garma package - A package for generation, estimation and bootstrap inference of GARMA processes

Matheus de Vasconcellos Barroso

2018-05-12

Contents

1. Introduction

This is an introduction to the *garma* package. The package is designed to generate and estimate time series following the *GARMA* process of Benjamin et al. (2003), as wel as to perform moving block bootstrap resamples and diagnosis for this class of models. The package has four core *classes* and respective *methods*, all related to the evaluation of simulated *GARMA* process. The S4 classes are: `GarmaSpec`, `GarmaSim`, `GarmaFit` and `GarmaSimBoot`. The syntax is straightforward, in the firt class 'Spec' stands for **Specification** of a *GARMA* process, while 'Sim' for *Simulation* of the specified model in the second, 'Fit' for **estimating** in the third, and 'SimBoot' for a Moving Block Bootstrap evaluation of the simulated process. All classes have the built-in methods `print`, `summary` and `plot`, and their behaviour depends on some factors as the number of Monte Carlo simulations being specified.

How to install the package:

The **garma** package depends on the **dboot** package. So the first step in the installation process is to certify that you have the package **devtools** installed:

```r
if (!require("devtools")) install.packages("devtools")
```

Now the **dboot** and **garma** pkgs:

```r
if (!require("dboot")) install_github("matheusbarroso/dboot")
if (!require("garma"))
install_github("matheusbarroso/garma")
```

The next section gives a formal definition of the forementioned process followed by a section for the classes and methods explaining their usage and providing examples.

2. The *GARMA* model

Let Y be a stochastic process, i.e. a collection of random variables $Y = Y_t(\omega), t \in \mathcal{T}, \omega \in \Omega$, defined in the filtered probability space $(\Omega, \mathcal{F}, \mathcal{F}_{t\geq 0}, \mathcal{P})$, where Ω is the set of all possible states, $\mathcal{F}$ is the $\sigma - field$ of all subsets of Ω, $\mathcal{F}_{t\geq 0}$ is a filtration, $\mathcal{P}$ is a probability measure under $\mathcal{F}$ and $\mathcal{T}$ an arbitrary set. In the garma model we have that each realization of $Y_t, t = 1, ...n$, has conditional distribution belonging to the exponential family of distributions, where the conditioning is w.r.t. to $\mathcal{F}_{t-1} = \{\mathbf{x_1}, \ldots, \mathbf{x_{t-1}}; y_1, \ldots, y_{t-1}; \mu_1, \ldots, \mu_{t-1}\}$ such that the conditional density $f_{Y_t|\mathcal{F}_{t-1}}$ is of the form:

$$f_{Y_t|\mathcal{F}_{t-1}}(y_t|\mathcal{F}_{t-1}) = exp\left(\frac{y_t v_t - a(v_t)}{\varphi} + b(y_t, \varphi)\right) \tag{1}$$

where $a(\cdot)$ and $b(\cdot)$ are specific functions defining the particular member of the exponential familiy, with v_t as the canonical and φ as the scale parameter of the member of the family, $\mathbf{x}$ is a r dimensional vector of explanatory variables and μ is the mean vector. Additionaly, the predictor η is such that $\eta = g(\mu_t)$ where g is the link function.

$$\eta_t = \mathbf{x_t'}\boldsymbol{\beta} + \sum_{j=1}^{p} \phi_j \{g(y_{t-j}) - \mathbf{x_{t-j}'}\boldsymbol{\beta}\} + \sum_{j=1}^{q} \theta_j \{g(y_{t-j}) - \eta_{t-j}\} \tag{2}$$

where $\boldsymbol{\beta}' = (\beta_1, \beta_2, \ldots, \beta_r)$ is the vector of parameters of the linear predictor, $\boldsymbol{\phi}' = (\phi_1, \phi_2, \ldots, \phi_p)$ the vector of autoregressive parameters and $\boldsymbol{\theta}' = (\theta_1, \theta_2, \ldots, \theta_p)$ the vector of moving average parameters. Eq.1 and Eq.2 define the *GARMA* model.
In this package only the conditional *Gamma* and *Poisson* were implemented, though the application to all the exponential family is already in the pipeline. Thus, the next two subections deal with the specifics of the aforementioned models.

Poisson-GARMA model

If $Y_t|F_{t-1}$ follows a Poisson distribution with mean parameter μ_t then its p.m.f. is:

$$f_{Y_t|\mathcal{F}_{t-1}}(y_t|\mathcal{F}_{t-1}) = \frac{e^{-\mu_t}\mu_t^{y_t}}{y_t!} = exp\left(y_t log(\mu_t) - \mu_t - log(y_t!)\right) \tag{3}$$

Consequently, we have that $y_t|\mathcal{F}_{t-1}$ belongs to the exponential distribution, $v_t = log(\mu_t)$, $a(v_t) = e^{-\mu_t}$, $b(y_t, \varphi) = -log(y_t!)$ and $\varphi = 1$. The canonical *link function* is $log \equiv ln$. Hence η_t is such that:

$$\eta_t = log(\mu_t) = \mathbf{x_t'}\boldsymbol{\beta} + \sum_{j=1}^{p} \phi_j \{log(y^*_{t-j}) - \mathbf{x_{t-j}'}\boldsymbol{\beta}\} + \sum_{j=1}^{q} \theta_j \{g(y_{t-j}) - \eta_{t-j}\} \tag{4}$$

where $y^*_{t-j} = max(y_{t-j}, \alpha), 0 < \alpha < 1$. α is the offset parameter and its default value in the package is 0.1

***Gamma-GARMA* model**

If $Y_t|F_{t-1}$ follows a Gamma distribution with shape parameter δ and scale parameter γ (so a $\Gamma(\delta,\gamma)$ distribution) its p.d.f is given by:

$$f_{Y_t|\mathcal{F}_{t-1}}(y_t|\mathcal{F}_{t-1}) = \frac{y_t^{\delta-1}e^{-y_t/\delta}}{\Gamma(\delta)\gamma^{\delta}} \tag{5}$$

with $E_{Y_t|\mathcal{F}_{t-1}}(y_t|\mathcal{F}_{t-1}) = \delta\gamma$ and $Var_{Y_t|\mathcal{F}_{t-1}}(y_t|\mathcal{F}_{t-1}) = \delta\gamma^2$. Howoever, here we adopt a more useful parametrization of the gamma density given by this suitable transformation: $\delta = 1/\sigma^2$ and $\gamma = \sigma^2\mu_t$. With these new parameters at hand we have that $E_{Y_t|\mathcal{F}_{t-1}}(y_t|\mathcal{F}_{t-1}) = \mu_t$ and $Var_{Y_t|\mathcal{F}_{t-1}}(y_t|\mathcal{F}_{t-1}) = \sigma^2\mu_t^2$. In the *Gamma* case the canonical *link function* is the reciprocal function, though, we make usage of the *logarithmic* function and thus the equation for η_t is the same of that for the *Poisson* case in Eq. 4.

3. The garma package

After a formal definition of the *GARMA* model we are now able to understand better each class and its respective methods in the **garma** package. This section contains four subsections, one for each main class in the package.

3.1 GarmaSpec

The GarmaSpec class is the more general class for specifying the *GARMA* model and contains the slots that are common to all members of the exponential family. For those unfamiliar with S4 classes, slots might be seen as the parameters that specify the class, and are defined as follows: * `family`: A character vector specifying the family of the specification object. Accepted values are: “Po” for poisson and “GA”for gamma families.

- `beta.x`:A numeric vector with length of the desired specification. In the current version the intercept term must be included. The default value is `beta.x = 1L`.
- `phi`: A numeric vector with length of the desired autoregressive term order specification.The default behaviour is `phi = 0L`.
- `theta`: A numeric vector with length of the desired moving-average term order specification. The default behaviour is `theta = 0L`.
- `X`: A $n \times m$ matrix, where `n = nsteps + burnin + max.order` and `m = length(beta.x)`. Here, **nsteps** is the number of simulations desired, **burnin** the number of discarded observations in the begining of the process and **max.order** is equal to the highest order of the GARMA model (i.e. `max(length(phi),length(theta)`.)

This class has two child classes: `PoissonSpec` and `GammaSpec`. Each one specifies the additional slots required for a correct specification of a *Poisson* or *Gamma GARMA* models.

PoissonSpec

Inherited from `GarmaSpec`, it has the slots:

- `family`: A character with the tag “PO”.
- `alpha`: Numeric, specifying the **offset** (α) parameter. The default value is $\alpha = 0.01$
- `mu0`: A numeric vector with length equal to the **max.order** of the Poisson-GARMA model. The default value is $\mu_0 = 10$.

GammaSpec

Inherited from the `GarmaSpec`, it has the slots:

- `family`: A character with the tag "GA".
- `sigma2`: Numeric, specifying the σ^2 parameter. The default value is $\sigma^2 = 1$
- `mu0`: A numeric vector with length equal to the **max.order** of the Poisson-GARMA model. The default value is $\mu_0 = 10$.

Methods

To avoid problems in the model specification the easiest way to create an object of the GarmaSpec(Gamma/Poisson) is through the constructor function (method) `GarmaSpec`. At least the `family` argument must be provided in order to create an object of this class. If no further arguments are passed the default behaviour is used.

