Package ‘dlmodeler’

February 15, 2013

Title Generalized Dynamic Linear Modeler

Version 1.2-1

Date 2012-07-11

Author Cyrille Szymanski <cnszym@gmail.com>

Maintainer Cyrille Szymanski <cnszym@gmail.com>

Description dlmodeler is a set of user-friendly functions to simplify the state-space modelling, fitting, analysis and forecasting of Generalized Dynamic Linear Models (DLMs). It includes functions to name and extract individual components of a DLM, build classical seasonal time-series models (monthly, quarterly,yearly, etc. with calendar adjustments) and provides a unified interface compatible with other state-space packages including: dlm, FKF and KFAS.

License GPL (>= 2) | BSD

Suggests KFAS (>= 0.9.9), FKF (>= 0.1.2), dlm (>= 1.1-2)

Depends R (>= 2.15.0)

Repository CRAN

Date/Publication 2012-07-12 07:56:06

NeedsCompilation no

R topics documented:

---

dlmodeler-package *Generalized Dynamic Linear Modeler*

Description

Package `dlmodeler` is a set of user friendly functions to simplify the state-space modelling, fitting, analysis and forecasting of Generalized Dynamic Linear Models (DLMs).

It includes functions to name and extract components of a DLM, and provides a unified interface compatible with other state-space packages including: `dlm`, `KFAS`, `FKF` and `sspir`.

Details

The distinguishing aspect of this package is that it provides functions for naming and extracting components of DLMs (see below), and a unified interface compatible with other state-space R packages for filtering and smoothing:

- package `KFAS`: implements exact diffuse initialization and supports filtering, smoothing and likelihood computation for the exponential family state-space models (as of v0.9.9), used by default
- package `dlm`: good general purpose package with many helper functions available and some support for Bayesian analysis (as of v1.1-2)
- package `FKF`: very fast and memory efficient but has no smoothing algorithm (as of v0.1.2)
- package `sspir`: provides (extended) Kalman filter and Kalman smoother for models with support for the exponential family, but it has no support for the multivariate case, exact diffuse initialization or importance sampling, and does not support missing values in covariates (as of v0.2.8)

Introduction

Generalized Dynamic Linear Models are a powerful approach to time-series modelling, analysis and forecasting. This framework is closely related to the families of regression models, ARIMA models, exponential smoothing, and structural time-series (also known as unobserved component models, UCM).

The origin of DLM time-series analysis has its roots in the world of engineering. In order to control dynamic physical systems, unknown quantities such as velocity and position (the state of the system) need to be estimated from noisy measurements such as readings from various sensors (the observations). The state of the system evolves from one state (e.g. position and speed at time t) to another (position and speed at time t+1) according to a known transition equation, possibly including random perturbations and intervention effects. The observations are derived from the state values by a an observation equation (e.g. observation at time t = position + noise), also possibly including random disturbances and intervention effects.

The challenge is to obtain the best estimate of the unknown state considering the set of available observations at a given point in time. Due to the presence of noise disturbances, it is generally not possible to simply use the observations directly because they lead to estimators which are too erratic. During the 1960s, the Kalman filtering and smoothing algorithm was developed and popularized to efficiently and optimally solve this etimation problem. The technique is based on an iterative procedure in which state values are successively predicted given the knowledge of the past observations, and then updated upon the reception of the next observation. Because of the predict-and-update nature of Kalman filtering, it can also be interpreted under a Bayesian perspective.

Dynamic linear models

The theory developed for the control of dynamic systems has a direct application to the general analysis of time-series. By having a good estimate of the current state and dynamics of the system, it is possible to derive assumptions about their evolution and subsequent values; and therefore to obtain a forecast for the future observations.

Dynamic Linear Models are a special case of general state-space models where the state and the observation equations are linear, and the distributions follow a normal law. They are also referred to as gaussian linear state-space models. Generalized DLMs relax the assumption of normality by allowing the distribution to be any of the exponential family of functions (which includes the Bernoulli, binomial and Poisson distributions, useful in particular for count data).

There are two constitutive operations for dynamic linear models: filtering and smoothing. In a few words, filtering is the operation consisting in estimating the state values at time t, using only observations up to (and including) t-1. On the contrary, smoothing is the operation which aims at estimating the state values using the whole set of observations.

State-space form and notations

The state-space model is represented as follows:

- initial state: $alpha(0) \sim N(a(0), P(0))$
- observation equation: $y(t) = Z(t)alpha(t) + eta(t)$
- transition equation: $alpha(t+1) = T(t)alpha(t) + R(t)eps(t)$
- observation disturbance: $eta(t) \sim N(0, H(t))$
- state disturbance: $eps(t) \sim N(0, Q(t))$
- state mean and covariance matrix: $a(t) = E[alpha(t)]$ and $P(t) = cov(alpha(t))$

With:

- n = number of time-steps $(t = 1..n)$

- m = dimension of state vector
- $alpha(t)$ = state vector $(m, 1)$
- $a(0)$ = initial state vector $(m, 1)$
- $P(0)$ = initial state covariance matrix (m, m)
- $Pinf(0)$ = diffuse part of $P(0)$ matrix (m, m)
- d = dimension of observation vector
- $y(t)$ = observation vector $(d, 1)$
- $Z(t)$ = observation design matrix (d, m) if constant, or (d, m, n)
- $eta(t)$ = observation disturbance vector $(d, 1)$
- $H(t)$ = observation disturbance covariance matrix (d, d) if constant, or (d, d, n)
- $T(t)$ = state transition matrix (m, m) if constant, or (m, m, n)
- r = dimension of the state disturbance covariance matrix
- $eps(t)$ = state disturbance vector $(r, 1)$
- $R(t)$ = state disturbance selection matrix (m, r) if constant, or (m, r, n)
- $Q(t)$ = state disturbance covariance matrix (r, r) if constant, or (r, r, n)

Components

DLMs are constructed by combining several terms together. The model consisiting of $level + trend + seasonal + cycle$ is an example of how individual elements can be added together to form a more complete model. A typical analysis will consider the model as a whole, but also look into the values of indivdual terms, for example the variations in the $level$, and the evolution of the shape of the $seasonal$ terms. This is also known as seasonal adjustment.

This package introduces a notion called the "component" to facilitate the analysis of the DLMs. Mathematically speaking, a component is a named subset of state variables. Components are automatically created when the model is built, and the package provides functions which makes it easier to access and analyze their values afterwards: expectation, variance and prediction/confidence bands.

Notes

Work is in progress to provide generalized DLM support.

Maintainer

Cyrille Szymanski <cnszym@gmail.com>

Author(s)

Cyrille Szymanski <cnszym@gmail.com>

References

Harvey, A.C. Forecasting, Structural Time Series Models and the Kalman Filter. Cambridge University Press (1989).

