniform distribution can be defined using this approach with the following code (equivalent to `my_prior=list(c("unif",0,1))`):

```
> my_prior=list(list(c("runif",1,0,1), c("dunif",0,1)))

[[1]]
[[1]][[1]]
[1] "runif" "1"     "0"     "1"

[[1]][[2]]
[1] "dunif" "0"     "1"
```

3.6 Adding constraints to prior distributions

To add constraints to prior distributions (for instance, parameter 1 < parameter 2), users need to use the parameter `prior_test` in the ABC functions of the package (see their online documentation). This parameter `prior_test` will be evaluated as a logical expression, you can use all the logical operators including `"<"`, `">"`, ... to define whether a parameter set respects the constraint. Each parameter should be designated with `"X1"`, `"X2"`, ... in the same order as in the prior definition.

Here is an example where the second parameter should be greater than the first one:

```
prior = list(c("unif",0,1),c("unif",0,10))
ABC_rejection(model=a_model,prior=prior,nb_simul=3, prior_test="X2 > X1")
```

3.7 The target summary statistics

A vector containing the summary statistics of the data must be supplied. The statistics must be in the same order as in the simulation outputs. The function `SABC` allows for a semi-automatic generation of summary statistics according to Fearnhead et al. (2012).

3.8 The option verbose

Intermediary results can be written in output files in the working directory. Users solely need to choose the option `verbose=TRUE` when launching the `EasyABC` functions (otherwise, the default value for `verbose` is `FALSE`). Intermediary results consist in the progressive writing of simulation outputs for the functions `ABC_rejection` and `ABC_mcmc` and in the writing of intermediary results at the end of each step for the function `ABC_sequential`. Additional details are provided in the help files of the functions.

3.9 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(1,1,1,1,1,1)) {
  .C("trait_model",input=input,stat_to_return=array(0,4))$stat_to_return
}
```

Now, as our model will have two parameters with constant values (see 5), we can fix them as following:

```
trait_model <- function(input=c(1,1,1,1,1,1)) {
  .C("trait_model",input=c(input[1], 500, input[2:3], 1, input[4:5]),
    stat_to_return=array(0,4))$stat_to_return
}
```

3.10 Example of integration of an external program: `fastsimcoal`

This example is provided by an EasyABC user Albert Min-Shan Ko (currently at the Department of genetics, Max Planck Institute of Evolutionary Anthropology, Leipzig, Germany). The purpose is to plug a third-party software related to population genetics into the EasyABC workflow. This software needs input data in a given format, so the idea is to wrap the call to the `fastsimcoal` software into a script that will link EasyABC to `fastsimcoal`.

Here are the scripts as provided by courtesy of Albert Min-Shan Ko.

- First, a R script reformats the parameters to be used by `fastsimcoal` (here named `mod.input.r`).

```
r<-read.table('input',head=F)
sink('mod.input')
cat(paste('1','p1','unif',round(r[1,],0),round(r[1,],0),sep='\t'))
cat('\n')
cat(paste('1','p2','unif',round(r[2,],0),round(r[2,],0),sep='\t'))
cat('\n')
cat(paste('1','p3','unif',round(r[3,],0),round(r[3,],0),sep='\t'))
sink()
```

- Second, a GNU Bash script (here names `run_sim.sh`) invokes the latter R script and builds a parameter file for `fastsimcoal` (`sim.est`), runs `fastsimcoal` and computes some summary statistics with the arlequin program.

```
#!/bin/bash
rm -fr sim
Rscript mod.input.r
cat <(sed -n 1p template.est) <(sed -n '1,3'p mod.input) \
  <(sed -n '5,\$'p template.est) > sim.est
  until [ -f sim/arl_output ]; do
    ./fastsimcoal -t sim.tpl -e sim.est -E1 -n1 -q
    ./arlsumstat sim/sim_1_1.arp sim/arl_output 1 0 run_silent
  done
cat sim/arl_output > output
```

Then, the user can invoke EasyABC like this :

```
prior=list(c("unif",500,1000),c("unif",100,500),c("unif",50,200))
ABC_sim<-ABC_rejection(model=binary_model('./run_sim.sh'),prior=prior,nb_simul=3)
```

3.11 Example of integration of a java model

If your model runs with a Java Virtual Machine (can be written in Java, Scala, Groovy, ...), you can of course use the `binary_model` wrapper to run the JVM within your model. But, you can achieve a tighter integration that will simplify the process and save computing time. This section propose to use the R package `rJava`.

Let's consider the toy model written in Java (in a file named Model.java):

```
public class Model {
  public static double[] run(double[] x) {
      double[] result = new double[2];
      result[0] = x[0] + x[1];
      result[1] = x[0] * x[1];
      return result;
  }
}
```

We can compile it with the command: `javac Model.java` and then define our wrapper in R:

```
mymodel <- function(x) {
  library("rJava")
  .jinit(classpath=".")
  result = .jcall(J("Model"),"[D","run",.jarray(x))
  result
}
```

Then, the user can invoke EasyABC like this :

```
prior=list(c("unif",0,1),c("normal",1,2))
ABC_sim<-ABC_rejection(model=mymodel,prior=prior,nb_simul=3)
```

4 A first worked example

4.1 The toy model

We here consider a very simple stochastic model coded in the `R` language:

```
> toy_model<-function(x){
+   c( x[1] + x[2] + rnorm(1,0,0.1) , x[1] * x[2] + rnorm(1,0,0.1) )
+ }

function(x){
  c( x[1] + x[2] + rnorm(1,0,0.1) , x[1] * x[2] + rnorm(1,0,0.1) )
}
```

We will use two different types of prior distribution for the two model parameters ($x[1]$ and $x[2]$): a uniform distribution between 0 and 1 and a normal distribution with mean 1 and standard deviation 2.

