Introduction to the garma Package

Matheus de Vasconcellos Barroso

2018-03-15

Contents

Introduction

This is an introduction to the *garma* package. The package is designed to simulate, estimate and generate Moving Block Bootstrap resamples and diagnosis for the *GARMA* process of Benjamin et al. (2003). It 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:

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

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

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

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

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 unter $\mathcal{F}$ 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 $\mathcal{F}_{t-1}$. $\mathcal{F}_{t-1} = \{\mathbf{x_1}, \dots, \mathbf{x_{t-1}}; y_1, \dots, y_{t-1}; \mu_1, \dots, \mu_{t-1}\}$ such that the conditional density $f_{Y_t|\mathcal{F}_{t-1}}$ is of the form:

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

where $a(\cdot)$ and $b(\cdot)$ are specific functions definining the particular member of the exponential familiy, with v_t as the canonical and φ as the scale parameter of the member, $\mathbf{x}$ 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 $\boldsymbol{\beta'} = (\beta_1, \beta_2, \dots, \beta_r)$ is the vector of parameters of the linear predictor, $\boldsymbol{\phi'} = (\phi_1, \phi_2, \dots, \phi_p)$ the vector of autoregressive parameters and $\boldsymbol{\theta'} = (\theta_1, \theta_2, \dots, \theta_p)$ 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:

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

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:

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

with $E_{Y_t|\mathcal{F}_{t-1}} = \delta\gamma$ and $Var_{Y_t|\mathcal{F}_{t-1}} = \delta\gamma^2$. 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 $E_{Y_t|\mathcal{F}_{t-1}} = \mu_t$ and $Var_{Y_t|\mathcal{F}_{t-1}} = \sigma^2\mu_t^2$. In the *Gamma* case the canonical *link function* is the reciprocal function, though, we make usage of the *logarithmic* function. A new version of the package with all reasonable link functions will be developed but keep in mind that the *log* is currently being used in the *Gamma* case so that the equation for η_t is the same for the *Poisson* case, that is, Eq. 4.

The garma package

After a formal definition of the *GARMA* model now we are 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.

GarmaSpec

The GarmaSpec class is the more general class for specificating the *GARMA* model and contain 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 includer here. The default behaviour 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)`. Where **nsteps** is the number of simulations desired, **burnin** the number of burnin observations 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 correc specification of a *Poisson* or *Gamma GARMA* models.

PoissonSpec

Inherits from `GarmaSpec` and 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

Inherits from `GarmaSpec` and has the slots:

- `family`: A character with the tag "PO".
- `sigma2`: Numeric, specifying the σ^2 parameter.
- `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 construtor 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:

```r
library(garma)
GarmaSpec("PO") # This results in no ARMA terms.
#> An object of class "PoissonSpec"
#> Slot "alpha":
#> [1] 0.1
#>
#> Slot "mu0":
#> [1] 10
#>
#> Slot "family":
#> [1] "PO"
#>
#> Slot "beta.x":
#> [1] 1
#>
#> Slot "phi":
#> [1] 0
#>
#> Slot "theta":
#> [1] 0
#>
#> Slot "X":
#> <0 x 0 matrix>
GarmaSpec("GA") # This results in no ARMA terms.
#> An object of class "GammaSpec"
#> Slot "sigma2":
#> [1] 1
#>
#> Slot "mu0":
#> [1] 10
#>
#> Slot "family":
#> [1] "GA"
#>
#> Slot "beta.x":
#> [1] 1
#>
#> Slot "phi":
#> [1] 0
#>
#> Slot "theta":
#> [1] 0
#>
#> Slot "X":
#> <0 x 0 matrix>
```

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

```r
GarmaSpec("PO",phi=c(0.3,0.2),mu0=c(1,1))
#> An object of class "PoissonSpec"
#> Slot "alpha":
#> [1] 0.1
#>
#> Slot "mu0":
#> [1] 1 1
#>
#> Slot "family":
#> [1] "PO"
#>
#> Slot "beta.x":
#> [1] 1
#>
#> Slot "phi":
#> [1] 0.3 0.2
#>
#> Slot "theta":
#> [1] 0
#>
#> Slot "X":
#> <0 x 0 matrix>
```

Now we are ready to move to some more elaborate examples:

1. A GA-GARMA(1,0) with an intercept term, 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.

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

2. A PO-GARMA(1,3) with an intercept term, here the default behaviour of μ_0 will not suffice, so we need three initial values. The intercept will be generated automatically

```r
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. 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)

```r
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:

```
spec4 <- 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)))))
```

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` parameters 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 perform, 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 simulate. The default value is **100**.
- `burnin`: A numeric vector indicating the number of burn in 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 build in the session Garma Spec:

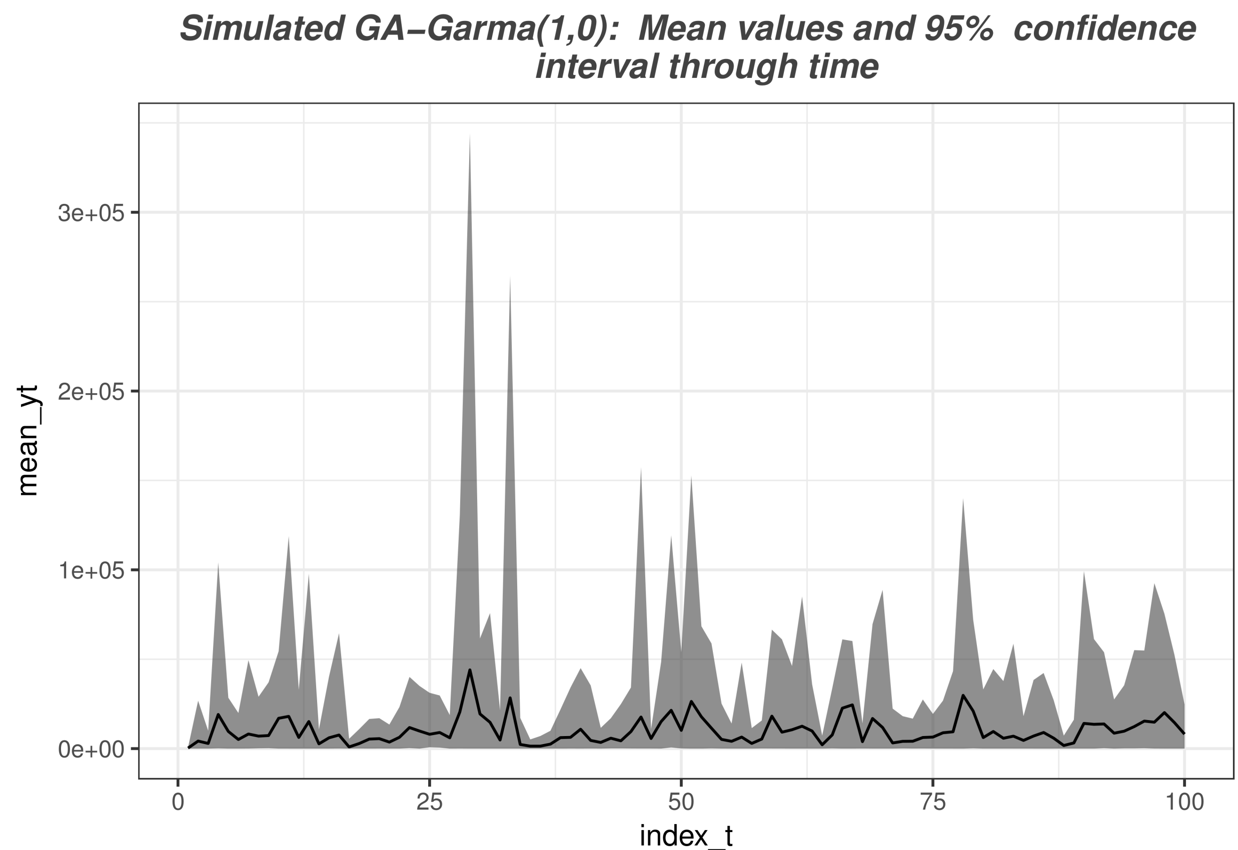
```
sim1 <- GarmaSim(spec1,
                 nmonte = 10, #10 monte carlo simulations
                 burnin = 0) # no burnin
```

