Package 'sarima'

May 23, 2017

Type Package

Title Simulation and Prediction with Seasonal ARIMA Models

Version 0.4-5

Date 2017-05-20

Author Georgi N. Boshnakov

Maintainer Georgi N. Boshnakov <georgi.boshnakov@manchester.ac.uk>

Description Functions, classes and methods for time series modelling with ARIMA and related models. The aim of the package is to provide consistent interface for the user. For example, a single function autocorrelations() computes various kinds of theoretical and sample autocorrelations. This is work in progress, see the documentation and vignettes for the current functionality.

Depends methods, stats4

Imports PolynomF, ltsa, FitAR, FitARMA, portes, lagged

Suggests fGarch, fImport

License GPL (>= 2)

LazyLoad yes

Collate utils.R filterClasses.R modelClasses.R sarima.R autocovariances.R armacalc.R

NeedsCompilation no

Repository CRAN

Date/Publication 2017-05-23 03:44:31 UTC

R topics documented:

`sarima-package` *Package sarima* *Simulation and Prediction with Seasonal ARIMA Models*

Description

Functions, classes and methods for time series modelling with ARIMA and related models. The aim of the package is to provide consistent interface for the user. For example, a single function autocorrelations() computes various kinds of theoretical and sample autocorrelations. This is work in progress, see the documentation and vignettes for the current functionality.

Details

Package:	sarima
Type:	Package
Version:	0.4-5
Date:	2017-05-20
License:	GPL (>= 2)
LazyLoad:	yes
Built:	R 3.3.2; ; 2017-05-22 13:13:07 UTC; windows

There is a large number of packages for time series modelling. They provide a huge number of functions, often with similar or overlapping functionality and different argument conventions. One of the aims of package **sarima** is to provide consistent interface to some frequently used functionality.

In package **sarima** names of functions and S4 classes generally consist of whole words stringed together in camel case. Only the first letter is capitalised in common abbreviations, such as ARIMA. The first word in class names is also capitalised, while function names start with lowercase letters.

This is work in progress, see also the vignette(s).

Author(s)

Georgi N. Boshnakov

Maintainer: Georgi N. Boshnakov <georgi.boshnakov@manchester.ac.uk>

Examples

```
## simulate a white noise ts (model from Francq & Zakoian)
n <- 5000
x <- sarima:::rgarch1p1(n, alpha = 0.3, beta = 0.55, omega = 1, n.skip = 100)

## acf and pacf
x.acf <- autocorrelations(x)
x.acf
x.pacf <- partialAutocorrelations(x)
x.pacf

## portmanteau test for iid, by default gives also ci's for the acf under H0
x.iid <- whiteNoiseTest(x.acf, h0 = "iid", nlags = c(5,10,20), x = x, method = "LiMcLeod")
x.iid

x.iid2 <- whiteNoiseTest(x.acf, h0 = "iid", nlags = c(5,10,20), x = x, method = "LjungBox")
x.iid2

## portmanteau test for garch H0
x.garch <- whiteNoiseTest(x.acf, h0 = "garch", nlags = c(5,10,20), x = x)
x.garch

## plot methods give the CI's under H0
plot(x.acf)
## if the data are given, the CI's under garch H0 are also given.
plot(x.acf, data = x)

## Tests based on partial autocorrelations are also available:
plot(x.pacf)
plot(x.pacf, data = x)
```

`acfIidTest` *Carry out IID tests using sample autocorrelations*

Description

Carry out tests for IID from sample autocorrelations.

Usage

```
acfIidTest(acf, n, npar = 0, nlags = npar + 1,
           method = c("LiMcLeod", "LjungBox", "BoxPierce"),
           interval = 0.95, expandCI = TRUE)
```

Arguments

`acf`	autocorrelations.
`n`	length of the corresponding time series.
`npar`	number of df to subtract.
`nlags`	number of autocorrelations to use for the portmonteau statistic, can be a vector to request several such statistics.
`method`	a character string, one of "LiMcLeod", "LjungBox" or "BoxPierce".
`interval`	a number or NULL.
`expandCI`	logical flag, if `TRUE` return a CI for each lag up to `max(nlags)`. Used only if CI's are requested.

Details

Performs one of several tests for IID based on sample autocorrelations. A correction of the degerees of freedom for residuals from fitted models can be specified with argument `npar`. `nlags` specifies the number of autocorrelations to use in the test, it can be a vector to request several tests.

The results of the test are gathered in a matrix with one row for each element of `nlags`. The test statistic is in column "ChiSQ", degrees of freedom in "DF" and the p-value in "pvalue". The method is in attribute "method".

If `interval` is not `NULL` confidence intervals for the autocorrelations are computed, under the null hypothesis of independence. The coverage probability (or probabilities) is speciified by `interval`.

If argument `expandCI` is `TRUE`, there is one row for each lag, up to `max(nlags)`. It is best to use this feature with a single coverage probability.

If `expandCI` to `FALSE` the confidence intervals are put in a matrix with one row for each coverage probability.

Value

a list with components "test" and (if requested) "ci", as described in Details

Author(s)

Georgi N. Boshnakov

Examples

```
ts1 <- rnorm(100)

a1 <- drop(acf(ts1)$acf)
acfIidTest(a1, n = 100, nlags = c(5, 10, 20))
acfIidTest(a1, n = 100, nlags = c(5, 10, 20), method = "LjungBox")
acfIidTest(a1, n = 100, nlags = c(5, 10, 20), interval = NULL)
acfIidTest(a1, n = 100, method = "LjungBox", interval = c(0.95, 0.90), expandCI = FALSE)


## acfIidTest() is called behind the scenes by methods for autocorrelation objects
ts1_acrf <- autocorrelations(ts1)
class(ts1_acrf)  # "SampleAutocorrelations"
whiteNoiseTest(ts1_acrf, h0 = "iid", nlags = c(5,10,20), method = "LiMcLeod")
plot(ts1_acrf)
```

---

armaccf_xe *Crosscovariances of ARMA process and its innovations*

Description

Crosscovariances of ARMA process and its innovations.