Durbin, J. and Koopman, S. J. Time Series Analysis by State Space Methods. Oxford University Press (2001). http://www.ssfpack.com/dkbook/

Commandeur, and Koopman, An Introduction to State Space Time Series Analysis, Oxford University Press (2007).

Petris, Petrone, and Campagnoli, Dynamic Linear Models with R, Springer (2009).

See Also

Other R packages and functions of interest (in alphabetical order):

- decompose() from {stats}: Decompose a time series into seasonal, trend and irregular components using moving averages. Deals with additive or multiplicative seasonal component.
- Package `dlm`: Maximum likelihood, Kalman filtering and smoothing, and Bayesian analysis of Normal linear State Space models, also known as Dynamic Linear Models.
- Package `dse`: Package dse provides tools for multivariate, linear, time-invariant, time series models. It includes ARMA and state-space representations, and methods for converting between them. It also includes simulation methods and several estimation functions. The package has functions for looking at model roots, stability, and forecasts at different horizons. The ARMA model representaion is general, so that VAR, VARX, ARIMA, ARMAX, ARIMAX can all be considered to be special cases. Kalman filter and smoother estimates can be obtained from the state space model, and state-space model reduction techniques are implemented. An introduction and User's Guide is available in a vignette.
- Package `FKF`: This is a fast and flexible implementation of the Kalman filter, which can deal with NAs. It is entirely written in C and relies fully on linear algebra subroutines contained in BLAS and LAPACK. Due to the speed of the filter, the fitting of high-dimensional linear state space models to large datasets becomes possible. This package also contains a plot function for the visualization of the state vector and graphical diagnostics of the residuals.
- Package `forecast`: Methods and tools for displaying and analysing univariate time series forecasts including exponential smoothing via state space models and automatic ARIMA modelling.
- HoltWinters() from {stats}: Computes Holt-Winters Filtering of a given time series. Unknown parameters are determined by minimizing the squared prediction error.
- Package `KFAS`: Package KFAS provides functions for Kalman filtering, state, disturbance and simulation smoothing, forecasting and simulation of state space models. All functions can use exact diffuse initialisation when distributions of some or all elements of initial state vector are unknown. Filtering, state smoothing and simulation functions use sequential processing algorithm, which is faster than standard approach, and it also allows singularity of prediction error variance matrix. KFAS also contains function for computing the likelihood of exponential family state space models and function for state smoothing of exponential family state space models.
- Package `MARSS`: The MARSS package fits constrained and unconstrained linear multivariate autoregressive state-space (MARSS) models to multivariate time series data.

- Package `sspir`: A glm-like formula language to define dynamic generalized linear models (state space models). Includes functions for Kalman filtering and smoothing. Estimation of variance matrices can be perfomred using the EM algorithm in case of Gaussian models. Read help(sspir) to get started.
- stl() from {stats}: Decompose a time series into seasonal, trend and irregular components using loess, acronym STL.
- StructTS() and tsSmooth() from {stats}: Fit a structural model for a time series by maximum likelihood. Performs fixed-interval smoothing on a univariate time series via a state-space model. Fixed-interval smoothing gives the best estimate of the state at each time point based on the whole observed series.

Other software of interest (in alphabetical order):

- SAS PROC UCM `http://www.sas.com/products/ets`
- SsfPack `http://www.ssfpack.com`
- STAMP `http://www.stamp-software.com`
- S+FinMetrics `http://www.insightful.com/products/finmetrics`
- TRAMO-SEATS `http://www.bde.es/servicio/software/econome.htm`
- X12-ARIMA `http://www.census.gov/srd/www/x12a/`
- X13-ARIMA-SEATS `http://www.census.gov/ts/papers/jsm09bcm.pdf`