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.

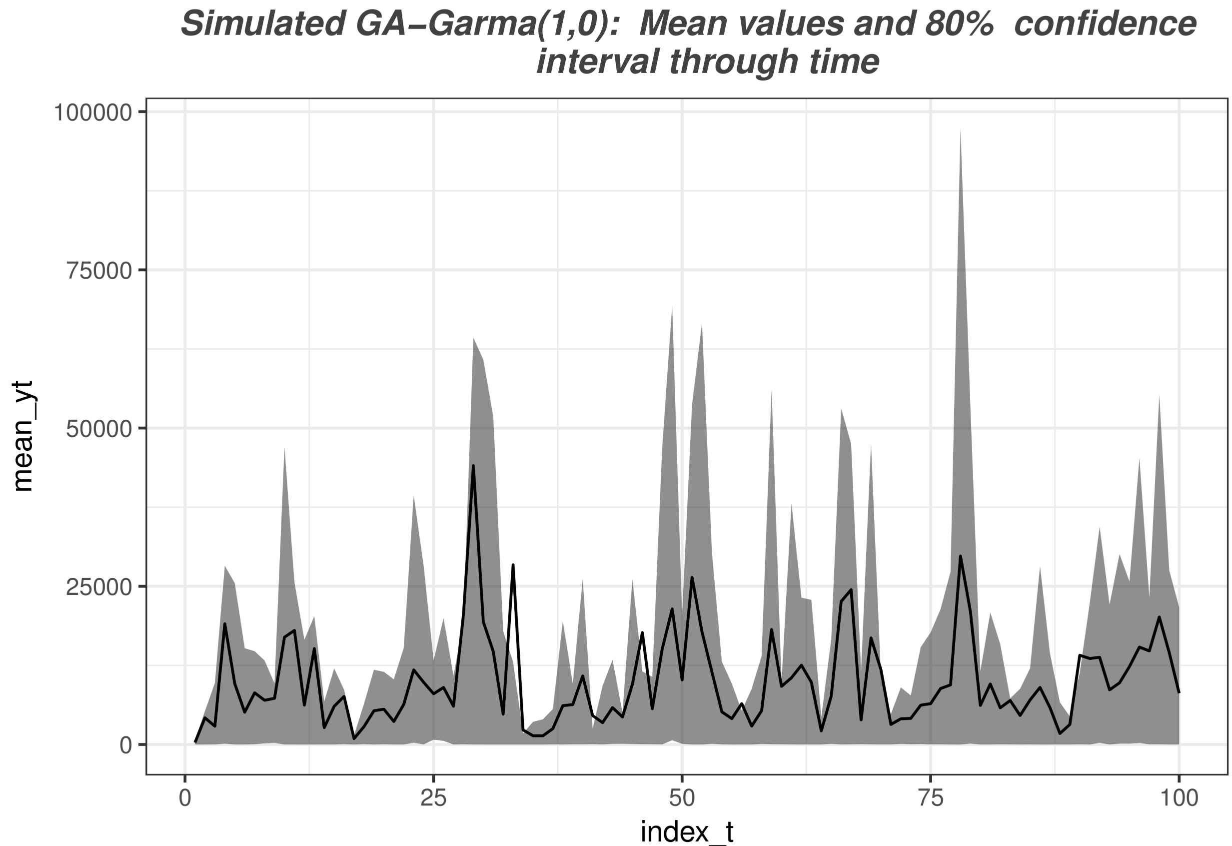
```
print(sim1)
```

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)
```




```
plot(sim1,
        confInt = .80)
```

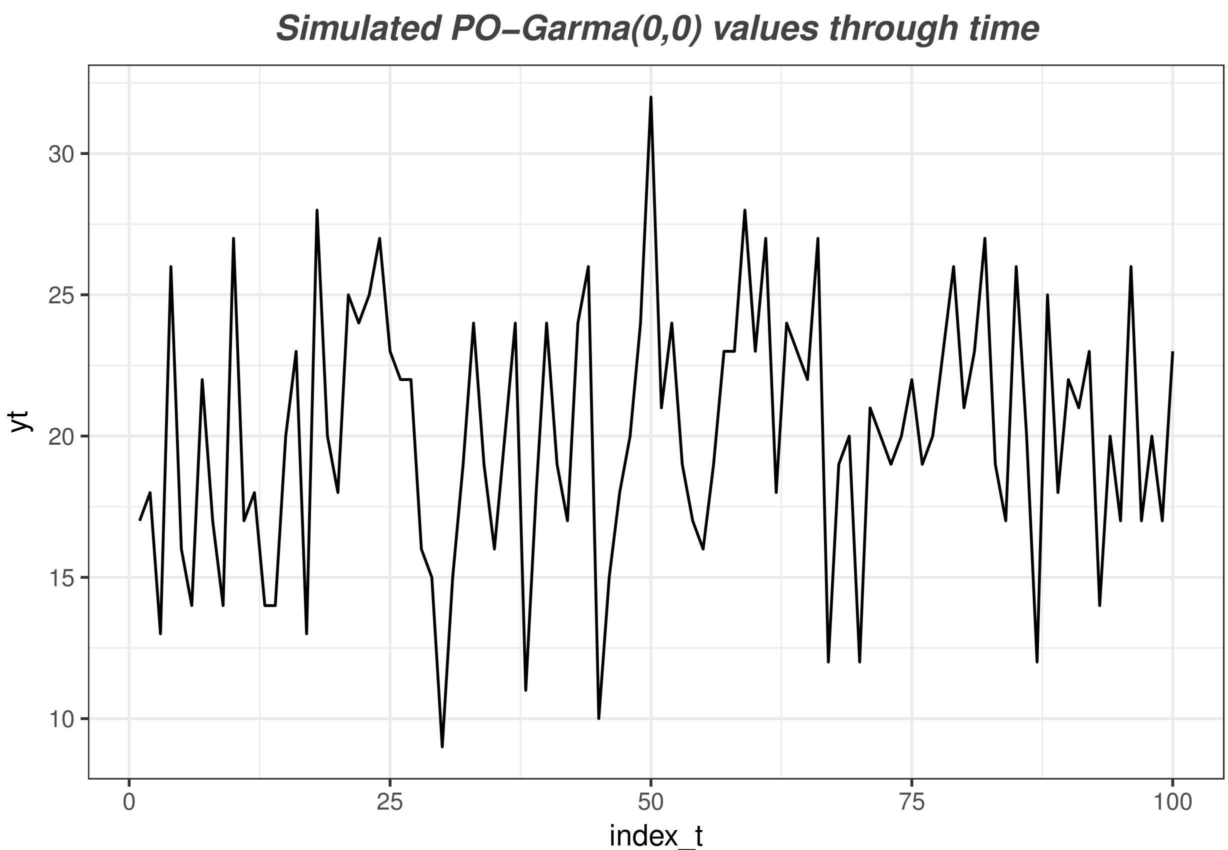


The summary(`GarmaSim`) method has a different behaviour for `nmonte > 1` and `nmonte = 1`