If we wish to specify a second order poisson model we would need at least:

```
GarmaSpec("PO",phi=c(0.3,0.2),mu0=c(1,1))
```

Now we are ready to move to some examples First of all, it is required to load the garma package: `library(garma)`

1. A GA-GARMA(1,0) with $\phi = 0.5, \sigma^2 = 2, \beta = 1$ and $\boldsymbol{x'_1} = 1$ note that the length of x1 is 101, as we will simulate 100 steps, and the other 1 is for the initial value of Phi. The initial value for μ_0 is the default.

```
spec1 <- GarmaSpec("GA",
phi = 0.5,
beta.x = 1,
sigma2 =2,
X = as.matrix(
 data.frame(
   x1 = rep(1,101))))
```

2. A PO-GARMA(1,3), where $\phi = 0.2, \boldsymbol{\theta} = \{\theta_1, \theta_2, \theta_3\} = \{0.1, 0.3, 0.5\}, \mu_0 = \{1, 2, 3\}$ and $\boldsymbol{x'} = 0$ here the default behaviour of μ_0 will not suffice, so we need three initial values. The intercept will be generated automatically

```
spec2 <- GarmaSpec("PO",
phi = 0.2,
theta = c(0.1, 0.3, 0.5),
mu0 = 1:3)
```

3. A PO-GARMA(2,0) with an intercept term and one covariate. Here, $\boldsymbol{\phi} = \{\phi_1, \phi_2\} = \{0.5, 0.15\}, \boldsymbol{\beta} = \{\beta_{intercept}, \beta_1\} = \{1, 1\}$ and $\mu_0 = \{2, 2\}$.A series of length 100 will be generated, though, 1000 burnin observations will be deleted and the covariates must take into consideration these obs and the order of the process, so that the nrow of X should be 100 (desired length)+1000 (burnin)+2 (order of the process)

```
spec3 <- GarmaSpec("PO",
phi = c(0.5,0.15),
beta.x = c(1,1),
mu0 =  c(2,2),
X = as.matrix(
 data.frame(
 intercept = rep(1,1102),
   x1 = c(rep(7,100),
       rep(2,1002)))))
```

4. A PO-GARMA(0,0), with an intercept and covariate $\boldsymbol{\beta} = \{\beta_{intercept}, \beta_1\} = \{1, 0.1\}$:

```
spec4 <- GarmaSpec("PO",
beta.x = c(1,0.1),
X = as.matrix(
 data.frame(
 intercept = rep(1,1100),
   x1 = c(rep(7,100),
       rep(2,1000)))))
```

3.2 GarmaSim

The `GarmaSim` class is used to simulate an object of the class `GarmaSpec`. The methods `GarmaSim`, `print`, `plot` and `summary` are implemented. The `GarmaSim` arguments are:

- `spec`: An object of the GarmaSpec class, as provided by the method `GarmaSpec`
- `nmonte`: A positive integer, specifying the number of Monte Carlo simulations to be performed. The default value is **1000**.
- `nsteps`: A numeric vector with the number of steps in the Garma model simulation, that is, the length of the time series to be simulated. The default value is **100**.
- `burnin`: A numeric vector indicating the number of discarded observations. If you want to generate only `nsteps` this argument should be set equal to zero. Otherwise, provide a positive integer. The default value is **1000**.
- `allow.parallel`: Logical `TRUE/FALSE` indicating whether parallel computation via the *foreach* package should be used. The default value is `TRUE`. OBS:paralllel backend must be registered prior to calling GarmaSim.
- `seed`: Numeric, the seed to `set.seed()` for replicable examples. Default value is *123*. Before moving to the examples it is useful to register the parallel back end:

```
library(doParallel)
no_cores <- detectCores() - 1
registerDoParallel(no_cores)
```

Now we can move to some examples:

1. Using the specifications (models) built in the session Garma Spec:

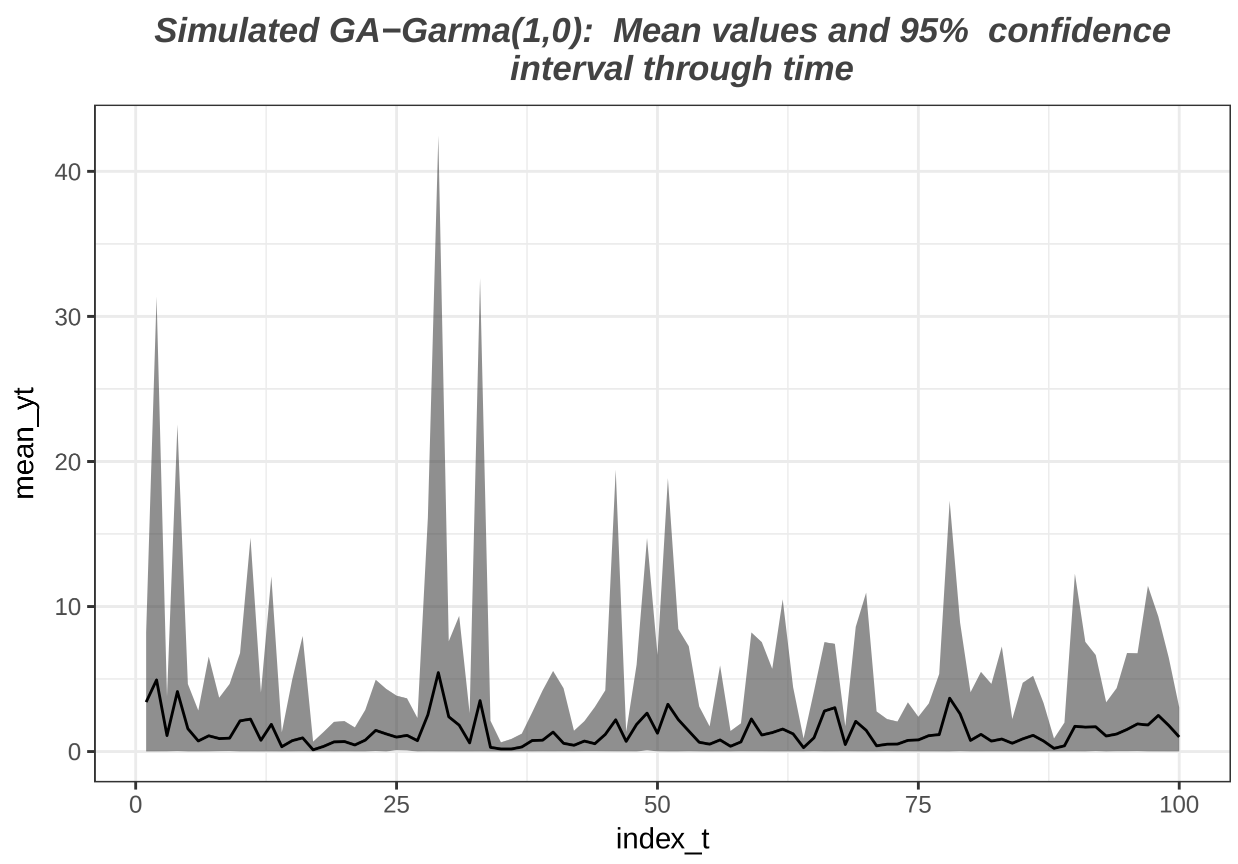
```
sim1 <- GarmaSim(spec1,
                 nmonte = 10, #10 monte carlo simulations
                 burnin = 0) # no burnin
```

The print method give us a tidy output in the monte carlo setting, with a brief descritpion of the model and each generated series in the columns.

```
print(sim1)
```

When `nmonte` > 1, the plot comes with th 95% empirical confidence interval for the mean of each observation. The confidence level can be changed through the `confInt` parameter (`plot(sim1,confInt = .80)`):

```
plot(sim1)
```

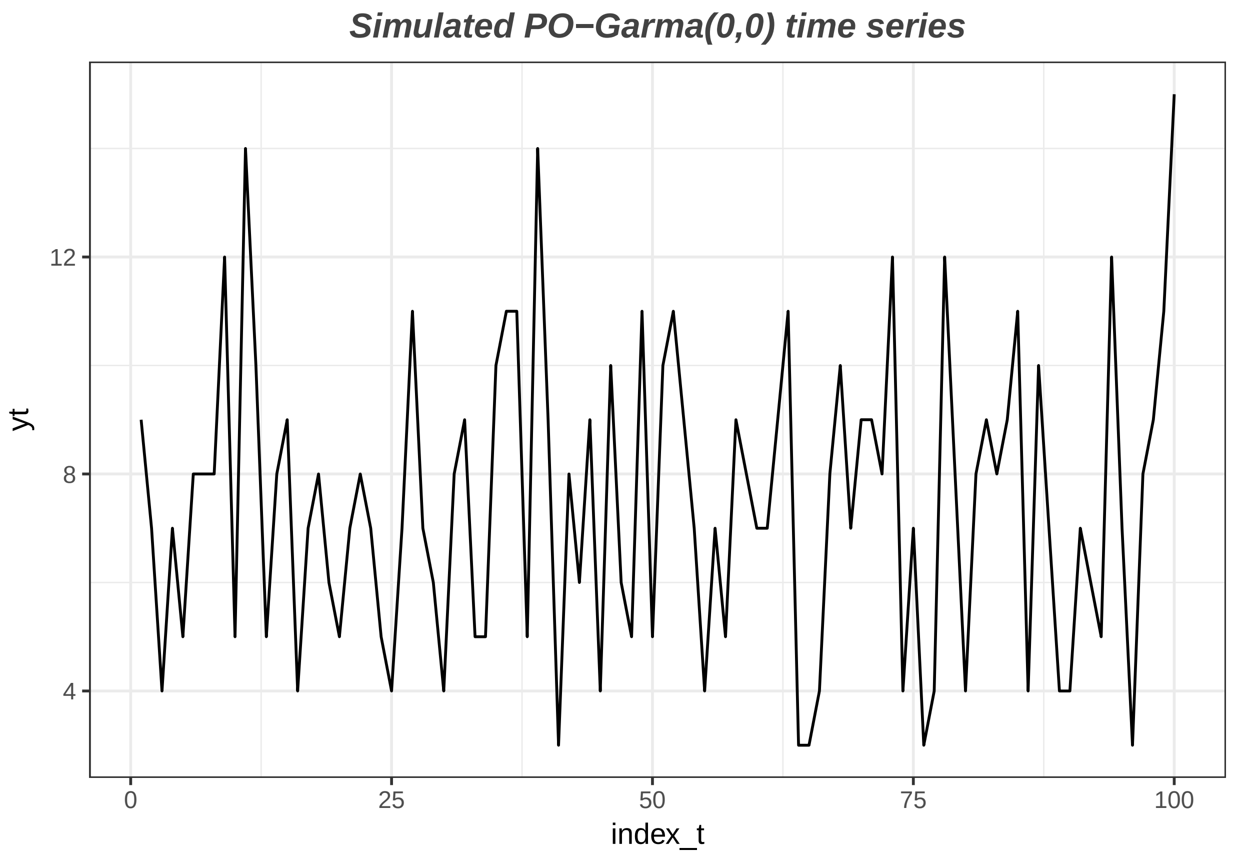


The summary(`GarmaSim`) method has a different behaviour for `nmonte > 1` and `nmonte = 1`

```
summary(sim1)
#>  -------------------------------------------------------
#>  A GA-Garma(1,0)  simulation object:
#>
#>  Number of Monte Carlo Simulations ('nmonte') =  10
#>  Time Series Length ('nsteps') =  100
#>  Burn-in ('burnin') =  0
#>  parallel =  TRUE
#>  -------------------------------------------------------
#> Mean Monte Carlo mean values =  1.3351 with distribution:
#>     2.5%    50%  97.5%
#> 2 0.7308 1.0938 2.3591
#>
#> Mean Monte Carlo min values =  1e-04 with distribution:
#>   2.5% 50% 97.5%
#> 1    0   0 2e-04
#>
#> Mean Monte Carlo max values =  18.485 with distribution:
#>     2.5%     50%   97.5%
#> 3 6.5795 11.4202 42.4766
#>
#>         mean   2.5%     50%   97.5%
#> min   0.0001 0.0000  0.0000  0.0002
#> mean  1.3351 0.7308  1.0938  2.3591
#> max  18.4850 6.5795 11.4202 42.4766
```

