Package 'sarima'

December 16, 2025

Type Package

Title Simulation and Prediction with Seasonal ARIMA Models

Version 0.9.5

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. Function sarima() fits extended multiplicative seasonal ARIMA models with trends, exogenous variables and arbitrary roots on the unit circle, which can be fixed or estimated (for the algebraic basis for this see <doi:10.48550/arXiv.2208.05055>, a paper on the methodology is being prepared).

URL https://geobosh.github.io/sarima/ (doc)

https://github.com/GeoBosh/sarima (devel)

BugReports https://github.com/GeoBosh/sarima/issues

Depends R (>= 2.10), methods, stats4

Imports graphics, stats, utils, PolynomF (>= 1.0-0), Formula, lagged (>= 0.2.1), Rcpp (>= 0.12.14), Rdpack, numDeriv, ltsa

Suggests testthat, KFAS, FKF, fGarch, forecast

RdMacros Rdpack

License GPL (>= 2)

LazyLoad yes

Collate RcppExports.R utils.R generics.R filterClasses.R modelClasses.R sarima.R autocovariances.R armacalc.R fit.R wrapKFAS.R arma_Q0dotdotstats.R Kalman.R fitTools.R periodogram.R predict.Sarima.R zzz.R

LinkingTo Rcpp, RcppArmadillo

RoxygenNote 7.1.1

NeedsCompilation yes

Author Georgi N. Boshnakov [aut, cre] (ORCID:
<https://orcid.org/0000-0003-2839-346X>),
Jamie Halliday [aut]

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

Repository CRAN

Date/Publication 2025-12-16 06:10:20 UTC

Contents

`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. Function sarima() fits extended multiplicative seasonal ARIMA models with trends, exogenous variables and arbitrary roots on the unit circle, which can be fixed or estimated (for the algebraic basis for this see <doi:10.48550/arXiv.2208.05055>, a paper on the methodology is being prepared).

Details

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** a consistent naming scheme is used as much as possible. Names of functions start with a lowercase letter and consist of whole words, acronyms or commonly used abbreviations. In multiword names, the second and subsequent words start with capital letters (camelCase). Only the first letter in acronyms is capitalised, e.g. Arma stands for ARMA. Formal (S4) classes follow the same rules but the first letter of the first word is capitalised, as well.

For example, the functions that compute autocorrelations, autocovariances, partial autocorrelations are called `autocorrelations`, `autocovariances`, and `partialAutocorrelations`, respectively. Moreover, they recognise from their argument(s) what exactly is needed. If they are given times series, they compute sample autocorrelations, etc; if they are given model specifications, they compute the corresponding theoretical properties.

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

Author(s)

Georgi N. Boshnakov [aut, cre] (ORCID: <https://orcid.org/0000-0003-2839-346X>), Jamie Halliday [aut]

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

References

Boshnakov GN (1996). "Bartlett's formulae—closed forms and recurrent equations." *Ann. Inst. Statist. Math.*, **48**(1), 49–59. ISSN 0020-3157, doi:10.1007/BF00049288.

Halliday J, Boshnakov GN (2022). "Partial autocorrelation parameterisation of models with unit roots on the unit circle." doi:10.48550/ARXIV.2208.05055, https://arxiv.org/abs/2208.05055.

Brockwell PJ, Davis RA (1991). *Time series: theory and methods. 2nd ed.*. Springer Series in Statistics. Berlin etc.: Springer-Verlag..

Francq C, Zakoian J (2010). *GARCH models: structure, statistical inference and financial applications*. John Wiley & Sons. ISBN 978-0-470-68391-0.

Li WK (2004). *Diagnostic checks in time series*. Chapman & Hall/CRC Press.

McLeod AI, Yu H, Krougly Z (2007). "Algorithms for Linear Time Series Analysis: With R Package." *Journal of Statistical Software*, **23**(5). doi:10.18637/jss.v023.i05.

See Also

ArmaModel autocorrelations

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.pacf <- partialAutocorrelations(x) )

## 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)

## Models
## AR
( ar2a1 <- ArModel(ar = c(-0.3, -0.7), sigma2 = 1) )
autocorrelations(ar2a1, maxlag = 6)
partialAutocorrelations(ar2a1, maxlag = 6)
autocovariances(ar2a1, maxlag = 6)
partialVariances(ar2a1, maxlag = 6)

## see examples for ArmaModel()
```

`acfGarchTest` *Tests for weak white noise*

Description

Carry out tests for weak white noise under GARCH, GARCH-type, and stochastic volatility null hypotheses.

Usage

```
acfGarchTest(acr, x, nlags, interval = 0.95)

acfWnTest(acr, x, nlags, interval = 0.95, ...)
```

Arguments

`acr`	autocorrelations.
`x`	time series.
`nlags`	how many lags to use.
`interval`	If not NULL, compute also confidence intervals with the specified coverage probability.
`...`	additional arguments for the computation of the variance matrix under the null hypothesis, passed on to `nvarOfAcfKP`.

Details

Unlike the autocorrelation IID test, the time series is needed here to estimate the covariance matrix of the autocorrelations under the null hypothesis.

`acfGarchTest` performs a test for uncorrelatedness of a time series. The null hypothesis is that the time series is GARCH, see Francq and Zakoian (2010).

`acfWnTest` performs a test for uncorrelatedness of a time series under a weaker null hypothesis. The null hypothesis is that the time series is GARCH-type or from a stochasitc volatily model, see Kokoszka and Politis (2011).

See the references for details and precise specification of the hypotheses.

The format of the return value is the same as for `acfIidTest`.

Value

a list with components "test" and "ci"

Author(s)

Georgi N. Boshnakov

References

Francq C, Zakoian J (2010). *GARCH models: structure, statistical inference and financial applications*. John Wiley & Sons. ISBN 978-0-470-68391-0.

Kokoszka PS, Politis DN (2011). "Nonlinearity of ARCH and stochastic volatility models and Bartlett's formula." *Probability and Mathematical Statistics*, **31**(1), 47–59.

See Also

`whiteNoiseTest`, `acfIidTest`;

`plot-methods` for graphical representations of results

Examples

```
## see also the examples for \code{\link{whiteNoiseTest}}
set.seed(1234)
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)

acfGarchTest(x.acf, x = x, nlags = c(5,10,20))
acfGarchTest(x.pacf, x = x, nlags = c(5,10,20))

# do not compute CI's:
acfGarchTest(x.pacf, x = x, nlags = c(5,10,20), interval = NULL)

## plot methods call acfGarchTest() suitably if 'x' is given:
plot(x.acf, data = x)
plot(x.pacf, data = x)

## use 90% limits:
plot(x.acf, data = x, interval = 0.90)

acfWnTest(x.acf, x = x, nlags = c(5,10,20))
```

```
nvarOfAcfKP(x, maxlag = 20)
whiteNoiseTest(x.acf, h0 = "arch-type", x = x, nlags = c(5,10,20))
```

---

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.
...	additional arguments passed on to methods. In particular, some methods have argument x for the time series.

Details

Performs one of several tests for IID based on sample autocorrelations. A correction of the degrees 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

Methods

`signature(acf = "ANY")` In this method `acf` contains the autocorrelations.

`signature(acf = "missing")` The autocorrelations are computed from argument `x` (the time series).

`signature(acf = "SampleAutocorrelations")` This is a convenience method in which argument `n` is taken from `acf` and thus does not need to be specified by the user.

Author(s)

Georgi N. Boshnakov

References

Li WK (2004). *Diagnostic checks in time series*. Chapman & Hall/CRC Press.

See Also

`whiteNoiseTest`, `acfGarchTest`, `acfMaTest`;

`plot-methods` for graphical representations of results

Examples

```
set.seed(1234)
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)

## use 10% level of significance in the plot:
plot(ts1_acrf, interval = 0.9)
```

`acfMaTest` *Autocorrelation test for MA(q)*

Description

Carry out autocorrelation test for MA(q).

Usage

```
acfMaTest(acf, ma, n, nlags, interval = 0.95)
```

Arguments

`acf`	autocorrelations.
`ma`	a positive integer, the moving average order.
`n`	length of the corresponding time series.
`nlags`	number of autocorrelations to use for the portmonteau statistic, can be a vector to request several such statistics.
`interval`	a number or NULL.

Details

`acfMaTest` performs a test that the time series is MA(`ma`), under the classical assumptions of Bartlett's formulas.

When intervals are requested, they are confidence intervals for lags from 1 to `ma`. For lags greater than the moving average order, `ma`, autocorrelations outside them suggest to reject the null hypothesis that the process is MA(`ma`).

Value

a list with components "test" and (if requested) "ci"

Author(s)

Georgi N. Boshnakov

See Also

`whiteNoiseTest`, `acfIidTest` `acfGarchTest`

armaccf_xe *Crosscovariances between an ARMA process and its innovations*

Description

Compute autocovariances of ARMA models and crosscovariances between an 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 sigma2 (for the time being, sigmasq is also accepted, if model$sigma2 is NULL).
lag.max	maximal lag for the result.
compare	if TRUE compute the autocovariances also using tacvfARMA() and return both results for comparison.

Details

Given a causal ARMA model, armaccf_xe computes theoretical crosscovariances $R_{xe}(0)$, $R_{xe}(1)$, $R_{xe}(lag.max)$, where $R_{xe}(k) = E(X_t e_{t-k})$, between an ARMA process and its innovations. Negative lags are not considered since $R_{xe}(k) = 0$ for $k < 0$. The moving average polynomial may have roots on the unit circle.

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 (which computes autocorrelations, not autocovariances). 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

armaacf is useful for exploratory computations but autocovariances is more convenient and eliminates the need to know function names for particular cases.

Author(s)

Georgi N. Boshnakov

References

McLeod AI, Yu H, Krougly Z (2007). "Algorithms for Linear Time Series Analysis: With R Package." *Journal of Statistical Software*, **23**(5). doi:10.18637/jss.v023.i05.

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

## In the original example in ?ltsa::tacvfARMA
## 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])

## There is no need to use specific functions,
## autocovariances() is most convenient for routine use:
armalist <- list(ar = phi, ma = theta, sigma2 = sigma2)
autocovariances(armalist, maxlag = 10)

## even better, set up an ARMA model:
mo <- new("ArmaModel", ar = phi, ma = theta, sigma2 = sigma2)
autocovariances(mo, maxlag = 10)
```

ArmaModel *Create ARMA objects*

Description

Create ARMA objects.

Usage

```
ArmaModel(...)
ArModel(...)
MaModel(...)
```

Arguments

...	the arguments are passed to new(). Typical arguments are ar, ma and mean.

Value

an object representing an ARMA, AR or MA model

Author(s)

Georgi N. Boshnakov

See Also

ArmaModel, ArModel, MaModel

Examples

```
## MA
( ma2a1 <- MaModel(ma = c(0.3, 0.7), sigma2 = 1) )
autocorrelations(ma2a1, maxlag = 6)
partialAutocorrelations(ma2a1, maxlag = 6)
autocovariances(ma2a1, maxlag = 6)
partialVariances(ma2a1, maxlag = 6)

## sigma2 is set to NA if not specified
## but things that don't depend on it are computed:
( ma2a2 <- MaModel(ma = c(0.3, 0.7)) )
autocorrelations(ma2a2, maxlag = 6)
partialAutocorrelations(ma2a2, maxlag = 6)

## AR
( ar2a1 <- ArModel(ar = c(-0.3, -0.7), sigma2 = 1) )
autocorrelations(ar2a1, maxlag = 6)
partialAutocorrelations(ar2a1, maxlag = 6)
autocovariances(ar2a1, maxlag = 6)
```

```
partialVariances(ar2a1, maxlag = 6)

## ARMA
( arma2a1 <- ArmaModel(ar = 0.5, ma = c(0.3, 0.7), sigma2 = 1) )
autocorrelations(arma2a1, maxlag = 6)
partialAutocorrelations(arma2a1, maxlag = 6)


## modelCoef() returns a list with components 'ar' and 'ma'
modelCoef(arma2a1)
modelCoef(ma2a1)
modelCoef(ar2a1)

## modelOrder() returns a list with components 'ar' and 'ma'
modelOrder(arma2a1)
modelOrder(ma2a1)
modelOrder(ar2a1)

as(ma2a1, "ArmaModel")  # success, as expected
as(ar2a1, "ArModel")  # success, as expected
as(ArmaModel(ar = c(-0.3, -0.7)), "ArModel")
## But these fail:
## as(ma2a1, "ArModel") # fails
## as(arma2a1, "ArModel") # fails
## as(arma2a1, "MaModel") # fails
```

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

Description

Classes ArmaModel, ArModel and MaModel in package sarima.