Examples

```
require(dlmodeler)

# This section illustrates most of the possibilities offered by the
# package by reproducing famous examples of state-space time-series
# analysis found in the litterature.

###############################################
# analysis from Durbin & Koopman book page 32 #
# random walk                                 #
###############################################

# load and show the data
y <- matrix(Nile,nrow=1)
plot(y[1,],type='l')

# y(t)   = a(t) + eta(t)
# a(t+1) = a(t) + eps(t)
# with the parametrization (phi) proposed in the book
build.fun <- function(p) {
varH <- exp(p[1])
varQ <- exp(p[2])*varH
dlmodeler.build.polynomial(0,sqrt(varH),sqrt(varQ),name='p32')
}

# fit the model by maximum likelihood estimation
fit <- dlmodeler.fit.MLE(y, build.fun, c(0,0), verbose=FALSE)
```

```
# compare the fitted parameters with those reported by the authors
fit$par[2]        # psi = -2.33
fit$model$Ht[1,1] # H   = 15099
fit$model$Qt[1,1] # Q   = 1469.1

# compute the filtered and smoothed values
f <- dlmodeler.filter(y, fit$mod, smooth=TRUE)

# f.ce represents the filtered one steap ahead observation
# prediction expectations E[y(t) | y(1), y(2), ..., y(t-1)]
f.ce <- dlmodeler.extract(f, fit$model,
                          type="observation", value="mean")

# s.ce represents the smoothed observation expectations
# E[y(t) | y(1), y(2), ..., y(n)]
s.ce <- dlmodeler.extract(f$smooth, fit$model,
                          type="observation", value="mean")

# plot the components
plot(y[1,],type='l')
lines(f.ce$p32[1,],col='light blue',lty=2)
lines(s.ce$p32[1,],col='dark blue')


################################################
# analysis from Durbin & Koopman book page 163 #
# random walk + stochastic seasonal            #
################################################

# load and show the data
y <- matrix(log(Seatbelts[,'drivers']),nrow=1)
plot(y[1,],type='l')

# y(t)    = a(t) + s1(t) + eta(t)
# a(t+1)  = a(t) + eps_L(t)
# s1(t+1) = -s2(t) - s3(t) - ... - s12(t) + eps_S(t)
# s2(t+1) = s1(t)
# s3(t+1) = s2(t), etc.
mod <- dlmodeler.build.structural(
pol.order=0,
pol.sigmaQ=NA,
dseas.order=12,
dseas.sigmaQ=NA,
sigmaH=NA,
name='p163')

# fit the model by maximum likelihood estimation
fit <- dlmodeler.fit(y, mod, method="MLE")

# compare the fitted parameters with those reported by the authors
fit$model$Ht[1,1] # H  = 0.0034160
fit$model$Qt[1,1] # Q1 = 0.00093585
```

```
fit$model$Qt[2,2] # Q2 = 5.0109e-007

# compute the filtered and smoothed values
f <- dlmodeler.filter(y, fit$model, smooth=TRUE)

# f.ce represents the filtered one steap ahead observation
# prediction expectations E[y(t) | y(1), y(2), ..., y(t-1)]
f.ce <- dlmodeler.extract(f, fit$model,
                          type="observation", value="mean")

# s.ce represents the smoothed observation expectations
# E[y(t) | y(1), y(2), ..., y(n)]
s.ce <- dlmodeler.extract(f$smooth, fit$model,
                          type="observation", value="mean")

# plot the components
plot(y[1,])
lines(f.ce$level[1,],col='light blue',lty=2)
lines(s.ce$level[1,],col='dark blue')

# note that the smoothed seasonal component appears to be constant
# throughout the serie, this is due to Qt[2,2]=sigmaQS being close to
# zero. Durbin & Koopman treat the seasonal component as deterministic
# in the remainder of their models.
plot(y[1,]-s.ce$level[1,],ylim=c(-.5,.5))
lines(f.ce$seasonal[1,],type='l',col='light green',lty=2)
lines(s.ce$seasonal[1,],type='l',col='dark green')


#########################################################
# analysis from Durbin & Koopman book page 166          #
# random walk + seasonal + seat belt law + petrol price #
#########################################################

# load and show the data
y <- matrix(log(Seatbelts[,'drivers']),nrow=1)
law <- matrix(Seatbelts[,'law'],nrow=1)
petrolprice <- matrix(log(Seatbelts[,'PetrolPrice']),nrow=1)
par(mfrow=c(3,1))
plot(y[1,],type='l')
plot(petrolprice[1,],type='l')
plot(law[1,],type='l')

# y(t)    = a(t) + s1(t) + lambda*law + mu*petrolprice + eta(t)
# a(t+1)  = a(t) + eps(t)
# s1(t+1) = -s2(t) - s3(t) - ... - s12(t)
# s2(t+1) = s1(t)
# s3(t+1) = s2(t), etc.
m1 <- dlmodeler.build.structural(
pol.order=0,
dseas.order=12,
sigmaH=NA, pol.sigmaQ=NA)
m2 <- dlmodeler.build.regression(
```

```
law,
sigmaQ=0,
name='law')
m3 <- dlmodeler.build.regression(
petrolprice,
sigmaQ=0,
name='petrolprice')
mod <- dlmodeler.add(dlmodeler.add(m1,m2),m3,name='p166')

# fit the model by maximum likelihood estimation
fit <- dlmodeler.fit(y, mod, method="MLE")

# compute the filtered and smoothed values
f <- dlmodeler.filter(y, fit$model, smooth=TRUE)

# E[y(t) | y(1), y(2), ..., y(t-1)]
f.ce <- dlmodeler.extract(f, fit$model,
                          type="observation", value="mean")
# E[y(t) | y(1), y(2), ..., y(n)]
s.ce <- dlmodeler.extract(f$smooth, fit$model,
                          type="observation", value="mean")
# E[a(t) | y(1), y(2), ..., y(t-1)]
fa.ce <- dlmodeler.extract(f, fit$model,
                           type="state", value="mean")
# E[a(t) | y(1), y(2), ..., y(n)]
sa.ce <- dlmodeler.extract(f$smooth, fit$model,
                           type="state", value="mean")

# see to which values the model has converged and
# compare them with those reported by the authors
fa.ce$law[1,193]         # law         = -0.23773
fa.ce$petrolprice[1,193] # petrolprice = -0.29140

# plot the smoothed de-seasonalized serie
par(mfrow=c(1,1))
plot(y[1,])
lines(s.ce$level[1,]+s.ce$law[1,]+s.ce$petrolprice[1,],
      col='dark blue')

# show the AIC of the model
AIC(fit)


###################################
# testing other fitting functions #
###################################

# load and show the data
y <- matrix(Nile,nrow=1)
plot(y[1,],type='l')

# random walk
# y(t)   = a(t) + eta(t)
```

```
# a(t+1) = a(t) + eps(t)
mod <- dlmodeler.build.polynomial(0,sigmaQ=NA,name='p32')

# fit the model by maximum likelihood estimation and compute the
# 1-step ahead MSE
fit.mle <- dlmodeler.fit(y, mod, method="MLE")
mean((fit.mle$filtered$f[1,10:100]-y[1,10:100])^2)

# fit the model by minimizing the 1-step ahead MSE
fit.mse1 <- dlmodeler.fit(y, mod, method="MSE", ahead=1, start=10)
mean((fit.mse1$filtered$f[1,10:100]-y[1,10:100])^2)

# fit the model by minimizing the 4-step ahead MSE
fit.mse4 <- dlmodeler.fit(y, mod, method="MSE", ahead=4, start=10)
mean((fit.mse4$filtered$f[1,10:100]-y[1,10:100])^2)

# compare the 1-step ahead forecasts for these models
# as can be expected, the MLE and MSE1 models roughly
# have the same means
plot(y[1,],type='l')
lines(fit.mle$filtered$f[1,],col='dark blue')
lines(fit.mse1$filtered$f[1,],col='dark green')
lines(fit.mse4$filtered$f[1,],col='dark red')


##################################################
# looking at variances and prediction intervals #
##################################################

# load and show the data
y <- matrix(log(Seatbelts[,'drivers']),nrow=1)
plot(y[1,],type='l')

# model with level + seasonal
mod <- dlmodeler.build.structural(
pol.order=0,
pol.sigmaQ=NA,
dseas.order=12,
dseas.sigmaQ=NA,
sigmaH=NA,
name='p163')

# fit the model by maximum likelihood estimation, filter & smooth
fit <- dlmodeler.fit(y, mod, method="MLE")
fs <- dlmodeler.filter(y, fit$model, smooth=TRUE)

# value we will be using to compute 90% prediction intervals
prob <- 0.90

# true output mean + prediction interval
output.intervals <- dlmodeler.extract(fs,fit$model,
                    type="observation",value="interval",prob=prob)
plot(y[1,],xlim=c(100,150))
```

```
lines(output.intervals$p163$mean[1,],col='dark green')
lines(output.intervals$p163$lower[1,],col='dark grey')
lines(output.intervals$p163$upper[1,],col='dark grey')

# true state level mean + prediction interval
state.intervals <- dlmodeler.extract(fs, fit$model,type="state",
                                     value="interval",prob=prob)
plot(y[1,])
lines(state.intervals$level$mean[1,],col='dark green')
lines(state.intervals$level$lower[1,],col='dark grey')
lines(state.intervals$level$upper[1,],col='dark grey')

# true state seasonal mean + prediction interval
plot(state.intervals$seasonal$mean[1,],
     ylim=c(-.4,.4),xlim=c(100,150),
     type='l',col='dark green')
lines(state.intervals$seasonal$lower[1,],col='light grey')
lines(state.intervals$seasonal$upper[1,],col='light grey')
```

`AIC.dlmodeler.fit` *Log-likelihood and AIC of a model*

Description

Returns the log-likelihood or the AIC for a fitted DLM object.