Usage

```
armaccf_xe(model, lag.max = 1)
armaacf(model, lag.max, compare)
```

Arguments

model	the model, a list with components ar, ma and sigmasq.
lag.max	maximal lag for the result.
compare	if TRUE compute the autocovariances also using tacvfARMA() and return both results for comparison.

Details

Given an ARMA model, armaccf_xe computes theoretical crosscovariances between the ARMA process and its innovations. This is a simple illustration of the equations I give in my time series courses.

armaacf computes ARMA autocovariances. The default method computes computes the zero lag autocovariance using armaccf_xe() and multiplies the autocorrelations obtained from ARMAacf. If

`compare = TRUE` it also uses `tacvfARMA` from package **ltsa** and returns both results in a matrix for comparison. The matrix has columns `"native"`, `"tacvfARMA"` and `"difference"`, where the last column contains the (zapped) differences between the autocorrelations obtained by the two methods.

The ARMA parameters in argument `model` follow the Brockwell-Davis convention for the signs. Since `tacvfARMA()` uses the Box-Jenkins convention for the signs, the moving average parameters are negated for calls to `tacvfARMA()`.

Value

for `armaccf_xe`, the crosscovariances for lags 0, ..., maxlag.

for `armaacf`, the autocovariances, see Details.

Note

The built-in R function `ARMAacf()` computes autocorrelations, not the autocovariances.

Examples

```
## Example 1 from ?ltsa::tacvfARMA
z <- sqrt(sunspot.year)
n <- length(z)
p <- 9
q <- 0
ML <- 5
out <- arima(z, order = c(p, 0, q))

phi <- theta <- numeric(0)
if (p > 0) phi <- coef(out)[1:p]
if (q > 0) theta <- coef(out)[(p+1):(p+q)]
zm <- coef(out)[p+q+1]
sigma2 <- out$sigma2

armaacf(list(ar = phi, ma = theta, sigmasq = sigma2), lag.max = 20)
## this illustrates that the methods
## based on ARMAacf and tacvARMA are equivalent:
armaacf(list(ar = phi, ma = theta, sigmasq = sigma2), lag.max = 20, compare = TRUE)

## In the original example
## the comparison is with var(z), not with the theoretical variance:
rA <- ltsa::tacvfARMA(phi, - theta, maxLag=n+ML-1, sigma2=sigma2)
rB <- var(z) * ARMAacf(ar=phi, ma=theta, lag.max=n+ML-1)
## so rA and rB are different.
## but the difference is due to the variance:
rB2 <- rA[1] * ARMAacf(ar=phi, ma=theta, lag.max=n+ML-1)
cbind(rA[1:5], rB[1:5], rB2[1:5])
```

`ArmaModel-class` *Classes ArmaModel, ArModel and MaModel in package sarima*

Description

Classes ArmaModel, ArModel and MaModel in package sarima.

Objects from the Class

Objects can be created by calls of the form `new("ArmaModel", ..., ar, ma, sar, sma, mean)`.

Classes `"ArModel"` and `"MaModel"` are subclasses of `"ArmaModel"` with the corresponding order always zero.

Slots

`center`: Object of class `"numeric"` ~~

`intercept`: Object of class `"numeric"` ~~

`sigma2`: Object of class `"numeric"` ~~

`ar`: Object of class `"BJFilter"` ~~

`ma`: Object of class `"SPFilter"` ~~

Extends

Class `"ArmaSpec"`, directly. Class `"VirtualArmaModel"`, directly. Class `"ArmaFilter"`, by class "ArmaSpec", distance 2. Class `"VirtualFilterModel"`, by class "VirtualArmaModel", distance 2. Class `"VirtualStationaryModel"`, by class "VirtualArmaModel", distance 2. Class `"VirtualArmaFilter"`, by class "ArmaSpec", distance 3. Class `"VirtualAutocovarianceModel"`, by class "VirtualArmaModel", distance 3. Class `"VirtualMeanModel"`, by class "VirtualArmaModel", distance 3. Class `"VirtualMonicFilter"`, by class "ArmaSpec", distance 4.

Methods

modelOrder `signature(object = "ArmaModel", convention = "ArFilter")`: ...

modelOrder `signature(object = "ArmaModel", convention = "MaFilter")`: ...

modelOrder `signature(object = "ArmaModel", convention = "missing")`: ...

modelOrder `signature(object = "SarimaModel", convention = "ArmaModel")`: ...

sigmaSq `signature(object = "ArmaModel")`: ...

Examples

```
arma1p1 <- new("ArmaModel", ar = 0.5, ma = 0.9, sigma2 = 1)
autocovariances(arma1p1, maxlag = 10)
autocorrelations(arma1p1, maxlag = 10)
partialAutocorrelations(arma1p1, maxlag = 10)
partialAutocovariances(arma1p1, maxlag = 10)
```

```
new("ArmaModel", ar = 0.5, ma = 0.9, intercept = 4)
new("ArmaModel", ar = 0.5, ma = 0.9, center = 1.23)

new("ArModel", ar = 0.5,  center = 1.23)
new("MaModel", ma = 0.9,  center = 1.23)

# argument 'mean' is an alias for 'center':
new("ArmaModel", ar = 0.5, ma = 0.9, mean = 1.23)

##  both center and intercept may be given
## (the mean is not equal to the intercept in this case)
new("ArmaModel", ar = 0.5, ma = 0.9, center = 1.23, intercept = 2)

## Don't use 'mean' together with 'cener' and/or 'intercept'.
##    new("ArmaModel", ar = 0.5, ma = 0.9, center = 1.23, mean  = 4)
##    new("ArmaModel", ar = 0.5, ma = 0.9, intercept = 2, mean  = 4)
## Both give error message:
##   Use argument 'mean' only when 'center' and 'intercept' are missing or zero
```

`autocorrelations` *Compute autocorrelations and related quantities*

Description

Generic functions for computation of autocorrelations, autocovariances and related qqunatities.