Objects from the Class

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

The recommended way to create objects from these classes is with the functions `ArmaModel`, `ArModel` and `MaModel`. Objects can also be created by calls of the form `new("ArmaModel", ..., ar, ma, mean, check)`. See also `coerce-methods` for further ways to create objects from these classes.

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")`: ...

Author(s)

Georgi N. Boshnakov

See Also

ArmaModel, ArModel, MaModel, coerce-methods

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 'center' 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
```

ArmaSpectrum-class *Class* "ArmaSpectrum"

Description

Objects from class "ArmaSpectrum" spectra computed by spectrum.

Details

The methods for show, print and plot work analogously to those for class "Spectrum" (which is a super class of "ArmaSpectrum"). In addition, print and show print also the parameters of the ARMA model.

Objects from the Class

Objects contain spectra produced by sarima::spectrum (recommended), see spectrum for details.

Objects can also be created by calls of the form new("ArmaSpectrum", ar = , ma = , sigma2 = , ...), where ar and ma are numeric vectors and sigma2 is a number. sigma2 may be omitted but then only normalized spectra can be computed. There further possibilities for the arguments but they should be considered internal and subject to change.

Slots

All slots are inherited from class "Spectrum".

.Data: Object of class "function".

call: Object of class "call".

model: Object of class "ANY".

Methods

initialize signature(.Object = "ArmaSpectrum"): ...

Author(s)

Georgi N. Boshnakov

See Also

class "Spectrum" for further details,

spectrum for further examples

Examples

```
## spectral density of the stationary part of a fitted 'airline model'
fit0 <- arima(AirPassengers, order = c(0,1,1),
              seasonal = list(order = c(0,1,1), period = 12))
sd.air <- spectrum(fit0)
show(sd.air)
plot(sd.air, log = "y") # plot log of the spectral density

## use the "ArmaSpectrum" object as a function to evaluate the sp. density:
sd.air(seq(0, 0.5, length.out = 13))
sd.air(seq(0, 0.5, length.out = 13), standardize = FALSE)

## white noise (constant spectral density)
sp.wn <- spectrum(ArmaModel(sigma2 = 2))
sp.wn
print(sp.wn)
print(sp.wn, standardize=FALSE)
show(sp.wn)
```

---

arma_Q0Gardner *Computing the initial state covariance matrix of ARMA*

Description

Wrappers for the internals 'stats' functions used by arima() to compute the initial state covariance matrix of ARMA models.

Usage

```
arma_Q0naive(phi, theta, tol = .Machine$double.eps)

arma_Q0gnbR(phi, theta, tol = .Machine$double.eps)
```

Arguments

phi	autoregressive coefficients.
theta	moving average coefficients.
tol	tollerance.

Details

`arima()` uses one of two methods to compute the initial state covariance matrix of a stationary ARMA model. Both methods are implemented by internal non-exported C functions. `arma_Q0Gardner()` and `arma_Q0bis` are simple R wrappers for those functions. They are defined in the tests (**TODO:** put in the examples?) bit are not defined in the namespace of the package since they use unexported functions.

`arma_Q0Gardner()` implements the original method from Gardner et al (1980). `arma_Q0bis()` is a more recent method that deals better with roots very close to the unit circle.

These functions can be useful for comparative testing. They cannot be put in package 'sarima' since they use `` `:::` `` operator and are hence inadmissible to CRAN.

Value

a matrix

References

Gardner G, Harvey AC, Phillips GDA (1980). "Algorithm AS154. An algorithm for exact maximum likelihood estimation of autoregressive-moving average models by means of Kalman filtering." *Applied Statistics*, 311–322.

Examples

```
## arma_Q0Gardner(phi, theta, tol = .Machine$double.eps)
## arma_Q0bis(phi, theta, tol = .Machine$double.eps)
```

`arma_Q0gnb` *Compute the initial state covariance of ARMA model*

Description

Compute the initial state covariance of ARMA model

Usage

```
arma_Q0gnb(phi, theta, tol = 2.220446e-16)
```

Arguments

`phi`	autoregression parameters.
`theta`	moving average parameters.
`tol`	tollerance. (TODO: explain)

Details

Experimental computation of the initial state covariance matrix of ARMA models.

The numerical results are comparable to `SSinit = "Rossignol2011"` method in `arima` and related functions. The method seems about twice faster than "Rossignol2011" on the models I tried but I haven't done systematic tests.

See section 'examples' below and, for more tests based on the tests from **stats**, the tests in `test/testthat/test-arma-q0.R`.

Value

a matrix

Author(s)

Georgi N. Boshnakov

References

Gardner G, Harvey AC, Phillips GDA (1980). "Algorithm AS154. An algorithm for exact maximum likelihood estimation of autoregressive-moving average models by means of Kalman filtering." *Applied Statistics*, 311–322.

R bug report PR#14682 (2011-2013) <URL: https://bugs.r-project.org/bugzilla3/show_bug.cgi?id=14682>.

See Also

makeARIMA, arima

Examples

```
Q0a <- arma_Q0gnb(c(0.2, 0.5), c(0.3))
Q0b <- makeARIMA(c(0.2, 0.5), c(0.3), Delta = numeric(0))$Pn
all.equal(Q0a, Q0b) ## TRUE

## see test/testthat/test-arma-q0.R for more;
## these functions cannot be defined in the package due to their use of
## \code{:::} on exported base R functions.
##
## "Gardner1980"
arma_Q0Gardner <- function(phi, theta, tol = .Machine$double.eps){
    ## tol is not used here
    .Call(stats:::C_getQ0, phi, theta)
}
## "Rossignol2011"
arma_Q0bis <- function(phi, theta, tol = .Machine$double.eps){
    .Call(stats:::C_getQ0bis, phi, theta, tol)
}

arma_Q0Gardner(c(0.2, 0.5), c(0.3))
arma_Q0bis(c(0.2, 0.5), c(0.3))
Q0a
Q0b
```

as.SarimaModel *Convert S3 model objects to class SarimaModel*

Description

Convert S3 model objects to class SarimaModel.

Usage

```
as.SarimaModel(x, ...)

## S3 method for class 'Arima'
as.SarimaModel(x, ...)
```

Arguments

x	an objects from a class representing Seasonal ARIMA models.
...	further arguments for methods.

Details

This function can be useful when one needs to manipulate the components of SARIMA models.

The method for class Arima (objects generated by stats::arima()) extracts the model information and convert it to "SarimaModel".

For S4 classes, there are methods for as(), where suitable. modelCoef provides a more powerful conversion mechanism.

Value

an object from class "SarimaModel"

See Also

SarimaModel

autocorrelations *Compute autocorrelations and related quantities*

Description

Generic functions for computation of autocorrelations, autocovariances and related quantities. The idea is to free the user from the need to look for specific functions that compute the desired property for their object.

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, all inheriting from "Lagged". The latter means that indexing starts from zero, so the value for lag zero is accessed by r[0]). Subscripting always returns the underlying data unclassed (i.e. ordinary vectors or arrays). In particular, "[]" extracts the underlying data.

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.

There are plot methods for sample autocorrelations and sample partial autocorrelations with overlaid significance limits under null hypotheses for independence or weak white noise, see plot-methods and the examples there. More details can be found in the vignettes, see section 'See also' below.

Value

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

Author(s)

Georgi N. Boshnakov

See Also

plot-methods for plotting with significance limits computed under strong white noise and weak white noise hypotheses;

autocorrelations-methods, partialAutocorrelations-methods for details on individual methods;

vignette("white_noise_tests", package = "sarima") and
vignette("garch_tests_example", package = "sarima") for extensive worked examples.

armaccf_xe, armaacf

Examples

```
set.seed(1234)
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)

## compute 2nd order properties from raw data
autocovariances(v1)
autocovariances(v1, maxlag = 10)
partialAutocovariances(v1, maxlag = 6)
partialAutocovariances(v1)
partialVariances(v1, maxlag = 6)
pv1 <- partialVariances(v1)

## compute 2nd order properties from raw data
autocovariances(AirPassengers, maxlag = 6)
autocorrelations(AirPassengers, maxlag = 6)
partialAutocorrelations(AirPassengers, maxlag = 6)
partialAutocovariances(AirPassengers, maxlag = 6)
partialVariances(AirPassengers, maxlag = 6)

acv <- autocovariances(AirPassengers, maxlag = 6)
autocovariances(acv) # no-op
autocovariances(acv, maxlag = 4) # trim the available lags

## compute 2nd order properties from sample autocovariances
acr <- autocorrelations(acv)
acr
partialAutocorrelations(acv)
partialAutocovariances(acv)
partialVariances(acv)

## compute 2nd order properties from sample autocorrelations
acr
partialAutocorrelations(acr)
```

```
## These cannot be computed, since the variance is needed but unknown:
##     autocovariances(acr)
##     partialAutocovariances(acr)
##     partialVariances(acr)

## to treat autocorrelations as autocovariances,
## convert them to autocovariances explicitly:
as(acr, "Autocovariances")
as(acr, "SampleAutocovariances")

partialVariances(as(acr, "Autocovariances"))
partialVariances(as(acr, "SampleAutocovariances"))
```

`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")`

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

Author(s)

Georgi N. Boshnakov

Examples

```
## see the examples for ?autocorrelations
```

autocovariances-methods *Methods for function autocovariances()*

Description

Methods for function autocovariances().

Methods

`signature(x = "ANY", maxlag = "ANY")`

`signature(x = "Autocovariances", maxlag = "ANY")`

`signature(x = "Autocovariances", maxlag = "missing")`

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

`signature(x = "VirtualAutocovariances", maxlag = "ANY")`

Author(s)

Georgi N. Boshnakov

See Also

`autocorrelations`

Examples

```
## see the examples for ?autocorrelations
```

coerce-methods *setAs methods in package sarima*

Description

Methods for as() in package sarima.

Methods

This section shows the methods for converting objects from one class to another, defined via `setAs()`. Use `as(obj, "classname")` to convert object `obj` to class "classname".

`signature(from = "ANY", to = "Autocorrelations")`

`signature(from = "ANY", to = "ComboAutocorrelations")`

`signature(from = "ANY", to = "ComboAutocovariances")`

`signature(from = "ANY", to = "PartialAutocorrelations")`

`signature(from = "ANY", to = "PartialAutocovariances")`

`signature(from = "ANY", to = "PartialVariances")`

`signature(from = "ArmaSpec", to = "list")`

`signature(from = "Autocorrelations", to = "ComboAutocorrelations")`

`signature(from = "Autocorrelations", to = "ComboAutocovariances")`

`signature(from = "Autocovariances", to = "ComboAutocorrelations")`

`signature(from = "Autocovariances", to = "ComboAutocovariances")`

`signature(from = "BJFilter", to = "SPFilter")`

`signature(from = "numeric", to = "BJFilter")` Convert a numeric vector to a BJFilter object. This is a way to state that the coefficients follow the Box-Jenkins convention for the signs, see the examples.

`signature(from = "numeric", to = "SPFilter")` Convert a numeric vector to an SPFilter object. This is a way to state that the coefficients follow the signal processing (SP) convention for the signs, see the examples.

`signature(from = "PartialVariances", to = "Autocorrelations")`

`signature(from = "PartialVariances", to = "Autocovariances")`

`signature(from = "PartialVariances", to = "ComboAutocorrelations")`

`signature(from = "PartialVariances", to = "ComboAutocovariances")`

`signature(from = "SarimaFilter", to = "ArmaFilter")`

`signature(from = "SarimaModel", to = "list")`

`signature(from = "SPFilter", to = "BJFilter")`

`signature(from = "vector", to = "Autocorrelations")`

`signature(from = "vector", to = "Autocovariances")`

`signature(from = "vector", to = "PartialAutocorrelations")`

`signature(from = "vector", to = "PartialAutocovariances")`

`signature(from = "VirtualArmaFilter", to = "list")`

`signature(from = "VirtualSarimaModel", to = "ArmaModel")`

Author(s)

Georgi N. Boshnakov

Examples