Usage

```
## S3 method for class 'dlmodeler.filtered'
logLik(object, ...)

## S3 method for class 'dlmodeler.fit'
logLik(object, ...)

## S3 method for class 'dlmodeler.fit'
AIC(object, ..., k = 2)
```

Arguments

`object`	fitted DLM as given by a call to one of the `dlmodeler.fit()` functions, or filtered DLM as given by a call to `dlmodeler.filter`.
`...`	not used.
`k`	penalty parameter.

Details

The AIC is computed according to the formula $-2 * log(likelihood) + k * npar$, where $npar$ represents the number of parameters in the fitted model, and $k = 2$ for the usual AIC, or $k = log(n)$ (n the number of observations) for the BIC or SBC (Schwarz's Bayesian criterion).

Value

Returns a numeric value with the corresponding log-likelihiid, AIC, BIC, or ..., depending on the value of k.

Author(s)

Cyrille Szymanski <cnszym@gmail.com>

References

Durbin, and Koopman, Time Series Analysis by State Space Methods, Oxford University Press (2001), page 152.

See Also

dlmodeler.fit.MLE

Examples

```
## Example TODO
```

---

dlmodeler.add *Add two models*

Description

Add two DLMs together, performing an outer sum.

Usage

```
dlmodeler.add(mod1, mod2, name = NULL)
```

Arguments

mod1, mod2	objects of class dlmodeler.
name	an optional name to be given to the resulting DLM.

Details

The state vector of the resulting DLM is equal to the concatenation of the state vectors of mod1 and mod2.

The observation vector of the resulting DLM is equal to the sum of the observation vectors of mod1 and mod2.

Value

An object of class dlmodeler representing the sum of mod1 and mod2.

Author(s)

Cyrille Szymanski <cnszym@gmail.com>

References

Giovanni Petris, An R Package for Dynamic Linear Models. Journal of Statistical Software, 36(12), 1-16. http://www.jstatsoft.org/v36/i12/.

See Also

dlmodeler, dlmodeler.bind

Examples

```
require(dlmodeler)

# create the following model:
# deterministic level + quarterly seasonal + disturbance
mod1 <- dlmodeler.build.polynomial(0,sigmaH=.1)
mod2 <- dlmodeler.build.dseasonal(4,sigmaH=0)
mod <- dlmodeler.add(mod1, mod2)
```

---

dlmodeler.bind   *Bind two models*

Description

Bind two DLMs together, creating a multi-variate model.

Usage

```
dlmodeler.bind(mod1, mod2, name = NULL)
```

Arguments

mod1, mod2	objects of class dlmodeler.
name	an optional name to be given to the resulting DLM.

Details

The state vector of the resulting DLM is equal to the concatenation of the state vectors of mod1 and mod2.

The observation vector of the resulting DLM is equal to the concatenation of the observation vectors of mod1 and mod2.

Value

An object of class dlmodeler representing the sum of mod1 and mod2.

Author(s)

Cyrille Szymanski <cnszym@gmail.com>

See Also

dlmodeler, dlmodeler.add

Examples

```
## Example TODO
```

---

dlmodeler.build *Build a DLM*

Description

Builds a DLM with the supplied design matrices, or an "empty" DLM of the specified dimensions.

Usage

```
dlmodeler.build(a0 = NULL, P0 = NULL, P0inf = NULL,
                Tt = NULL, Rt = NULL, Qt = NULL,
                Zt = NULL, Ht = NULL,
                dimensions = NULL,
                name = 'noname', components = list())
```

Arguments

a0	initial state vector.
P0	initial state covariance matrix.
P0inf	diffuse part of P0, matrix of zeros and ones.
Tt	state transition matrix.
Rt	state disturbance selection matrix.
Qt	state disturbance covariance matrix.
Zt	observation design matrix.
Ht	observation disturbance covariance matrix.
dimensions	vector of dimensions (m, r, d).
name	an optional name to be given to the resulting DLM.
components	optional list of components.

Details

A DLM can be constructed either by specifying all the elements a0, P0, P0inf,Tt, Rt, Qt, Zt and Ht or by simply giving the dimensions m, r and d (in which case the DLM is created with zero-filled elements of the appropriate dimension).

See dlmodeler for information about the state-space representation adopted in this package.

This function is called by the helper functions referenced below.

Value

An object of class dlmodeler representing the model.

Author(s)

Cyrille Szymanski <cnszym@gmail.com>

See Also

dlmodeler, dlmodeler.check, dlmodeler.build.polynomial, dlmodeler.build.dseasonal, dlmodeler.build.tseasonal, dlmodeler.build.structural, dlmodeler.build.arima, dlmodeler.build.regressio

Examples

```
require(dlmodeler)

# a stochastic level+trend DLM
mod <- dlmodeler.build(
a0 = c(0,0), # initial state: (level, trend)
P0 = diag(c(0,0)), # initial state variance set to...
P0inf = diag(2), # ...use exact diffuse initialization
matrix(c(1,0,1,1),2,2), # state transition matrix
diag(c(1,1)), # state disturbance selection matrix
diag(c(.5,.05)), # state disturbance variance matrix
matrix(c(1,0),1,2), # observation design matrix
matrix(1,1,1) # observation disturbance variance matrix
)
# print the model
mod
# check if it is valid
dlmodeler.check(mod)$status

# an empty DLM with 4 state variables (3 of which are stocastic)
# and bi-variate observations
mod <- dlmodeler.build(dimensions=c(4,3,2))
# print the model
mod
# check if it is valid
dlmodeler.check(mod)$status
```

dlmodeler.build.arima *Build an ARIMA model*

Description

Builds an univariate ARIMA DLM of the specified order and coefficients.

Usage

```
dlmodeler.build.arima(ar=c(), ma=c(), d=0,
                      sigmaH = NA, sigmaQ = 0,
                      name = "arima")
```

Arguments

ar	vector of autoregressive coefficients.
ma	vector of moving average coefficients.
d	order of differenciation.
sigmaH	std dev of the observation disturbance.
sigmaQ	std dev of the state disturbances.
name	an optional name to be given to the resulting DLM.

Details

The autoregressive terms of the model are $ar[1] + ar[2]L + ...ar[p]L^p$ where L is the lag operator.

The moving average terms of the model are $1 + ma[1]L + ...ma[q]L^q$ where L is the lag operator.

The initial value `P0inf` is parametered to use exact diffuse initialisation (if supported by the back-end).

Remember: all functions of this package assume by default a zero state disturbance (deterministic states). By looking at the prototype for this function, you can assume sigmaH refers to the observation and sigmaQ refers to the state. This rule of thumb is useful to remember how this function should be used.

Value

An object of class `dlmodeler` representing the ARIMA model.

Note

State representations are not unique, so other forms could be used to achieve the same goals.

Currently, only ARMA models (d=0) are implemented.

Author(s)

Cyrille Szymanski <cnszym@gmail.com>

References

Durbin, and Koopman, Time Series Analysis by State Space Methods, Oxford University Press (2001), pages 46-48.