Usage

```
autocovariances(x, maxlag, ...)

autocorrelations(x, maxlag, lag_0, ...)

partialAutocorrelations(x, maxlag, lag_0 = TRUE, ...)

partialAutocovariances(x, maxlag, ...)

partialVariances(x, ...)
```

Arguments

`x`	an object for which the requested property makes sense.
`maxlag`	the maximal lag to include in the result.
`lag_0`	if TRUE include lag zero.
`...`	further arguments for methods.

Details

autocorrelations is a generic function for computation of autocorrelations. It deduces the appropriate type of autocorrelation from the class of the object. For example, for models it computes theoretical autocorrelations, while for time series it computes sample autocorrelations.

The other functions described are similar for other second order properties of x.

These functions return objects from suitable classes. A value for lag zero is included (and is accessed by r[0]). Functions computing autocorrelations and partial autocorrelations have argument lag_0 — if it is set to FALSE, the value for lag zero is dropped from the result and the returned object is an ordinary vector or array, as appropriate.

See the individual methods for the format of the result and further details.

Value

an object from a class suitable for the requested property and x

Author(s)

Georgi N. Boshnakov

Examples

```
v1 <- rnorm(100)
autocorrelations(v1)
v1.acf <- autocorrelations(v1, maxlag = 10)

v1.acf[1:10] # drop lag zero value (and the class)
autocorrelations(v1, maxlag = 10, lag_0 = FALSE) # same

partialAutocorrelations(v1)
partialAutocorrelations(v1, maxlag = 10)

autocovariances(v1)
autocovariances(v1, maxlag = 10)
partialAutocovariances(v1, maxlag = 6)
partialAutocovariances(v1)
partialVariances(v1, maxlag = 6)
pv1 <- partialVariances(v1)

autocovariances(AirPassengers, maxlag = 6)
autocorrelations(AirPassengers, maxlag = 6)
partialAutocorrelations(AirPassengers, maxlag = 6)
partialAutocovariances(AirPassengers, maxlag = 6)
partialVariances(AirPassengers, maxlag = 6)
```

autocorrelations-methods

Methods for function autocorrelations()

Description

Methods for function autocorrelations().

Methods

```
signature(x = "ANY", maxlag = "ANY", lag_0 = "ANY")
signature(x = "ANY", maxlag = "ANY", lag_0 = "missing")
signature(x = "Autocorrelations", maxlag = "ANY", lag_0 = "missing")
signature(x = "Autocorrelations", maxlag = "missing", lag_0 = "missing")
signature(x = "Autocovariances", maxlag = "ANY", lag_0 = "missing")
signature(x = "PartialAutocorrelations", maxlag = "ANY", lag_0 = "missing")
signature(x = "PartialAutocovariances", maxlag = "ANY", lag_0 = "missing")
signature(x = "SamplePartialAutocorrelations", maxlag = "ANY", lag_0 = "missing")

signature(x = "SamplePartialAutocovariances", maxlag = "ANY", lag_0 = "missing")

signature(x = "VirtualArmaModel", maxlag = "ANY", lag_0 = "missing")
```

autocovariances-methods

Methods for function autocovariances()

Description

Methods for function autocovariances().

Methods

```
signature(x = "ANY")
signature(x = "VirtualArmaModel")
```

filterCoef *Coefficients and other basic properties of filters*

Description

Coefficients and other basic properties of filters.

Usage

```
filterCoef(object, convention, ...)

filterOrder(object, ...)

filterPoly(object, ...)

filterPolyCoef(object, lag_0 = TRUE, ...)
```

Arguments

object	object.
convention	convention for the sign.
lag_0	if FALSE, drop the constant term.
...	further arguments for methods.

Details

Generic functions to extract basic properties of filters: `filterCoef` returns coefficients, `filterOrder` returns the order, `filterPoly`, returns the characteristic polynomial, `filterPolyCoef` gives the coefficients of the characteristic polynomial.

What exactly is returned depends on the specific filter classes, see the description of the corresponding methods. For the core filters, the values are as can be expected. For "ArmaFilter", the value is a list with components "ar" and "ma" giving the requested property for the corresponding part of the filter. Similarly, for "SarimaFilter" the values are lists, maybe with additional quantities.

Value

the requested property as described in Details.

filterCoef-methods *Methods for filterCoef()*

Description

Methods for filterCoef().

Methods

```
signature(object = "BJFilter", convention = "character")
signature(object = "MonicFilterSpec", convention = "missing")
signature(object = "SPFilter", convention = "character")
signature(object = "VirtualArmaFilter", convention = "character")
signature(object = "VirtualArmaFilter", convention = "missing")
signature(object = "VirtualBJFilter", convention = "character")
signature(object = "VirtualSPFilter", convention = "character")
signature(object = "SarimaFilter", convention = "ANY")
signature(object = "SarimaFilter", convention = "character")
```

filterOrder-methods *Methods for filterOrder()*

Description

Methods for filterOrder().

Methods

```
signature(object = "MonicFilterSpec")
signature(object = "SarimaFilter")
signature(object = "VirtualArmaFilter")
```

filterPoly-methods *Methods for filterPoly*

Description

Methods for filterPoly.

Methods

signature(object = "BJFilter")

signature(object = "MonicFilterSpec")

signature(object = "SarimaFilter")

signature(object = "SPFilter")

signature(object = "VirtualArmaFilter")

filterPolyCoef-methods *Methods for filterPolyCoef*

Description

Methods for filterPolyCoef.

Methods

signature(object = "BJFilter")

signature(object = "SarimaFilter")

signature(object = "SPFilter")

signature(object = "VirtualArmaFilter")

signature(object = "VirtualBJFilter")

signature(object = "VirtualSPFilter")

`fun.forecast` *Forecasting functions for seasonal ARIMA models*

Description

Forecasting functions for seasonal ARIMA models.