```
## the default for ARMA model is BJ for ar and SP for ma:
mo <- new("ArmaModel", ar = 0.9, ma = 0.4, sigma2 = 1)
modelPoly(mo)

## here we declare explicitly that 0.4 uses the SP convention
##    (not necessary, the result is the same, but the intention is clear).
mo1 <- new("ArmaModel", ar = 0.9, ma = as(0.4, "SPFilter"), sigma2 = 1)
modelPoly(mo1)
identical(mo, mo1) ## TRUE
```

```
## if the sign of theta follows the BJ convention, this can be stated unambiguously.
##   This creates the same model:
mo2 <- new("ArmaModel", ar = 0.9, ma = as(-0.4, "BJFilter"), sigma2 = 1)
modelPoly(mo2)
identical(mo, mo2) ## TRUE

## And this gives the intended model whatever the default conventions:
ar3 <- as(0.9, "BJFilter")
ma3 <- as(-0.4, "BJFilter")
mo3 <- new("ArmaModel", ar = ar3, ma = ma3, sigma2 = 1)
modelPoly(mo3)
identical(mo, mo3) ## TRUE

## The coefficients can be extracted in any particular form,
## e.g. to pass them to functions with specific requirements:
modelCoef(mo3) # coefficients of the model with the default (BD) sign convention
modelCoef(mo3, convention = "BD") # same result
modelCoef(mo3, convention = "SP") # signal processing convention


## for ltsa::tacvfARMA() the convention is BJ, so:
co <- modelCoef(mo3, convention = "BJ") # Box-Jenkins convention

ltsa::tacvfARMA(co$ar, co$ma, maxLag = 6, sigma2 = 1)
autocovariances(mo3, maxlag = 6) ## same
```

`confint` *Confidence and acceptance intervals in package sarima*

Description

Compute confidence and acceptance intervals for sample autocorrelations under assumptions chosen by the user.

Usage

```
## S4 method for signature 'SampleAutocorrelations'
confint(object, parm, level = 0.95, se = FALSE, maxlag, ..., assuming)
```

Arguments

`object`	an object containing sample autocorrelations (sacfs).
`parm`	which parameters to include, as for `stats::confint`.
`level`	coverage level, such as 0.95.
`se`	If `TRUE` return also standard errors.
`assuming`	under what assumptions to do the computations? Currently can be `"iid"`, `"garch"`, a fitted model, or a theoretical model, see Details.
`maxlag`	maximal lag to include
`...`	further arguments for `se`.

Details

For lags not fixed by the assumed model the computed intervals are confidence intervals.

The autocorrelations postulated by the null model (argument `assuming`) are usually fixed for some lags. For such lags it doesn't make sense to talk about confidence intervals. We use the term *acceptance interval* in this case since the sacfs for such lags fall in the corresponding intervals with high probability if the null model is correct.

If `assuming` is `"iid"` (strong white noise), then all autocorrelations in the null model are fixed (to zero) and the limits of the resulting acceptance intervals are ethose from the familiar plots produced by base-R's function `acf`.

If `assuming` is a fitted MA(q) model, e.g. obtained from `arima()`, then for lags $1, \ldots, q$ we get confidence intervals, while for lags greater than q the intervals are acceptance intervals.

The autocorrelations of ARMA models with non-trivial autoregressive part may also have structural patterns of zeroes (for example some seasonal models), leading to acceptance intervals for such lags.

If `assuming` specifies a theoretical (non-fitted) model, then the autocorrelation function of the null model is completely fixed and we get acceptance intervals for all lags.

The return value is a matrix with one row for each requested lag, containg the lag, lower bound, upper bound, estimate for acf(lag) and the value of acf(lag) under H0 (if fixed) and `NA` if not fixed under H0. The null model is stored as attribute `"assuming"`.

Note: When `assuming = "garch"` it is currently necessary to submit the time series from which the autocorrelations were computed as argument `x`.

Value

a matrix as described in section 'Details';

if `se = TRUE`, a column giving the standard errors of the sample autocorrelations is appended.

See Also

`se`

`vignette("white_noise_tests", package = "sarima")`

`vignette("garch_tests_example", package = "sarima")`

Examples

```
set.seed(1234)
v1 <- arima.sim(n = 100, list(ma = c(0.8, 0.1), sd = 1))
v1.acf <- autocorrelations(v1, maxlag = 10)

confint(v1.acf, parm = 1:4, assuming = "iid")
confint(v1.acf,  assuming = "iid", maxlag = 4) # same

## a fitted MA(2) - rho_1, rho_2 not fixed, the rest fixed
ma2fitted <- arima(v1, order = c(0,0,2), include.mean=FALSE)
confint(v1.acf, assuming = ma2fitted, maxlag = 4)

## a theoretical MA(2) model, all acfs fixed under H0
```

```
ma2 <- MaModel(ma = c(0.8, 0.1), sigma2 = 1)
confint(v1.acf, assuming = ma2, maxlag = 4)

# a weak white noise null
confint(v1.acf, assuming = "garch", maxlag = 4, x = v1)
```

`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 coefficient of order zero.
`...`	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.

For further details on argument `convention` see `filterCoef-methods`.

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.

Note

The `filterXXX()` functions are somewhat low level and technical. They should be rarely needed in routine work. The corresponding `modelXXX` are more flexible.

Author(s)

Georgi N. Boshnakov

See Also

`modelOrder`, `modelCoef`, `modelPoly`, `modelPolyCoef`, for the recommended higher level alternatives for models.

`filterOrder-methods`, `filterCoef-methods`, `filterPoly-methods`, `filterPolyCoef-methods`, for more information on the methods and the arguments.

Examples

```
filterPoly(as(c(0.3, 0.5), "BJFilter"))   # 1 - 0.3*x - 0.5*x^2
filterPoly(as(c(0.3, 0.5), "SPFilter"))   # 1 + 0.3*x + 0.5*x^2

## now two representations of the same filter:
fi1 <- as(c(0.3, 0.5), "BJFilter")
fi2 <- as(c(-0.3, -0.5), "SPFilter")
identical(fi2, fi1) # FALSE, but
## fi1 and fi2 represent the same filter, eg. same ch. polynomials:
filterPoly(fi1)
filterPoly(fi2)
identical(filterPolyCoef(fi2), filterPolyCoef(fi1))

# same as above, using new()
fi1a <- new("BJFilter", coef = c(0.3, 0.5))
identical(fi1a, fi1) # TRUE

fi2a <- new("SPFilter", coef = c(-0.3, -0.5))
identical(fi2a, fi2) # TRUE

## conversion by as() changes the internal representation
## but represents the same filter:
identical(as(fi1, "SPFilter"), fi2) # TRUE

c(filterOrder(fi1), filterOrder(fi2))

## these give the internally stored coefficients:
filterCoef(fi1)
filterCoef(fi2)

## with argument 'convention' the result doesn't depend
## on the internal representation:
co1 <- filterCoef(fi1, convention = "SP")
co2 <- filterCoef(fi2, convention = "SP")
identical(co1, co2) # TRUE
```

`filterCoef-methods` *Methods for filterCoef()*

Description

Methods for `filterCoef` in package **sarima**.

Methods

`filterCoef()` returns the coefficients of `object`. The format of the result depends on the type of filter, see the descriptions of the individual methods below.

If argument `convention` is omitted, the sign convention for the coefficients is the one used in the object. `convention` can be set to "BJ" or "SP" to request, respectively, the Box-Jenkins or the signal processing convention. Also, "-" is equivalent to "BJ" and "+" to "SP".

For ARMA filters, "BJ" and "SP" request the corresponding convention for both parts (AR and MA). A widely used convention, e.g., by base R and (Brockwell and Davis 1991), is "BJ" for the AR part and "SP" for the MA part. It can be requested with `convention = "BD"`. For convenience, "–" is equivalent to "BJ", "++" to "SP", "-+" to "BD". For completeness, "+-" can be used to request "SP" for the AR part and "BJ" for the MA part.

Invalid values of `convention` throw error. In particular, low level filters, such as "BJFilter" don't know if they are AR or MA, so they throw error if `convention` is "BD" or "+-" (but "++" and "–" are ok, since they are unambiguous). Similarly and to avoid subtle errors, the ARMA filters do not accept "+" or "-".

`signature(object = "VirtualMonicFilterSpec", convention = "missing")` returns `object@coef`.

`signature(object = "VirtualBJFilter", convention = "character")` returns the filter coefficients in the requested convention.

`signature(object = "VirtualSPFilter", convention = "character")` returns the filter coefficients in the requested convention.

`signature(object = "BJFilter", convention = "character")` returns the filter coefficients in the requested convention.

`signature(object = "SPFilter", convention = "character")` returns the filter coefficients in the requested convention.

`signature(object = "VirtualArmaFilter", convention = "missing")`

`signature(object = "VirtualArmaFilter", convention = "character")` Conceptually, calls `filterCoef()`, with one argument, on the AR and MA parts of the model. If `convention` is present, converts the result to the specified convention. Returns a list with the following components:

ar AR coefficients.

ma MA coefficients.

`signature(object = "SarimaFilter", convention = "missing")`

`signature(object = "SarimaFilter", convention = "character")` If `convention` is present, converts the coefficients to the specified convention. AR-like coefficients get the convention for the AR part, Ma-like coefficients get the convention for the MA part. Returns a list with the following components:

nseasons number of seasons.

iorder integration order, number of (non-seasonal) differences.

siorder seasonal integration order, number of seasonal differences.

ar ar coefficients.

ma ma coefficients.

sar seasonal ar coefficients.

sma seasonal ma coefficients.

Author(s)

Georgi N. Boshnakov

References

Brockwell PJ, Davis RA (1991). *Time series: theory and methods. 2nd ed.*. Springer Series in Statistics. Berlin etc.: Springer-Verlag..

See Also

`filterCoef` for examples and related functions

Examples

```
## see the examples for ?filterCoef
```

`filterOrder-methods` *Methods for function* `filterOrder` *in package* **sarima**

Description

Methods for function `filterOrder` in package **sarima**.

Methods

The following methods ensure that all filters in package **sarima** have a method for `filterOrder`.

`signature(object = "VirtualMonicFilterSpec")` Returns `object@order`.

`signature(object = "SarimaFilter")` Returns a list with the following components:

nseasons number of seasons.

iorder integration order, number of (non-seasonal) differences.

siorder seasonal integration order, number of seasonal differences.

ar autoregression order

ma moving average order

sar seasonal autoregression order

sma seasonal moving average order

`signature(object = "VirtualArmaFilter")` Returns a list with the following components:

ar autoregression order.

ma moving average order.

Author(s)

Georgi N. Boshnakov

See Also

`filterCoef` for examples and related functions

Examples

```
## see the examples for ?filterCoef
```

`filterPoly-methods` *Methods for* `filterPoly` *in package* **sarima**

Description

Methods for `filterPoly` in package **sarima**.

Methods

The methods for `filterPoly` take care implicitly for the sign convention used to store the coefficients in the object.

`signature(object = "BJFilter")` A polynomial whose coefficients are the negated filter coefficients.

`signature(object = "SPFilter")` A polynomial whose coefficients are as stored in the object.

`signature(object = "SarimaFilter")` Returns a list with the following components:

nseasons number of seasons.

iorder integration order, number of (non-seasonal) differences.

siorder seasonal integration order, number of seasonal differences.

arpoly autoregression polynomial

mapoly moving average polynomial

sarpoly seasonal autoregression polynomial

smapoly seasonal moving average polynomial

fullarpoly the polynomial obtained by multiplying out all AR-like terms, including differences.

fullmapoly the polynomial obtained by multiplying out all MA terms

core_sarpoly core seasonal autoregression polynomial. It is such that sarpoly(z) = core_sarpoly($z^{nseasons}$)

core_smapoly core seasonal moving average polynomial. It is such that smapoly(z) = core_smapoly($z^{nseasons}$)

`signature(object = "VirtualArmaFilter")` Returns a list with the following components:

ar autoregression polynomial.

ma moving average polynomial.

`signature(object = "VirtualMonicFilterSpec")` Calls `filterPolyCoef(object)` and converts the result to a polynomial. Thus, it is sufficient to have a method for `filterPolyCoef()`.

Author(s)

Georgi N. Boshnakov

See Also

`filterCoef` for examples and related functions

Examples