```
> toy_prior=list(c("unif",0,1),c("normal",1,2))

[[1]]
[1] "unif" "0"    "1"

[[2]]
[1] "normal" "1"      "2"
```

And we will consider an imaginary dataset of two summary statistics that the toy_model is aiming at fitting:

```
> sum_stat_obs=c(1.5,0.5)

[1] 1.5 0.5
```

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 and the proportion of simulations which are to be retained p:

```
> set.seed(1)

NULL

> n=10

[1] 10

> p=0.2

[1] 0.2

> ABC_rej<-ABC_rejection(model=toy_model, prior=toy_prior, nb_simul=n,
+ summary_stat_target=sum_stat_obs, tol=p)

$param
           [,1]      [,2]
param 0.6927316 0.8877425
param 0.3162717 1.0934523

$stats
         [,1]      [,2]
[1,] 1.564895 0.4678920
[2,] 1.386153 0.2915392
```

```
$weights
[1] 0.5 0.5

$stats_normalization
[1] 0.7266951 0.5603033

$nsim
[1] 10

$nrec
[1] 2

$computime
[1] 0.001829386
```

Alternatively, `ABC_rejection` can be used to solely launch the simulations and to store the simulation outputs without performing the rejection step. This option enables the user to make use of the `R` package `abc` (Csilléry et al. 2012) which offers an array of more sophisticated post-processing treatments than the simple rejection procedure:

```
> # Run the ABC rejection on the model
> set.seed(1)

NULL

> n=10

[1] 10

> ABC_rej<-ABC_rejection(model=toy_model, prior=toy_prior, nb_simul=n)

$param
           [,1]      [,2]
param 0.2655087 0.3475333
param 0.6607978 1.6590155
param 0.7698414 0.9884657
param 0.2121425 1.7796865
param 0.8696908 0.1769783
param 0.6684667 2.6424424
param 0.7829328 1.2666727
param 0.6927316 0.8877425
param 0.3162717 1.0934523
param 0.3323947 1.7753432

$stats
           [,1]       [,2]
 [1,] 0.7460219 0.21951603
 [2,] 2.2377665 1.14501671
 [3,] 1.9987724 0.83732115
 [4,] 1.9297049 0.15607719
 [5,] 1.0718915 0.06472432
 [6,] 3.3702993 1.85828258
 [7,] 2.1300244 0.98600890
 [8,] 1.5648945 0.46789202
 [9,] 1.3861534 0.29153921
[10,] 2.1023574 0.45240868
```

```
$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] 0.7266951 0.5603033

$nsim
[1] 10

$computime
[1] 0.001153946
```

```
> # Install if needed the "abc" package
> install.packages("abc")
```

```
> # Post-process the simulations outputs
> library(abc)
```

```
 [1] "EasyABC"   "abc"       "MASS"      "quantreg"  "SparseM"   "nnet"
 [7] "stats"     "graphics"  "grDevices" "utils"     "datasets"  "methods"
[13] "base"
```

```
> rej<-abc(sum_stat_obs, ABC_rej$param, ABC_rej$stats, tol=0.2, method="rejection")
```

```
Call:
abc(target = sum_stat_obs, param = ABC_rej$param, sumstat = ABC_rej$stats,
    tol = 0.2, method = "rejection")
Method:
Rejection

Parameters:
P1, P2

Statistics:
S1, S2

Total number of simulations 10

Number of accepted simulations:  2
```

```
> # simulations selected:
> rej$unadj.values
```

```
           [,1]      [,2]
param 0.6927316 0.8877425
param 0.3162717 1.0934523
```

```
> # their associated summary statistics:
> rej$ss
```

```
         [,1]      [,2]
[1,] 1.564895 0.4678920
[2,] 1.386153 0.2915392
```

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

```
 [1] 1.6103923 1.9542368 1.2025193 1.0981535 1.2163667 4.6145013 1.5879393
 [8] 0.1448057 0.4716867 1.2112846
```

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(1.25,0.75)

[1] 1.25 0.75

> ABC_Beaumont<-ABC_sequential(method="Beaumont", model=toy_model,
+ prior=toy_prior, nb_simul=n, summary_stat_target=sum_stat_obs,
+ tolerance_tab=tolerance)

$param
           [,1]      [,2]
 [1,] 0.7800180 0.4830061
 [2,] 0.3181763 2.3673583
 [3,] 0.1811065 2.6700808
 [4,] 0.8456229 0.2467030
 [5,] 0.1418500 1.9097104
 [6,] 0.3282295 2.8256064
 [7,] 0.1976839 1.2863106
 [8,] 0.3323167 2.3794734
 [9,] 0.2252921 1.5824478
[10,] 0.6077307 0.4225812

$stats
           [,1]       [,2]
 [1,] 1.3838109 0.49279371
 [2,] 2.6442826 0.65600856
 [3,] 2.8649926 0.47168981
 [4,] 1.0743703 0.19859866
 [5,] 2.0575765 0.21200292
 [6,] 3.0582520 0.80427662
 [7,] 1.4783048 0.06284696
 [8,] 2.6188539 0.64199282
 [9,] 1.7743399 0.35304040
[10,] 0.8912953 0.24977382

$weights
 [1] 0.09597405 0.09105900 0.10164109 0.10169384 0.12145842 0.08442337
 [7] 0.11758458 0.08987372 0.11027429 0.08601765

$stats_normalization
[1] 2.290391 1.433019

$epsilon
[1] 0.5079519

$nsim
[1] 31
```

```
$computime
[1] 0.01041698
```

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

> tolerance=0.75

[1] 0.75

> c_drov=0.7

[1] 0.7

> ABC_Drovandi<-ABC_sequential(method="Drovandi", model=toy_model,
+ prior=toy_prior, nb_simul=n, summary_stat_target=sum_stat_obs,
+ tolerance_tab=tolerance, c=c_drov)

$param
           [,1]      [,2]
 [1,] 0.6988245 0.8190878
 [2,] 0.6860284 0.1430693
 [3,] 0.3080524 2.0202168
 [4,] 0.4009210 0.8702183
 [5,] 0.4295584 0.4770350
 [6,] 0.6366469 0.8473816
 [7,] 0.9931522 0.2698854
 [8,] 0.3444874 0.5230873
 [9,] 0.5484915 1.4070225
[10,] 0.3991026 0.7969931

$stats
           [,1]      [,2]
 [1,] 1.5335613 0.4986674
 [2,] 0.9236163 0.1415199
 [3,] 2.3118317 0.6644022
 [4,] 1.3034399 0.4532500
 [5,] 0.8410152 0.2013222
 [6,] 1.5010774 0.4530792
 [7,] 1.3363126 0.3626958
 [8,] 1.0170970 0.2974128
 [9,] 1.9362341 0.9295290
[10,] 1.2003607 0.1584102