2. An **Poisson-GARMA(0,0)**, that is, $\phi = 0, \theta = 0$ and $\boldsymbol{\beta} = \{\beta_{intercept}, \beta_1\} = \{1, 1\}$, with only one Monte Carlo simulation:

```
sim2 <-
  GarmaSim(
    GarmaSpec("PO",
              beta.x = c(0.1,1),
              X = as.matrix(
              data.frame(
                intercept = rep(1,1100),
                x1 = c(rep(7,100),rep(2,1000))))),
  nmonte = 1,
  allow.parallel = TRUE)
print(sim2)
#>  -------------------------------------------------------
#>  A PO-Garma(0,0) simulation object:
#>
#>  beta.intercept =  0.1
#>  beta.x1 =  1
#>  mu0[1] =  10
#>  Number of Monte Carlo Simulations ('nmonte') =  1
#>  Time Series Length ('nsteps') =  100
#>  Burn-in ('burnin') =  1000
#>  parallel =  TRUE
#>  -------------------------------------------------------
#>
#>   [1]  9  7  4  7  5  8  8  8 12  5 14 10  5  8  9  4  7  8  6  5  7  8  7
#>  [24]  5  4  7 11  7  6  4  8  9  5  5 10 11 11  5 14  9  3  8  6  9  4 10
#>  [47]  6  5 11  5 10 11  9  7  4  7  5  9  8  7  7  9 11  3  3  4  8 10  7
#>  [70]  9  9  8 12  4  7  3  4 12  8  4  8  9  8  9 11  4 10  7  4  4  7  6
#>  [93]  5 12  7  3  8  9 11 15
plot(sim2)
```



```
summary(sim2)
#>  -------------------------------------------------------
#>  A PO-Garma(0,0)  simulation object:
#>
#>  Number of Monte Carlo Simulations ('nmonte') =  1
#>  Time Series Length ('nsteps') =  100
#>  Burn-in ('burnin') =  1000
#>  parallel =  TRUE
#>  -------------------------------------------------------
#>    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
#>    3.00    5.00    7.00    7.45    9.00   15.00
```

Further specification examples can be found at the `GarmaSim-class` help menu.

3.3 GarmaFit

Given a `GarmaSim` object we can simultaneously estimate all the `nmonte` series (if `parallel = TRUE` and the parallel back-end registered). The methods `GarmaFit`, `print`, `plot` and `summary` are implemented. The `GarmaSim` method has the following parameters:

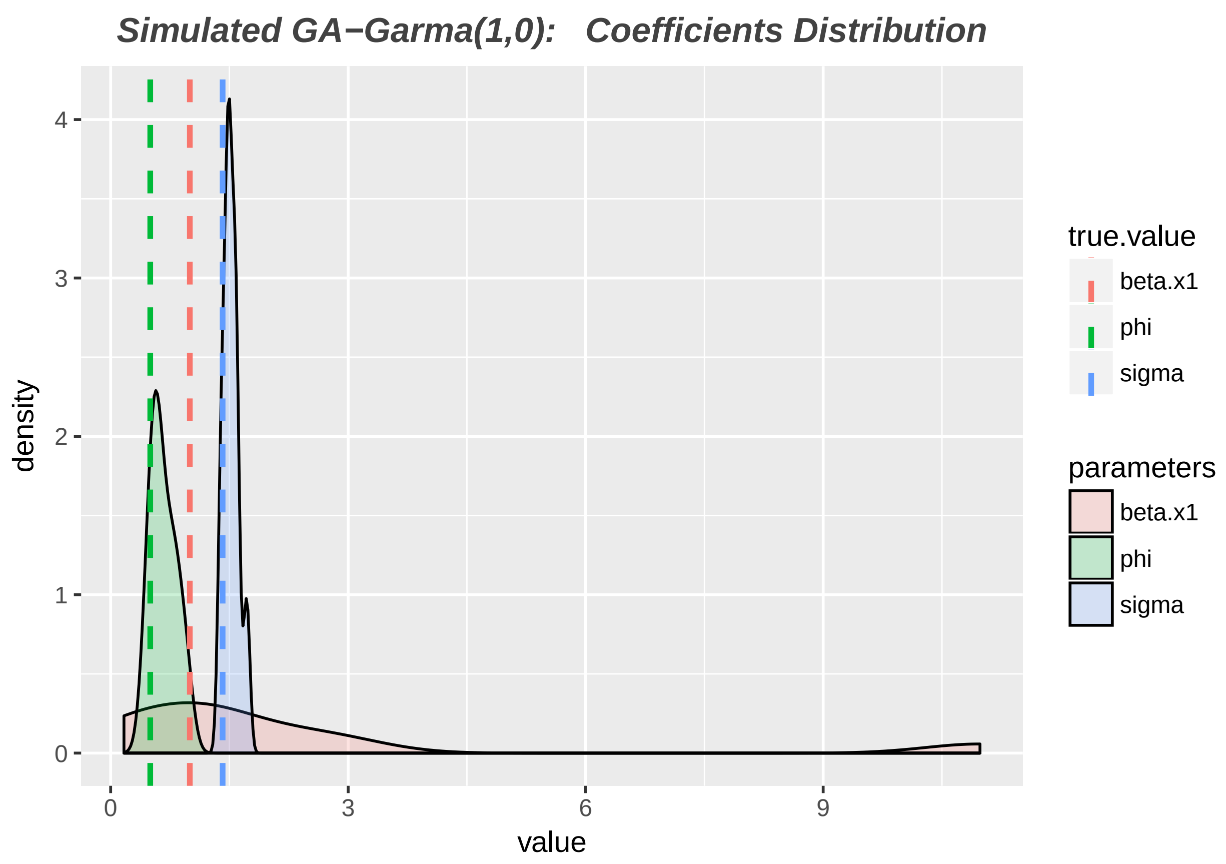
- `garma`: An object of the **GarmaSim** class, as provided by `GarmaSim`.
- `allow.parallel`: Logical `TRUE/FALSE` indicating whether parallel computation via the foreach package should be used. The default value is `TRUE`. OBS:paralllel backend must be registered prior to calling GarmaSim.
- `seed`: Numeric, the seed to `set.seed()` for replicable examples. Default value is 123.

- `errorhandling`: Character, either 'try' or 'pass'
- `n.try`: Positive integer. If `errorhandling = 'try'`, this specifies the number of iterations in the algorithm.
- `control`: List. This is passed to the garmaFit2 function. The options are given by `garmaFit` function in the **gamlss.util** package.

The examples:

1.

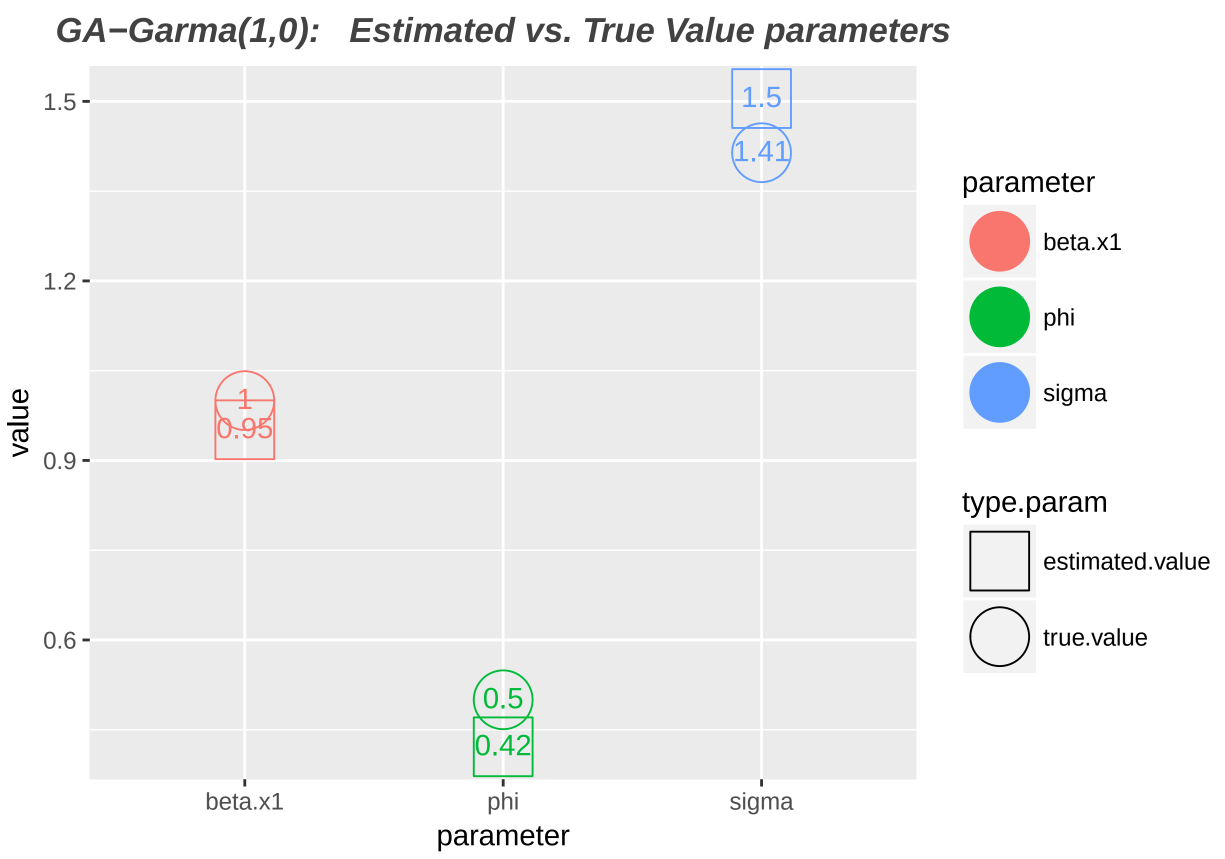
```r
fit1 <- GarmaFit(sim1)
print(fit1)
#>  -------------------------------------------------------
#>  A GA-Garma(1,0) simulation fitted object:
#>
#>  phi 0.5
#>  beta.x1 =  1
#>  mu0[1] =  10
#>  sigma2 =  2
#>  Number of Monte Carlo Simulations ('nmonte') =  10
#>  Time Series Length ('nsteps') =  100
#>  Burn-in ('burnin') =  0
#>  parallel =  TRUE
#>  errorhandling =  try
#>  n.try =  5
#>  -------------------------------------------------------
#>
#>  Estimated parameters:
#>              [,1]      [,2]      [,3]      [,4]      [,5]      [,6]
#> beta.x1 0.2836789 2.8930311 1.1386888 1.3663394 2.4575526 1.1995227
#> phi     0.4589851 0.7847901 0.5618464 0.7369287 0.8544735 0.6093733
#> sigma   1.5987879 1.4810814 1.4072223 1.4776656 1.5128949 1.3946111
#>               [,7]      [,8]      [,9]     [,10]
#> beta.x1 10.9807448 0.3657657 1.4026019 0.1680009
#> phi      0.9568498 0.5452537 0.6356124 0.5221223
#> sigma    1.7181861 1.4815231 1.5731927 1.5651013
plot(fit1)
```



```
summary(fit1)
#>  --------------------------------------------------------
#>  A GA-Garma(1,0)  simulation object:
#>
#>  Number of Monte Carlo Simulations ('nmonte') =  10
#>  Time Series Length ('nsteps') =  100
#>  Burn-in ('burnin') =  0
#>  parallel =  TRUE
#>  --------------------------------------------------------
#>   parameters      Min.   1st Qu.    Median.      Mean.   3rd Qu.       Max.
#> 1    beta.x1 0.1680009 0.3657657 1.1995227 2.2255927 2.4575526 10.9807448
#> 2        phi 0.4589851 0.5452537 0.6093733 0.6666235 0.7847901  0.9568498
#> 3      sigma 1.3946111 1.4776656 1.4815231 1.5210266 1.5731927  1.7181861
#>   true.value
#> 1   1.000000
#> 2   0.500000
#> 3   1.414214
```