```
## see the examples for ?filterCoef
```

`filterPolyCoef-methods`

Methods for filterPolyCoef

Description

Methods for `filterPolyCoef` in package **sarima**.

Methods

The `filterPolyCoef` methods return results with the same structure as the corresponding methods for `filterPoly` but with polynomials replaced by their coefficients. If `lag_0` is `FALSE` the order 0 coefficients are dropped.

`signature(object = "VirtualBJFilter")` Calls `filterCoef(object)`, negates the result and prepends 1 if `lag_0` is `TRUE`.

`signature(object = "VirtualSPFilter")` Calls `filterCoef(object)` and prepends 1 to the result if `lag_0` is `TRUE`.

`signature(object = "VirtualArmaFilter")` Returns a list with the following components:

ar coefficients of the autoregression polynomial.

ma coefficients of the moving average polynomial.

`signature(object = "BJFilter")` The coefficients of a polynomial whose coefficients are the negated filter coefficients. This is equivalent to the method for "VirtualBJFilter" but somewhat more efficient.

`signature(object = "SPFilter")` The coefficients of a polynomial whose coefficients are as stored in the object. This is equivalent to the method for "VirtualSPFilter" but somewhat more efficient.

`signature(object = "SarimaFilter")` Returns a list with the same components as the "SarimaFilter" method for `filterPoly`, where the polynomials are replaced by their coefficients.

Author(s)

Georgi N. Boshnakov

See Also

`filterCoef` for examples and related functions

Examples

```
## see the examples for ?filterCoef
```

`FisherInformation-methods`
Fisher information

Description

Compute the Fisher information for the parameters of a model.

Usage

```
FisherInformation(model, ...)

## S3 method for class 'Arima'
FisherInformation(model, ...)
```

Arguments

`model`	a fitted or theoretical model for which a method is available.
`...`	further arguments for methods.

Details

`FisherInformation` computes the information matrix for the parameters of `model`. This is a generic function. The methods for objects from S4 classes are listed in section 'Methods'.

Currently package **sarima** defines methods for objects representing fitted or theoretical ARMA and seasonal ARMA models. For integrated models the result should be interpreted as the information matrix after differencing.

For ARMA models the implementation is based on Friedlander (1984) and (for the seasonal specifics) Godolphin and Godolphin (2007).

Value

a square matrix with attribute "nseasons"

Methods

This section lists FisherInformation methods for S4 classes.

signature(model = "ANY")

signature(model = "SarimaModel")

signature(model = "ArmaModel")

Author(s)

Georgi Boshnakov

References

Friedlander B (1984). "On the computation of the Cramer-Rao bound for ARMA parameter estimation." *IEEE Transactions on Acoustics, Speech, and Signal Processing*, **32**(4), 721-727. doi:10.1109/TASSP.1984.1164391.

Godolphin EJ, Godolphin JD (2007). "A Note on the Information Matrix for Multiplicative Seasonal Autoregressive Moving-Average Models." *Journal of Time Series Analysis*, **28**, 783-791. doi:10.1111/j.14679892.2007.00531.x.

Examples

```
## a fitted model (over-parameterised)
seas_spec <- list(order = c(1,0,1), period = 4)
fitted <- arima(rnorm(100), order = c(1,0,1), seasonal = seas_spec)
(fi <- FisherInformation(fitted))
## asymptotic covariance matrix of the ARMA parameters:
fitted$sigma2 * solve(fi) / 100

## a theoretical seasonal ARMA model:
sarima1 <- new("SarimaModel", ar = 0.9, ma = 0.1, sar = 0.5, sma = 0.9, nseasons = 12)
FisherInformation(sarima1)

## a non-seasonal ARMA model:
arma2a1 <- ArmaModel(ar = 0.5, ma = c(0.3, 0.7), sigma2 = 1)
FisherInformation(arma2a1)
## sigma2 is not needed for the information matrix:
arma2a1a <- ArmaModel(ar = 0.5, ma = c(0.3, 0.7))
FisherInformation(arma2a1a) # same as above
```

`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). Currently not used!
`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, sigma2 = 1)
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,
           sigma2 = 1)
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)
```

`InterceptSpec-class` *Class InterceptSpec*

Description

A helper class from which a number of models inherit intercept, centering and innovations variance.

Objects from the Class

Objects can be created by calls of the form `new("InterceptSpec", ...)`.

Slots

`center`**:** Object of class `"numeric"`, centering parameter, defaults to zero.

`intercept`**:** Object of class `"numeric"`, intercept parameter, defaults to zero.

`sigma2`**:** Object of class `"numeric"`, innovations variance, defaults to NA.

Methods

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

Author(s)

Georgi N. Boshnakov

See Also

`ArmaModel`, `SarimaModel`

Examples

```
showClass("InterceptSpec")
```

`isStationaryModel` *Check if a model is stationary*

Description

Check if a model is stationary.

Usage

```
isStationaryModel(object)
```

Arguments

`object`	an object

Details

This is a generic function. It returns `TRUE` if `object` represents a stationary model and FALSE otherwise.

Value

TRUE or FALSE

Methods

```
signature(object = "SarimaSpec")
signature(object = "VirtualIntegratedModel")
signature(object = "VirtualStationaryModel")
```

Author(s)

Georgi N. Boshnakov

See Also

`nUnitRoots`

modelCenter *model center*

Description

model center

Usage

```
modelCenter(object)
```

Arguments

object	an object

Methods

```
signature(object = "InterceptSpec")
```

Author(s)

Georgi N. Boshnakov

modelCoef *Get the coefficients of models*

Description

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

Usage

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

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.
component	if not missing, specifies a component to extract, see Details.
...	not used, further arguments for methods.

Details

`modelCoef` is a generic function for extraction of coefficients of model objects. What 'coefficients' means depends on the class of `object` but it can be changed with the optional argument `convention`. In effect, `modelCoef` provides a very flexible and descriptive way of extracting coefficients from models in various forms.

The one-argument form, `modelCoef(object)`, gives the coefficients of `object`. In effect it defines, for the purposes of `modelCoef`, the meaning of 'coefficients' for class `class(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. Conceptually, the value will have the format and meaning of the value that would be returned by a call to `modelCoef(obj)` with `obj` from class `class(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"))`. Note that this is conceptual - argument `convention` can be the name of a virtual class, for example. Also, for some classes of `object` character values other than names of classes may be supported.

For example, if `obj` is from class "ArmaModel", `modelCoef(obj)` returns a list with components "ar" and "ma", which follow the "BD" convention. So, to get such a list of coefficients from an `object` from any class capable of representing ARMA models, set `convention = "ArmaModel"` in the call to `modelCoef{}`.

`modelCoef()` will signal an error if `object` is not compatible with `target` (e.g. if it contains unit roots). (see `filterCoef` if you need to expand any multiplicative filters). **TODO: rethink this, it does not reflect current behaviour!**

If there is no class which returns exactly what is needed some additional computation may be necessary. In the above "ArmaModel" example we might need the coefficients in the "BJ" convention, so we would need to change the signs of the MA coefficients to achieve this. Since this is a very common operation, a convenience feature is available. Setting `convention = "BJ"` requests ARMA coefficients with "BJ" convention. For completeness, the the settings "SP" (signal processing) and "BD" (Brockwell-Davis) are also available.

The methods for `modelCoef()` in package "sarima" return a list with components depending on argument "convention", as outlined above.

Value

a list, with components depending on the target class, as described in Details

Author(s)

Georgi N. Boshnakov

See Also

`modelOrder`, `modelPoly`, `modelPolyCoef`

Examples

```
## define a seasonal ARIMA model, it has a number of components
m1 <- new("SarimaModel", iorder = 1, siorder = 1, ma = -0.3, sma = -0.1, nseasons = 12)
m1
## Get the coefficients corresponding to a 'flat' ARMA model,
## obtained by multiplying out AR-like and MA-like terms.
## A simple way is to use modelCoef() with a suitable convention:
modelCoef(m1, "ArmaModel")
modelCoef(m1, "ArmaFilter") ## same

## Here is another model
m1a <- new("SarimaModel", iorder = 1, siorder = 1,  ar = 0.6, nseasons = 12)
modelCoef(m1a, "ArmaModel")
modelCoef(m1a, "ArmaFilter") ## same

## if only AR-like terms are allowed in a computation,
## use convention = "ArModel" to state it explicitly.
##
## this works, since m1a contains only AR-like terms:
modelCoef(m1a, "ArModel")
modelCoef(m1a, "ArFilter") ## same
## ... but these would throw errors if evaluated,
## since model m1a contains both AR-like and MA-like terms,
## Not run:
modelCoef(m1, "ArModel")
modelCoef(m1, "ArFilter")
modelCoef(m1, "MaModel")
modelCoef(m1, "MaFilter")

## End(Not run)
```

---

modelCoef-methods *Methods for generic function modelCoef*

Description

Methods for generic function modelCoef.

Methods

signature(object = "Autocorrelations", convention = "ComboAutocorrelations", component = "missing")

signature(object = "Autocorrelations", convention = "PartialAutocorrelations", component = "missing")

signature(object = "Autocovariances", convention = "Autocorrelations", component = "missing")

signature(object = "Autocovariances", convention = "ComboAutocorrelations", component = "missing")

```
signature(object = "Autocovariances", convention = "ComboAutocovariances", component = "missing")

signature(object = "Autocovariances", convention = "PartialAutocorrelations", component = "missing")

signature(object = "ComboAutocorrelations", convention = "Autocorrelations", component = "missing")

signature(object = "ComboAutocorrelations", convention = "PartialAutocorrelations", component = "missin

signature(object = "ComboAutocovariances", convention = "Autocovariances", component = "missing")

signature(object = "ComboAutocovariances", convention = "PartialAutocovariances", component = "missing"

signature(object = "ComboAutocovariances", convention = "PartialVariances", component = "missing")

signature(object = "ComboAutocovariances", convention = "VirtualAutocovariances", component = "missing"

signature(object = "PartialAutocorrelations", convention = "Autocorrelations", component = "missing")

signature(object = "SarimaModel", convention = "ArFilter", component = "missing")
signature(object = "SarimaModel", convention = "ArmaFilter", component = "missing")

signature(object = "SarimaModel", convention = "MaFilter", component = "missing")
signature(object = "SarimaModel", convention = "SarimaFilter", component = "missing")

signature(object = "VirtualAutocovariances", convention = "character", component = "missing")

signature(object = "VirtualAutocovariances", convention = "missing", component = "missing")

signature(object = "VirtualAutocovariances", convention = "VirtualAutocovariances", component = "missin

signature(object = "SarimaModel", convention = "ArModel", component = "missing")
signature(object = "SarimaModel", convention = "MaModel", component = "missing")
signature(object = "VirtualFilterModel", convention = "BD", component = "missing")
signature(object = "VirtualFilterModel", convention = "BJ", component = "missing")
signature(object = "VirtualFilterModel", convention = "character", component = "missing")

signature(object = "VirtualFilterModel", convention = "missing", component = "missing")

signature(object = "VirtualFilterModel", convention = "SP", component = "missing")
signature(object = "ArmaModel", convention = "ArmaFilter", component = "missing")
signature(object = "VirtualAutocovariances", convention = "Autocovariances", component = "missing")
```

Author(s)

Georgi N. Boshnakov

`modelIntercept` *Give the intercept parameter of a model*

Description

Give the intercept parameter of a model.