$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] 1.953418 1.214883

$epsilon
[1] 0.1910321

$nsim
[1] 45

$computime
[1] 0.0105145
```

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

> alpha_delmo=0.5

[1] 0.5

> tolerance=0.75

[1] 0.75

> ABC_Delmoral<-ABC_sequential(method="Delmoral", model=toy_model,
+ prior=toy_prior, nb_simul=n, summary_stat_target=sum_stat_obs,
+ alpha=alpha_delmo, tolerance_target=tolerance)

$param
            [,1]       [,2]
 [1,] 0.78127811 -0.2702393
 [2,] 0.05077116  2.1359039
 [3,] 0.06904047  2.8398604
 [4,] 0.68078598  0.8291069
 [5,] 0.09484793  1.6327956
 [6,] 0.49198560 -0.2347007
 [7,] 0.64359353  0.8969070
 [8,] 0.20850839  1.1239227
 [9,] 0.08957006  1.0364054
[10,] 0.42904699  0.1828383

$stats
           [,1]         [,2]
 [1,] 0.4462728 -0.146496105
 [2,] 2.1294208 -0.027886795
 [3,] 2.6991126  0.095629112
 [4,] 1.4473701  0.655829205
 [5,] 1.6872698  0.178016895
 [6,] 0.2232969 -0.054842893
 [7,] 1.4422756  0.577653747
```

```
 [8,] 1.1639790  0.219924657
 [9,] 0.9855821  0.236920205
[10,] 0.7760881  0.001486995

$weights
 [1] 0.0000000 0.1428571 0.0000000 0.1428571 0.1428571 0.0000000 0.1428571
 [8] 0.1428571 0.1428571 0.1428571

$stats_normalization
[1] 1.8358809 0.9232688

$epsilon
[1] 0.8447518

$nsim
[1] 33

$computime
[1] 0.01206541
```

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). Also note that the method "Lenormand" is only supported with uniform prior distributions (since it performs a Latin Hypercube sampling at the beginning). Here, we therefore need to alter the prior distribution of the second model parameter:

```
> toy_prior2=list(c("unif",0,1),c("unif",0.5,1.5))

[[1]]
[1] "unif" "0"    "1"

[[2]]
[1] "unif" "0.5"  "1.5"

> n=10

[1] 10

> pacc=0.4

[1] 0.4

> ABC_Lenormand<-ABC_sequential(method="Lenormand", model=toy_model,
+ prior=toy_prior2, nb_simul=10, summary_stat_target=sum_stat_obs,
+ p_acc_min=pacc)

$param
          [,1]      [,2]
[1,] 0.4238934 1.0930780
[2,] 0.4014618 1.3217679
[3,] 0.4641413 0.9797916
[4,] 0.3823968 0.9414760
[5,] 0.4517912 1.0619879
```

```
$stats
         [,1]      [,2]
[1,] 1.539900 0.4970336
[2,] 1.580238 0.5560530
[3,] 1.430449 0.5941463
[4,] 1.392671 0.4194371
[5,] 1.591849 0.3816762

$weights
[1] 0.45057481 0.28520208 0.16563157 0.06030091 0.03829063

$stats_normalization
[1] 0.5679161 0.4919836

$epsilon
[1] 0.08399851

$nsim
[1] 30

$computime
[1] 0.01588416
```

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_rec* 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*, a vector *tab_normalization* precising the scale of each summary statistics, and a vector *proposal_range* precising the maximal distances in each dimension of the parameter space for a jump of the MCMC. All these arguments have default values (see the package help for the function `ABC_mcmc`), so that `ABC_mcmc` will work without user-defined values.

```
> n=10

[1] 10

> ABC_Marjoram_original<-ABC_mcmc(method="Marjoram_original", model=toy_model,
+ prior=toy_prior, summary_stat_target=sum_stat_obs, n_rec=n)

[1] "Warning: summary statistics are normalized by default through a division by the target summa
[1] "Consider providing normalization constants for each summary statistics in the option 'tab_no
[1] "Warning: default values for proposal distributions are used - they may not be appropriate to
[1] "Consider providing proposal range constants for each parameter in the option 'proposal_range
[1] "Warning: a default value for the tolerance has been computed - it may not be appropriate to
[1] "Consider providing a tolerance value in the option 'dist_max' or using the method 'Marjoram'
$param
            [,1]      [,2]
 [1,] 0.56259845 1.4003546
 [2,] 0.54894245 1.2773139
 [3,] 0.45760461 1.2563753
 [4,] 0.53848397 1.0236945
 [5,] 0.44819730 1.3325929
 [6,] 0.31813265 1.2246407
 [7,] 0.27645825 0.4381652
```

```
 [8,] 0.23817187 1.2712988
 [9,] 0.16680816 1.2391745
[10,] 0.07056915 0.6239996

$stats
          [,1]       [,2]
 [1,] 1.878234 0.71215059
 [2,] 1.748384 0.52208320
 [3,] 1.550066 0.47266403
 [4,] 1.641593 0.48570519
 [5,] 1.794420 0.78160536
 [6,] 1.649409 0.46514911
 [7,] 0.650774 0.14511484
 [8,] 1.601802 0.21169643
 [9,] 1.421674 0.14952471
[10,] 0.585113 0.08518684

$dist
 [1] 0.243614137 0.029370496 0.004103076 0.009727851 0.355732224 0.014779640
 [7] 0.824300506 0.337081871 0.494058380 1.060287912

$stats_normalization
[1] 1.5 0.5

$epsilon
[1] 1.060288

$nsim
[1] 92

$n_between_sampling
[1] 10

$computime
[1] 0.05546999
```