```
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 =  9932.798 with distribution:
#>       2.5%     50%    97.5%
#> 2 5638.614 8282.88 18799.36
#>
#> Mean Monte Carlo min values =  0.4406 with distribution:
#>   2.5%   50% 97.5%
#> 1    0 0.027 1.512
#>
#> Mean Monte Carlo max values =  134052.1 with distribution:
#>       2.5%      50%    97.5%
#> 3 53313.97 92538.95 344191.5
#>
#>              mean      2.5%       50%      97.5%
#> min        0.4406     0.000     0.027      1.512
#> mean    9932.7980  5638.614  8282.880  18799.355
#> max   134052.0926 53313.974 92538.953 344191.509
```

2. An **Poisson-GARMA(0,0)**, with only one Monte Carlo simulation:

```
sim2 <-
  GarmaSim(
    GarmaSpec("PO",
              beta.x = c(0.1,1),
              X = as.matrix(
              data.frame(
                intercept = rep(10,1100),
                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] 17 18 13 26 16 14 22 17 14 27 17 18 14 14 20 23 13 28 20 18 25 24 25
#>  [24] 27 23 22 22 16 15  9 15 19 24 19 16 20 24 11 18 24 19 17 24 26 10 15
#>  [47] 18 20 24 32 21 24 19 17 16 19 23 23 28 23 27 18 24 23 22 27 12 19 20
#>  [70] 12 21 20 19 20 22 19 20 23 26 21 23 27 19 17 26 20 12 25 18 22 21 23
#>  [93] 14 20 17 26 17 20 17 23
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.
#>    9.00   17.00   20.00   20.07   23.00   32.00
```

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

GarmaFit

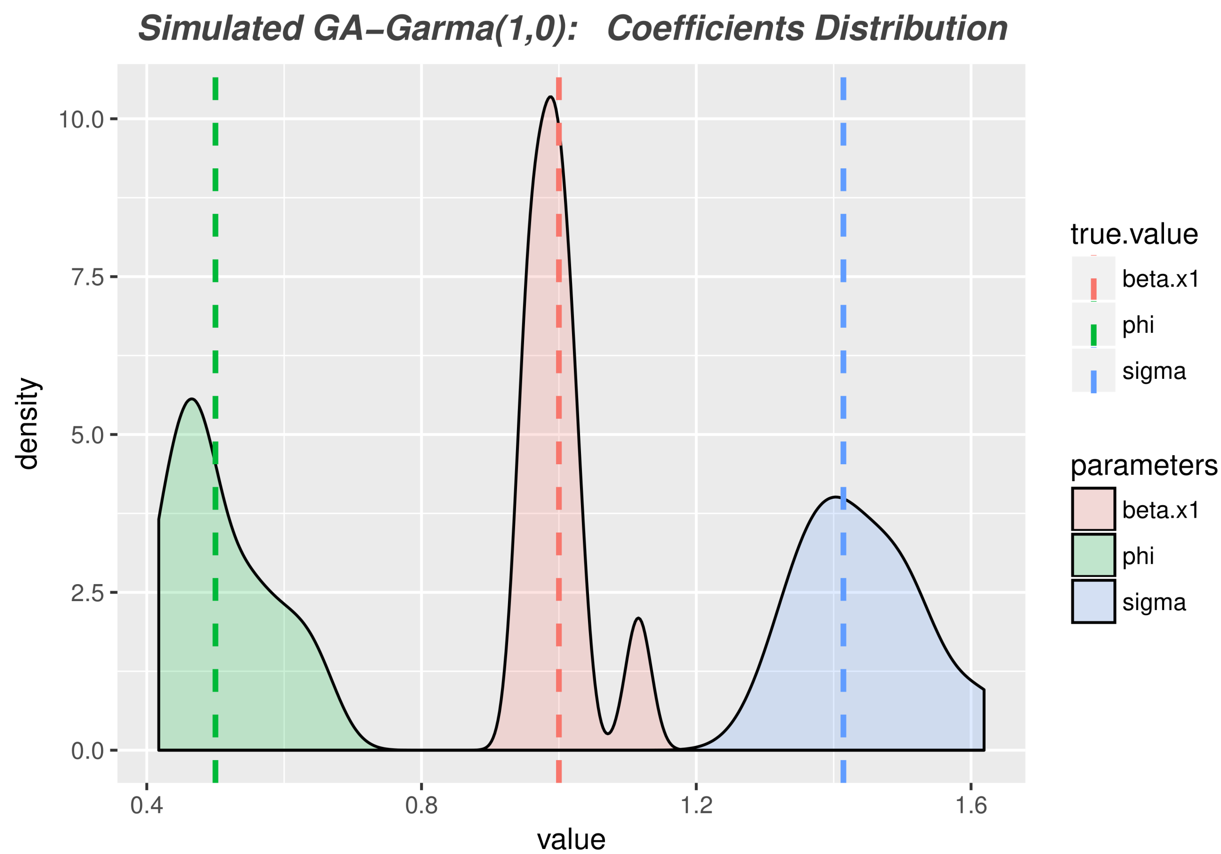
Given a `GarmaSim` object we can simultaneously estimate all the `nmonte` series (if `parallel = TRUE` and the parallel back-end registered). The methods `GarmaFit`, `print`, `plot` and `summary` are implemented. The `GarmaSim` method has the following parameters:

- `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 attempts 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.