Usage

```
modelIntercept(object)
```

Arguments

`object`	an object from a class for which intercept is defined.

Methods

```
signature(object = "InterceptSpec")
```

Author(s)

Georgi N. Boshnakov

`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.

Author(s)

Georgi N. Boshnakov

See Also

`modelCoef`

Examples

```
m1 <- new("SarimaModel", iorder = 1, siorder = 1, ma = -0.3, sma = -0.1, nseasons = 12)
modelOrder(m1)
modelOrder(m1, "ArmaFilter")
modelOrder(m1, new("ArmaFilter"))

modelPoly(m1, "ArmaModel")
modelPolyCoef(m1, "ArmaModel")
```

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 = "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 = "VirtualFilterModel", convention = "missing")
signature(object = "VirtualFilterModel", convention = "character")
```

Author(s)

Georgi N. Boshnakov

modelPoly-methods *Get polynomials associated with SARIMA models*

Description

Get polynomials associated with SARIMA models.

Methods

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

Author(s)

Georgi N. Boshnakov

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 = "VirtualMonicFilter", convention = "missing")`

`signature(object = "VirtualFilterModel", convention = "character")`

Author(s)

Georgi N. Boshnakov

nSeasons *Number of seasons*

Description

Number of seasons.

Usage

```
nSeasons(object)
```

Arguments

`object`	an object for which the notion of number of seasons makes sense.

Details

This is a generic function.

Value

an integer number

Methods

`signature(object = "SarimaFilter")`

`signature(object = "VirtualArmaFilter")`

Author(s)

Georgi N. Boshnakov

`nUnitRoots` *Number of unit roots in a model*

Description

Gives the number of roots with modulus one in a model.

Usage

```
nUnitRoots(object)
```

Arguments

`object`	an object.

Details

`nUnitRoots()` gives the number of roots with modulus one in a model. This number is zero for stationary models, see also `isStationaryModel()`.

Value

a non-negative integer number

Methods

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

```
signature(object = "VirtualStationaryModel")
```

Author(s)

Georgi N. Boshnakov

nvarOfAcfKP *Compute variances of autocorrelations under ARCH-type hypothesis*

Description

Compute variances of autocorrelations under ARCH-type hypothesis.

Usage

```
nvarOfAcfKP(x, maxlag, center = FALSE, acfscale = c("one", "mom"))
```

Arguments

x	time series.
maxlag	a positive integer, the maximal lag.
center	logical flag, if FALSE, the default, don't center the time series before squaring, see Details.
acfscale	character string, specifying what factor to use for the autocovariances. "one" stands for $1/n$, "mom" for $1/(n-k)$, where n is the length of x and k is lag.

Details

nvarOfAcfKP computes estimates of n times the variances of sample autocorrelations of white noise time series. It implements the result of (Kokoszka and Politis 2011) which holds under weak assumptions. In particular, it can be used to test if the true autocorrelations of a time series are equal to zero in GARCH modelling.

Value

a numeric vector

Author(s)

Georgi N. Boshnakov

References

Kokoszka PS, Politis DN (2011). "Nonlinearity of ARCH and stochastic volatility models and Bartlett's formula." *Probability and Mathematical Statistics*, **31**(1), 47–59.

See Also

whiteNoiseTest

Examples

```
## see examples for whiteNoisTest()
```

nvcovOfAcf *Covariances of sample autocorrelations*

Description

Compute covariances of autocorrelations.

Usage

```
nvcovOfAcf(model, maxlag)

nvcovOfAcfBD(acf, ma, maxlag)

acfOfSquaredArmaModel(model, maxlag)
```

Arguments

model	a model, see Details.
maxlag	a positive integer number, the maximal lag.
acf	autocorrelations.
ma	a positive integer number, the order of the MA(q) model. The default is the maximal lag available in acf.

Details

nvcovOfAcf computes the unscaled asymptotic autocovariances of sample autocorrelations of ARMA models, under the classical assumptions when the Bartlett's formulas are valid. It works directly with the parameters of the model and uses Boshnakov (1996). Argument model can be any specification of ARMA models for which autocorrelations() will work, e.g. a list with components "ar", "ma", and "sigma2".

nvcovOfAcfBD computes the same quantities but uses the formula given by Brockwell & Davis (1991) (eq. (7.2.6.), p. 222), which is based on the autocorrelations of the model. Argument acf contains the autocorrelations.

For nvcovOfAcfBD, argument ma asks to treat the provided acf as that of MA(ma). Only the values for lags up to ma are used and the rest are set to zero, since the autocorrelations of MA(ma) models are zero for lags greater than ma. To force the use of all autocorrelations provided in acf, set ma to the maximal lag available in acf or omit ma, since this is its default.

acfOfSquaredArmaModel(model, maxlag) is a convenience function which computes the autocovariances of the "squared" model, see Boshnakov (1996).

Value

an (maxlag,maxlag)-matrix

Note

The name of `nvcovOfAcf` stands for "`n` times the variance-covariance matrix", so it needs to be divided by `n` to get the asymptotic variances and covariances.

Author(s)

Georgi N. Boshnakov

References

Boshnakov GN (1996). "Bartlett's formulae—closed forms and recurrent equations." *Ann. Inst. Statist. Math.*, **48**(1), 49–59. ISSN 0020-3157, doi:10.1007/BF00049288.

Brockwell PJ, Davis RA (1991). *Time series: theory and methods. 2nd ed.*. Springer Series in Statistics. Berlin etc.: Springer-Verlag..

See Also

`whiteNoiseTest`

Examples

```
## MA(2)
ma2 <- list(ma = c(0.8, 0.1), sigma2 = 1)
nv <- nvcovOfAcf(ma2, maxlag = 4)
d <- diag(nvcovOfAcf(ma2, maxlag = 7))
cbind(ma2 = 1.96 * sqrt(d) / sqrt(200), iid = 1.96/sqrt(200))

acr <- autocorrelations(list(ma = c(0.8, 0.1)), maxlag = 7)
nvBD <- nvcovOfAcfBD(acr, 2, maxlag = 4)
all.equal(nv, nvBD) # TRUE
```

---

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")`

Author(s)

Georgi N. Boshnakov

`periodogram` *Obtain the most important period lags of a time series according to a periodogram.*

Description

Obtain the most important period lags of a time series according to a periodogram.

Usage

```
periodogram(x, ..., no.results = 20)
```

Arguments

`x`	A vector containing the time series values
`...`	Arguments to be passed to `spectrum`
`no.results`	The number of results to be printed at the end. Defaults to the 20 most important frequencies.

Details

Using the `spectral` function, obtain spectral density estimates at a number of frequencies and rather than plotting, obtain the rank and period of the values. Return a given number of results based on the level of interest of the user.

Value

A data.frame containing the following columns:

`rank`	numeric vector containing the ranked importance of the frequency.
`spectrum`	estimates of the spectral density at frequencies corresponding to `freq`.
`frequency`	vector at which the spectral density is estimated.
`period`	vector of corresponding periods.

plot-methods *Plot methods in package sarima*

Description

Plot methods in package sarima.

Methods

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

`signature(x = "SampleAutocorrelations", y = "missing")` plots the sample autocorrelations with (individual) rejection limits computed under the null hypothesis of i.i.d. (strong white noise) If argument `data` is provided, it should be the time series from which the autocorrelations were computed. In this case rejection limits for null hypothesis that the time series is (garch) weak white noise are provided, as well.

Additional arguments can be supplied, see `whiteNoiseTest` the examples, and the vignettes.

`signature(x = "SamplePartialAutocorrelations", y = "missing")` plots the sample partial autocorrelations with rejection limits for the hypotheses and controlling arguments as for `"SampleAutocorrelations"`.

Author(s)

Georgi N. Boshnakov

See Also

`whiteNoiseTest` for the computations for the rejection levels;

Examples

```
set.seed(1234)
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)
## add limits for a weak white noise test:
plot(x.acf, data = x)

## similarly for pacf
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`	an object from a suitable class or a list, see Details.
`x`	initial/before values of the time series, a list, a numeric vector or time series, 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 burn-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.

Argument `model` can be from any class representing models in the SARIMA family, such as "SarimaModel", or a list with components suitable to be passed to =new()= for such models.

The canonical form of argument `x` is a list with components `before`, `init` and `main`. If any of these components is missing or NULL, it is filled suitably. Any other components of `x` are ignored. If `x` is

not a list, it is put in component `main`. Conceptually, the three components are concatenated in the given order, the simulated values are put in `main` (`before` and `init` are not changed), the `before` part is dropped and the rest is returned. In effect, `before` and `init` can be viewed as initial values but `init` is considered part of the generated series.

The format for `eps` is the same as that of `x`. The lengths of missing components in `x` are inferred from the corresponding components of `eps`, and vice versa.

The format for `xintercept` is the same as that of `x` and `eps`.

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

Value

for `prepareSimSarima`, a function to simulate time series, see Details. it is typically called multiple times without arguments. All arguments have defaults set by `prepareSimSarima`.

`n`	length of the simulated time series,
`rand.gen`	random number generator,
`...`	arguments for the random number generator, passed on to `arima.sim`.

Author(s)

Georgi N. Boshnakov

See Also

`sim_sarima`

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()
tmp1
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))
```

`sarima` *Fit extended SARIMA models*

Description

Fit extended SARIMA models, which can include lagged exogeneous variables, general unit root non-stationary factors, multiple periodicities, and multiplicative terms in the SARIMA specification. The models are specified with a flexible formula syntax and contain as special cases many models with specialised names, such as ARMAX and reg-ARIMA.

Usage