Usage

```
fun.forecast(past, n = max(2 * length(past), 12), eps = numeric(n), pasteps, ...)
```

Arguments

`past`	past values of the time series, by default zeroes.
`n`	number of forecasts to compute.
`eps`	values of the white noise sequence (for simulation of future).
`pasteps`	past values of the white noise sequence for models with MA terms, 0 by default.
`...`	specification of the model, passed to `new()` to create a `"SarimaModel"` object, see Details.

Details

`fun.forecast` computes predictions from a SARIMA model. The model is specified using the "`...`" arguments which are passed to `new("SarimaModel", ...)`, see the description of class `"SarimaModel"` for details.

Argument `past`, if provided, should contain a least as many values as needed for the prediction equation. It is harmless to provide more values than necessary, even a whole time series.

`fun.forecast` can be used to illustrate, for example, the inherent difference for prediction of integrated and seasonally integrated models to corresponding models with roots close to the unit circle.

Value

the forecasts as an object of class "ts"

Author(s)

Georgi N. Boshnakov

Examples

```
f1 <- fun.forecast(past=1,n=100,ar=c(0.85),center=5)
plot(f1)

f2 <- fun.forecast(past=8,n=100,ar=c(0.85),center=5)
plot(f2)
```

```
f3 <- fun.forecast(past=10,n=100,ar=c(-0.85),center=5)
plot(f3)

frw1 <- fun.forecast(past=1,n=100,iorder = 1)
plot(frw1)

frw2 <- fun.forecast(past=3,n=100,iorder = 1)
plot(frw2)

frwa1 <- fun.forecast(past=c(1,2),n=100,ar=c(0.85),iorder = 1)
plot(frwa1)

fi2a <- fun.forecast(past=c(3,1),n=100,iorder = 2)
plot(fi2a)

fi2b <- fun.forecast(past=c(1,3),n=100,iorder = 2)
plot(fi2b)

fari1p2 <- fun.forecast(past=c(0,1,3),ar=c(0.9),n=20,iorder = 2)
plot(fari1p2)

fsi1 <- fun.forecast(past=rnorm(4),n=100,siorder = 1,nseasons=4)
plot(fsi1)

fexa <- fun.forecast(past=rnorm(5),n=100,ar=c(0.85),siorder = 1,nseasons=4)
plot(fexa)

fi2a <- fun.forecast(past=rnorm(24,sd=5),n=120,siorder = 2,nseasons=12)
plot(fi2a)

fi1si1a <- fun.forecast(past=rnorm(24,sd=5),n=120,iorder = 1,siorder = 1,nseasons=12)
plot(fi1si1a)

fi1si1a <- fun.forecast(past=AirPassengers[120:144],n=120,iorder = 1,siorder = 1,nseasons=12)
plot(fi1si1a)


m1 <- list(iorder = 1, siorder = 1, ma=0.8, nseasons=12)
m1
x <- sim_sarima(model=m1,n=500)
acf(diff(diff(x),lag=12), lag.max=96)
pacf(diff(diff(x),lag=12), lag.max=96)

m2 <- list(iorder = 1, siorder = 1, ma=0.8, sma=0.5, nseasons=12)
m2
x2 <- sim_sarima(model=m2, n=500)
acf(diff(diff(x2),lag=12), lag.max=96)
pacf(diff(diff(x2),lag=12), lag.max=96)
fit2 <- arima(x2, order=c(0,1,1), seasonal=list(order=c(0,1,0), nseasons=12))
fit2
tsdiag(fit2)
tsdiag(fit2,gof.lag=96)
```

```
x2past <- rnorm(13, sd=10)
x2 <- sim_sarima(model=m2, n=500, x = list(init = x2past))
plot(x2)

fun.forecast(ar=0.5, n=100)
fun.forecast(ar=0.5, n=100, past=1)
fun.forecast(ma=0.5, n=100, past=1)
fun.forecast(iorder = 1, ma=0.5, n=100, past=1)
fun.forecast(iorder = 1, ma=0.5, ar=0.8, n=100, past=1)

fun.forecast(m1, n=100)
fun.forecast(m2, n=100)
fun.forecast(iorder = 1, ar=0.8, ma=0.5, n=100, past=1)
```

`modelCoef` *Get the coefficients of models*

Description

Get the coefficients of an object, optionally specifying the expected format.

Usage

```
modelCoef(object, convention, ...)
```

Arguments

`object`	an object.
`convention`	the convention to use for the return value, a character string or any object from a supported class, see Details.
`...`	not used, further arguments for methods.

Details

`modelCoef` is a generic function for extraction of coefficients of model objects. What 'coeffcients' means depends on the class of `object` but it can be changed with the optional argument `convention`

The one-argument form, `modelCoef(object)`, gives the coefficients of `object`. In effect, it defines the meaning of 'coefficients' for the purposes of `modelCoef`.

Argument `convention` can be used to specify what kind of value to return.

If `convention` is not a character string, only its class is used and the value will have the format and meaning of the value that would be returned by a call `modelCoef(convention)`.

If `convention` is a character string, it is typically the name of a class. In this case `modelCoef(object, "someclass")` is equivalent to `modelCoef(object, new("someclass"))`. For some classes of `object` character values other than names of classes may be supported.

Value

the value is defined by the methods as described in Details

modelCoef-methods *Methods for generic function modelCoef*

Description

Methods for generic function modelCoef.