```
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 0.9512578 1.1156606 0.9952164 0.9638448 1.0227713 0.9951840
#> phi     0.4214639 0.6464954 0.4174278 0.4640413 0.6021564 0.4757756
#> sigma   1.5047123 1.3715135 1.3989626 1.4281057 1.4208644 1.3231409
#>              [,7]      [,8]      [,9]     [,10]
#> beta.x1 0.9919034 0.9755214 1.0169597 0.9530157
#> phi     0.5313133 0.5566795 0.4767271 0.4704674
#> sigma   1.6189999 1.3584369 1.5088017 1.4912128
plot(fit1)
```

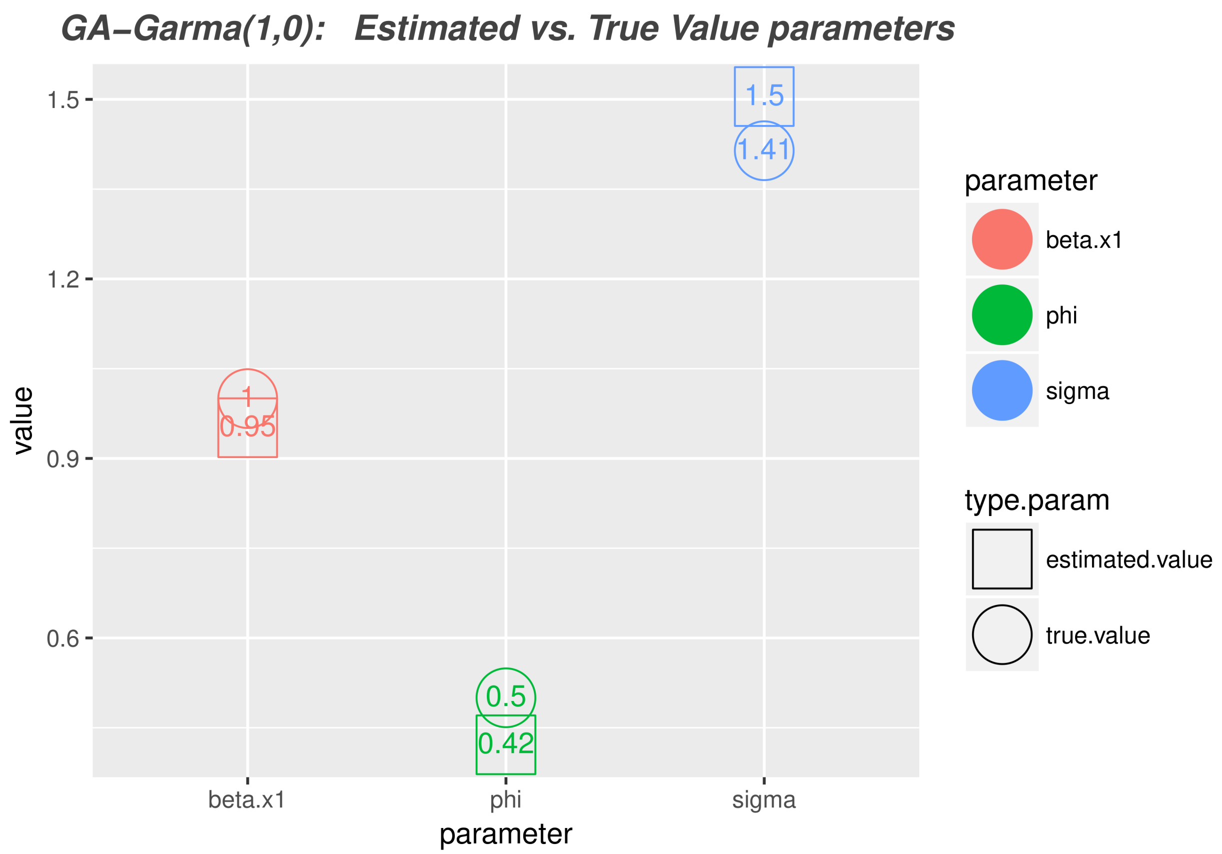



```
summary(fit1)
#>  ------------------------------------------------------
#>  A GA-Garma(1,0)  simulation object:
#>
#>  Number of Monte Carlo Simulations ('nmonte') =  10
#>  Time Series Length ('nsteps') =  100
#>  Burn-in ('burnin') =  0
#>  parallel =  TRUE
#>  ------------------------------------------------------
#>   parameters      Min.   1st Qu.    Median      Mean.   3rd Qu.      Max.
#> 1    beta.x1 0.9512578 0.9638448 0.9919034 0.9981335 1.0169597 1.1156606
#> 2        phi 0.4174278 0.4640413 0.4757756 0.5062548 0.5566795 0.6464954
#> 3      sigma 1.3231409 1.3715135 1.4208644 1.4424751 1.5047123 1.6189999
#>   true.value
#> 1   1.000000
#> 2   0.500000
#> 3   1.414214
```

2. An example with `nmonte =  1`:

```
sim2 <-  GarmaSim(
  GarmaSpec("GA",
            phi = 0.5,
            beta.x = 1,
            sigma2 =2,
            X = as.matrix(
              data.frame(
```

```r
                 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)
```




```r
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
```

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 parameters:

- `sim`: An object of the **GarmaSim** class, as provided by `GarmaSim`.
- `l`: `l` is the fixed block length used in generating the replicate 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.

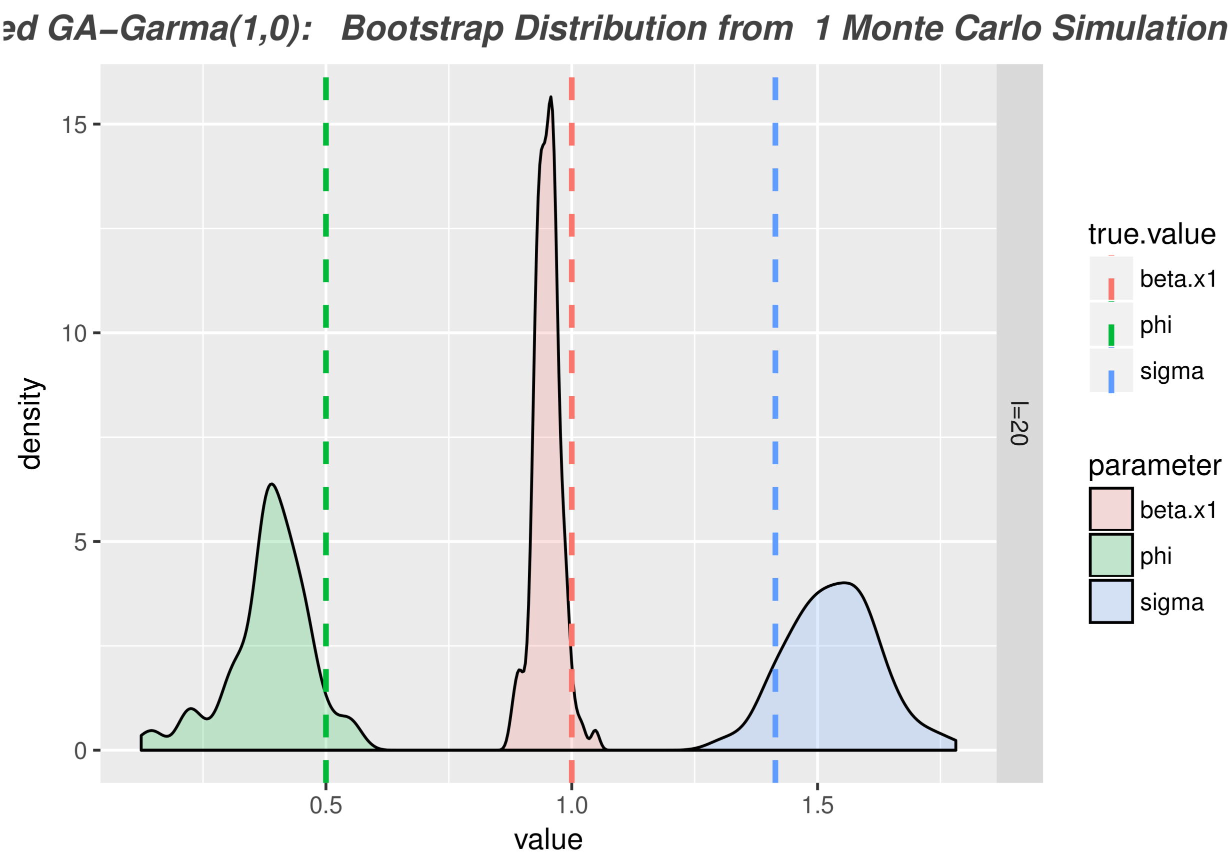
- `seed`: Numeric, the seed to `set.seed()` for replicable examples. Default value is 123.
- `errorhandling`: Character, either 'try' or 'pass'
- `n.try`: Positive integer. If `errorhandling = 'try'`, this specifies the number of attempts in the algorithm.
- `boot.function`: A function to summarise the bootstrap replicates. The default function returns 0. Be aware that this is not a problem, as by default the mean values is already being returned. This is useful if the user wants to specify a quantity not being reported, as an 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.`nmonte = 1` and only one block length for the MBB:

```
# specification and simulation:
Sim1 <- GarmaSim(
  GarmaSpec("GA",
            phi = 0.5,
            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=  1853.087
#> deviance of  garma model=  1805.009
#print(mbb1)
plot(mbb1)
```

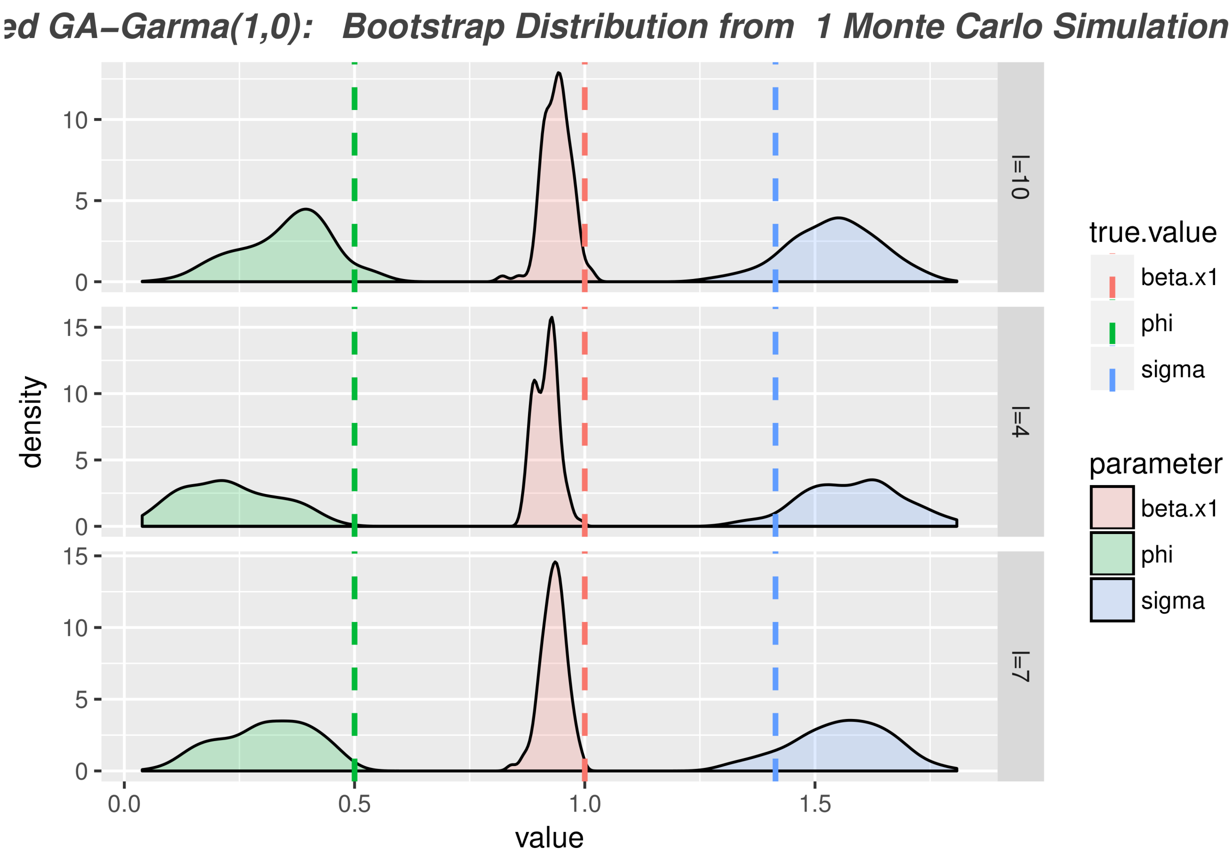



```
summary(mbb1)
#>  -------------------------------------------------------
#>  A GA-Garma(1,0)  bootstrap object:
#>
#>  Number of Monte Carlo Simulations ('nmonte') =  1
#>  Time Series Length ('nsteps') =  100
#>  Burn-in ('burnin') =  0
#>  parallel =  TRUE
#>  errorhandling =  try
#>  n.try =  5
#>  block length =  20
#>  R =  100
#>  -------------------------------------------------------
#>
#>  Bootstrap Statistic Summary:
#> $`True values`
#>   parameter true.value
#> 1   beta.x1   1.000000
#> 2       phi   0.500000
#> 3     sigma   1.414214
#>
#> $Estimates
#>   length parameter variable Mean Est.
#> 1   l=20   beta.x1 original 0.9502413
#> 2   l=20       phi original 0.3853386
#> 3   l=20     sigma original 1.5316826
```

2. `nmonte = 10`, multiple block lengths

```
mbb2 <- GarmaSimBoot(Sim1,
                     l = c(4,7,10))
#> deviance of linear model=  1853.087
#> deviance of  garma model=  1805.009
#> deviance of linear model=  1853.087
#> deviance of  garma model=  1805.009
#> deviance of linear model=  1853.087
#> deviance of  garma model=  1805.009