2. A GA-GARMA(1,0) with $\phi = 0.5, \sigma^2 = 2, \beta = 1$ and $\boldsymbol{x}_1' = 1$. An example with `nmonte =  1`:

```
sim2 <-  GarmaSim(
  GarmaSpec("GA",
            phi = 0.5,
            beta.x = 1,
            sigma2 =2,
            X = as.matrix(
              data.frame(
                x1 = rep(10,101)))),
  nmonte = 1,
  burnin = 0)
fit2 <- GarmaFit(sim2)
print(fit2)
#>  -------------------------------------------------------
#>  A GA-Garma(1,0) simulation fitted object:
#>
#>  phi 0.5
#>  beta.x1 =  1
#>  mu0[1] =  10
#>  sigma2 =  2
#>  Number of Monte Carlo Simulations ('nmonte') =  1
#>  Time Series Length ('nsteps') =  100
#>  Burn-in ('burnin') =  0
#>  parallel =  TRUE
#>  errorhandling =  try
#>  n.try =  5
#>  -------------------------------------------------------
#>
#>  Estimated parameters:
#>             [,1]
#> beta.x1 0.9512578
#> phi     0.4214639
#> sigma   1.5047123
plot(fit2)
```



```
summary(fit2)
#>  ------------------------------------------------------
#>  A GA-Garma(1,0)  simulation object:
#>
#>  Number of Monte Carlo Simulations ('nmonte') =  1
#>  Time Series Length ('nsteps') =  100
#>  Burn-in ('burnin') =  0
#>  parallel =  TRUE
#>  ------------------------------------------------------
#>   parameters true.value  estimate
#> 1    beta.x1   1.000000 0.9512578
#> 2        phi   0.500000 0.4214639
#> 3      sigma   1.414214 1.5047123
```

3.4 GarmaSimBoot

Given a `GarmaSim` object we can simultaneously perform the Moving Block Bootstrap for all the `nmonte` series (if `parallel = TRUE` and the parallel back-end registered). The methods `GarmaSimBoot`, `print`, `plot` and `summary` are implemented. The `GarmaSimBoot` method has the following arguments:

- `sim`: An object of the **GarmaSim** class, as provided by `GarmaSim`.
- `l`: the fixed block length used in generating the replicated time series.
- `R`: A positive integer giving the number of bootstrap replicates required.
- `allow.parallel`: Logical `TRUE/FALSE` indicating whether parallel computation via the foreach package

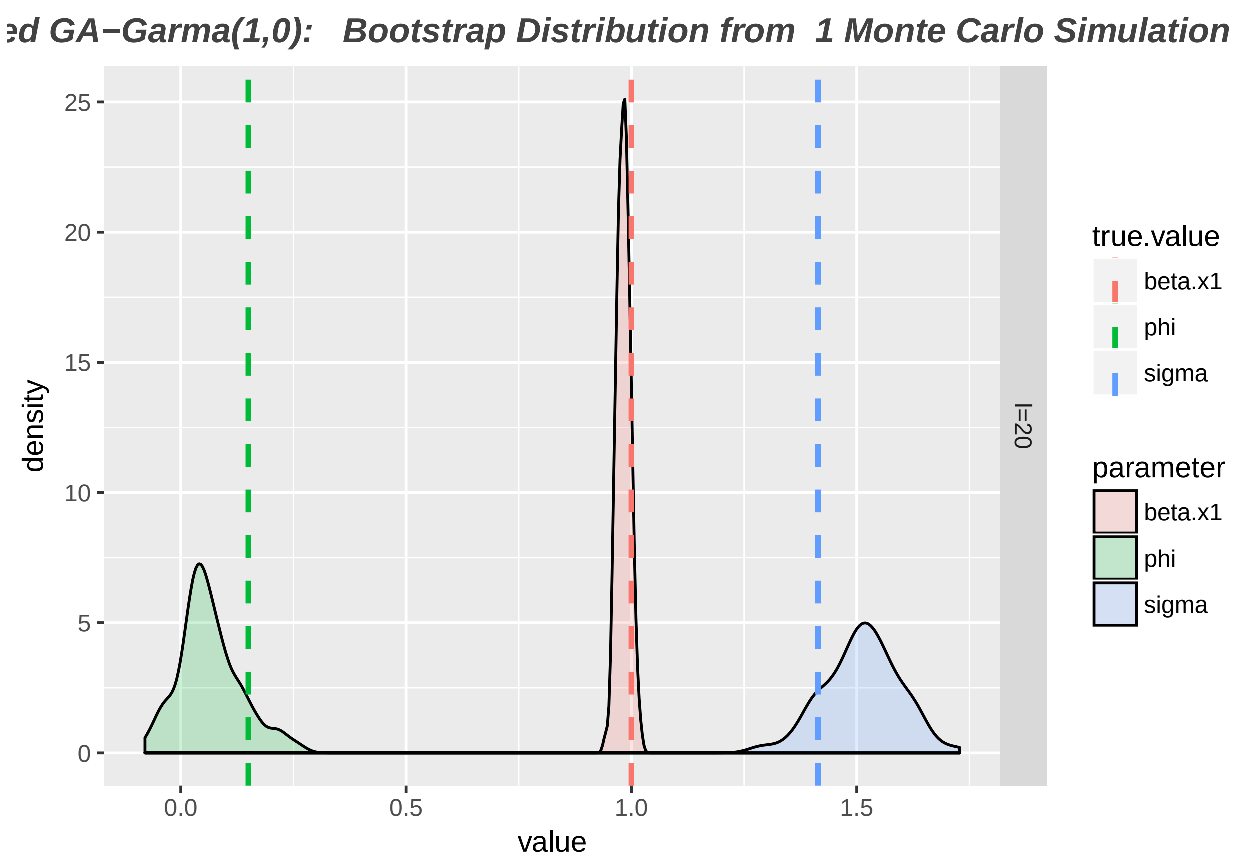
should be used. The default value is `TRUE`. OBS:paralllel backend must be registered prior to calling GarmaSim.

- `seed`: Numeric, the seed to `set.seed()` for replicable examples. Default value is 123.
- `errorhandling`: Character, either 'try' or 'pass'
- `n.try`: Positive integer. If `errorhandling = 'try'`, this specifies the number of attempts in the algorithm.
- `boot.function`: A function to summarise the bootstrap replicates. The default function returns 0. Be aware that this is not a problem, as by default the mean value is already being returned. This is useful if the user wants to specify a quantity not being reported (for example consider the 0.2 quantile). *Warning*: this feature is only enabled if `nmonte > 1`.
- `control`: List. This is passed to the garmaFit2 function. The options are given by `garmaFit` function in the **gamlss.util** package.

The examples:

1. A GA-GARMA(1,0) with $\phi = 0.15, \sigma^2 = 2, \beta = 1$ and $\boldsymbol{x}_1' = 1$ with `nmonte = 1` and only one block length for the MBB:

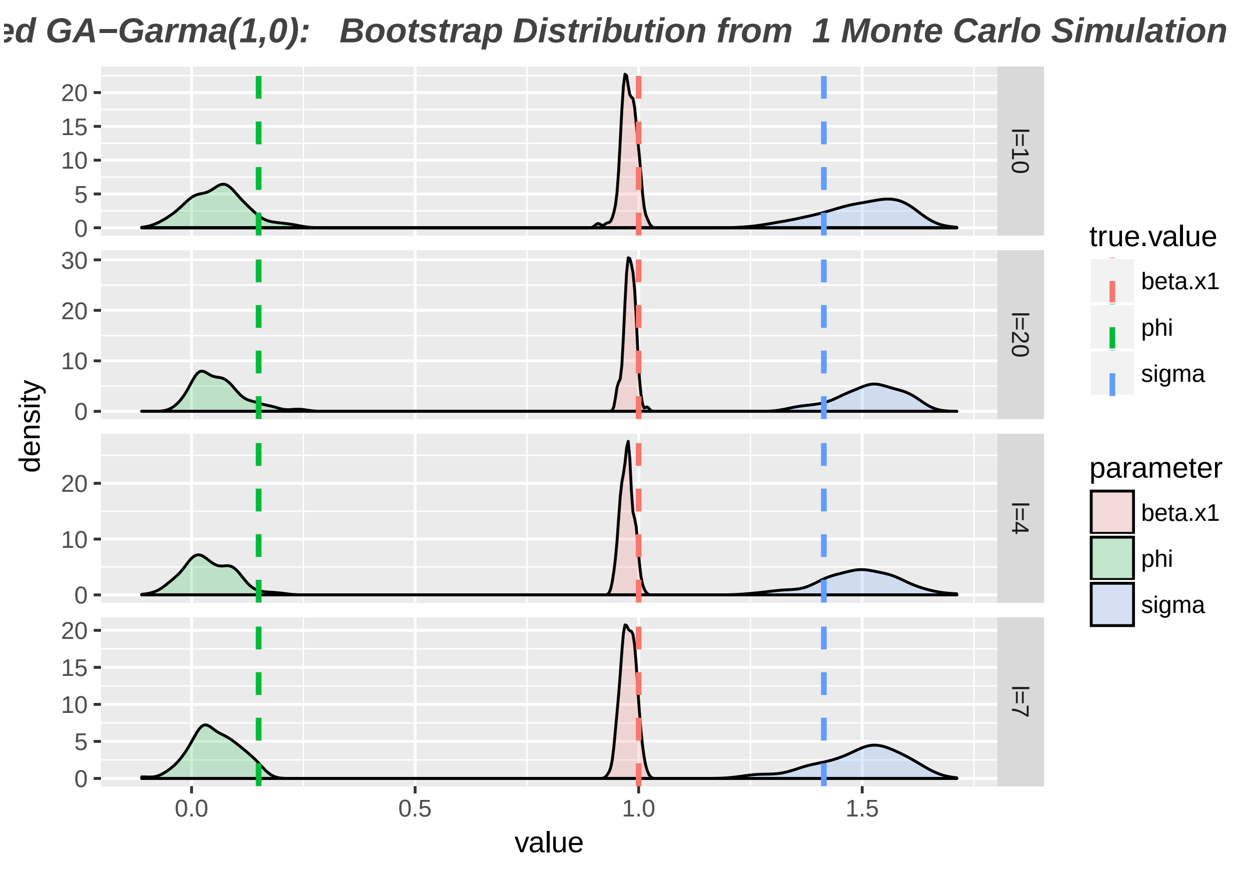
```
# specification and simulation:
Sim1 <- GarmaSim(
  GarmaSpec("GA",
            phi = 0.15,
            beta.x = 1,
            sigma2 =2,
            X = as.matrix(
              data.frame(
                x1 = rep(10,101)))),
  nmonte = 1,
  burnin = 0)
#MBB

mbb1 <- GarmaSimBoot(Sim1,l = 20)
#> deviance of linear model=  2074.691
#> deviance of  garma model=  2052.728
#print(mbb1)
plot(mbb1)
```



```
summary(mbb1)
#>  ------------------------------------------------------
#>  A GA-Garma(1,0)  bootstrap object:
#>
#>  Number of Monte Carlo Simulations ('nmonte') =  1
#>  Time Series Length ('nsteps') =  100
#>  Burn-in ('burnin') =  0
#>  parallel =  TRUE
#>  errorhandling =  try
#>  n.try =  5
#>  block length =  20
#>  R =  100
#>  ------------------------------------------------------
#>
#>  Bootstrap Statistic Summary:
#> $`True values`
#>   parameter true.value
#> 1   beta.x1   1.000000
#> 2       phi   0.150000
#> 3     sigma   1.414214
#>
#> $Estimates
#>   length parameter variable  Mean Est.
#> 1   l=20   beta.x1 original 0.98253643
#> 2   l=20       phi original 0.06443289
#> 3   l=20     sigma original 1.51186002
```

2. `nmonte = 10`, multiple block lengths

```
mbb2 <- GarmaSimBoot(Sim1,
                     l = c(4,7,10,20))
#> deviance of linear model=  2074.691
#> deviance of  garma model=  2052.728
#> deviance of linear model=  2074.691
#> deviance of  garma model=  2052.728
#> deviance of linear model=  2074.691
#> deviance of  garma model=  2052.728
#> deviance of linear model=  2074.691
#> deviance of  garma model=  2052.728

#print(mbb2)
plot(mbb2)
```



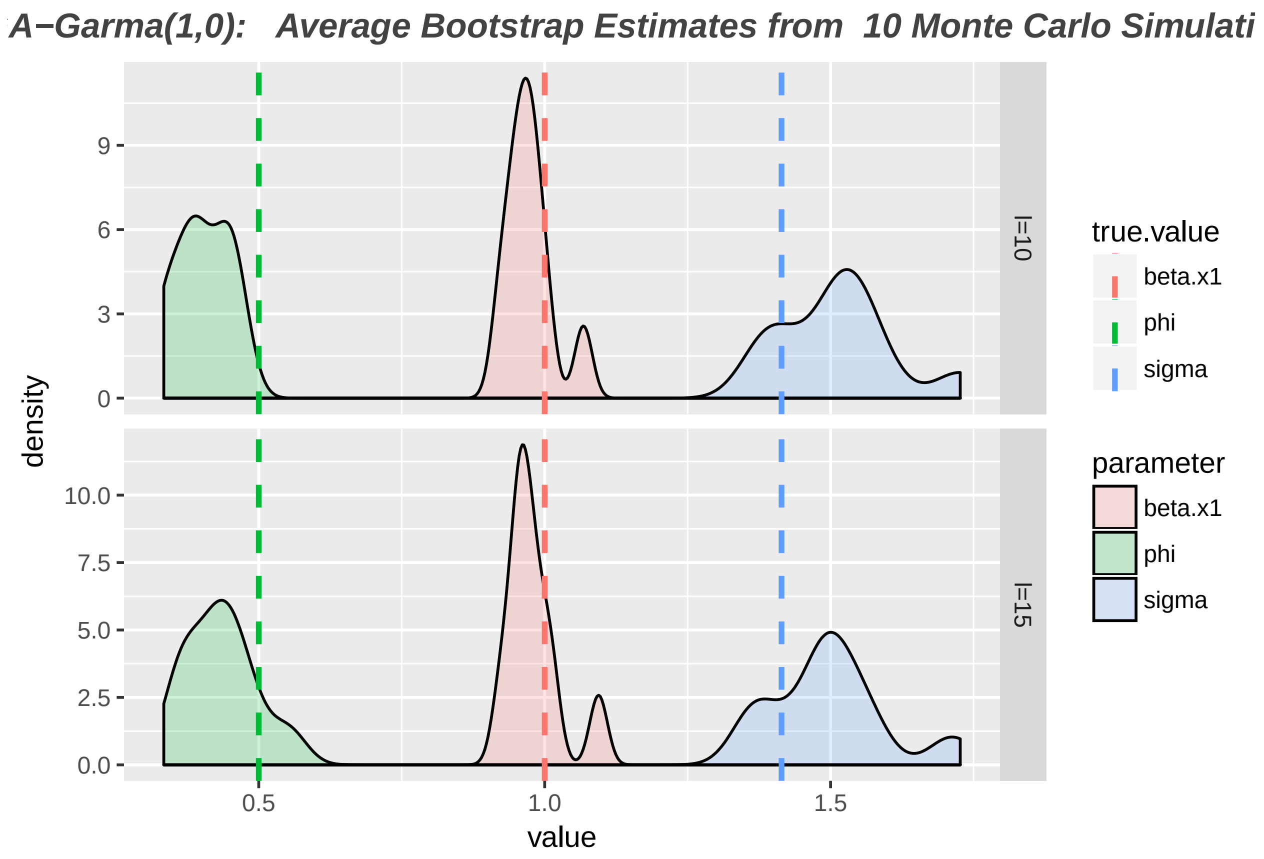
```
summary(mbb2)
#>  -------------------------------------------------------
#>  A GA-Garma(1,0)  bootstrap object:
#>
#>  Number of Monte Carlo Simulations ('nmonte') =  1
#>  Time Series Length ('nsteps') =  100
#>  Burn-in ('burnin') =  0
#>  parallel =  TRUE
#>  errorhandling =  try
#>  n.try =  5
#>  block length =  4 7 10 20
#>  R =  100
#>  -------------------------------------------------------
#>
#>  Bootstrap Statistic Summary:
#> $`True values`
#>   parameter true.value
#> 1   beta.x1   1.000000
#> 2       phi   0.150000
#> 3     sigma   1.414214
#>
#> $Estimates
#>    length parameter variable  Mean Est.
#> 10    l=7   beta.x1 original 0.97685236
#> 11    l=7       phi original 0.05191608
#> 12    l=7     sigma original 1.49685315
#> 7     l=4   beta.x1 original 0.97398193
#> 8     l=4       phi original 0.03845332
#> 9     l=4     sigma original 1.49204184
#> 4    l=20   beta.x1 original 0.98011355
#> 5    l=20       phi original 0.05894198
#> 6    l=20     sigma original 1.51715134
#> 1    l=10   beta.x1 original 0.97779395
#> 2    l=10       phi original 0.05789009
#> 3    l=10     sigma original 1.50199295
```

3. `nmonte = 10`, multiple block lengths + a user defined function to apply in the bootstrap resamples (the 0.1 and 0.9 quantiles)