Methods

```
signature(object = "Autocorrelations", convention = "ComboAutocorrelations")
signature(object = "Autocorrelations", convention = "PartialAutocorrelations")
signature(object = "Autocovariances", convention = "Autocorrelations")
signature(object = "Autocovariances", convention = "ComboAutocorrelations")
signature(object = "Autocovariances", convention = "ComboAutocovariances")
signature(object = "Autocovariances", convention = "PartialAutocorrelations")
signature(object = "ComboAutocorrelations", convention = "Autocorrelations")
signature(object = "ComboAutocorrelations", convention = "PartialAutocorrelations")

signature(object = "ComboAutocovariances", convention = "Autocovariances")
signature(object = "ComboAutocovariances", convention = "PartialAutocovariances")

signature(object = "ComboAutocovariances", convention = "PartialVariances")
signature(object = "ComboAutocovariances", convention = "VirtualAutocovariances")

signature(object = "PartialAutocorrelations", convention = "Autocorrelations")
signature(object = "SarimaModel", convention = "ArFilter")
signature(object = "SarimaModel", convention = "ArmaFilter")
signature(object = "SarimaModel", convention = "character")
signature(object = "SarimaModel", convention = "MaFilter")
signature(object = "SarimaModel", convention = "missing")
signature(object = "SarimaModel", convention = "SarimaFilter")
signature(object = "VirtualArmaModel", convention = "character")
signature(object = "VirtualArmaModel", convention = "missing")
signature(object = "VirtualAutocovariances", convention = "character")
signature(object = "VirtualAutocovariances", convention = "missing")
signature(object = "VirtualAutocovariances", convention = "VirtualAutocovariances")

signature(object = "PartialAutocovariances", convention = "PartialAutocorrelations")
```

`modelOrder` *Get the model order and other properties of models*

Description

Get the model order and other properties of models.

Usage

```
modelOrder(object, convention, ...)

modelPoly(object, convention, ...)

modelPolyCoef(object, convention, lag_0 = TRUE, ...)
```

Arguments

`object`	a model object.
`convention`	convention.
`lag_0`	if TRUE include lag_0 coef, otherwise drop it.
`...`	further arguments for methods.

Details

These functions return the requested quantity, optionally requesting the returned value to follow a specific convention, see also `modelCoef`.

When called with one argument, these functions return corresponding property in the native format for the object's class.

Argument `convention` requests the result in some other format. The mental model is that the returned value is as if the object was first converted to the class specified by `convention` and then the property extracted or computed. Normally, the object is not actually converted to that class. one obvious reason is efficiency but it may also not be possible, for example if argument `convention` is the name of a virtual class.

For example, the order of a seasonal SARIMA model is specified by several numbers. The call `modelOrder(object)` returns it as a list with components ar, ma, sar, sma, iorder, siorder and nseasons. For some computations all that is needed are the overall AR and MA orders obtained by multiplying out the AR-like and MA-like terms in the model. The result would be an ARMA filter and could be requested by `modelOrder(object, "ArmaFilter")`.

The above operation is valid for any ARIMA model, so will always succeed. On the other hand, if further computation would work only if there are no moving average terms in the model one could use `modelOrder(object, "ArFilter")`. Here, if `object` contains MA terms an error will be raised.

The concept is powerful and helps in writing expressive code. In this example a simple check on the returned value would do but even so, such a check may require additional care.

See Also

modelCoef

`modelOrder-methods` *Get the order of a model*

Description

Get the order of a model.

Methods

```
signature(object = "ArmaModel", convention = "ArFilter")
signature(object = "ArmaModel", convention = "MaFilter")
signature(object = "ArmaModel", convention = "missing")
signature(object = "SarimaModel", convention = "ArFilter")
signature(object = "SarimaModel", convention = "ArmaFilter")
signature(object = "SarimaModel", convention = "ArmaModel")
signature(object = "SarimaModel", convention = "ArModel")
signature(object = "SarimaModel", convention = "MaFilter")
signature(object = "SarimaModel", convention = "MaModel")
signature(object = "SarimaModel", convention = "missing")
```

`modelPoly-methods` *Get polynomials associated with SARIMA models*

Description

Get polynomials associated with SARIMA models.

Methods

```
signature(object = "SarimaModel", convention = "ArmaFilter")
signature(object = "SarimaModel", convention = "missing")
```

modelPolyCoef-methods *Methods for modelPolyCoef*

Description

Methods for modelPolyCoef, e generic function for getting the coefficients of polynomials associated with SARIMA models.

Methods

```
signature(object = "SarimaModel", convention = "ArmaFilter")
signature(object = "SarimaModel", convention = "missing")
```

partialAutocorrelations-methods *Methods for function partialAutocorrelations*

Description

Methods for function partialAutocorrelations.

Methods

```
signature(x = "ANY", maxlag = "ANY", lag_0 = "ANY")
signature(x = "mts", maxlag = "ANY", lag_0 = "missing")
signature(x = "PartialAutocovariances", maxlag = "ANY", lag_0 = "missing")
signature(x = "ts", maxlag = "ANY", lag_0 = "missing")
```

plot-methods *Plot methods in package sarima*

Description

Plot methods in package sarima.

Methods

```
signature(x = "SampleAutocorrelations", y = "matrix")
signature(x = "SampleAutocorrelations", y = "missing")
signature(x = "SamplePartialAutocorrelations", y = "missing")
```

Examples

```
n <- 5000
x <- sarima:::rgarch1p1(n, alpha = 0.3, beta = 0.55, omega = 1, n.skip = 100)
x.acf <- autocorrelations(x)
x.acf
x.pacf <- partialAutocorrelations(x)
x.pacf

plot(x.acf)
plot(x.acf, data = x)

plot(x.pacf)
plot(x.pacf, data = x)

plot(x.acf, data = x, main = "Autocorrelation test")
plot(x.pacf, data = x, main = "Partial autocorrelation test")

plot(x.acf, ylim = c(NA,1))
plot(x.acf, ylim.fac = 1.5)
plot(x.acf, data = x, ylim.fac = 1.5)
plot(x.acf, data = x, ylim = c(NA, 1))
```

---

prepareSimSarima *Prepare SARIMA simulations*

Description

Prepare SARIMA simulations.

Usage

```
prepareSimSarima(model, x = NULL, eps = NULL, n, n.start = NA,
                 xintercept = NULL, rand.gen = rnorm)

## S3 method for class 'simSarimaFun'
print(x, ...)
```

Arguments

model	a list specifying the SARIMA model.
x	initial/before values of the time series, a list or a numeric vector, see Details.
eps	initial/before values of the innovations, a list or a numeric vector, see Details.
n	number of observations to generate, if missing an attempt is made to infer it from x and eps.
n.start	number of burin-in observations.
xintercept	non-constant intercept which may represent trend or covariate effects.
rand.gen	random number generator, defaults to N(0,1).
...	ignored.