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 $proposal\_phi = 1$. This way of automatic determination of *dist_max*, *tab_normalization* and *proposal_range* is strongly recommended, compared to the crude automatic determination proposed in the method `Marjoram_original`.

```
> n=10

[1] 10

> ABC_Marjoram<-ABC_mcmc(method="Marjoram", model=toy_model,
+ prior=toy_prior, summary_stat_target=sum_stat_obs, n_rec=n)

$param
          [,1]      [,2]
```

```
 [1,] 0.4261381 1.2195024
 [2,] 0.2612382 1.2746695
 [3,] 0.2975477 1.2931405
 [4,] 0.3198904 1.1987988
 [5,] 0.4606831 1.0626738
 [6,] 0.4055440 0.9647904
 [7,] 0.3954450 1.0434970
 [8,] 0.5460065 0.9694137
 [9,] 0.6009988 0.8865792
[10,] 0.5748806 0.9669014

$stats
          [,1]      [,2]
 [1,] 1.526002 0.4791648
 [2,] 1.524407 0.4434820
 [3,] 1.594139 0.5080067
 [4,] 1.433806 0.5415198
 [5,] 1.579296 0.5326551
 [6,] 1.436487 0.4785101
 [7,] 1.586596 0.4711887
 [8,] 1.565939 0.4908674
 [9,] 1.542017 0.5616337
[10,] 1.497396 0.4379953

$dist
 [1] 0.0004596454 0.0023264926 0.0021802229 0.0022343522 0.0022445583
 [6] 0.0012880529 0.0023750524 0.0011051788 0.0030215350 0.0026289138

$stats_normalization
[1] 2.036691 1.209682

$epsilon
[1] 0.003021535

$nsim
[1] 10091

$n_between_sampling
[1] 10

$computime
[1] 0.7450817
```

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
```

```
> ABC_Wegmann<-ABC_mcmc(method="Wegmann", model=toy_model,
+ prior=toy_prior, summary_stat_target=sum_stat_obs, n_rec=n)

$param
           [,1]      [,2]
 [1,] 0.6039391 0.6217049
 [2,] 0.6039391 0.6217049
 [3,] 0.6862493 0.7345618
 [4,] 0.6444770 0.7826811
 [5,] 0.5771824 0.8634498
 [6,] 0.5771824 0.8634498
 [7,] 0.5321456 0.9708301
 [8,] 0.6284189 0.9781222
 [9,] 0.6970519 0.9518443
[10,] 0.5806452 0.9653508

$stats
          [,1]      [,2]
 [1,] 1.513577 0.5570033
 [2,] 1.513577 0.5570033
 [3,] 1.440597 0.4963756
 [4,] 1.448160 0.5055089
 [5,] 1.506074 0.5445662
 [6,] 1.506074 0.5445662
 [7,] 1.517320 0.4695789
 [8,] 1.444572 0.4821234
 [9,] 1.528422 0.4941067
[10,] 1.436026 0.5521671

$dist
 [1] 0.0025526933 0.0025526933 0.0008277370 0.0006434536 0.0015420497
 [6] 0.0015420497 0.0007812638 0.0009632257 0.0002124702 0.0030265966

$epsilon
[1] 0.003026597

$nsim
[1] 10091

$n_between_sampling
[1] 10

$min_stats
[1] -6.007372 -4.873890

$max_stats
[1] 9.273201 8.222326

$lambda
[1] 0.6060606 0.6060606

$geometric_mean
[1] 1.483335 1.406518
```

```
$boxcox_mean
[1] 0.5237961 0.4351582

$boxcox_sd
[1] 0.13159747 0.08961018

$pls_transform
           [,1]       [,2]
[1,] -0.7122335 -0.7048425
[2,] -0.6566722  0.7541762

$n_component
[1] 2

$computime
[1] 73.95887
```

4.5 Performing a SABC scheme

For the SABC algorithm by Albert et al. (2014) we need to provide the prior in the form of a sampler and a density:

```
> r.prior <- function()   c(runif(1,0,1),rnorm(1,1,2))

function()   c(runif(1,0,1),rnorm(1,1,2))

> d.prior <- function(x)  dunif(x[1],0,1)*dnorm(x[2],1,2)

function(x)  dunif(x[1],0,1)*dnorm(x[2],1,2)
```

Furthermore, we need to specify the size of the ensemble, the number of simulations and the initial tolerance

```
> n.sample <- 300

[1] 300

> iter.max <- n.sample * 30

[1] 9000

> eps.init <- 2

[1] 2
```

Since, for this example, the prior carries relevant information, we choose the method "informative":

```
> ABC_Albert <-SABC(r.model = toy_model,
+                   r.prior = r.prior,
+                   d.prior = d.prior,
+                   n.sample = n.sample,
+                   eps.init = eps.init,
+                   iter.max = iter.max,
+                   method = "informative",
+                   y = sum_stat_obs
+                   )
```

```
ensemble updates 4.33          eps.1 1.764944          eps.2 -0.2214506
ensemble updates 5.33          eps.1 1.420851          eps.2 -0.1854335
ensemble updates 6.33          eps.1 1.194413          eps.2 -0.02209231
ensemble updates 7.33          eps.1 1.024058          eps.2 -0.03447357
ensemble updates 8.33          eps.1 1.024058          eps.2 -0.03447357
ensemble updates 9.33          eps.1 0.9104874          eps.2 -0.1805792
ensemble updates 10.33          eps.1 0.8008526          eps.2 -0.1807286
ensemble updates 11.33          eps.1 0.7103717          eps.2 -0.2743043
ensemble updates 12.33          eps.1 0.7103717          eps.2 -0.2743043
ensemble updates 13.33          eps.1 0.6526791          eps.2 -0.5022296
ensemble updates 14.33          eps.1 0.6303172          eps.2 -0.3395959
ensemble updates 15.33          eps.1 0.6303172          eps.2 -0.3395959
ensemble updates 16.33          eps.1 0.5539463          eps.2 -0.4591772
Resampling. Effective sampling size:  296.6063
ensemble updates 17.33          eps.1 0.3926393          eps.2 0.4639381
ensemble updates 18.33          eps.1 0.3926393          eps.2 0.4639381
ensemble updates 19.33          eps.1 0.3494795          eps.2 0.2220292
ensemble updates 20.33          eps.1 0.3494795          eps.2 0.2220292
ensemble updates 21.33          eps.1 0.308418          eps.2 0.1141941
ensemble updates 22.33          eps.1 0.2624806          eps.2 0.08698512
ensemble updates 23.33          eps.1 0.2624806          eps.2 0.08698512
ensemble updates 24.33          eps.1 0.222708          eps.2 0.03872068
ensemble updates 25.33          eps.1 0.222708          eps.2 0.03872068
ensemble updates 26.33          eps.1 0.2054514          eps.2 -0.2532238
ensemble updates 27.33          eps.1 0.2054514          eps.2 -0.2532238
ensemble updates 28.33          eps.1 0.1806688          eps.2 -0.4118633
ensemble updates 29.33          eps.1 0.1806688          eps.2 -0.4118633
```

4.6 Using multiple cores

The functions of the package `EasyABC` can launch the simulations on multiple cores of a computer: users have to indicate the number of cores they wish to use in the argument `n_cluster` of the functions, and they have to use the option `use_seed=TRUE`. Users also need to design their code in a slightly different way so that it is compatible with the option `use_seed=TRUE` (see Section 3.3 for additional details). For the toy model above, the modifications needed are the following:

```
> toy_model_parallel<-function(x){
+         set.seed(x[1]) # so that each core is initialized with a different seed value.
+         c( x[2] + x[3] + rnorm(1,0,0.1) , x[2] * x[3] + rnorm(1,0,0.1) )
+ }

function(x){
        set.seed(x[1]) # so that each core is initialized with a different seed value.
        c( x[2] + x[3] + rnorm(1,0,0.1) , x[2] * x[3] + rnorm(1,0,0.1) )
}

> set.seed(1)

NULL

> n=10

[1] 10

> p=0.2
```

```
[1] 0.2
```

```
> ABC_rej<-ABC_rejection(model=toy_model_parallel, prior=toy_prior,
+ nb_simul=n, summary_stat_target=sum_stat_obs, tol=p, n_cluster=2,
+ use_seed=TRUE)

$param
          [,1]      [,2]
[1,] 0.6870228 0.4105591
[2,] 0.2121425 1.7796865

$stats
         [,1]      [,2]
[1,] 1.013496 0.4204994
[2,] 1.983370 0.4615872

$weights
[1] 0.5 0.5

$stats_normalization
[1] 1.420082 0.687775

$nsim
[1] 10

$nrec
[1] 2

$computime
[1] 0.645165
```

5 A second worked example

5.1 The trait model

We turn now to a stochastic ecological model hereafter called `trait_model` to illustrate how to use `EasyABC` with models not initially coded in the `R` language. `trait_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 have been fixed (see section 3.9) into the function `trait_model`. But if it didn't, we would have included these constants in the prior list using

uniform distributions with a trivial ranges, like `c("unif",500,500)` for example.

```
> trait_prior=list(c("unif",3,5),c("unif",-2.3,1.6),
+ c("unif",-25,125), c("unif",-0.7,3.2))

[[1]]
[1] "unif" "3"    "5"

[[2]]
[1] "unif" "-2.3" "1.6"

[[3]]
[1] "unif" "-25"  "125"

[[4]]
[1] "unif" "-0.7" "3.2"
```

We will consider an imaginary 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
```

5.2 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 and the proportion p of retained simulations. Note that the option `use_seed=TRUE` is used, since `trait_model` requires a seed initializing value for its pseudo-random number generator:

```
> set.seed(1)

NULL

> n=10

[1] 10

> p=0.2

[1] 0.2

> ABC_rej<-ABC_rejection(model=trait_model, prior=trait_prior, nb_simul=n,
+ summary_stat_target=sum_stat_obs, tol=p, use_seed=TRUE)

$param
         [,1]      [,2]      [,3]     [,4]
[1,] 4.435237  1.568434  32.00528 2.332036
[2,] 3.534441 -0.794155 -22.99145 0.791313

$stats
     [,1]     [,2]    [,3]     [,4]
[1,]  116 4.104226 32.9882 3029.800
[2,]   84 3.994615 49.1732 1950.147

$weights
```

```
[1] 0.5 0.5

$stats_normalization
[1] 3.981680e+01 6.631974e-01 1.451333e+01 1.526571e+04

$nsim
[1] 10

$nrec
[1] 2

$computime
[1] 5.21084
```

Alternatively, `ABC_rejection` can be used to solely launch the simulations and to store the simulation outputs without performing the rejection step. This option enables the user to make use of the `R` package `abc` (Csilléry et al. 2012) which offers an array of more sophisticated post-processing treatments than the simple rejection procedure:

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

> library(abc)

 [1] "EasyABC"   "abc"       "MASS"      "quantreg"  "SparseM"   "nnet"
 [7] "stats"     "graphics"  "grDevices" "utils"     "datasets"  "methods"
[13] "base"

> set.seed(1)

NULL

> n=10

[1] 10

> p=0.2

[1] 0.2

> ABC_rej<-ABC_rejection(model=trait_model, prior=trait_prior, nb_simul=n, use_seed=TRUE)

$param
          [,1]       [,2]       [,3]        [,4]
 [1,] 3.531017 -0.8487168  60.928005  2.84201038
 [2,] 3.403364  1.2037198 116.701290  1.87711139
 [3,] 4.258228 -2.0590335   5.896186 -0.01142867
 [4,] 4.374046 -0.8019955  90.476213  1.24102704
 [5,] 4.435237  1.5684338  32.005277  2.33203636
 [6,] 4.869410 -1.4726442  72.751065 -0.21033513
 [7,] 3.534441 -0.7941550 -22.991450  0.79131303
 [8,] 4.739382 -0.9726389  47.312017  1.63830672
 [9,] 3.987083 -1.5737514  99.105998  1.90702028
[10,] 4.588480 -1.8790199  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
[1] 3.981680e+01 6.631974e-01 1.451333e+01 1.526571e+04

$nsim
[1] 10

$computime
[1] 5.212872

> rej<-abc(sum_stat_obs, ABC_rej$param, ABC_rej$stats,
+ tol=0.2, method="rejection")

Call:
abc(target = sum_stat_obs, param = ABC_rej$param, sumstat = ABC_rej$stats,
    tol = 0.2, method = "rejection")
Method:
Rejection

Parameters:
P1, P2, P3, P4

Statistics:
S1, S2, S3, S4

Total number of simulations 10

Number of accepted simulations:  2

> # simulations selected:
> rej$unadj.values

         [,1]      [,2]      [,3]     [,4]
[1,] 4.435237  1.568434  32.00528 2.332036
[2,] 3.534441 -0.794155 -22.99145 0.791313