#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
#>  R =  100
#>  -------------------------------------------------------
```

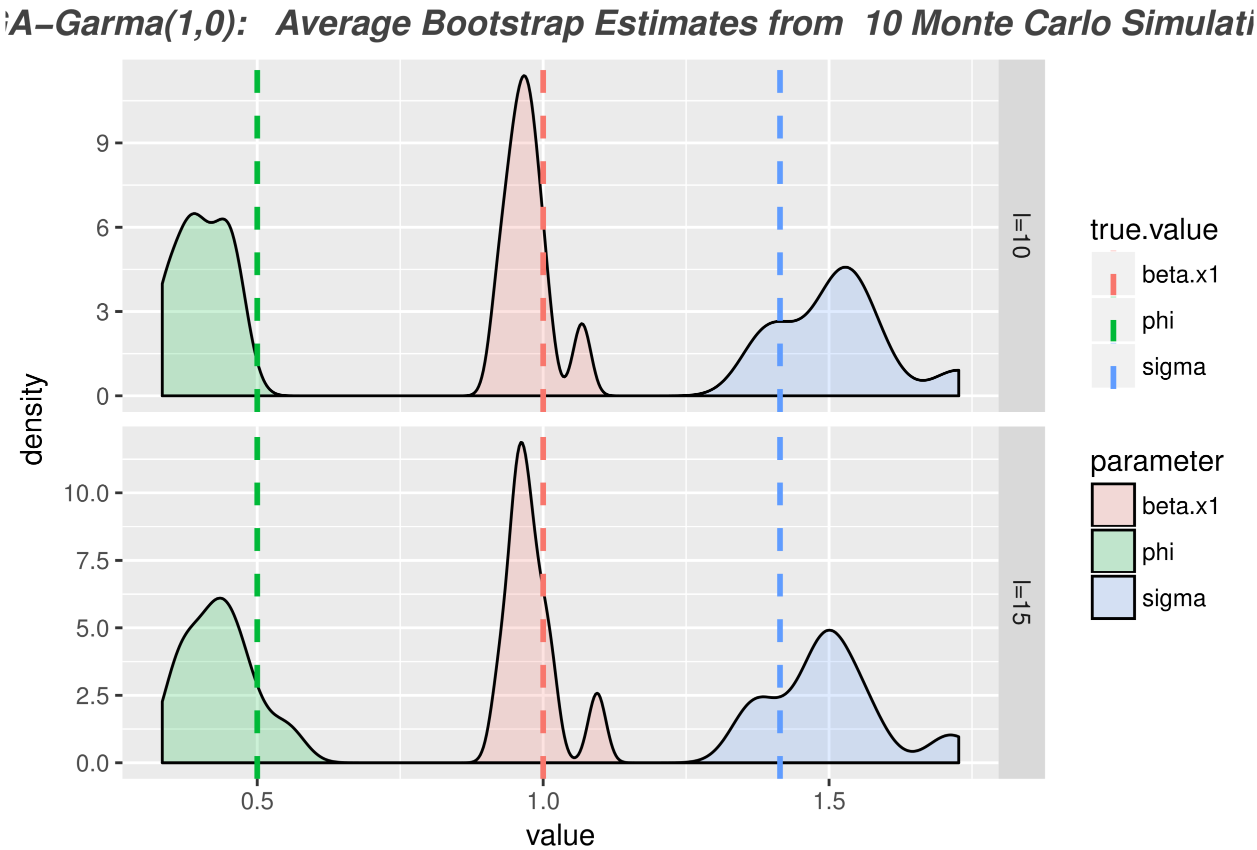
```
#>
#>  Bootstrap Statistic Summary:
#> $`True values`
#>   parameter true.value
#> 1   beta.x1   1.000000
#> 2       phi   0.500000
#> 3     sigma   1.414214
#>
#> $Estimates
#>   length parameter variable Mean Est.
#> 7    l=7   beta.x1 original 0.9320765
#> 8    l=7       phi original 0.3079951
#> 9    l=7     sigma original 1.5559196
#> 4    l=4   beta.x1 original 0.9161941
#> 5    l=4       phi original 0.2294262
#> 6    l=4     sigma original 1.5817752