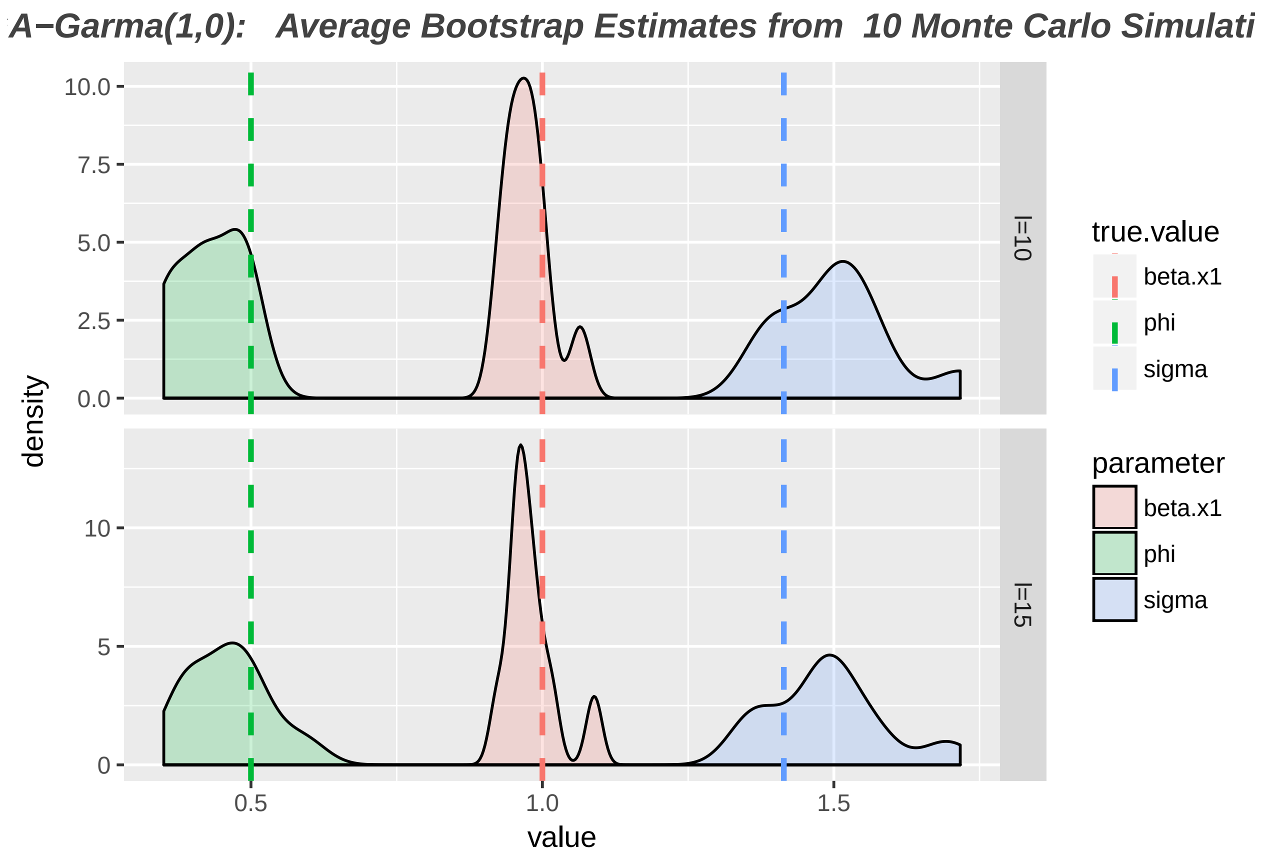
```
Sim2 <-  GarmaSim(
  GarmaSpec("GA",
            phi = 0.5,
            beta.x = 1,
            sigma2 =2,
            X = as.matrix(
              data.frame(
                x1 = rep(10,101)))),
  nmonte = 10,
  burnin = 0)
```

```
mbb3 <- GarmaSimBoot(Sim2,
                     l = c(10,15),
                     boot.function =
                       function(x)
                         quantile(x,
                                  probs = c(0.1,0.9)))
```

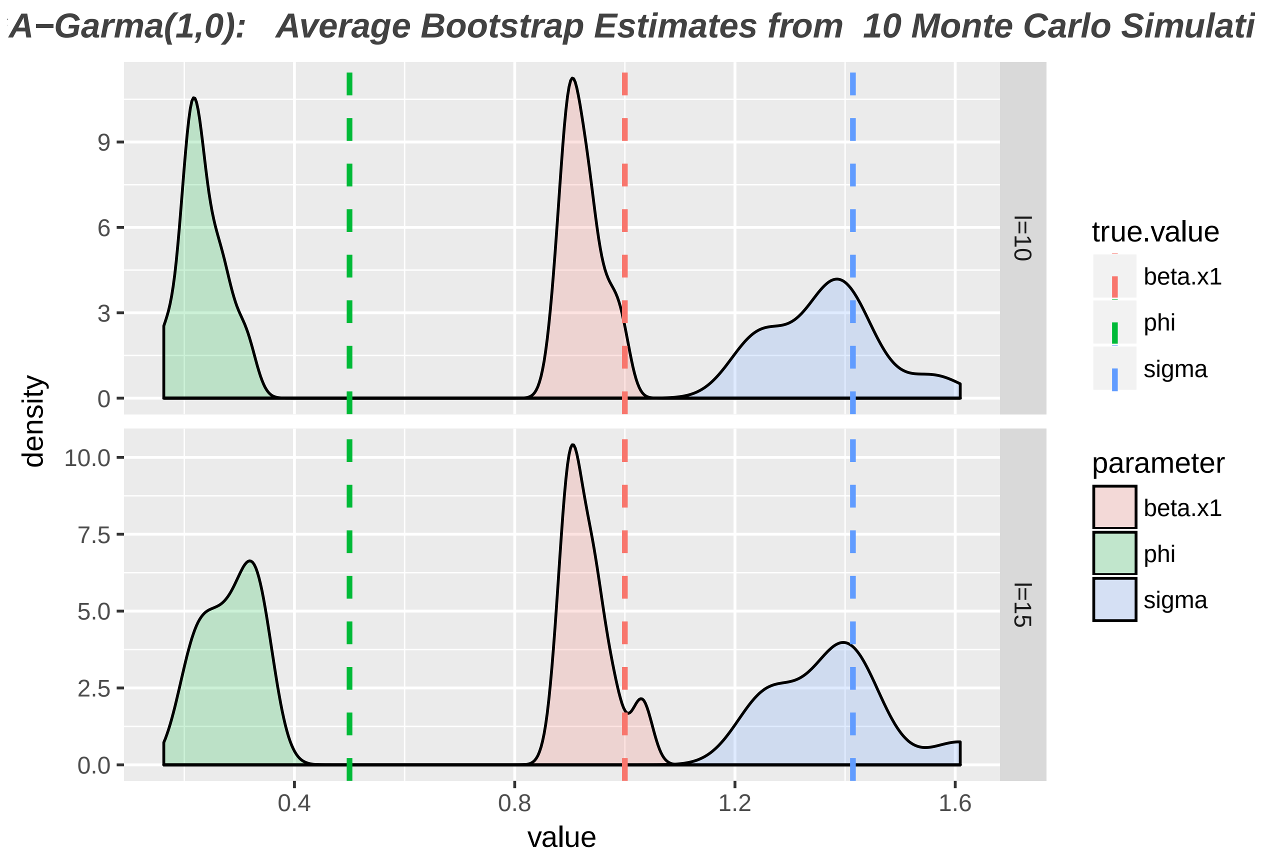
```
#print(mbb3)
plot(mbb3)
```



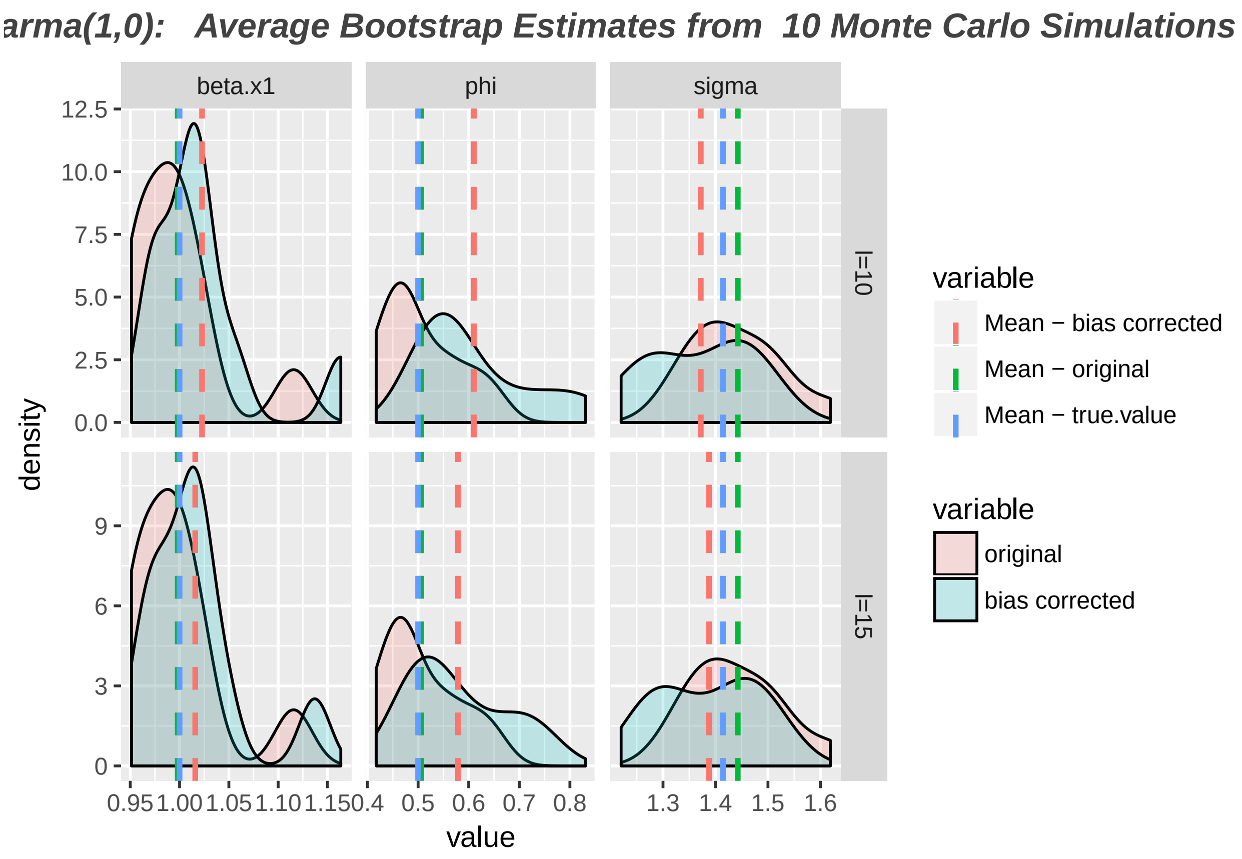
```r
plot(mbb3, variable = "Median.")
```



```r
plot(mbb3, variable = "boot.func.10%")
```



```r
#plot(mbb3, variable = "test") # if you type a variable that #does not exists you can see all available
plot(mbb3, type = "original-bias", scales = 'free')
```



```r
summary(mbb3)
```

3.5 ConfidenceInterval

Provided a MBB object, another useful thing to do is to examine the behaviour of the simulated confidence intervals. This task is easy, with built-in functions handling it and also the task of computing coverage rates and default plots.