See Also

`dlmodeler`, `dlmodeler.build`, `dlmodeler.build.polynomial`, `dlmodeler.build.dseasonal`, `dlmodeler.build.tseasonal`, `dlmodeler.build.structural`, `dlmodeler.build.regression`

Examples

```
# Example TODO
```

---

dlmodeler.build.dseasonal

Build a "dummy seasonal" model

Description

Builds an univariate "dummy seasonal" DLM of the specified order.

Usage

```
dlmodeler.build.dseasonal(ord, sigmaH = NA, sigmaQ = 0,
                          name = "dseasonal")
```

Arguments

`ord`	period of the seasonal pattern.
`sigmaH`	std dev of the observation disturbance.
`sigmaQ`	std dev of the state disturbance.
`name`	an optional name to be given to the resulting DLM.

Details

The seasonal pattern is represented by `ord` seasonal indices $a[1], a[2], ..., a[ord]$. The indices are constrained such that their sum equals 0, with $a[ord] = -a[1] - a[2] - a[3]... - a[ord-1]$. This only requires `ord-1` state variables.

The initial value `P0inf` is parametered to use exact diffuse initialisation (if supported by the back-end).

Remember: all functions of this package assume by default a zero state disturbance (deterministic states). By looking at the prototype for this function, you can assume sigmaH refers to the observation and sigmaQ refers to the state. This rule of thumb is useful to remember how this function should be used.

Value

An object of class dlmodeler representing the dummy seasonal model.

Note

State representations are not unique, so other forms could be used to achieve the same goals.

Author(s)

Cyrille Szymanski <cnszym@gmail.com>

References

Durbin, and Koopman, Time Series Analysis by State Space Methods, Oxford University Press (2001), pages 38-45.

See Also

dlmodeler, dlmodeler.build, dlmodeler.build.polynomial, dlmodeler.build.tseasonal, dlmodeler.build.structural, dlmodeler.build.arima, dlmodeler.build.regression

Examples

```
require(dlmodeler)

# generate some quarterly data
n <- 80
level <- 12
sigma <- .75
season <- c(5,6,8,2)
y <- level + rep(season,n/4) + rnorm(n, mean=0, sd=sigma)

# deterministic level + quarterly seasonal + disturbance
mod1 <- dlmodeler.build.polynomial(0,sigmaH=sigma)
mod2 <- dlmodeler.build.dseasonal(4,sigmaH=0)
mod <- dlmodeler.add(mod1, mod2)
f <- dlmodeler.filter(y, mod)

# show the one step ahead forecasts
plot(y,type='l')
lines(f$f[1,],col='light blue')
```

---

dlmodeler.build.polynomial

Build a polynomial model

Description

Builds an univariate polynomial DLM of the specified order.

Usage

```
dlmodeler.build.polynomial(ord, sigmaH = NA, sigmaQ = 0,
                           name = "polynomial")
```

Arguments

`ord`	order of the polynomial (0 = constant, 1 = linear, ...).
`sigmaH`	std dev of the observation disturbance.
`sigmaQ`	std dev of the state disturbances.
`name`	an optional name to be given to the resulting DLM.

Details

The polynomial term is of the form $a[1] + a[2]t + a[3]t^2 ... + a[ord]t^ord$.

The initial value `P0inf` is parametered to use exact diffuse initialisation (if supported by the back-end).

Remember: all functions of this package assume by default a zero state disturbance (deterministic states). By looking at the prototype for this function, you can assume sigmaH refers to the observation and sigmaQ refers to the state. This rule of thumb is useful to remember how this function should be used.

Value

An object of class `dlmodeler` representing the polynomial model.

Note

State representations are generally not unique, so other forms could be used to achieve the same goals.

Author(s)

Cyrille Szymanski <cnszym@gmail.com>

References

Durbin, and Koopman, Time Series Analysis by State Space Methods, Oxford University Press (2001), pages 38-45.

See Also

`dlmodeler`, `dlmodeler.build`, `dlmodeler.build.dseasonal`, `dlmodeler.build.tseasonal`, `dlmodeler.build.structural`, `dlmodeler.build.arima`, `dlmodeler.build.regression`

Examples

```
require(dlmodeler)

# generate some quarterly data
n <- 80
level <- 12
sigma <- .75
season <- c(5,6,8,2)
y <- level + rep(season,n/4) + rnorm(n, mean=0, sd=sigma)

# deterministic level + quarterly seasonal + disturbance
mod1 <- dlmodeler.build.polynomial(0,sigmaH=sigma)
mod2 <- dlmodeler.build.dseasonal(4,sigmaH=0)
mod <- dlmodeler.add(mod1, mod2)
f <- dlmodeler.filter(y, mod)

# show the one step ahead forecasts
plot(y,type='l')
lines(f$f[1,],col='light blue')
```

`dlmodeler.build.regression` *Build a regression model*

Description

Builds an univariate (multi-linear) regression DLM.

Usage

```
dlmodeler.build.regression(covariates, sigmaH = NA, sigmaQ = 0,
                              intercept = FALSE, name = "regression")
```

Arguments

`covariates`	covariate matrix (one row per covariate).
`sigmaH`	std dev of the observation disturbance.
`sigmaQ`	std dev of the state disturbance.
`intercept`	should an intercept be added to the model?
`name`	an optional name to be given to the resulting DLM.

Details

The regression term is of the form $a[1]x_1(t) + a[2]x_2(t) ... + a[k]x_k(t)$, where x_k is the k-th covariate.

The initial value `P0inf` is parametered to use exact diffuse initialisation (if supported by the back-end).

Remember: all functions of this package assume by default a zero state disturbance (deterministic states). By looking at the prototype for this function, you can assume sigmaH refers to the observation and sigmaQ refers to the state. This rule of thumb is useful to remember how this function should be used.

Value

An object of class `dlmodeler` representing the regression model.

Note

State representations are not unique, so other forms could be used to achieve the same goals.

Author(s)

Cyrille Szymanski <cnszym@gmail.com>

References

Durbin, and Koopman, Time Series Analysis by State Space Methods, Oxford University Press (2001), pages 38-45.