```
sarima(model, data = NULL, ss.method = "sarima", use.symmetry = FALSE,
       SSinit = "Rossignol2011")
```

Arguments

`model`	a model formula specifying the model.
`data`	a list or data frame, usually can be omitted.
`ss.method`	state space engine to use, defaults to `"sarima"`. (**Note:** this argument will probably be renamed.)
`use.symmetry`	a logical argument indicating whether symmetry should be used to estimate the unit polynomial.
`SSinit`	method to use for computation of the stationary part of the initial covariance matrix, one of "Rossignol2011", "gnb", "Gardner1980".

Details

`sarima` fits extended SARIMA models, which can include exogeneous variables, general unit root non-stationary factors and multiplicative terms in the SARIMA specification.

Let $\{Y_t\}$ be a time series and $f(t)$ and $g(t)$ be functions of time and/or (possibly lagged) exogeneous variables.

An extended pure SARIMA model for Y_t can be written with the help of the backward shift operator as

$$U(B)\Phi(B)Y_t = \Theta(B)\varepsilon_t,$$

where $\{\varepsilon_t\}$ is white noise, and $U(z)$, $\Phi(z)$, and $\Theta(z)$ are polynomials, such that all roots of $U(z)$ are on the unit circle, while the roots of $\Phi(z)$ and $\Theta(z)$ are outside the unit circle. If unit roots are missing, ie $U(z) \equiv 1$, the model is stationary with mean zero.

A reg-SARIMA or X-SARIMA model can be defined as a regression with SARIMA residuals:

$$Y_t = f(t) + Y_t^c$$

$$U(B)\Phi(B)Y_t^c = \Theta(B)\varepsilon_t,$$

where $Y_t^c = Y_t - f(t)$ is the centred Y_t. This can be written equivalently as a single equation:

$$U(B)\Phi(B)(Y_t - f(t)) = \Theta(B)\varepsilon_t.$$

The regression function $f(t)$ can depend on time and/or (possibly lagged) exogeneous variables. We call it centering function. If Y_t^c is stationary with mean zero, `f(t)` is the mean of Y_t. If `f(t)` is constant, say `mu`, Y_t is stationary with mean `mu`. Note that the two-equation form above shows that in that case `mu` is the intercept in the first equation, so it is perfectly reasonable to refer to it also as intercept but to avoid confusion we reserve the term **intercept** for `g(t)` below.

If the SARIMA part is stationary, then $EY_t = f(t)$, so $f(t)$ can be interpreted as trend. In this case the above specification is often referred to as **mean corrected form** of the model.

An alternative way to specify the regression part is to add the regression function, say $\{g(t)\}$, to the right-hand side of the SARIMA equation:

$$U(B)\Phi(B)Y_t = g(t) + \Theta(B)\varepsilon_t.$$

In the stationary case this is the classical ARMAX specification. This can be written in two-stage form in various ways, eg

$$U(B)\Phi(B)Y_t = (1 - \Theta(B))\varepsilon_t + u_t,$$
$$u_t = g(t) + \varepsilon_t.$$

So, in a sense, g(t) is a trend associated with the residuals from the SARIMA modelling. We refer to this form as intercept form of the model (as opposed to the mean-corrected form discussed previously).

In general, if there are no exogeneous variables the mean-corrected model is equivalent to some intercept model, which gives some justification of the terminology, as well. If there are exogeneous variables equivalence may be achievable but at the expense of introducing more lags in the model, whish is not desirable in general.

Some examples of equivalence. Let Y be a stationary SARIMA process ($U(z) = 1$) with mean μ. Then the mean-corrected form of the SARIMA model is

$$\Phi(B)(Y_t - \mu) = \Theta(B)\varepsilon_t,$$

while the intercept form is

$$\Phi(B)Y_t = c + \Theta(B)\varepsilon_t,$$

where $c = \Phi(B)\mu$. So, in this case the mean-corrected model X-SARIMA model with $f(t) = \mu$ is equivalent to the intercept model with $g(t) = \Phi(B)\mu$.

As another example, with $f(t) = bt$, the mean-corrected model is $(1-B)(Y_t - bt) = \varepsilon_t$. Expanding the left-hand side we obtain the intercept form $(1 - B)Y_t = b + \varepsilon_t$, which demonstrates that Y_t is a random walk with drift $g(t) = b$.

Model specification

Argument `model` specifies the model with a syntax similar to other model fitting functions in R. A formula can be given for each of the components discussed above as `y ~ f | SARIMA | g`, where `f`, `SARIMA` and `g` are model formulas giving the specifications for the centering function `f`, the `SARIMA` specification, and the intercept function `g`. In normal use only one of `f` or `g` will be different from zero. `f` should always be given (use `0` to specify that it is identical to zero), but `g` can be omitted altogether. Sometimes we refer to the terms specified by `f` and `g` by `xreg` and `regx`, respectively.

Model formulas for trends and exogeneous regressions

The formulas for the centering and intercept (ie `f` and `g`) use the same syntax as in linear models with some additional functions for trigonometric trends, polynomial trends and lagged variables.

Here are the available specialised terms:

.p(d) Orthogonal polynomials over `1:length(y)` of degree d (starting from degree 1, no constant).

t Stands for `1:length(y)`. Note that powers need to be protected by I(), e.g. `y ~ 1 + .t + I(.t^2)`.

.cs(s, k) cos/sin pair for the k-th harmonic of 2pi/s. Use vector k to specify several harmonics.

.B(x, lags) Include lagged terms of x, $B^{lags}(x[t]) = x[t - lags]$. `lags` can be a vector. If `x` is a matrix, the specified lags are taken from each column.

Model formulas for SARIMA models

A flexible syntax is provided for the specification of the SARIMA part of the model. It is formed using a number of primitives for stationary and unit root components, which have non-seasonal and seasonal variants. Arbitrary number of multiplicative factors and multiple seasonalities can be specified.

The SARIMA part of the model can contain any of the following terms. They can be repeated as needed. The first argument for all seasonal operators is the number of seasons.

ar(p) autoregression term of order p

ma(q) moving average term of order q

sar(s,p) seasonal autoregression term (s seasons, order p)

sma(s,q) seasonal moving average term (s seasons, order q)

i(d) $(1 - B)^d$

s(seas) summation operator, $(1 + B + \cdots + B^{seas-1})$

u(x) quadratic unit root term, corresponding to a complex pair on the unit circle. If x is real, it specifies the argument of one of the roots as a fraction of 2π. If z is complex, it is the root itself.

The real roots of modulus one (1 and -1) should be specified using `i(1)` and `s(2)`, which correspond to $1 - B$ and $1 + B$, respectively.

su(s, h) quadratic unit root terms corresponding to seasonal differencing factors. h specifies the desired harmonic which should be one of 1,2, ..., [s/2]. Several harmonics can be specified by setting h to a vector.

ss(s, p) seasonal summation operator, $(1 + B^s + \cdots + B^{(s-1)p})$

Terms with parameters can contain additional arguments specifying initial values, fixed parameters, and transforms. For `ar`, `ma`, `sar`, `sma`, values of the coefficients can be specified by an unnamed argument after the parameters given in the descriptions above. In estimation these values will be taken as initial values for optimisation. By default, all coefficients are taken to be non-fixed.

Argument `fixed` can be used to fix some of them. If it is a logical vector it should be of length one or have the same length as the coefficients. If `fixed` is of length one and `TRUE`, all coefficients are fixed. If FALSE, all are non-fixed. Otherwise, the TRUE/FALSE values in `fixed` determine the fixedness of the corresponding coefficients.

`fixed` can also be a vector of positive integer numbers specifying the indices of fixed coefficients, the rest are non-fixed.

Sometimes it may be easier to declare more (e.g. all) coefficients as fixed and then 'unfix' selectively. Argument `nonfixed` can be used to mark some coefficients as non-fixed after they have been declared fixed. Its syntax is the same as for `fixed`.

TODO: streamline "atanh.tr"

TODO: describe `SSinit`

Value

an object from S3 class Sarima

(**Note:** the format of the object is still under development and may change; use accessor functions, such as `coef()`, where provided.)

Note

Currently the implementation of the intercept form (ie the third part of the model formula) is incomplete.

Author(s)

Georgi N. Boshnakov

References

Halliday J, Boshnakov GN (2022). "Partial autocorrelation parameterisation of models with unit roots on the unit circle." doi:10.48550/ARXIV.2208.05055, https://arxiv.org/abs/2208.05055.

See Also

arima

Examples

```
## AirPassengers example
## fit the classic airline model using arima()
ap.arima <- arima(log(AirPassengers), order = c(0,1,1), seasonal = c(0,1,1))

## same model using two equivalent ways to specify it
ap.baseA <- sarima(log(AirPassengers) ~
                   0 | ma(1, c(-0.3)) + sma(12,1, c(-0.1)) + i(1) + si(12,1),
                   ss.method = "base")
ap.baseB <- sarima(log(AirPassengers) ~
                   0 | ma(1, c(-0.3)) + sma(12,1, c(-0.1)) + i(2) + s(12),
                   ss.method = "base")

ap.baseA
summary(ap.baseA)
ap.baseB
summary(ap.baseB)

## as above, but drop 1-B from the model:
ap2.arima <- arima(log(AirPassengers), order = c(0,0,1), seasonal = c(0,1,1))
ap2.baseA <- sarima(log(AirPassengers) ~
                    0 | ma(1, c(-0.3)) + sma(12,1, c(-0.1)) +     si(12,1),
                    ss.method = "base")
ap2.baseB <- sarima(log(AirPassengers) ~
                    0 | ma(1, c(-0.3)) + sma(12,1, c(-0.1)) + i(1) + s(12),
                    ss.method = "base")
```

```
## for illustration, here the non-stationary part is
##     (1-B)^2(1+B+...+B^5) = (1-B)(1-B^6)
##     (  compare to (1-B)(1-B^{12}) = (1-B)(1-B^6)(1+B^6) )
ap3.base <- sarima(log(AirPassengers) ~
                   0 | ma(1, c(-0.3)) + sma(12,1, c(-0.1)) + i(2) + s(6),
                   ss.method = "base")

## further unit roots, equivalent specifications for the airline model
tmp.su <- sarima(log(AirPassengers) ~
                 0 | ma(1, c(-0.3)) + sma(12,1, c(-0.1)) + i(2) + s(2) + su(12,1:5),
                 ss.method = "base")
tmp.su$interna$delta_poly
prod(tmp.su$interna$delta_poly)
zapsmall(coef(prod(tmp.su$interna$delta_poly)))
tmp.su

tmp.u <- sarima(log(AirPassengers) ~
                0 | ma(1, c(-0.3)) + sma(12,1, c(-0.1)) + i(2) + s(2) + u((1:5)/12),
                ss.method = "base")
tmp.u
```

`SarimaModel-class` *Class SarimaModel in package sarima*

Description

Class SarimaModel in package sarima.

Objects from the Class

Class `"SarimaModel"` represents standard SARIMA models. Objects can be created by calls of the form `new("SarimaModel", ..., ar, ma, sar, sma)`, using named arguments in the form `slotname = value`, where `slotname` is one of the slots, see below. The arguments have natural defaults. It may be somewhat surprising though that the default for the variance of the innovations (slot `"sigma2"`) is `NA`. The rationale for this choice is that for some calculations the innovations' variance is not needed and, more importantly, it is far too easy to forget to include it in the model (at least for the author) when the variance matters. The latter may may lead silently to wrong results if the "natural" default value of one is used when `sigma2` matters.

The models may be specified in intercept (`center = 0`) or mean-corrected (`intercept = 0`) form. Setting both to non-zero values is accepted but rarely needed.

If you waih to modify an existing object from class `"SarimaModel"`, give it as an unnamed argument to `"new"` and specify only the slots to be changed, see the examples.

Use `as.SarimaModel` to convert a model fitted with `stats::arima()` to `"SarimaModel"`.

Slots

`center`**:** Object of class `"numeric"`, a number, the ARIMA equation is for `X(t) - center`.

`intercept`**:** Object of class `"numeric"`, a number, the intercept in the ARIMA equation.

`sigma2`: Object of class `"numeric"`, a positive number, the innovations variance.

`nseasons`: Object of class `"numeric"`, a positive integer, the number of seasons. For non-seasonal models this is NA.

`iorder`: Object of class `"numeric"`, non-negative integer, the integration order.

`siorder`: Object of class `"numeric"`, non-negative integer, the seasonal integration order.

`ar`: Object of class `"BJFilter"`, the non-seasonal AR part of the model.

`ma`: Object of class `"SPFilter"`, the non-seasonal MA part of the model.

`sar`: Object of class `"BJFilter"`, the seasonal AR part of the model.

`sma`: Object of class `"SPFilter"`, the seasonal MA part of the model.

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

SARIMA models contain as special cases a number of models. The one-argument method of `modelCoef` is essentially a definition of model coefficients for SARIMA models. The two-argument methods request the model coefficients according to the convention of the class of the second argument. The second argument may also be a character string naming the target class.

Essentially, the methods for `modelCoef` are a generalisation of `as()` methods and can be interpreted as such (to an extent, the result is not necessarilly from the target class, not least because the target class may be virtual).

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

modelCoef `signature(object = "SarimaModel", convention = "SarimaFilter")`: Converts `object` to "SarimaFilter", equivalent to the one-argument call `modelCoef(object)`.

modelCoef `signature(object = "SarimaModel", convention = "ArFilter")`: Convert `object` to "ArFilter". An error is raised if `object` has non-trivial moving average part.

modelCoef `signature(object = "SarimaModel", convention = "MaFilter")`: Convert `object` to "MaFilter". An error is raised if `object` has non-trivial autoregressive part.

modelCoef `signature(object = "SarimaModel", convention = "ArmaFilter")`: Convert `object` to "ArmaFilter". This operation always successeds.

modelCoef `signature(object = "SarimaModel", convention = "character")`: The second argument gives the name of the target class. This is conceptually equivalent to `modelCoef(object, new(convention))`.

`modelOrder` gives the order of the model according to the conventions of the target class. An error is raised if `object` is not compatible with the target class.

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")`: ...

The polynomials associated with `object` can be obtained with the following methods. Note that target "ArmaFilter" gives the fully expanded products of the AR and MA polynomials, as needed, e.g., for filtering.

modelPoly `signature(object = "SarimaModel", convention = "ArmaFilter")`: ‘ Gives the fully expanded polynomials as a list

modelPoly `signature(object = "SarimaModel", convention = "missing")`: Gives the polynomials associated with the model as a list.

modelPolyCoef `signature(object = "SarimaModel", convention = "ArmaFilter")`: Give the coefficients of the fully expanded polynomials as a list.

modelPolyCoef `signature(object = "SarimaModel", convention = "missing")`: Gives the coefficients of the polynomials associated with the model as a list.

Author(s)

Georgi N. Boshnakov

See Also

`ArmaModel`

Examples