Details

prepareSimSarima does the preparatory work for simulation from a Sarima model, given the specifications and returns a function, which can be called as many times as needed.

The variance of the innovations is specified by the model and the simulated innovations are multiplied by the corresponding standard deviation. So, it is expected that the random number generator simulates from a standardised distribution.

print.simSarimaFun is a print method for the objects generated by prepareSimSarima.

Value

for prepareSimSarima, a function to simulate time series

Examples

```
mo1 <- list(ar=0.9, iorder = 1, siorder = 1, nseasons = 4, sigma2 = 2)
fs1 <- prepareSimSarima(mo1, x = list(before = rep(0,6)),  n = 100)
tmp1 <- fs1()
plot(ts(tmp1))

fs2 <- prepareSimSarima(mo1, x = list(before = rep(1,6)),  n = 100)
tmp2 <- fs2()
plot(ts(tmp2))

mo3 <- mo1
mo3[["ar"]] <- 0.5
fs3 <- prepareSimSarima(mo3, x = list(before = rep(0,6)),  n = 100)
tmp3 <- fs3()
plot(ts(tmp3))
```

SarimaModel-class *Class SarimaModel in package sarima*

Description

Class SarimaModel in package sarima.

Objects from the Class

Objects can be created by calls of the form new("SarimaModel", ..., ar, ma, sar, sma).

Slots

center: Object of class "numeric" ~~

intercept: Object of class "numeric" ~~

sigma2: Object of class "numeric" ~~

nseasons: Object of class "numeric" ~~

`iorder`: Object of class "numeric" ~~

`siorder`: Object of class "numeric" ~~

`ar`: Object of class "BJFilter" ~~

`ma`: Object of class "SPFilter" ~~

`sar`: Object of class "BJFilter" ~~

`sma`: Object of class "SPFilter" ~~

Extends

Class "VirtualFilterModel", directly. Class "SarimaSpec", directly. Class "SarimaFilter", by class "SarimaSpec", distance 2. Class "VirtualSarimaFilter", by class "SarimaSpec", distance 3. Class "VirtualCascadeFilter", by class "SarimaSpec", distance 4. Class "VirtualMonicFilter", by class "SarimaSpec", distance 5.

Methods

modelCoef `signature(object = "SarimaModel", convention = "ArFilter")`: ...

modelCoef `signature(object = "SarimaModel", convention = "ArmaFilter")`: ...

modelCoef `signature(object = "SarimaModel", convention = "character")`: ...

modelCoef `signature(object = "SarimaModel", convention = "MaFilter")`: ...

modelCoef `signature(object = "SarimaModel", convention = "missing")`: ...

modelCoef `signature(object = "SarimaModel", convention = "SarimaFilter")`: ...

modelOrder `signature(object = "SarimaModel", convention = "ArFilter")`: ...

modelOrder `signature(object = "SarimaModel", convention = "ArmaFilter")`: ...

modelOrder `signature(object = "SarimaModel", convention = "ArmaModel")`: ...

modelOrder `signature(object = "SarimaModel", convention = "ArModel")`: ...

modelOrder `signature(object = "SarimaModel", convention = "MaFilter")`: ...

modelOrder `signature(object = "SarimaModel", convention = "MaModel")`: ...

modelOrder `signature(object = "SarimaModel", convention = "missing")`: ...

modelPolyCoef `signature(object = "SarimaModel", convention = "ArmaFilter")`: ...

modelPolyCoef `signature(object = "SarimaModel", convention = "missing")`: ...

modelPoly `signature(object = "SarimaModel", convention = "ArmaFilter")`: ...

modelPoly `signature(object = "SarimaModel", convention = "missing")`: ...

See Also

ArmaModel

Examples

```
new("SarimaModel", nseasons = 12)
new("SarimaModel", nseasons = 12, intercept = 3)
new("SarimaModel", ar = 0.9, nseasons = 12, intercept = 3)
new("SarimaModel", ar = 0.9, sar= 0.8, nseasons = 12, intercept = 3)

showClass("SarimaModel")
```

sigmaSq-methods *Get the innovation variance of models*

Description

Get the innovation variance of models.

Usage

```
sigmaSq(object)
```

Arguments

object	an object from a suitable class.

Details

`sigmaSq()` gives the innovation variance of objects from classes for which it makes sense, such as ARMA models.

The value depends on the class of the object, e.g. for ARMA models it is a scalar in the univariate case and a matrix in the multivariate one.

Methods

```
signature(object = "ArmaModel")
signature(object = "SarimaSpec")
```

sim_sarima *Simulate trajectories of seasonal arima models*

Description

Simulate trajectories of seasonal arima models.

Usage

```
sim_sarima(model, n = NA, rand.gen = rnorm, n.start = NA, x, eps,
           xcenter = NULL, xintercept = NULL, ...)
```

Arguments

model	specification of the model, a list, see 'Details'.
rand.gen	random number generator for the innovations.
...	additional arguments for arima.sim and rand.gen, see 'Details'.
n	~~ TODO: describe this argument. ~~
n.start	~~ TODO: describe this argument. ~~
x	~~ TODO: describe this argument. ~~
eps	~~ TODO: describe this argument. ~~
xcenter	~~ TODO: describe this argument. ~~
xintercept	~~ TODO: describe this argument. ~~

Details

The model is specified by the argument "model" which is a list with elements suitable to be passed to new("SarimaModel", ...), see the description of class "SarimaModel". Here are some of the possible components:

nseasons number of seasons in a year (or whatever is the larger time unit)

iorder order of differencing, specifies the factor $(1 - B)^{d1}$ for the model.

siorder order of seasonal differencing, specifies the factor $(1 - B^{period})^{ds}$ for the model.

ar ar parameters (non-seasonal)

ma ma parameters (non-seasonal)

sar seasonal ar parameters

sma seasonal ma parameters

Additional arguments for rand.gen may be specified via the "..." argument. In particular, the length of the generated series is specified with argument n. Arguments for rand.gen can also be passed via the "..." argument.

sim_sarima calls internally arima.sim to simulate the ARMA part of the model. Then undifferences the result to obtain the end result.