#> 1   l=10   beta.x1 original 0.9389050
#> 2   l=10       phi original 0.3516731
#> 3   l=10     sigma original 1.5401262
```

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

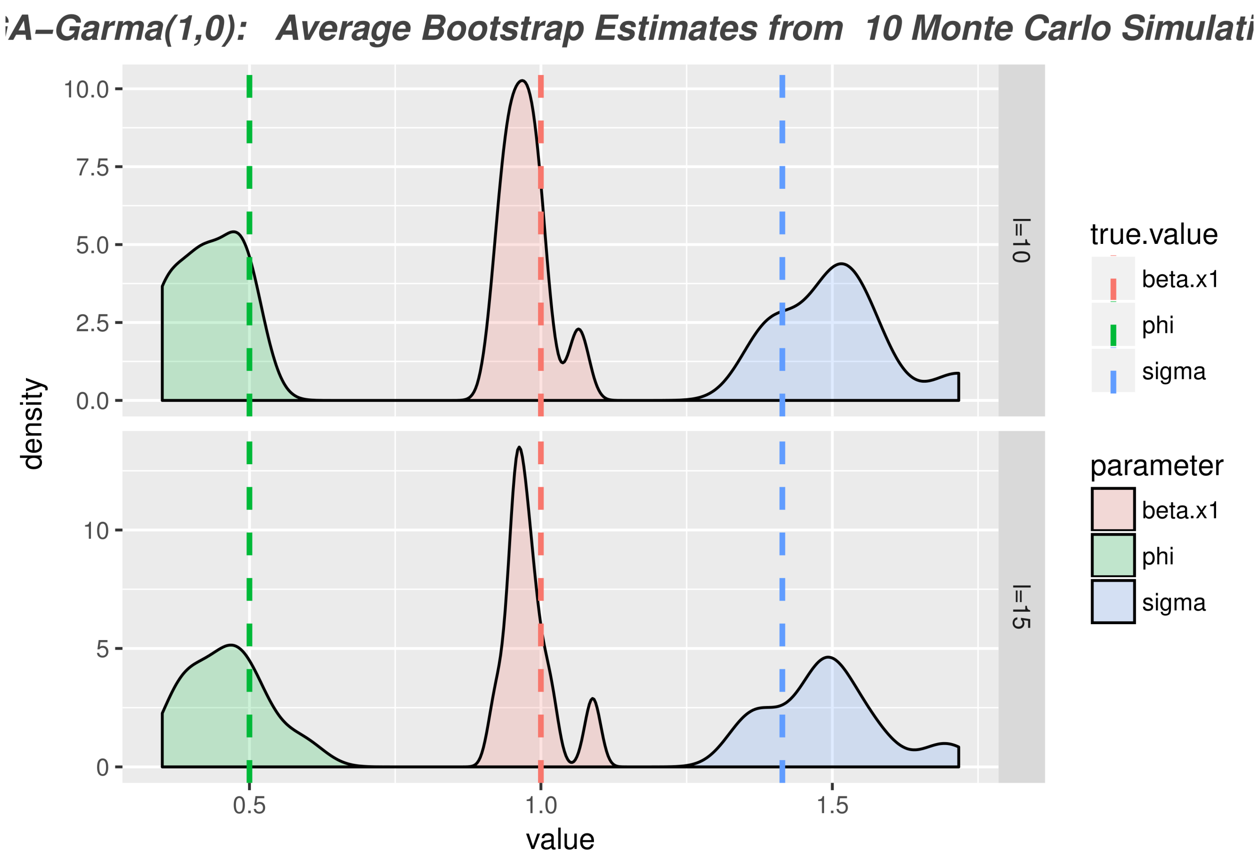
```
Sim2 <-  GarmaSim(
  GarmaSpec("GA",
            phi = 0.5,
            beta.x = 1,
            sigma2 =2,
            X = as.matrix(
              data.frame(
                x1 = rep(10,101)))),
  nmonte = 10,
  burnin = 0)
```

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

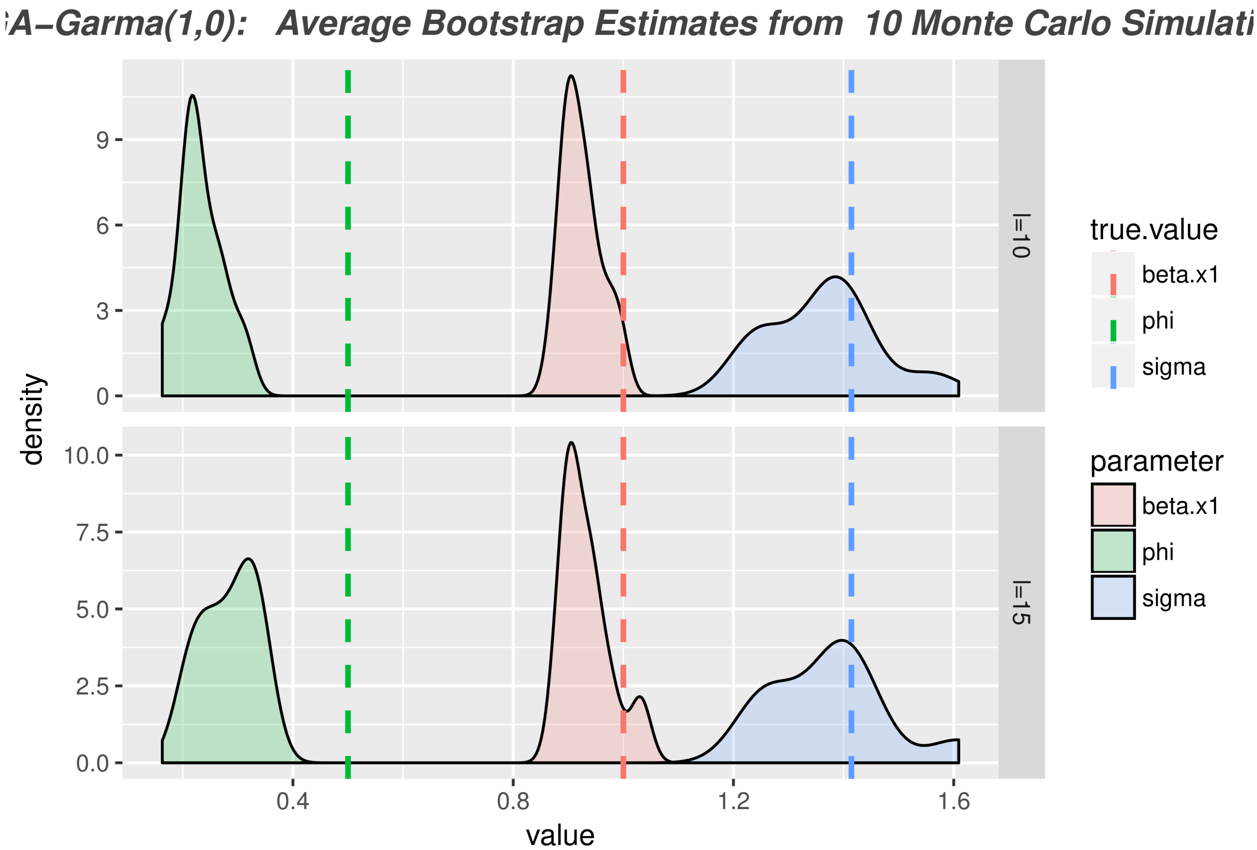
```
#print(mbb3)
plot(mbb3)
```



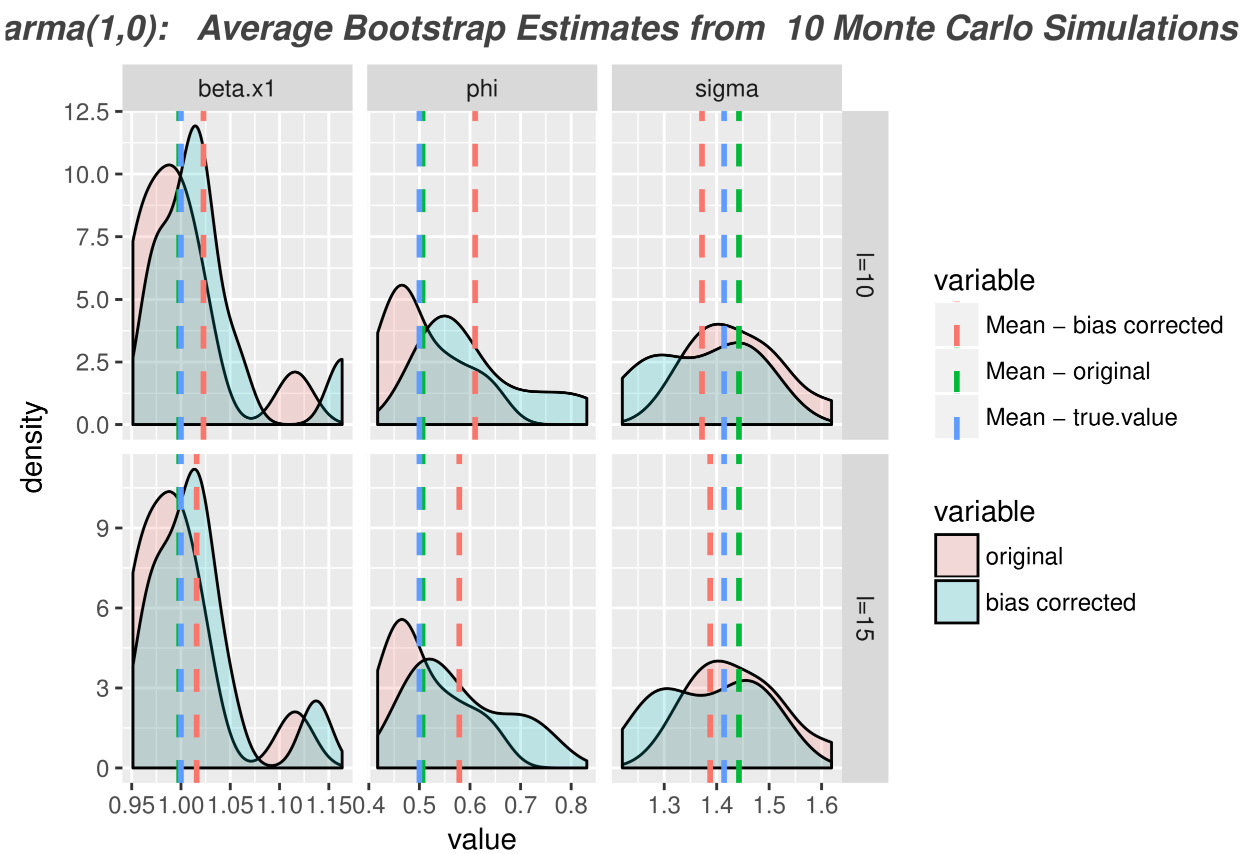