See Also

`dlmodeler`, `dlmodeler.build`, `dlmodeler.build.polynomial`, `dlmodeler.build.dseasonal`, `dlmodeler.build.tseasonal`, `dlmodeler.build.structural`, `dlmodeler.build.arima`

Examples

```
require(dlmodeler)

# generate some data
N <- 365*5
t <- c(1:N,rep(NA,365))
a <- rnorm(N+365,0,.5)
y <- pi + cos(2*pi*t/365.25) + .25*sin(2*pi*t/365.25*3) +
     exp(1)*a + rnorm(N+365,0,.5)

# build a model for this data
m1 <- dlmodeler.build.polynomial(0,sigmaH=.5,name='level')
m2 <- dlmodeler.build.dseasonal(7,sigmaH=0,name='week')
m3 <- dlmodeler.build.tseasonal(365.25,3,sigmaH=0,name='year')
m4 <- dlmodeler.build.regression(a,sigmaH=0,name='reg')
m <- dlmodeler.add(m1,dlmodeler.add(m2,dlmodeler.add(m3,m4)),
                   name='mymodel')

system.time(f <- dlmodeler.filter(y, m, raw.result=TRUE))

# extract all the components
m.state.mean <- dlmodeler.extract(f,m,type="state",
                                  value="mean")
m.state.cov <- dlmodeler.extract(f,m,type="state",
```

```
                                 value="covariance")
m.obs.mean <- dlmodeler.extract(f,m,type="observation",
                                value="mean")
m.obs.cov <- dlmodeler.extract(f,m,type="observation",
                               value="covariance")
m.obs.int <- dlmodeler.extract(f,m,type="observation",
                               value="interval",prob=.99)

par(mfrow=c(2,1))

# show the one step ahead forecasts & 99% prediction intervals
plot(y,xlim=c(N-10,N+30))
lines(m.obs.int$mymodel$upper[1,],col='light grey')
lines(m.obs.int$mymodel$lower[1,],col='light grey')
lines(m.obs.int$mymodel$mean[1,],col=2)

# see to which values the filter has converged:
m.state.mean$level[,N] # should be close to pi
mean(abs(m.state.mean$week[,N])) # should be close to 0
m.state.mean$year[1,N] # should be close to 1
m.state.mean$year[6,N] # should be close to .25
m.state.mean$reg[,N] # should be close to e

# show the filtered level+year components
plot(m.obs.mean$level[1,]+m.obs.mean$year[1,],
type='l',ylim=c(pi-2,pi+2),col='light green',
ylab="smoothed & filtered level+year")

system.time(s <- dlmodeler.smooth(f,m))

# show the smoothed level+year components
s.obs.mean <- dlmodeler.extract(s,m,type="observation",
                                value="mean")
lines(s.obs.mean$level[1,]+s.obs.mean$year[1,],type='l',
ylim=c(pi-2,pi+2),col='dark green')
```

`dlmodeler.build.structural` *Build a structural time series model*

Description

Builds a DLM for a structural time series, consisting of a polynomial term (level, trend, ...), a "dummy seasonal" pattern, a trigonometric cycle term, and an observation disturbance.

Usage

```
dlmodeler.build.structural(pol.order = NULL, dseas.order = NULL,
                           tseas.order = NULL, tseas.period = NULL,
                           sigmaH = NA, pol.sigmaQ = 0,
```

```
dseas.sigmaQ = 0, tseas.sigmaQ = 0,
name = "structural")
```

Arguments

`pol.order`	order of the polynomial (0=constant, 1=linear, ...), or `NULL`.
`dseas.order`	period of the dummy seasonal pattern, or `NULL`.
`tseas.order`	number of harmonics in the trigonometric seasonal pattern, or `NULL`.
`tseas.period`	period of the trigonometric seasonal pattern, or `NULL`.
`sigmaH`	std dev of the observation disturbance.
`pol.sigmaQ`	std dev of the polynomial state disturbances.
`dseas.sigmaQ`	std dev of the dummy seasonal state disturbances.
`tseas.sigmaQ`	std dev of the trigonometric seasonal state disturbances.
`name`	an optional name to be given to the resulting DLM.

Details

The initial value `P0inf` is parametered to use exact diffuse initialisation (if supported by the back-end).

Remember: all functions of this package assume by default a zero state disturbance (deterministic states). By looking at the prototype for this function, you can assume sigmaH refers to the observation and sigmaQ refers to the state. This rule of thumb is useful to remember how this function should be used.

Value

An object of class `dlmodeler` representing the structural model. This object can have the following components:

`level`	component representing the level (when `pol.order` = 0)
`level+trend`	component representing the level+trend (when `pol.order` = 1)
`polynomial`	component representing the level, trend, ... (when `pol.order` > 1)
`seasonal`	component representing the dummy seasonal pattern
`cycle`	component representing the trigonometric seasonal cycle

Note

State representations are not unique, so other forms could be used to achieve the same goals.

Author(s)

Cyrille Szymanski <cnszym@gmail.com>

References

Durbin, and Koopman, Time Series Analysis by State Space Methods, Oxford University Press (2001), pages 38-45.

See Also

dlmodeler, dlmodeler.build, dlmodeler.build.polynomial, dlmodeler.build.dseasonal, dlmodeler.build.tseasonal, dlmodeler.build.arima, dlmodeler.build.regression

Examples

```
require(dlmodeler)

# generate some quarterly data
n <- 80
level <- 12
sigma <- .75
season <- c(5,6,8,2)
y <- level + rep(season,n/4) + rnorm(n, mean=0, sd=sigma)

# deterministic level + quarterly seasonal
mod <- dlmodeler.build.structural(pol.order=0, dseas.order=4,
                                  sigmaH=sigma)
f <- dlmodeler.filter(y, mod)

# show the one step ahead forecasts
par(mfrow=c(2,1))
plot(y,type='l')
lines(f$f[1,],col='light blue')

# show the filtered level and seasonal components
c <- dlmodeler.extract(f,mod,type="state")
lines(c$level[1,],col='blue')
plot(c$seasonal[1,],type='l',col='dark green')
```

---

dlmodeler.build.tseasonal

Build a trigonometric seasonal model

Description

Builds an univariate trigonometric seasonal DLM of the specified order.

Usage

```
dlmodeler.build.tseasonal(per, ord = NULL,
                             sigmaH = NA, sigmaQ = 0,
                             name = "tseasonal")
```

Arguments

per	period of the seasonal pattern.
ord	optional order (number of harmonics) of the seasonal pattern.

`sigmaH`	std dev of the observation disturbance.
`sigmaQ`	std dev of the state disturbance.
`name`	an optional name to be given to the resulting DLM.

Details

The trigonometric decomposition has the form $a[1]cos(2pi/per)+a[2]sin(2pi/per)+a[3]cos(2pi/per*2) + a[4]sin(2pi/per*2)... + a[2*ord-1]cos(2pi/per*ord) + a[2*ord]sin(2pi/per*ord)$.

If `ord` is not specified, the order is selected such that there are `per-1` coefficients in the decomposition. In this case, `per` must be an integer value.

The initial value `P0inf` is parametered to use exact diffuse initialisation (if supported by the backend).

Remember: all functions of this package assume by default a zero state disturbance (deterministic states). By looking at the prototype for this function, you can assume sigmaH refers to the observation and sigmaQ refers to the state. This rule of thumb is useful to remember how this function should be used.

Value

An object of class `dlmodeler` representing the trigonometric seasonal model.