The function returns the simulated time series from the requested model.

Information about the model is printed on the screen if info="print". To suppress this, set info to any other value.

Value

an object of class "ts"

Author(s)

Georgi N. Boshnakov

Examples

```
require("PolynomF") # package "sarima" imports it, so should not be absent here.

x <- sim_sarima(n=144, model = list(ma=0.8))             # MA(1)
x <- sim_sarima(n=144, model = list(ar=0.8))               # AR(1)

x <- sim_sarima(n=144, model = list(ar=c(rep(0,11),0.8)))   # SAR(1), 12  seasons
x <- sim_sarima(n=144, model = list(ma=c(rep(0,11),0.8))) # SMA(1)

                                        # more enlightened SAR(1) and SMA(1)
x <-sim_sarima(n=144,model=list(sar=0.8, nseasons=12))    # SAR(1), 12  seasons
x <-sim_sarima(n=144,model=list(sma=0.8, nseasons=12))  # SMA(1)

x <- sim_sarima(n=144, model = list(iorder=1)) # (1-B)X_t = e_t (random walk)
acf(x)
acf(diff(x))

x <-sim_sarima(n=144, model = list(iorder=2))            # (1-B)^2 X_t = e_t
x <-sim_sarima(n=144, model = list(siorder=1, nseasons=12)) # (1-B)^{12} X_t = e_t

x <- sim_sarima(n=144, model = list(iorder=1, siorder=1, nseasons=12))
x <- sim_sarima(n=144, model = list(ma=0.4, iorder=1, siorder=1, nseasons=12))
x <- sim_sarima(n=144, model = list(ma=0.4, sma=0.7, iorder=1, siorder=1, nseasons=12))

x <- sim_sarima(n=144, model = list(ar=c(1.2,-0.8), ma=0.4,
                                    sar=0.3, sma=0.7, iorder=1,siorder=1,nseasons=12))

x <- sim_sarima(n=144, model = list(iorder=1, siorder=1, nseasons=12),
                x = list(init=AirPassengers[1:13]))


p <- polynom(c(1,-1.2,0.8))
solve(p)
abs(solve(p))

sim_sarima(n=144, model = list(ar=c(1.2,-0.8), ma=0.4,
                               sar=0.3, sma=0.7, iorder=1, siorder=1, nseasons=12))

x <- sim_sarima(n=144, model=list(ma=0.4, iorder=1, siorder=1, nseasons=12))
acf(x, lag.max=48)
x <- sim_sarima(n=144, model=list(sma=0.4, iorder=1, siorder=1, nseasons=12))
acf(x, lag.max=48)
x <- sim_sarima(n=144, model=list(sma=0.4, iorder=0, siorder=0, nseasons=12))
acf(x, lag.max=48)
x <- sim_sarima(n=144, model=list(sar=0.4, iorder=0, siorder=0, nseasons=12))
acf(x, lag.max=48)
x <- sim_sarima(n=144, model=list(sar=-0.4, iorder=0, siorder=0, nseasons=12))
acf(x, lag.max=48)

x <- sim_sarima(n=144, model=list(ar=c(1.2, -0.8), ma=0.4,
                                  sar=0.3, sma=0.7, iorder=1, siorder=1, nseasons=12))
```

summary.SarimaModel *Methods for summary in package sarima*

Description

Methods for summary in package sarima.

Usage

```
## S3 method for class 'SarimaModel'
summary(object, ...)
## S3 method for class 'SarimaFilter'
summary(object, ...)
## S3 method for class 'SarimaSpec'
summary(object, ...)
```

Arguments

object	an object from the corresponding class.
...	further arguments for methods.

VirtualMonicFilter-class *Undocumented classes in package sarima*

Description

This page is for classes without proper documentation.

Objects from the Class

A virtual Class: No objects may be created from it.

This page exists only to remind me which classes do not have documentation yet. It exists to avoid cluttering the report from 'R CMD check' during early stages of development.

Methods

No methods defined with class "VirtualMonicFilter" in the signature.

`whiteNoiseTest` *White noise tests*

Description

White noise tests.

Usage

```
whiteNoiseTest(object, h0, ...)
```

Arguments

`object`	an object, such as sample autocorrelations or partial autocorrelations.
`h0`	the null hypothesis, currently "iid" or "garch".
`...`	additional arguments passed on to methods.

Details

`whiteNoiseTest` carries out tests for white noise. Argument `h0` gives a way to identify the null hypothesis.

If `h0 = "iid"`, the test statistics and rejection regions can be use to test if the underlying time series is iid. Argument `method` specifies the method for portmanteau tests: one of "LiMcLeod" (default), "LjungBox", "BoxPierce".

If `h0 = "garch"`, the null hypothesis is that the time series is GARCH, see FrancqZakoian (TODO: give the reference properly!). The tests in this case are based on a non-parametric estimate of the asymptotic covariance matrix.

Portmonteau statistics and p-values are computed for the lags specified by argument `nlags`. If it is missing, suitable lags are chosen automatically.

If argument `interval` is TRUE, confidence intervals for the individual autocorrelations or partial autocorrelations are computed.

Value

a list with component `test` and, if `ci=TRUE`, component `ci`.