```r
plot(mbb3, variable = "Median.")
```




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




```
#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')
```




```
summary(mbb3)
#>  -------------------------------------------------------
#>  A GA-Garma(1,0)  bootstrap object:
#>
#>  Number of Monte Carlo Simulations ('nmonte') =  10
#>  Time Series Length ('nsteps') =  100
#>  Burn-in ('burnin') =  0
#>  parallel =  TRUE
#>  errorhandling =  try
#>  n.try =  5
#>  block length =  10 15
#>  R =  100
#>  -------------------------------------------------------
#>
#>  Bootstrap Statistic Summary:
#> $`True values`
#>   parameter true.value
#> 1   beta.x1   1.000000
#> 2       phi   0.500000
#> 3     sigma   1.414214
#>
#> $Estimates
#> $Estimates$original
#>    length parameter variable summarise
#> 1    l=10   beta.x1 original 0.9981340
#> 13   l=10       phi original 0.5062552
```

```
#> 25   l=10     sigma original 1.4424760
#> 37   l=15   beta.x1 original 0.9981340
#> 49   l=15       phi original 0.5062552
#> 61   l=15     sigma original 1.4424760
#>
#> $Estimates$bias
#>    length parameter variable   summarise
#> 2    l=10   beta.x1     bias -0.02468396
#> 14   l=10       phi     bias -0.10392664
#> 26   l=10     sigma     bias  0.07047019
#> 38   l=15   beta.x1     bias -0.01776717
#> 50   l=15       phi     bias -0.07256006
#> 62   l=15     sigma     bias  0.05485127
#>
#> $Estimates$`bias corrected`
#>    length parameter       variable summarise
#> 3    l=10   beta.x1 bias corrected 1.0228179
#> 15   l=10       phi bias corrected 0.6101819
#> 27   l=10     sigma bias corrected 1.3720058
#> 39   l=15   beta.x1 bias corrected 1.0159012
#> 51   l=15       phi bias corrected 0.5788153
#> 63   l=15     sigma bias corrected 1.3876247
#>
#> $Estimates$` std. error`
#>    length parameter    variable  summarise
#> 4    l=10   beta.x1  std. error 0.03779253
#> 16   l=10       phi  std. error 0.11520966
#> 28   l=10     sigma  std. error 0.11798771
#> 40   l=15   beta.x1  std. error 0.03809448
#> 52   l=15       phi  std. error 0.10869360
#> 64   l=15     sigma  std. error 0.10148352
#>
#> $Estimates$Min.
#>    length parameter variable  summarise
#> 5    l=10   beta.x1     Min. 0.87451636
#> 17   l=10       phi     Min. 0.08138220
#> 29   l=10     sigma     Min. 1.26085910
#> 41   l=15   beta.x1     Min. 0.87138490
#> 53   l=15       phi     Min. 0.07255566
#> 65   l=15     sigma     Min. 1.26234587
#>
#> $Estimates$`1st Qu.`
#>    length parameter variable summarise
#> 6    l=10   beta.x1  1st Qu. 0.9484606
#> 18   l=10       phi  1st Qu. 0.3232478
#> 30   l=10     sigma  1st Qu. 1.4278401
#> 42   l=15   beta.x1  1st Qu. 0.9574524
#> 54   l=15       phi  1st Qu. 0.3751582
#> 66   l=15     sigma  1st Qu. 1.4280658
#>
#> $Estimates$Median.
#>    length parameter variable summarise
#> 7    l=10   beta.x1  Median. 0.9745827
```

```
#> 19    l=10        phi  Median. 0.4298073
#> 31    l=10      sigma  Median. 1.5046131
#> 43    l=15    beta.x1  Median. 0.9808227
#> 55    l=15        phi  Median. 0.4603785
#> 67    l=15      sigma  Median. 1.4889247
#>
#> $Estimates$Mean.
#>    length parameter variable summarise
#> 8     l=10    beta.x1    Mean. 0.9734500
#> 20    l=10        phi    Mean. 0.4023286
#> 32    l=10      sigma    Mean. 1.5129462
#> 44    l=15    beta.x1    Mean. 0.9803668
#> 56    l=15        phi    Mean. 0.4336952
#> 68    l=15      sigma    Mean. 1.4973273
#>
#> $Estimates$`3rd Qu.`
#>    length parameter variable summarise
#> 9     l=10    beta.x1  3rd Qu. 0.9986573
#> 21    l=10        phi  3rd Qu. 0.4844474
#> 33    l=10      sigma  3rd Qu. 1.5866617
#> 45    l=15    beta.x1  3rd Qu. 1.0071224
#> 57    l=15        phi  3rd Qu. 0.5076496
#> 69    l=15      sigma  3rd Qu. 1.5592318
#>
#> $Estimates$Max.
#>    length parameter variable summarise
#> 10    l=10    beta.x1     Max. 1.0622121
#> 22    l=10        phi     Max. 0.5977572
#> 34    l=10      sigma     Max. 1.8272891
#> 46    l=15    beta.x1     Max. 1.0629719
#> 58    l=15        phi     Max. 0.5909486
#> 70    l=15      sigma     Max. 1.7863893
#>
#> $Estimates$`boot.func.10%`
#>    length parameter      variable summarise
#> 11    l=10    beta.x1 boot.func.10% 0.9238160
#> 23    l=10        phi boot.func.10% 0.2322866
#> 35    l=10      sigma boot.func.10% 1.3632199
#> 47    l=15    beta.x1 boot.func.10% 0.9325777
#> 59    l=15        phi boot.func.10% 0.2830445
#> 71    l=15      sigma boot.func.10% 1.3765794
#>
#> $Estimates$`boot.func.90%`
#>    length parameter      variable summarise
#> 12    l=10    beta.x1 boot.func.90% 1.0208273
#> 24    l=10        phi boot.func.90% 0.5225375
#> 36    l=10      sigma boot.func.90% 1.6683108
#> 48    l=15    beta.x1 boot.func.90% 1.0261614
#> 60    l=15        phi boot.func.90% 0.5408858
#> 72    l=15      sigma boot.func.90% 1.6252759
```

#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
```

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
```

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.