Note

State representations are not unique, so other forms could be used to achieve the same goals.

Author(s)

Cyrille Szymanski <cnszym@gmail.com>

References

Durbin, and Koopman, Time Series Analysis by State Space Methods, Oxford University Press (2001), pages 38-45.

See Also

`dlmodeler`, `dlmodeler.build`, `dlmodeler.build.polynomial`, `dlmodeler.build.dseasonal`, `dlmodeler.build.structural`, `dlmodeler.build.arima`, `dlmodeler.build.regression`

Examples

```
require(dlmodeler)

# generate some data
N <- 365*5
t <- c(1:N,rep(NA,365))
a <- rnorm(N+365,0,.5)
y <- pi + cos(2*pi*t/365.25) + .25*sin(2*pi*t/365.25*3) +
     exp(1)*a + rnorm(N+365,0,.5)
```

```
# build a model for this data
m1 <- dlmodeler.build.polynomial(0,sigmaH=.5,name='level')
m2 <- dlmodeler.build.dseasonal(7,sigmaH=0,name='week')
m3 <- dlmodeler.build.tseasonal(365.25,3,sigmaH=0,name='year')
m4 <- dlmodeler.build.regression(a,sigmaH=0,name='reg')
m <- dlmodeler.add(m1,dlmodeler.add(m2,dlmodeler.add(m3,m4)),
                   name='mymodel')

system.time(f <- dlmodeler.filter(y, m, raw.result=TRUE))

# extract all the components
m.state.mean <- dlmodeler.extract(f,m,type="state",
                                  value="mean")
m.state.cov <- dlmodeler.extract(f,m,type="state",
                                 value="covariance")
m.obs.mean <- dlmodeler.extract(f,m,type="observation",
                                value="mean")
m.obs.cov <- dlmodeler.extract(f,m,type="observation",
                               value="covariance")
m.obs.int <- dlmodeler.extract(f,m,type="observation",
                               value="interval",prob=.99)

par(mfrow=c(2,1))

# show the one step ahead forecasts & 99% prediction intervals
plot(y,xlim=c(N-10,N+30))
lines(m.obs.int$mymodel$upper[1,],col='light grey')
lines(m.obs.int$mymodel$lower[1,],col='light grey')
lines(m.obs.int$mymodel$mean[1,],col=2)

# see to which values the filter has converged:
m.state.mean$level[,N] # should be close to pi
mean(abs(m.state.mean$week[,N])) # should be close to 0
m.state.mean$year[1,N] # should be close to 1
m.state.mean$year[6,N] # should be close to .25
m.state.mean$reg[,N] # should be close to e

# show the filtered level+year components
plot(m.obs.mean$level[1,]+m.obs.mean$year[1,],
type='l',ylim=c(pi-2,pi+2),col='light green',
ylab="smoothed & filtered level+year")

system.time(s <- dlmodeler.smooth(f,m))

# show the smoothed level+year components
s.obs.mean <- dlmodeler.extract(s,m,type="observation",
                                value="mean")
lines(s.obs.mean$level[1,]+s.obs.mean$year[1,],type='l',
ylim=c(pi-2,pi+2),col='dark green')
```

`dlmodeler.check` *Check dimensions and validity*

Description

Checks a `dlmodeler` object, in particular for the consistency of the dimensions of its elements.

Usage

```
dlmodeler.check(model, yt = NULL)
```

Arguments

`model`	an object of class `dlmodeler` to check.
`yt`	an optional data vector to check with the model.

Details

See `dlmodeler` for information about the state-space representation adopted in this package.

Value

An array with the following information:

`status`	a boolean indicating whether the model is valid or not
`m`	dimension of state vector m
`r`	dimension of state disturbance covariance matrix r
`d`	dimension of observation vector d
`timevar`	a boolean indicating if the model has time-varying terms or not
`timevar.Tt`	the number of time steps in Tt, or NA if the matrix is constant
`timevar.Rt`	the number of time steps in Rt, or NA if the matrix is constant
`timevar.Qt`	the number of time steps in Qt, or NA if the matrix is constant
`timevar.Zt`	the number of time steps in Zt, or NA if the matrix is constant
`timevar.Ht`	the number of time steps in Ht, or NA if the matrix is constant

Author(s)

Cyrille Szymanski <cnszym@gmail.com>

See Also

`dlmodeler`, `dlmodeler.build`

Examples

```
require(dlmodeler)

# a stochastic level+trend DLM
mod <- dlmodeler.build(
a0 = c(0,0), # initial state: (level, trend)
P0 = diag(c(0,0)), # initial state variance set to...
P0inf = diag(2), # ...use exact diffuse initialization
matrix(c(1,0,1,1),2,2), # state transition matrix
diag(c(1,1)), # state disturbance selection matrix
diag(c(.5,.05)), # state disturbance variance matrix
matrix(c(1,0),1,2), # observation design matrix
matrix(1,1,1) # observation disturbance variance matrix
)
# print the model
mod
# check if it is valid
dlmodeler.check(mod)$status

# an empty DLM with 4 state variables (3 of which are stocastic)
# and bi-variate observations
mod <- dlmodeler.build(dimensions=c(4,3,2))
# print the model
mod
# check if it is valid
dlmodeler.check(mod)$status
```

`dlmodeler.extract` *Extract the mean, covariance and prediction intervals for states and observations*

Description

Extracts the mean (expectation), the variance-covariance matrix, and the prediction intervals for the states and observations of a filtered or smoothed DLM component.

Usage

```
dlmodeler.extract(fs, model, compnames=NULL,
                    type=c("observation","state"),
                    value=c("mean","covariance","interval"),
                    prob=.90)
```

Arguments

`fs`	filtered or smoothed `dlmodeler`, as a result from a call to `dlmodeler.filter()` or `dlmodeler.smooth()`.
`model`	object of class `dlmodeler` which was used for filtering or smoothing.

compnames	an optional list of components to extract.
type	an optional string indicating the type to extract: observation (output, by default) or state.
value	an optional string indicating the value to extract: mean (expectation, by default), covariance matrix, or prediction intervals.
prob	an optional probability (default = 90%) for the computation of prediction intervals.

Details

A component is a named portion of the state vector matrix which can be extracted with this function. Components are automatically created when DLMs are added together which makes it easier to decompose it later into its building blocks (for example: level+trend+seasonal+cycle).

Let us assume model named `m` is constructed by adding models named `m1` and `m2`. Typically, `m` will be constructed with two components named `m1` and `m1`, which can be extracted by this function.

Value

When this function is used with a filtered `dlmodeler`, it returns the means and covariances of the one-step ahead forecasts for the components:

- `Zt %*% at` $= E(y(t)|y(1),y(2)...y(t-1))$ for observation means, in the form of a (d,n) matrix.
- `at` $= E(alpha(t)|y(1),y(2)...y(t-1))$ for state means, in the form of a (m,n) matrix.
- `Zt %*% Pt %*% t(Zt) + Ht` $= cov(y(t)|y(1),y(2)...y(t-1))$ for observation covariances, in the form of a (d,d,n) array.
- `Pt` $= cov(alpha(t)|y(1),y(2)...y(t-1))$ for state covariances, in the form of a (m,m,n) array.