```
ar1 <- new("SarimaModel", ar = 0.9)
ar1c <- new("SarimaModel", ar = 0.9, intercept = 3)
ar1c
ar1m <- new("SarimaModel", ar = 0.9, center = 1)
ar1m

sm0 <- new("SarimaModel", nseasons = 12)
sm1 <- new("SarimaModel", nseasons = 12, intercept = 3)
sm1
## alternatively, pass a model and modify with named arguments
sm1b <- new("SarimaModel", sm0, intercept = 3)
identical(sm1, sm1b) # TRUE

## in the above models sigma2 is NA

## sm2 - from scratch, the rest modefy an existing model
sm2  <- new("SarimaModel", ar = 0.9, nseasons = 12, intercept = 3, sigma2 = 1)
sm2a <- new("SarimaModel", sm0, ar = 0.9, intercept = 3, sigma2 = 1)
sm2b <- new("SarimaModel", sm1, ar = 0.9, sigma2 = 1)
```

```
sm2c <- new("SarimaModel", ar1c, nseasons =12, sigma2 = 1)
identical(sm2, sm2a) # TRUE
identical(sm2, sm2b) # TRUE
identical(sm2, sm2c) # TRUE

sm3 <- new("SarimaModel", ar = 0.9, sar = 0.8, nseasons = 12, intercept = 3,
           sigma2 = 1)
sm3b <- new("SarimaModel", sm2, sar = 0.8)
identical(sm3, sm3b) # TRUE

## The classic 'airline model' (from examples for AirPassengers)
(fit <- arima(log10(AirPassengers), c(0, 1, 1),
              seasonal = list(order = c(0, 1, 1), period = 12)))

as.SarimaModel(fit)
```

---

se *Compute standard errors*

Description

Compute standard errors.

Usage

```
se(object, ...)

## S4 method for signature 'SampleAutocorrelations'
vcov(object, assuming = "iid", maxlag = maxLag(object), ...)
```

Arguments

object	an object containing estimates, such as a fitted model.
...	further arguments for vcov.
assuming	under what assumptions to do the computations? Currently can be "iid", "garch", a fitted model, or a theoretical model, see Details.
maxlag	maximal lag to include

Details

se is a convenience function for the typical case where only the square root of the diagonal of the variance-covariance matrix is needed.

The method for vcov gives the variance-covariance matrix of the first maxlag autocorrelation coefficients in the object. The result depends on the underlying assumptions and the method of calculation. These can be specifyed with the additional arguments.

Argument "assuming" can be though also as specifying a null hypothesis. Setting it to "iid" or "garch" corresponds to strong white noise (iid) and weak white noise, respectively.

Setting "assuming" to an ARMA model (theoretical or fitted) specifies that as the null model.

Note: The method for vcov is not finalised yet. It is used by a method for confint. Bug reports and requests on the github repo may bring this closer to the top of my task list.

Value

for se, a numeric vector giving standard errors;

for the vcov method, a square matrix

See Also

link{confint}, vcov

---

sigmaSq *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 = "InterceptSpec")
```

Author(s)

Georgi N. Boshnakov

`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 or a model object, see 'Details'.
`rand.gen`	random number generator for the innovations.
`n`	length of the time series.
`n.start`	number of burn-in observations.
`x`	initial/before values of the time series, a list, a numeric vector or time series, see Details.
`eps`	initial/before values of the innovations, a list or a numeric vector, see Details.
`xintercept`	non-constant intercept which may represent trend or covariate effects.
`xcenter`	currently ignored.
`...`	additional arguments for `arima.sim` and `rand.gen`, see 'Details'.

Details

The model can be specified by a model object, e.g., from class SarimaModel. It can also be 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.

If the model is stationary the generated time series is stationary starting with the first value. In particular, there is no need for a 'warm-up' period.

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

For multple simulations with the same (or almost the same) setup, it is better to execute `prepareSimSarima` once and call the function returned by it as many times as needed.

Value

an object of class "ts", a simulated time series from the given model

Author(s)

Georgi N. Boshnakov

Examples

```
require("PolynomF") # guaranteed to be available since package "sarima" imports it.

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, sigma2 = 1))  # SAR(1), 12 seasons
x <- sim_sarima(n=144,model=list(sma=0.8, nseasons=12, sigma2 = 1))  # SMA(1)

x <- sim_sarima(n=144, model = list(iorder=1, sigma2 = 1)) # (1-B)X_t = e_t (random walk)
acf(x)
acf(diff(x))

x <- sim_sarima(n=144, model = list(iorder=2, sigma2 = 1))  # (1-B)^2 X_t = e_t
x <- sim_sarima(n=144, model = list(siorder=1,
                                    nseasons=12, sigma2 = 1)) # (1-B)^{12} X_t = e_t

x <- sim_sarima(n=144, model = list(iorder=1, siorder=1,
                                    nseasons=12, sigma2 = 1))
x <- sim_sarima(n=144, model = list(ma=0.4, iorder=1, siorder=1,
                                    nseasons=12, sigma2 = 1))
x <- sim_sarima(n=144, model = list(ma=0.4, sma=0.7, iorder=1, siorder=1,
                                    nseasons=12, sigma2 = 1))

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, sigma2 = 1))

x <- sim_sarima(n=144, model = list(iorder=1, siorder=1,
                                    nseasons=12, sigma2 = 1),
                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))
## use xintercept to include arbitrary trend/covariates
sim_sarima(n = 144, model = list(sma = 0.4, ma = 0.4, sar = 0.8, ar = 0.5,
    nseasons = 12, sigma2 = 1), xintercept = 1:144)
```

`spectrum` *Spectral Density*

Description

Estimate the spectral density of a time series or compute the spectral density associated with a time series model.

Usage

```
spectrum(x, standardize = TRUE, ...)

## Default S3 method:
spectrum(x, standardize = TRUE, raw = TRUE, taper = 0.1,
         demean = FALSE, detrend = TRUE, ...)

## S3 method for class 'genspec'
print(x, n.head = min(length(x$spec), 6), sort = TRUE, ...)

## S3 method for class 'Arima'
spectrum(x, standardize = TRUE, ...)

## S3 method for class 'ArmaModel'
spectrum(x, standardize = TRUE, ...)
```

```
## S3 method for class 'SarimaModel'
spectrum(x, standardize = TRUE, ...)

## S3 method for class 'function'
spectrum(x, standardize = TRUE, param = list(), ...)
```

Arguments

x	a model or a univariate or multivariate time series.
standardize	if TRUE, the default method standardises the time series before computing the periodogram, while the methods for models scale the spectral density so that it is a probability density function.
raw	if TRUE, the default, compute a completely raw periodogram, unless further arguments request otherwise, see section Details.
taper, demean, detrend	see spec.pgram and section 'Details'.
...	further arguments for the default method. Currently not used by other methods.
n.head	how many rows to print?
sort	TRUE, FALSE or "max", see section "Details".
param	a named list, specying model parameters for the "function" method, see section 'Details'.

Details

spectrum in package **sarima** is a generic function with a default method its namesake in package **stats**, see spectrum for a full description of its functionality.

Autoprinting of objects returned by spectrum prints concise information and plots the spectrum. This means that a plot is produced, for example, when the result of a call to spectrum() is not assigned to a variable or if a command containing just the name of the object is executed. If you don't want the graph, just assign the result to a variable. For more control over the printing (for example, number of digits) use print(object, ...) explicitly. In that case no plot is produced. If additional graphical parameters are desired, call plot, ...

All methods print some basic information about the object and a table giving the most influential frequencies and their contributions to the spectrum.

Methods for objects representing ARIMA and SARIMA models (fitted or theoretical) compute the corresponding spectral densities. For non-stationary models, the spectral density for the stationary part. These methods for spectrum return objects from class "Spectrum". If standardize = TRUE the spectral density is scaled, so that it integrates to one (and so is a probability density function). For fitted models confidence bands are not computed currently.

The method for class "function" can be used to create objects from class "Spectrum" using a user specified function. The first argument of that function needs to be a vector of frequencies for which to calculate the spectrum. It is conventionally called freq but this is not required. If there are parameters they should not be part of the signature of the function but need to be listed and given values as a named list via argument param, see the examples for class "Spectrum". This method is somewhat experimental but the restrictions might be relaxed in a future release.

The rest of this section describes the default method. For futher details on the other methods see "Spectrum".

The default method for `spectrum`:

The default method is a wrapper for `stats::spectrum()`.

The default method returns an object from class "genspec". It inherits from "spec", the class returned by `stats::spectrum`, and adds some additional components. The main difference though is that it has a print method, which plots the object as discussed above. `raw = FALSE` with no further arguments is equivalent to `stats::spectrum(object)` and computes a raw periodogram (for the standardised time series if `standardize = TRUE`). This still detrends and tapers the series though. `raw = TRUE` sets `detrend` to `FALSE`, `taper` to zero, and `demean` to `TRUE`, to compute a 'completely raw' periodogram. In both cases, further arguments are respected.

Argument `sort` of the `print` method for "genspec" controls the sorting order of the columns of the printed table. If `FALSE`, no sorting is done. If `TRUE`, the spectrum is sorted in decreasing order, so the first row contains the frequency with the highest value of the spectrum. If "max", the local maxima are found and sorted in decreasing order, followed by the rest, also sorted in decreasing order. Note that due to aliasing the local maxima may be shifted from the "true" frequency (e.g. not be exactly on the harmonics of the number of seasons). Tapering and smoothing parameters may help.

The plot method for class "genspec" is inherited from that for "spec", see `?plot.spec`.

Value

for the default method, an object of class "genspec", which inherits from "spec", and contains the following additional components:

`standardized`	TRUE or FALSE,
`nseasons`	number of seasons,
`freq.range`	`numeric(2)`, the frequency range - `(-m/2, m/2]`, where `m = floor(frequency(x))`;

for the remaining methods, an object of class "Spectrum".

Author(s)

Georgi N. Boshnakov

See Also

`spectrum` which is called by the default method to do the work.

class "Spectrum" for further details on the methods for objects returned by `spectrum()`.

Examples

```
## spectral density of the stationary part of a fitted 'airline model'
fit0 <- arima(AirPassengers, order = c(0,1,1),
              seasonal = list(order = c(0,1,1), period = 12))
spectrum(fit0)

## spectral densities of some ARMA models models from Chan and Gray ().
```

```
##   (TODO: complete the reference)
spectrum(ArmaModel(ma = c(-1, 0.6), sigma2 = 1))
spectrum(ArmaModel(ar = 0.5, sigma2 = 1))
spectrum(ArmaModel(ar = 0.5, ma = -0.8, sigma2 = 1))
spectrum(new("SarimaModel", ar = 0.5, sar = 0.9, nseasons = 12, sigma2 = 1))

mo <- new("SarimaModel", ma = -0.4, sma = -0.9, nseasons = 12, sigma2 = 1)
sp1.mo <- spectrum(mo)
## this also plots the object.  (if you are reading the web version, generated
##     by pkgdown, it may not be showing some of the graphs,
##     I haven't figured out why.)
show(sp1.mo)  # equivalently, just sp1.mo

print(sp1.mo)
print(sp1.mo, digits = 4)
plot(sp1.mo)
plot(sp1.mo, standardize = FALSE)

## the object can be used as a function:
head(sp1.mo())
sp1.mo(seq(0, 0.5, length.out = 12))
sp1.mo(seq(0, 0.5, length.out = 12), standardize = FALSE)

sarima1b <- new("SarimaModel", ar = 0.9, ma = 0.1, sar = 0.5, sma = 0.9,
                nseasons = 12, sigma2 = 1)
spectrum(sarima1b)

## default method for spectrum()

## frequency range is c(-1/2, 1/2] since frequency(x) = 1
frequency(lh)
spectrum(lh)

## frequency range is c(-12/2, 12/2] since frequency(x) = 12
frequency(ldeaths)
( sp <- spectrum(ldeaths) )
print(sp) # equivalently: print(sp, sort = TRUE)
print(sp, sort = FALSE, n.head = 3)
print(sp, sort = "max")
plot(sp)
plot(sp, log = "dB")  # see ?plot.spec for further arguments
```

Spectrum-class *Class* "Spectrum"

Description

Objects from class "Spectrum" spectra computed by spectrum.