Given a `GarmaSimBoot` object we can simultaneously perform confidence inteval computations for all the `nmonte` series (if `parallel = TRUE` and the parallel back-end registered). The methods `ConfidenceInterval`, `print`, `plot`, `summary` and `coverage` are implemented. The `ConfidenceInterval` method has the following parameters:

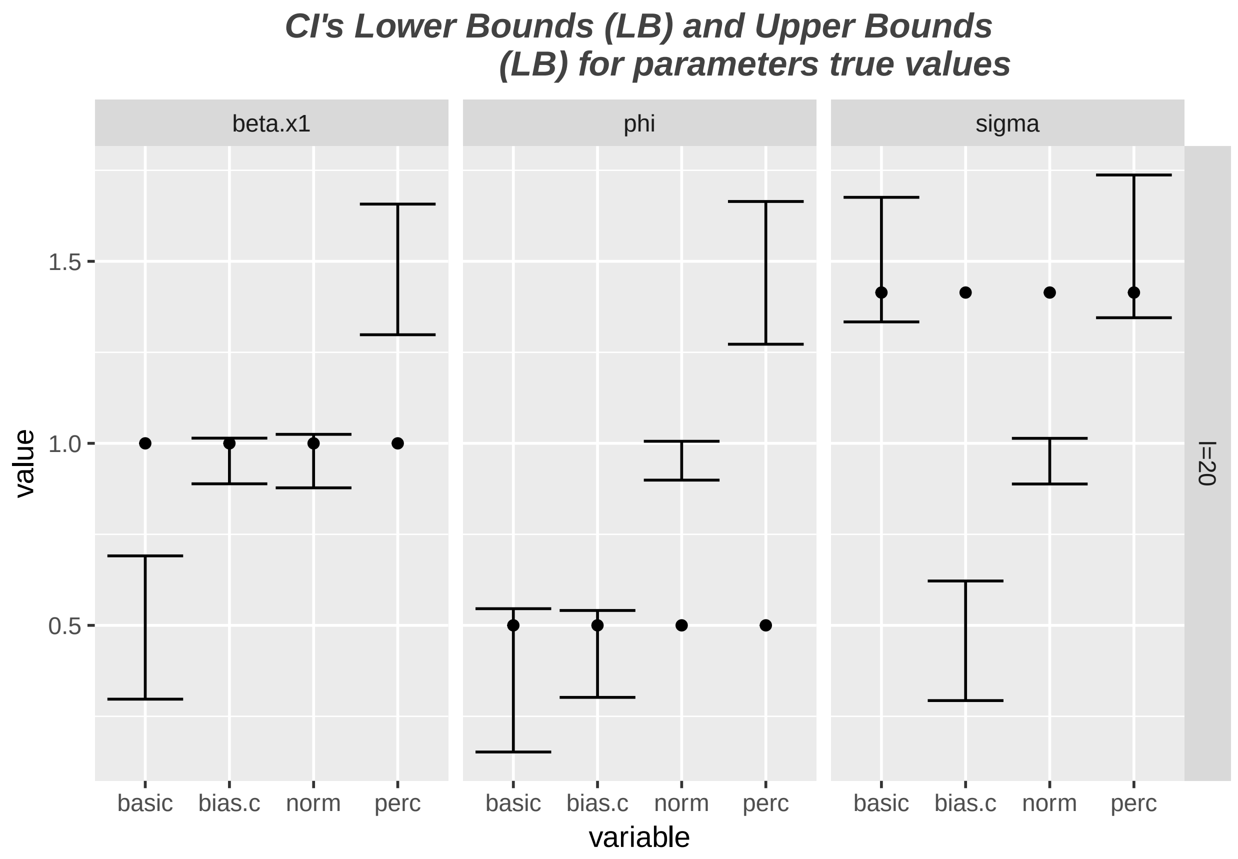
- `garma.boot`: An object of the **GarmaSimBoot** class, as provided by `GarmaSimBoot`.
- `allow.parallel`: Logical `TRUE/FALSE` indicating whether parallel computation via the foreach package should be used. The default value is `TRUE`. OBS:paralllel backend must be registered prior to calling GarmaSim.
- `conf`: A scalar containing the confidence level of the required interval(s). The default value is 0.95.

The examples:

1. `nmonte = 1`:

```r
spec1 <- GarmaSim(
GarmaSpec("GA",
          phi = c(0.5),
         beta.x = 1,
         sigma2 = 2,
         X = as.matrix(
           data.frame(
             x1 = rep(10,101)))),
nmonte = 1, burnin = 0)

boot1 <- GarmaSimBoot(spec1,l = 20 )

ci <- ConfidenceInterval(boot1)
print(ci)
summary(ci)
```

```
coverage (ci)
#>  -------------------------------------------------------
#>  A GA-Garma(1,0)   MBB ci's coverage object:
#>
#>  Number of Monte Carlo Simulations ('nmonte') =  1
#>  Time Series Length ('nsteps') =  100
#>  parallel =  TRUE
#>  block length =  20
#>  R =  100
#>  -------------------------------------------------------
#>
#>  Bootstrap Confidence Interval Coverage Rates:
#>   length parameter norm bias.c basic perc
#> 1   l=20   beta.x1    1      1     1    1
#> 2   l=20       phi    1      1     1    1
#> 3   l=20     sigma    1      1     1    1
plot(ci)
```

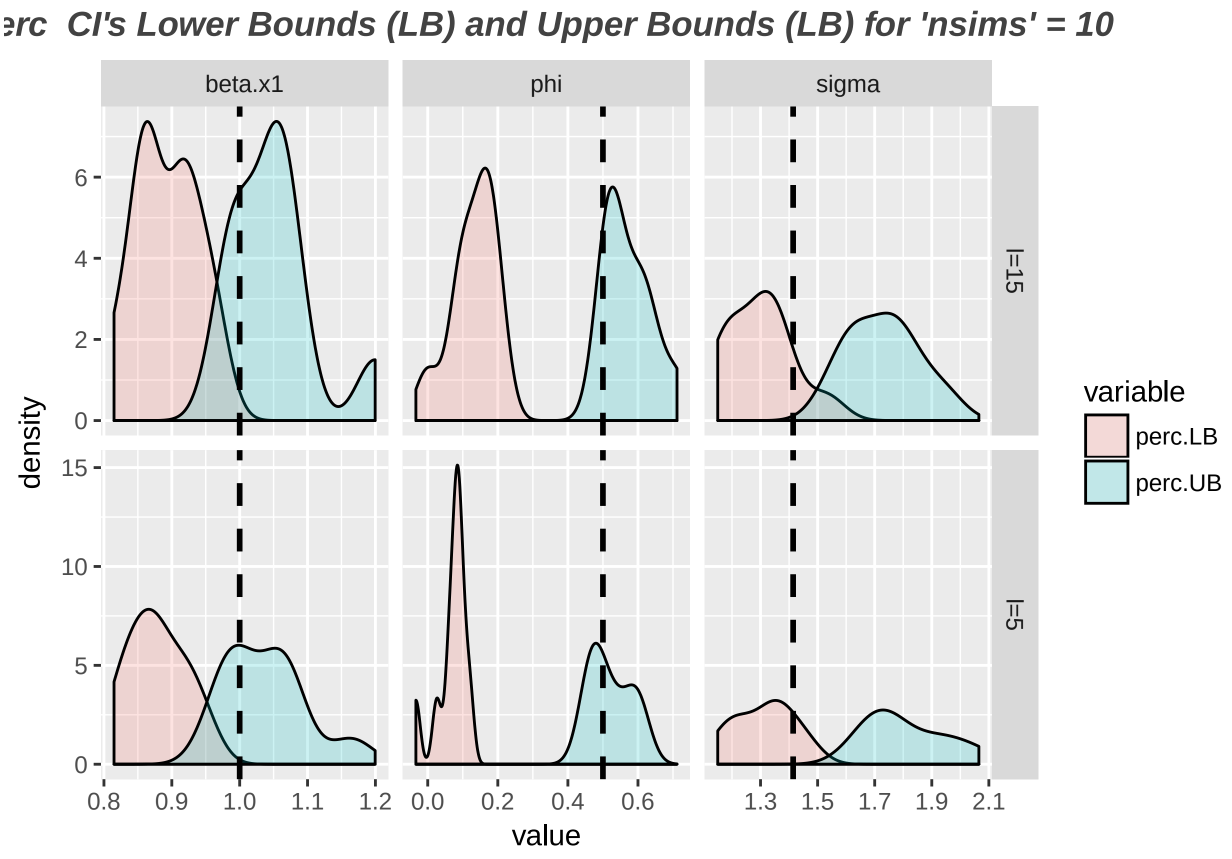


Note that we have information regarding the Normal, Basic, Bias Corrected and Percentile confidence intervals. In the case of `nmonte > 1` we just have do provide the respective tag for each ci (actually `type.ci` argument to the plot method): 'norm', 'basic', 'bias.c' and 'perc'.