Examples

```
n <- 5000
x <- sarima:::rgarch1p1(n, alpha = 0.3, beta = 0.55, omega = 1, n.skip = 100)
x.acf <- autocorrelations(x)
x.pacf <- partialAutocorrelations(x)

x.iid <- whiteNoiseTest(x.acf, h0 = "iid", nlags = c(5,10,20), x = x, method = "LiMcLeod")
x.iid
```

```
x.iid2 <- whiteNoiseTest(x.acf, h0 = "iid", nlags = c(5,10,20), x = x, method = "LjungBox")
x.iid2

x.garch <- whiteNoiseTest(x.acf, h0 = "garch", nlags = c(5,10,20), x = x)
x.garch
```

`xarmaFilter` *Applies an extended ARMA filter to a time series*

Description

Filter time series with an extended arma filter. If `whiten` is `FALSE` (default) the function applies the given ARMA filter to `eps` (`eps` is often white noise). If `whiten` is `TRUE` the function applies the "inverse filter" to x, effectively computing residuals.

Usage

```
xarmaFilter(model, x = numeric(length(eps)), eps = numeric(length(x)),
            from = NULL, whiten = FALSE, xcenter = NULL,
            xintercept = NULL)
```

Arguments

`x`	the time series to be filtered, a vector.
`eps`	residuals, a vector or NULL.
`model`	the model parameters, a list with components `"ar"`, `"ma"`, `"center"` and `"intercept"`, see Details.
`from`	the index from which to start filtering.
`whiten`	if TRUE use `x` as input and apply the inverse filter to produce `eps` ("whiten" `x`), if FALSE use `eps` as input and generate `x` ("colour" `eps`).
`xcenter`	a vector of means of the same length as the time series, see Details.
`xintercept`	a vector of intercepts having the length of the series, see Details.

Details

The model is specified by argument `model`, which is a list with the following components:

`ar` the autoregression parameters,

`ma` the moving average parameters,

`center` center by this value,

`intercept` intercept.

The relation between x and eps is assumed to be the following. Let

$$y_t = x_t - \mu_t$$

be the centered series, where μ_t is obtained from center and xcenter and is not necessarilly the mean, see below. The equation relating the centered series, $y_t = x_t - \mu_t$, and eps is the following:

$$y_t = c_t + \sum_{i=1}^{p} \phi(i) y_{t-i} + \sum_{i=1}^{q} \theta(i) \varepsilon_{t-i} + \varepsilon_t$$

where c_t is the intercept (basically the sum of intercept with xintercept. The inverse filter is obtained by writing this as an equation expressing ε_t in terms of the remaining quantities.

If whiten = TRUE, xarmaFilter uses the above formula to compute the filtered values of x for t=from,...,n, i.e. whitening the time series if eps is white noise. If whiten = FALSE, eps is computed, i.e. the inverse filter is applied to get x from eps, i.e. "colouring" eps. In both cases the first few values in x and/or eps are used as initial values.

Essentially, the centering is subtracted from the series to obtain the centred series, say y. Then either y is filtered to obtain eps or the inverse filter is applied to obtain y from eps finally the mean is added back to y and the result returned.

The centering is formed from model$center and argument xcenter. If model$center is supplied it is recycled to the length of the series, x, and subtracted from x. If argument xcenter is supplied, it is subtracted from x. If both model$center and xcenter are supplied their sum is subtracted from x.

The above gives a vector y, $y_t = x_t - \mu_t$, which is then filtered. If the mean is zero, $y_t = x_t$ in the formulas below.

Finally, the mean is added back, $x_t = y_t + \mu_t$, and the new x is returned.

xarmaFilter can be used to simulate arma series with the default value of whiten=FALSE. In this case eps is the input series and y the output.

$$y_t = c_t + \sum_{i=1}^{p} \phi(i) y_{t-i} + \sum_{i=1}^{q} \theta(i) \varepsilon_{t-i} + \varepsilon_t$$

Then model$center and/or xcenter are added to y to form the output vector x.

Residuals corresponding to a series y can be obtained by setting whiten=TRUE. In this case y is the input series. The elements of the output vector eps are calculated by the formula:

$$\varepsilon_t = -c_t - \sum_{i=1}^{q} \theta(i) \varepsilon_{t-i} - \sum_{i=1}^{p} \phi(i) y_{t-i} + y_t$$

There is no need in this case to restore x since eps is returned.

In both cases any necessary initial values are assumed to be already in the vectors. If from is not supplied it is set to max(p,q)+1

xarmaFilter calls the lower level function coreXarmaFilter to do the computation.

Value

The filtered series: the modified x if whiten=FALSE, the modified eps if whiten=TRUE.

Author(s)

Georgi N. Boshnakov

Examples

```
m1 <- new("SarimaModel", iorder = 1, siorder = 1, ma = -0.3, sma = -0.1, nseasons = 12)
model0 <- modelCoef(m1, "ArmaModel")
## model1 <- filterCoef(model1)
model1 <- as(model0, "list")

ap.1 <- xarmaFilter(model1, x = AirPassengers, whiten = TRUE)
ap.2 <- xarmaFilter(model1, x = AirPassengers, eps = ap.1, whiten = FALSE)
ap <- AirPassengers
ap[-(1:13)] <- 0 # check that the filter doesn't use x, except for initial values.
ap.2a <- xarmaFilter(model1, x = ap, eps = ap.1, whiten = FALSE)
ap.2a - ap.2 ## indeed = 0
##ap.3 <- xarmaFilter(model1, x = list(init = AirPassengers[1:13]), eps = ap.1, whiten = TRUE)

## now set some non-zero initial values for eps
eps1 <- numeric(length(AirPassengers))
eps1[1:13] <- rnorm(13)
ap.A <- xarmaFilter(model1, x = AirPassengers, eps = eps1, whiten = TRUE)
ap.Ainv <- xarmaFilter(model1, x = ap, eps = ap.A, whiten = FALSE)
AirPassengers - ap.Ainv # = 0

## compare with sarima.f (an old function)
pred1 <- sarima.f(past = AirPassengers[1:13], n = 131, ar = model1$ar, ma = model1$ma)
pred2 <- xarmaFilter(model1, x = ap, eps = numeric(length(AirPassengers)), whiten = FALSE)
pred2 <- pred2[-(1:13)]
all(pred1 == pred2) ##TRUE
```

Index