Usage

```
## S3 method for class 'Spectrum'
print(x, ..., n = 128, standardize = TRUE)

## S3 method for class 'Spectrum'
plot(x, y, to, from = y, n = 128, standardize = TRUE,
    log = NULL, main = "Spectral density", xlab = "Frequency", ylab = NULL, ...)
```

Arguments

x	a "Spectrum" object.
y	not used but same as from
from, to	interval of frequencies to plot, defaults to $[0, 1/2]$.
n	number of points to plot (for the plot method), number of points to look at for the peaks and troughs (print method).
standardize	if TRUE make the spectral density integrate to one (i.e., be a probability density function).
log	if log = "y" plot the logarithm of the spectral density, see ?plot.default.
main	a character string, the title of the plot.
xlab	a character string, the label for the x-axis.
ylab	a character string, the label for the y-axis. If NULL, the default, the label is set to "Spd" or "log(Spd)", depending on the value of argument log.
...	for print, further arguments for print.default(); for plot, further arguments for curve.

Details

"Spectrum" is an S4 class and as such autoprinting calls the "Spectrum" method for show(), which prints and plots. show has a single argument, the object. For more control over printing, call print which has additional arguments. Similarly, call plot for more flexible graphics.

print(object) (i.e., without further arguments) is equivalent to show(object), except that the former returns object while the latter returns NULL (both invisibly), as is standard for these functions. If print is called with further arguments. the spectrum is not plotted.

The peaks and troughs printed by print are computed by evaluating the spectral density at n equially spaced points and recording the maxima of the resulting discrete sequence. Set argument n to get a finer/coarser grid or to force calculations for particular frequencies. For example, a multiple of 12 may be suitable for n if the data is monthly.

Except for x and standardize the arguments of the plot method are as for curve. With the default standardize = TRUE the spectral density integrates to one over one whole period (usually $(-1/2, 1/2]$ but due to its symmetry it is usually plotted over the second half of that interval.

Objects from the Class

Objects contain spectra produced by sarima::spectrum, see spectrum for details.

Objects can also be created by calling "new" but this is not recommended and currently considered internal.

Slots

`.Data`: Object of class `"function"` ~~

`call`: Object of class `"call"` ~~

`model`: Object of class `"ANY"`, the underlying model.

Methods

plot `signature(x = "Spectrum", y = "ANY")`: plots `x`.

show `signature(object = "Spectrum")`:

plots `object` and prints succinct information about it, including the peaks and troughs in the spectral density. It is equivalent to calling `print` and `plot` with a single argument, see section 'Details'.

Author(s)

Georgi N. Boshnakov

See Also

`spectrum` for details and further examples,

`ArmaSpectrum` for ARMA spectra

Examples

```
## ARFIMA(0,d,0) with parameters 'freq' and 'd'
spARFIMA0d0 <- function(freq){  sigma2 / (2 * sin(2*pi*freq/2)^(2 * d)) }
sp <- spectrum(spARFIMA0d0, param = list(sigma2 = 1, d = 0.2))
print(sp, digits = 4)
## evaluate the spd at selected frequencies
sp(c(0:4 / 8))

## argument 'freq' doesn't need to be called 'freq' but it needs to be
##   the first one. This is equivalent to above:
spARFIMA0d0b <- function(x){  sigma2 / (2 * sin(2*pi*x/2)^(2 * d)) }
spb <- spectrum(spARFIMA0d0b, param = list(sigma2 = 1, d = 0.2))
plot(spb)

## An example without parameters, as above with sigma2 = 1, d = 0.2 hard
##   coded:
spARFIMA0d0c <- function(freq){  1 / (2 * sin(2*pi*freq/2)^(2 * 0.2)) }
spc <- spectrum(spARFIMA0d0c)
print(spc, digits = 4)

spc(c(0:4 / 8))
all.equal(spc(c(0:4 / 8)), sp(c(0:4 / 8))) # TRUE
```

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.

Author(s)

Georgi N. Boshnakov

tsdiag.Sarima *Diagnostic Plots for fitted seasonal ARIMA models*

Description

Produce diagnostics for fitted seasonal ARIMA models. The method offers several portmanteau tests (including Ljung-Box, Li-McLeod and Box-Pierce), plots of autocorrelations and partial autocorrelations of the residuals, ability to control which graphs are produced (including interactively), as well as their layout.

Usage

```
## S3 method for class 'Sarima'
tsdiag(object, gof.lag = NULL, ask = FALSE, ..., plot = 1:3, layout = NULL)

# if 'object' is produced by stats::arima(), forecast::auto.arima() and
# similar, use the full name, 'tsdiag.Sarima()', in the call. The
# arguments are the same.
```

Arguments

object	fitted (seasonal) ARIMA model. currently the output of sarima, stats::arima or compatible (e.g., forecast::Arima and forecast::auto.arima). If object is not from sarima, use the full name, tsdiag.Sarima, of the method when calling it, see the examples.
gof.lag	maximal lag for portmanteau tests.
ask	if TRUE present a menu of available plots, see section 'Details'.
...	not used.
plot	if TRUE all available plots; a vector of positive integers specifies a subset of the available plots.
layout	a list with arguments for graphics::layout for the plots. The default is as for the ARIMA method of stats::tsdiag.

Details

Compute and graph diagnostics for seasonal ARIMA models. For objects of class "Sarima" (produced by sarima) just call the generic, tsdiag. The method can be called also directly on the output from base R's arima() with tsdiag.Sarima() or sarima::tsdiag.Sarima().

The method offers several portmanteau tests (including Ljung-Box, Li-McLeod and Box-Pierce), plots of autocorrelations and partial autocorrelations of the residuals, ability to control which graphs are produced (including interactively), as well as their layout.

The method always makes a correction of the degrees of freedom of the portmanteau tests (roughly, subtracting the number of estimated ARMA parameters). Note that stats::tsdiag doesn't do that.

plot can be TRUE to ask for all plots or a vector of positive integers specifying which plots to consider. Currently the following options are available:

1 residuals
2 acf of residuals
3 p values for Ljung-Box statistic
4 p values for Li-McLeod statistic
5 p values for Box-Pierce statistic
6 pacf of residuals

The default is plot = 1:3, which produces a plot similar to the one from stats::tsdiag (but with adjusted d.f., see above). If plot is TRUE, you probably need also ask = TRUE.

If argument plot is of length two the graphics window is split into 2 equal subwindows. Argument layout can still be used to change this. If argument plot is of length one the graphics window is not split at all.

In interactive sessions, if the number of requested graphs (as specified by argument plot) is larger than the number of graphs specified by the layout (by default 3), the function makes the first graph and then presents a menu of the requested plots.

Argument layout can be used to change the layout of the plot, for example to put two graphs per plot, see the examples. Currently it should be a list of arguments for layout, see ?layout. Don't call layout youself, as that will change the graphics device prematurely.

The computed results are returned (invisibly). This is another difference from stats::tsdiag which doesn't return them.

Value

a list with components:

`residuals`	residuals
`LjungBox`	Ljung box test
`LiMcLeod`	LiMcLeod test
`BoxPierce`	BoxPierce test

Only components that are actually computed are included, the rest are NULL or absent.

Author(s)

Georgi N. boshnakov

See Also

`tsdiag`

Examples

```
ap.baseA <- sarima(log(AirPassengers) ~
    0 | ma(1, c(-0.3)) + sma(12,1, c(-0.1)) + i(1) + si(12,1),
    ss.method = "base")
tsdiag(ap.baseA)

## apply the method on objects from arima()
ap.arima <- arima(log(AirPassengers), order = c(0,1,1), seasonal = c(0,1,1))
tsdiag.Sarima(ap.arima)
## use Li-McLeod test instead of Ljung-Box
tsdiag.Sarima(ap.arima, plot = c(1:2,4))
## call R's tsdiag method, for comparison:
tsdiag(ap.arima, plot = c(1:2,4))

## plot only acf and p-values
tsd <- tsdiag.Sarima(ap.arima, plot = c(2:3), layout = list(matrix(1:2, nrow = 2)))
## the results can be used for further calculations:
head(tsd$LjungBox$test, 4)


## plot resid, acf, and p-values, leaving half the space for residuals
tsdiag.Sarima(ap.arima, plot = c(1:3), layout = list(matrix(1:3, nrow = 3),
                                                     heights = c(1,2,2)))


## four plots arranged as a 2x2 matrix.
tsdiag.Sarima(ap.arima, plot = c(2:5), layout = list(matrix(1:4, nrow = 2)))
```

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. The null hypothesis is identified by argument h0, based on which whiteNoiseTest chooses a suitable function to call. The functions implementing the tests are also available to be called directly and their documentation should be consulted for further arguments that are available.

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 Francq & Zakoian (2010). 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.

Note

Further methods will be added in the future.

Author(s)

Georgi N. Boshnakov

References

Francq C, Zakoian J (2010). *GARCH models: structure, statistical inference and financial applications*. John Wiley & Sons. ISBN 978-0-470-68391-0.

Li WK (2004). *Diagnostic checks in time series*. Chapman & Hall/CRC Press.

See Also

acfGarchTest (h0 = "garch"), acfIidTest (h0 = "iid");

acfMaTest

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 = NULL, eps = NULL, 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.

model$center and model$intercept are scalars and usually at most one of them is nonzero. They can be considered part of the model specification. In contrast, arguments xcenter and xintercept are vectors of the same length as x. They can represent contributions from covariate variables. Usually at most one of xcenter and xintercept is used.

The description below uses μ_t and c_t for the contributions by model$center plus xcenter and model$intercept plus xintercept, respectively. The time series $\{x_t\}$ and $\{\varepsilon_t\}$ are represented by x and eps in the R code. Let

$$y_t = x_t - \mu_t$$

be the centered series. where the centering term μ_t is essentially the sum of center and xcenter and is not necessarilly the mean. 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).

If whiten = FALSE, y_t is computed for t=from,...,n using the above formula, i.e. the filter is applied to get y from eps (and some initial values). If eps is white noise, it can be said that y is obtained by "colouring" the white noise eps. This can be used, for example, to simulate ARIMA time series. Finally, the centering term is added back, $x_t = y_t + \mu_t$ for t=from,...,n, and the modified x is returned. The first from - 1 elements of x are left unchanged.

The inverse filter is obtained by rewriting the above equation as an equation expressing ε_t in terms of the remaining quantities:

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

If whiten = TRUE, xarmaFilter uses this formula for t=from,...,n to compute eps from y (and some initial values). If eps is white noise, then it can be said that the time series y has been whitened.

In both cases the first few values in x and/or eps are used as initial values.

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.

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

Residuals corresponding to a series x can be obtained by setting whiten = TRUE. In this case x is the input series. The elements of the output vector eps are calculated by the formula for ε_t given above. 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 and provide the first from - 1 values in the returned vectors. Argument from should not be smaller than the default value max(p,q)+1.

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

Value

the result of applying the filter or its inverse, as descibed in Details: if whiten = FALSE, the modified x; if whiten = TRUE, the modified eps.

Author(s)

Georgi N. Boshnakov

Examples

```
## define a seasonal ARIMA model
m1 <- new("SarimaModel", iorder = 1, siorder = 1, ma = -0.3, sma = -0.1, nseasons = 12)

model0 <- modelCoef(m1, "ArmaModel")
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)
## compute predictions starting at from = 14
pred1 <- sarima.f(past = AirPassengers[1:13], n = 131, ar = model1$ar, ma = model1$ma)
pred2 <- xarmaFilter(model1, x = ap, whiten = FALSE)
pred2 <- pred2[-(1:13)]
```

```
all(pred1 == pred2) ##TRUE
```

Index