> # 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
```

```
> # 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
```

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

- if it is a `R` function
  `ABC_rejection(My_simulation_code, prior, nb_simul,...)`
- 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, nb_simul, use_seed=TRUE,...)`
- 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, nb_simul, n_cluster=2, use_seed=TRUE)`

5.3 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=trait_prior, nb_simul=n, summary_stat_target=sum_stat_obs,
+ tolerance_tab=tolerance, use_seed=TRUE)

$param
          [,1]       [,2]       [,3]         [,4]
 [1,] 3.362110 -1.9163965 23.0654334 -0.599666827
 [2,] 3.543818  0.9455070 17.2848491  0.716968486
 [3,] 3.260380 -0.9270426 25.3968857  0.779964720
 [4,] 3.398492  0.4703746 15.1099903  0.938328200
 [5,] 3.017150 -1.8711144 26.1860829 -0.002416091
 [6,] 4.053298 -1.7809923 14.1823387 -0.159038819
 [7,] 4.398392  1.0919829  5.7191547  2.949045622
 [8,] 4.358355  0.7294188 14.7202497  2.627312598
 [9,] 4.694502  1.1509709  0.9577352  3.095833860
[10,] 3.672407  0.7490497 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
[1] 4.530833e+01 9.914929e-01 1.550687e+01 1.311587e+04

$epsilon
[1] 4.782638

$nsim
[1] 72

$computime
[1] 29.2165
```

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

> tolerance=3

[1] 3

> c_drov=0.7

[1] 0.7

> ABC_Drovandi<-ABC_sequential(method="Drovandi", model=trait_model,
+ prior=trait_prior, nb_simul=n, summary_stat_target=sum_stat_obs,
+ tolerance_tab=tolerance, c=c_drov, use_seed=TRUE)

$param
          [,1]         [,2]       [,3]        [,4]
 [1,] 4.372120 -0.001121019  5.4715372 -0.12568710
 [2,] 4.051867 -0.125022011 12.1207510 -0.67154259
 [3,] 4.357840 -0.158636879  4.0214488 -0.14801890
 [4,] 4.003453  0.155106599 11.8676334 -0.43394316
 [5,] 4.497001  0.296629792  0.4179174  0.04238219
 [6,] 4.614254 -0.421605678 11.7121630 -0.13055283
 [7,] 4.255693 -0.030613125 13.7085732 -0.09558018
 [8,] 4.320190  0.177067473 15.5170178  0.26745481
```

```
 [9,] 4.704030 -0.007652111  9.6886842  0.34755989
[10,] 4.202139 -0.031661415  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
[1]    45.687583     1.130895    18.184548 14175.858029

$epsilon
[1] 1.204596

$nsim
[1] 40

$computime
[1] 12.12149
```

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

> alpha_delmo=0.5

[1] 0.5

> tolerance=3

[1] 3

> ABC_Delmoral<-ABC_sequential(method="Delmoral", model=trait_model,
+ prior=trait_prior, nb_simul=n, summary_stat_target=sum_stat_obs,
+ alpha=alpha_delmo, tolerance_target=tolerance, use_seed=TRUE)

$param
          [,1]        [,2]      [,3]        [,4]
```

```
 [1,] 4.340312  0.27803253 12.362276  1.38257144
 [2,] 4.142546 -0.85552836  8.562840  0.51965039
 [3,] 4.137010  0.08351261  3.671017  2.36717400
 [4,] 4.108617 -0.06937268 10.516238 -0.03101547
 [5,] 4.254593  0.35733560 13.356750  1.44322454
 [6,] 4.129874  0.13015864 10.201600 -0.10552508
 [7,] 4.157252 -0.16565665  4.865025  1.99019890
 [8,] 4.034561 -0.46067583  5.810586  1.22583149
 [9,] 4.186188 -0.13743236  7.820983  1.35175359
[10,] 4.254593  0.35733560 13.356750  1.44322454

$stats
      [,1]     [,2]    [,3]     [,4]
 [1,]  101 3.233236 20.2856 22151.89
 [2,]   93 2.580371 22.1176 22572.67
 [3,]  107 3.402067 15.8006 32264.44
 [4,]   77 1.631609 19.8354 25750.84
 [5,]   87 3.080927 19.0860 17732.19
 [6,]   81 2.024566 17.0106 18401.83
 [7,]   97 3.042123 14.1450 26042.48
 [8,]   87 3.104230 14.3686 28931.67
 [9,]   97 2.808620 17.5424 28791.39
[10,]   87 3.080927 19.0860 17732.19

$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]    32.616288     1.050966    14.887083 13081.213750

$epsilon
[1] 1.37042

$nsim
[1] 52

$computime
[1] 15.89171
```

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

> pacc=0.4

[1] 0.4

> ABC_Lenormand<-ABC_sequential(method="Lenormand", model=trait_model,
+ prior=trait_prior, nb_simul=n, summary_stat_target=sum_stat_obs,
+ p_acc_min=pacc, use_seed=TRUE)
```

```
$param
         [,1]       [,2]      [,3]      [,4]
[1,] 4.181565  0.1880921 -8.224741 2.1632959
[2,] 3.999284 -0.2759132 -9.953852 2.5511089
[3,] 4.616579  0.9458394  7.014727 0.4664729
[4,] 4.607724  1.2293201  2.027068 0.7340780
[5,] 4.508725  1.3191745  1.673027 0.6871023

$stats
     [,1]     [,2]    [,3]     [,4]
[1,]   94 2.593756 12.0412 33359.00
[2,]   99 2.792966 13.4086 34502.26
[3,]   95 2.625845 14.1682 20079.22
[4,]  113 3.096566 11.4918 29972.94
[5,]   77 2.804382  9.7772 31782.12

$weights
[1] 0.17057544 0.11047443 0.55469515 0.03091801 0.13333697

$stats_normalization
[1]    32.028460     0.789409    18.695073 20209.843809

$epsilon
[1] 0.9711427

$nsim
[1] 30

$computime
[1] 8.652279
```

5.4 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*, a vector *tab_normalization* precising the scale of each summary statistics, and a vector *proposal_range* precising the maximal distances in each dimension of the parameter space for a jump of the MCMC. All these arguments have default values (see the package help for the function `ABC_mcmc`), so that `ABC_mcmc` will work without user-defined values.