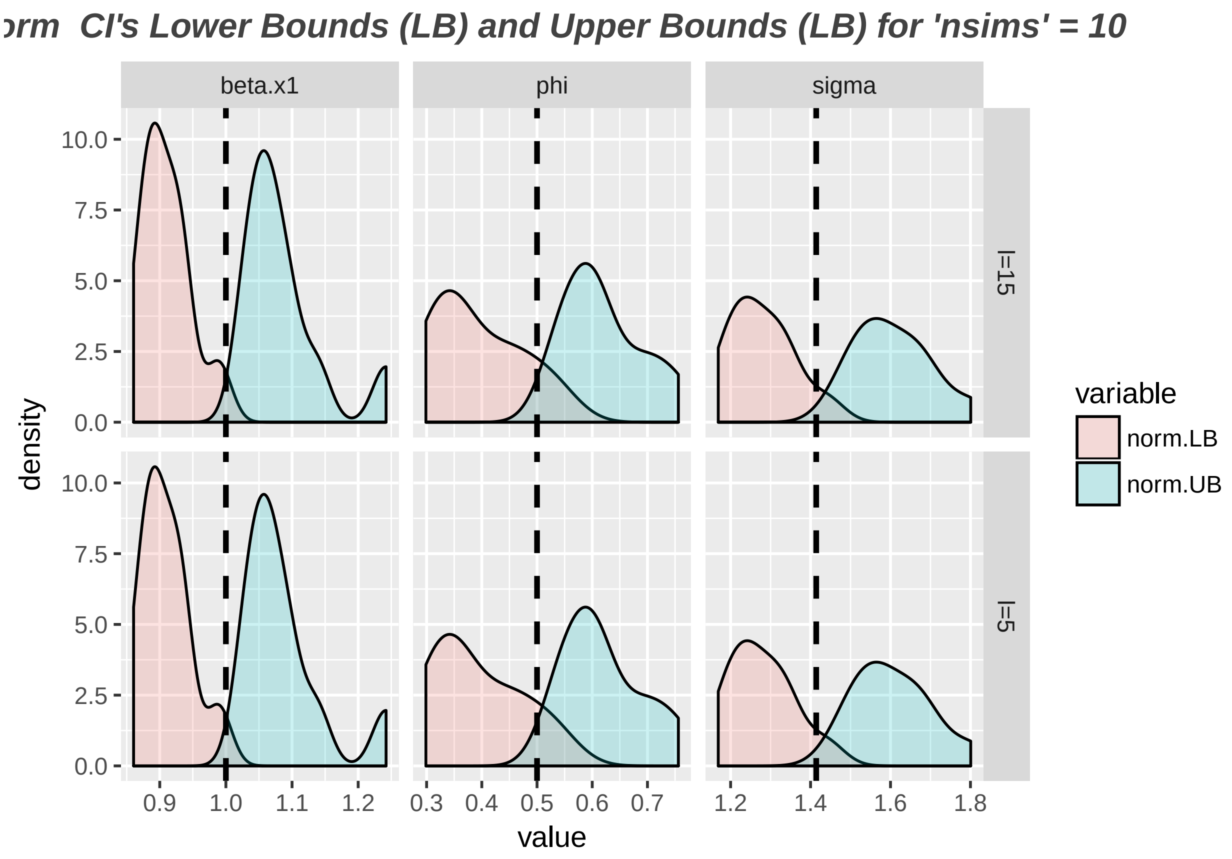
2. `nmonte > 1`:

```
spec2 <- GarmaSim(
GarmaSpec("GA",
          phi = c(0.5),
          beta.x = 1,
          sigma2 = 2,
          X = as.matrix(
            data.frame(
              x1 = rep(10,101)))),
nmonte = 10, burnin = 0)

boot2 <- GarmaSimBoot(spec2,l = c(5,15) )

ci <- ConfidenceInterval(boot2)
print(ci)
summary(ci)
```

```
coverage (ci)
#>  -------------------------------------------------------
#>  A GA-Garma(1,0)   MBB ci's coverage object:
#>
#>  Number of Monte Carlo Simulations ('nmonte') =  10
#>  Time Series Length ('nsteps') =  100
#>  parallel =  TRUE
#>  block length =  5 15
#>  R =  100
#>  -------------------------------------------------------
#>
#>  Bootstrap Confidence Interval Coverage Rates:
#>   length parameter norm bias.c basic perc
#> 1    l=5   beta.x1  1.0    0.9   0.9  0.7
#> 2    l=5       phi  0.9    0.7   0.7  0.5
#> 3    l=5     sigma  0.9    1.0   1.0  0.8
#> 4   l=15   beta.x1  1.0    0.9   0.9  0.7
#> 5   l=15       phi  0.9    1.0   0.8  1.0
#> 6   l=15     sigma  0.9    1.0   1.0  0.9
plot(ci,type.ci='perc')
```



```
plot(ci)  #norm ci
```



4. Estimation of a general Garma model

This section deals with the estimation of any garma model, making usage of the `garmaFit2` function from the **dboot** package (an adapted version of the `garmaFit` function from the **gamlss.util** pkg). This is a general purpose function, handling families other than the ones in the exponential family, for details check the **gamlss.util** package. The `GarmaFit` method is only a constructor from a `GarmaSim` object to the `garmaFit` function. The minimal structure for estimating a **GARMA** process is (as defined in the `garmaFit` function):

- `formula`: A formula for linear terms i.e. like in `lm()`
- `order`: the relevant `data.frame`.
- `data`: Numeric, the seed to `set.seed()` for replicable examples. Default value is 123.

So we can construc an example with the `GarmaSim` function or use the analogy into a real data example:

```
spec <- GarmaSpec("PO",
                  phi = c(0.5,0.15),
                  beta.x = c(1,1),
                  mu0 =  c(2,2),
                  X = as.matrix(
                    data.frame(
                      intercept = rep(1,1102),
                      x1 = c(rep(7,100),
                             rep(2,1002)))))
```

```
sim <- GarmaSim(spec,
                nmonte = 1,
                nsteps = 100)

#creating the required data.frame
db <- data.frame(cbind(yt = print(sim), intercept = rep(1,100)))
#>  -------------------------------------------------------
#>  A PO-Garma(2,0) simulation object:
#>
#>  phi1 =  0.5
#>  phi2 =  0.15
#>  beta.intercept =  1
#>  beta.x1 =  1
#>  mu0[1] =  2
#>  mu0[2] =  2
#>  Number of Monte Carlo Simulations ('nmonte') =  1
#>  Time Series Length ('nsteps') =  100
#>  Burn-in ('burnin') =  1000
#>  parallel =  TRUE
#>  -------------------------------------------------------
#>
set.seed(123)
fit <- garmaFit2(yt~.-1 ,data= db,order = sim@order )
#> deviance of linear model=  637.7676
#> deviance of  garma model=  571.322
library(gamlss.util)
#> Loading required package: zoo
#>
#> Attaching package: 'zoo'
#> The following objects are masked from 'package:base':
#>
#>     as.Date, as.Date.numeric
summary(fit)
#>
#> Family:  c("NO", "Normal")
#> Fitting method: "nlminb"
#>
#> Call:  garmaFit2(formula = yt ~ . - 1, order = sim@order, data = db)
#>
#>
#> Coefficient(s):
#>                 Estimate  Std. Error  t value   Pr(>|t|)
#> beta.intercept 19.0463703   1.5215935 12.51738 < 2.22e-16 ***
#> phi1            0.5060380   0.0982163  5.15228 2.5734e-07 ***
#> phi2            0.1976032   0.0973950  2.02889    0.04247 *
#> sigma           4.4636610   0.3188329 14.00000 < 2.22e-16 ***
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#>  Degrees of Freedom for the fit: 4 Residual Deg. of Freedom   96
#> Global Deviance:     571.322
#>             AIC:     579.322
#>             SBC:     589.743
```

```
#the same task using the built-in intercept:
db <- data.frame(yt = print(sim))
#>  -------------------------------------------------------
#>  A PO-Garma(2,0) simulation object:
#>
#>  phi1 =  0.5
#>  phi2 =  0.15
#>  beta.intercept =  1
#>  beta.x1 =  1
#>  mu0[1] =  2
#>  mu0[2] =  2
#>  Number of Monte Carlo Simulations ('nmonte') =  1
#>  Time Series Length ('nsteps') =  100
#>  Burn-in ('burnin') =  1000
#>  parallel =  TRUE
#>  -------------------------------------------------------
#>
set.seed(123)
fit <- garmaFit2(yt~. ,data= db,order = sim@order )
#> deviance of linear model=  637.7676
#> deviance of  garma model=  571.322
summary(fit)
#>
#> Family:  c("NO", "Normal")
#> Fitting method: "nlminb"
#>
#> Call:  garmaFit2(formula = yt ~ ., order = sim@order, data = db)
#>
#>
#> Coefficient(s):
#>                   Estimate  Std. Error  t value   Pr(>|t|)
#> beta.(Intercept) 19.0463703   1.5215935 12.51738 < 2.22e-16 ***
#> phi1              0.5060380   0.0982163  5.15228 2.5734e-07 ***
#> phi2              0.1976032   0.0973950  2.02889    0.04247 *
#> sigma             4.4636610   0.3188329 14.00000 < 2.22e-16 ***
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#>  Degrees of Freedom for the fit: 4 Residual Deg. of Freedom   96
#> Global Deviance:     571.322
#>             AIC:     579.322
#>             SBC:     589.743
```

5. MBB of a general Garma model

In this section we deal with the MBB estimation of a *GARMA* process using the `tsboot2` function from the **dboot** package. This is only a modified version of the `tsboot` function from the **boot** package, with similar arguments.

```
spec <- GarmaSpec("PO",
                  phi = c(0.5,0.15),
                  beta.x = c(1,1),
                  mu0 =  c(2,2),
                  X = as.matrix(
                    data.frame(
                      intercept = rep(1,1102),
                      x1 = c(rep(7,100),
                             rep(2,1002)))))

sim <- GarmaSim(spec,
                nmonte = 1,
                nsteps = 100)

#db <- data.frame(yt = print(sim))
db <- data.frame(cbind(yt = print(sim), intercept = rep(1,100)))
#>  -------------------------------------------------------
#>  A PO-Garma(2,0) simulation object:
#>
#>  phi1 =  0.5
#>  phi2 =  0.15
#>  beta.intercept =  1
#>  beta.x1 =  1
#>  mu0[1] =  2
#>  mu0[2] =  2
#>  Number of Monte Carlo Simulations ('nmonte') =  1
#>  Time Series Length ('nsteps') =  100
#>  Burn-in ('burnin') =  1000
#>  parallel =  TRUE
#>  -------------------------------------------------------
#>
set.seed(123)

boot.function <- function(data, order, family) {
 fit <- garmaFit2(yt~.-1,data = data ,order = order,
                  family = family)
 return(fit$coef)
}
ord <- sim@order ; fam <- sim@spec@family


MBB <- tsboot2(db,
               statistic = boot.function,
               R = 10,
               l = 5,
               order = ord,
               family = fam,
               export = "garmaFit2",
               package = "gamlss")
#> deviance of linear model=  655.8949
#> deviance of  garma model=  568.2738
#> Warning: <anonymous>: ... may be used in an incorrect context: 'statistic(ran.gen(ts.orig[inds, ], n

#The original statistic (applied to the original series ):
MBB$t0
#> beta.intercept           phi1           phi2
#>      2.9929513      0.4841739      0.2068798

#The resamples:
MBB$t
#>           [,1]      [,2]        [,3]
#>  [1,] 3.037345 0.4416777  0.14509357
#>  [2,] 2.935912 0.3022353  0.12467772
#>  [3,] 3.114831 0.5144340  0.14757825
#>  [4,] 2.981993 0.5308902 -0.03848191
#>  [5,] 3.003604 0.5799550 -0.07059052
#>  [6,] 2.982355 0.3891946 -0.04175624
#>  [7,] 2.945518 0.3152008  0.07419220
#>  [8,] 2.883092 0.4431917  0.12134359
#>  [9,] 2.966544 0.4938093  0.06522629
#> [10,] 2.976808 0.2649635  0.28420021

#Quick statistic for the resamples:
apply(MBB$t,2, mean)
#> [1] 2.98280007 0.42755522 0.08114832
```

Here we only dealt with the *Poisson-GARMA* model, for all available families:

```
library(gamlss.dist)
?gamlss.family
```

6. References

Benjamin, Michael A., Rigby, Robert A. and Stasinopoulos, D. Mikis. 2003. Generalized Autoregressive Moving Average Models. Journal of the American Statistical Association. Mar, 2003, Vol. 98, 461, pp. 214-223.