```
> n=10

[1] 10

> ABC_Marjoram_original<-ABC_mcmc(method="Marjoram_original", model=trait_model,
+ prior=trait_prior, summary_stat_target=sum_stat_obs, n_rec=n, use_seed=TRUE)

[1] "Warning: summary statistics are normalized by default through a division by the target summa
[1] "Consider providing normalization constants for each summary statistics in the option 'tab_no
[1] "Warning: default values for proposal distributions are used - they may not be appropriate to
[1] "Consider providing proposal range constants for each parameter in the option 'proposal_range
[1] "Warning: a default value for the tolerance has been computed - it may not be appropriate to
[1] "Consider providing a tolerance value in the option 'dist_max' or using the method 'Marjoram'
$param
```

```
          [,1]        [,2]      [,3]     [,4]
 [1,] 4.997627  0.84010920 25.536706 1.278158
 [2,] 4.755947  0.68249262 22.105614 1.612602
 [3,] 4.651135  0.33003172 21.718700 1.812796
 [4,] 4.695900  0.08157121 12.530543 1.643469
 [5,] 4.539471  0.37290377  5.105711 1.797239
 [6,] 4.548357  0.12496940  9.455071 1.713712
 [7,] 4.549036  0.11335499  2.889321 1.560856
 [8,] 4.501335  0.04634601  4.737510 1.537775
 [9,] 4.578767 -0.07938267 16.193415 1.710819
[10,] 4.648880 -0.13622707  3.653714 1.346123

$stats
      [,1]     [,2]    [,3]     [,4]
 [1,]  160 4.289764 34.2134 13233.20
 [2,]  136 3.830190 27.7274 13269.30
 [3,]  128 4.051209 26.7252 11257.43
 [4,]  136 4.025038 23.7432 23336.36
 [5,]  112 3.778224 14.9670 29343.19
 [6,]  138 4.191242 22.5144 26229.38
 [7,]  113 3.533753 18.0050 32206.39
 [8,]  114 3.767839 16.8016 33633.82
 [9,]  138 4.191255 25.6692 19795.19
[10,]  140 4.087263 21.3230 34907.71

$dist
 [1] 1.6899342 0.8730048 0.9667866 0.5860855 0.3396242 0.6336507 0.2032423
 [8] 0.3170328 0.7981136 0.5942420

$stats_normalization
[1]   100.0     2.5    20.0 30000.0

$epsilon
[1] 1.689934

$nsim
[1] 94

$n_between_sampling
[1] 10

$computime
[1] 18.83283
```

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 $proposal\_phi = 1$. This way of automatic determination of *dist_max*, *tab_normalization* and *proposal_range* is strongly recommended, compared to the crude automatic determination proposed in the method `Marjoram_original`.

```
> n=10

[1] 10

> n_calib=10

[1] 10

> tol_quant=0.2

[1] 0.2

> ABC_Marjoram<-ABC_mcmc(method="Marjoram", model=trait_model, prior=trait_prior,
+ summary_stat_target=sum_stat_obs,
+ n_rec=n, n_calibration=n_calib, tolerance_quantile=tol_quant, use_seed=TRUE)

$param
          [,1]      [,2]      [,3]     [,4]
 [1,] 3.442424 -1.209557  6.213494 3.014171
 [2,] 3.274600 -1.572532  9.170805 3.057381
 [3,] 3.511270 -1.032664  8.990505 2.819731
 [4,] 3.558531 -1.509734  5.059470 2.928636
 [5,] 3.182580 -1.360515 11.931425 2.716008
 [6,] 3.175917 -1.317759 16.864318 3.026080
 [7,] 3.412646 -1.275362 12.889961 2.879364
 [8,] 3.481502 -1.174408 12.556091 2.745612
 [9,] 3.181147 -1.479273  9.196864 3.064937
[10,] 3.020011 -2.099470 11.515074 2.727925

$stats
      [,1]     [,2]    [,3]     [,4]
 [1,]   71 3.285778 27.7368 24401.28
 [2,]   64 3.148762 19.2908 22632.50
 [3,]   67 3.141369 18.3856 19849.03
 [4,]   77 3.362678 21.1664 22601.58
 [5,]   57 2.931711 16.4910 24735.44
 [6,]   72 3.155310 25.1204 19218.44
 [7,]   74 2.984850 22.1110 14480.72
 [8,]   78 3.182465 22.8904 16945.87
 [9,]   66 3.189300 22.3966 33645.64
[10,]   61 2.951453 21.8548 28037.07

$dist
 [1] 2.972173 2.796552 2.804964 2.756005 2.750976 2.700694 2.624792 2.643574
 [9] 2.590242 2.256364

$stats_normalization
[1]    29.8596718     0.6306902    13.8140017 13868.3508997

$epsilon
[1] 2.972173

$nsim
[1] 101
```

```
$n_between_sampling
[1] 10

$computime
[1] 24.37261
```

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

> n_calib=10

[1] 10

> tol_quant=0.2

[1] 0.2

> ABC_Wegmann<-ABC_mcmc(method="Wegmann", model=trait_model, prior=trait_prior,
+ summary_stat_target=sum_stat_obs,
+ n_rec=n, n_calibration=n_calib, tolerance_quantile=tol_quant, use_seed=TRUE)

$param
          [,1]        [,2]       [,3]     [,4]
 [1,] 3.921204  0.81331156 -10.880017 3.033596
 [2,] 3.986454  0.49518954 -13.791213 2.979189
 [3,] 4.245415  0.50248137  -2.576517 2.885192
 [4,] 3.986349 -0.70777034  -2.252613 2.937485
 [5,] 3.828751 -0.43841092  -2.849862 3.003523
 [6,] 4.102651  0.02332906 -13.454664 2.827341
 [7,] 4.355581  0.12368403 -12.637506 2.615605
 [8,] 4.572630  1.40556236 -12.585316 2.546601
 [9,] 4.626577  0.84356696 -15.144880 2.471644
[10,] 4.471983  1.34451520  -8.925323 2.536161

$stats
      [,1]     [,2]    [,3]     [,4]
 [1,]   76 2.493128  6.6772 19123.95
 [2,]   76 2.220249 10.2498 29960.63
 [3,]   95 3.383096 10.5040 18247.19
 [4,]   99 3.416150 14.9762 23737.06
 [5,]   86 3.130847 13.7688 17717.01
 [6,]   92 2.683641 14.0050 36721.37
 [7,]   89 2.562782 12.1524 30007.04
 [8,]  103 3.036986  8.6100 24065.64
 [9,]  124 3.014251 18.7696 40800.38
[10,]   99 2.539588  9.2120 26439.87
```

```
$dist
 [1] 1.1232751 0.5065221 1.3255787 0.8378349 1.1404621 0.4300007 0.1610612
 [8] 0.5363059 1.0898174 0.2331846

$epsilon
[1] 1.325579

$nsim
[1] 101

$n_between_sampling
[1] 10

$min_stats
[1]     40.000000      1.278687      6.677200 -31870.984757

$max_stats
[1]   170.000000     4.782022    89.615800 21416.109998

$lambda
[1] -1.8181818  1.8181818  0.6060606  1.8181818

$geometric_mean
[1] 1.366608 1.487361 1.549825 1.492348

$boxcox_mean
[1] 0.5060675 0.4899755 0.6143174 0.4843456

$boxcox_sd
[1] 0.3325527 0.3655486 0.3096839 0.3273697

$pls_transform
           [,1]       [,2]       [,3]       [,4]
[1,] -0.5501260 -0.4874449 -0.4229914  0.5319001
[2,] -0.4649096 -0.5045054  0.5675415 -0.4794300
[3,]  0.3895548 -0.6838577 -0.4375262 -0.4976505
[4,]  0.5835209 -0.5759754  0.3729212  0.4343794

$n_component
[1] 4

$computime
[1] 17.52087
```

5.5 Performing a SABC scheme

For the SABC algorithm by Albert et al. (2014) we need to provide the prior in the form of a sampler and a density:

```
> r.prior <- function()   c(runif(1,3,5),runif(1,-2.3,1.6),runif(1,-25,125),runif(1,-0.7,3.2),1)

function()   c(runif(1,3,5),runif(1,-2.3,1.6),runif(1,-25,125),runif(1,-0.7,3.2),1)

> d.prior <- function(x)  dunif(x[1],3,5)*dunif(x[2],-2.3,1.6)*dunif(x[3],-25,125)*dunif(x[4],-0.
```

```
function(x)  dunif(x[1],3,5)*dunif(x[2],-2.3,1.6)*dunif(x[3],-25,125)*dunif(x[4],-0.7,3.2)
```

Furthermore, we need to specify the size of the ensemble, the number of simulations and the initial tolerance

```
> n.sample <- 300
[1] 300
> iter.max <- n.sample * 30
[1] 9000
> eps.init <- 20
[1] 20
```

Since, for this example, the prior is flat, we choose the method "noninformative":

```
> ABC_Albert <-SABC(r.model = trait_model,
+                   r.prior = r.prior,
+                   d.prior = d.prior,
+                   n.sample = n.sample,
+                   eps.init = eps.init,
+                   iter.max = iter.max,
+                   method = "noninformative",
+                   y = sum_stat_obs
+                   )
updates 9.277778 %          eps  0.2285382          u.mean 0.4281745
updates 12.61111 %          eps  0.212536          u.mean 0.4034268
updates 15.94444 %          eps  0.1974166          u.mean 0.3797819
updates 19.27778 %          eps  0.1854344          u.mean 0.3608474
updates 22.61111 %          eps  0.1799304          u.mean 0.3520872
updates 25.94444 %          eps  0.1736016          u.mean 0.3419626
updates 29.27778 %          eps  0.164586          u.mean 0.3274393
updates 32.61111 %          eps  0.1603159          u.mean 0.3205171
updates 35.94444 %          eps  0.1521838          u.mean 0.3072523
updates 39.27778 %          eps  0.1489183          u.mean 0.3018941
updates 42.61111 %          eps  0.1497981          u.mean 0.3033396
updates 45.94444 %          eps  0.1426213          u.mean 0.2915078
updates 49.27778 %          eps  0.1335824          u.mean 0.2764673
updates 52.61111 %          eps  0.1281264          u.mean 0.2673084
updates 55.94444 %          eps  0.1240465          u.mean 0.2604174
updates 59.27778 %          eps  0.1195864          u.mean 0.2528411
updates 62.61111 %          eps  0.1175052          u.mean 0.2492897
updates 65.94444 %          eps  0.11417          u.mean 0.2435763
updates 69.27778 %          eps  0.1101366          u.mean 0.2366293
updates 72.61111 %          eps  0.1030254          u.mean 0.2242741
updates 75.94444 %          eps  0.0961418          u.mean 0.2121745
updates 79.27778 %          eps  0.09283637          u.mean 0.2063115
updates 82.61111 %          eps  0.08752654          u.mean 0.1968162
updates 85.94444 %          eps  0.08371145          u.mean 0.1899315
updates 89.27778 %          eps  0.08051734          u.mean 0.1841246
updates 92.61111 %          eps  0.07670184          u.mean 0.177134
updates 95.94444 %          eps  0.07243722          u.mean 0.1692461
updates 99.27778 %          eps  0.0711558          u.mean 0.1668598
updates 102.6111 %          eps  0.0682838          u.mean 0.1614829
Resampling. Effective sampling size:  297.7532
```

5.6 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.

6 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.

7 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).

We thank Sylvie Huet, Albert Ko, Matteo Fasiolo and Wim Delva for their suggestions and inputs in the development of this version.

8 References

Albert C., Künsch HR., Scheidegger A. (2014) A Simulated Annealing Approach to Approximate Bayes Computations. *Stat. Comput.*, 1–16, arXiv:1208.2157 [stat.CO].

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.

Fearnhead, P. and Prangle, D. (2012) Constructing summary statistics for approximate Bayesian computation: semiautomatic approximate Bayesian computation. *J.Roy. Stat. Soc.: Series B* **74.3**, 419-474.

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