When this function is used with a smoothed `dlmodeler`, it returns the means and covariances of the smoothed components:

- `Zt %*% at` $= E(y(t)|y(1),y(2)...y(N))$ for observation means, in the form of a (d,n) matrix.
- `at` $= E(alpha(t)|y(1),y(2)...y(N))$ for state means, in the form of a (m,n) matrix.
- `Zt %*% Pt %*% t(Zt) + Ht` $= cov(y(t)|y(1),y(2)...y(N))$ for observation covariances, in the form of a (d,d,n) array.
- `Pt` $= cov(alpha(t)|y(1),y(2)...y(N))$ for state covariances, in the form of a (m,m,n) array.

When the value `interval` is requested, this function returns a list for each component containing:

- `mean` = the mean (expectaton) for the filtered or smoothed state or observation variable.
- `lower` = lower bound of the prediction interval computed as `mean-k*sd`, `k=-qnorm((1+prob)/2)`.
- `upper` = upper bound of the prediction interval computed as `mean+k*sd`, `k=-qnorm((1+prob)/2)`.

Author(s)

Cyrille Szymanski <cnszym@gmail.com>

See Also

dlmodeler, dlmodeler.filter, dlmodeler.smooth

Examples

```
require(dlmodeler)

# generate some data
N <- 365*5
t <- c(1:N,rep(NA,365))
a <- rnorm(N+365,0,.5)
y <- pi + cos(2*pi*t/365.25) + .25*sin(2*pi*t/365.25*3) +
     exp(1)*a + rnorm(N+365,0,.5)

# build a model for this data
m1 <- dlmodeler.build.polynomial(0,sigmaH=.5,name='level')
m2 <- dlmodeler.build.dseasonal(7,sigmaH=0,name='week')
m3 <- dlmodeler.build.tseasonal(365.25,3,sigmaH=0,name='year')
m4 <- dlmodeler.build.regression(a,sigmaH=0,name='reg')
m <- dlmodeler.add(m1,dlmodeler.add(m2,dlmodeler.add(m3,m4)),
                   name='mymodel')

system.time(f <- dlmodeler.filter(y, m, raw.result=TRUE))

# extract all the components
m.state.mean <- dlmodeler.extract(f,m,type="state",
                                  value="mean")
m.state.cov <- dlmodeler.extract(f,m,type="state",
                                 value="covariance")
m.obs.mean <- dlmodeler.extract(f,m,type="observation",
                                value="mean")
m.obs.cov <- dlmodeler.extract(f,m,type="observation",
                               value="covariance")
m.obs.int <- dlmodeler.extract(f,m,type="observation",
                               value="interval",prob=.99)

par(mfrow=c(2,1))

# show the one step ahead forecasts & 99% prediction intervals
plot(y,xlim=c(N-10,N+30))
lines(m.obs.int$mymodel$upper[1,],col='light grey')
lines(m.obs.int$mymodel$lower[1,],col='light grey')
lines(m.obs.int$mymodel$mean[1,],col=2)

# see to which values the filter has converged:
m.state.mean$level[,N] # should be close to pi
mean(abs(m.state.mean$week[,N])) # should be close to 0
m.state.mean$year[1,N] # should be close to 1
m.state.mean$year[6,N] # should be close to .25
m.state.mean$reg[,N] # should be close to e

# show the filtered level+year components
```

```
plot(m.obs.mean$level[1,]+m.obs.mean$year[1,],
type='l',ylim=c(pi-2,pi+2),col='light green',
ylab="smoothed & filtered level+year")

system.time(s <- dlmodeler.smooth(f,m))

# show the smoothed level+year components
s.obs.mean <- dlmodeler.extract(s,m,type="observation",
                                value="mean")
lines(s.obs.mean$level[1,]+s.obs.mean$year[1,],type='l',
ylim=c(pi-2,pi+2),col='dark green')
```

`dlmodeler.filter.smooth` *Filtering and smoothing for a DLM*

Description

Kalman filtering and smoothing for a Dynamic Linear Model, using the specified back-end for the computations.

Usage

```
dlmodeler.filter(yt, model,
                 backend = c("KFAS", "FKF", "dlm"),
                 smooth = FALSE,
                 raw.result = FALSE,
                 logLik = FALSE, filter = TRUE)

dlmodeler.smooth(filt, raw.result = FALSE)
```

Arguments

`yt`	matrix of observed values (one column per time step).
`model`	an object of class `dlmodeler`.
`backend`	an optional argument which specifies the back-end to use for the computations.
`smooth`	an optional argument which specifies if the back-end shoud also run the smoothing algorithm.
`raw.result`	if TRUE, the raw results from the back-end will be stored in `raw.result`.
`logLik`	an optional argument which specifies if the back-end shoud compute the log-likelihood.
`filter`	an optional argument which specifies if the back-end shoud also run the filtering algorithm.
`filt`	filtered `dlmodeler.filtered`, as a result from a call to `dlmodeler.filter()`.

Details

This function will automatically load the adequate back-end package.

Currently, packages KFAS (used by default), FKF and dlm are supported. Refer to dlmodeler for more information.

Value

An object of class dlmodeler.filtered which contains the following elements:

f	matrix containing the one step ahead predictions $E(y(t)\|y(1), y(2)...y(t-1))$
at	matrix containing the one step ahead predicted state variables $E(a(t)\|y(1), y(2)...y(t-1))$
Pt	matrix containing the one step ahead predicted state covariance matrices $cov(a(t)\|y(1), y(2)...y(t-1))$
logLik	the value of the log-likelihood for the model
backend	a character string indicating which back-end was used for the computation
raw.result	the raw result from the back-end, or NA if it wasn't requested

Or an object of class dlmodeler.smoothed which contains the following elements:

at	matrix containing the one step ahead smoothed state variables $E(a(t)\|y(1), y(2)...y(n))$
Pt	matrix containing the one step ahead predicted state covariance matrices $cov(a(t)\|y(1), y(2)...y(n))$
backend	a character string indicating which back-end was used for the computation
raw.result	the raw result from the back-end, or NA if it wasn't requested

Note

Package dlm does not offer a way to obtain the log-likelihood and the filtered values at the same time (as of v1.1-2). The log-likelihood is not computed by default, but this can be done by using the parameter logLik=TRUE. The computation of the filtered values can also be disabled with parameter filter=FALSE if these values are not needed.

Package FKF does not implement a smoothing algorithm (as of v0.1.1).

Author(s)

Cyrille Szymanski <cnszym@gmail.com>

See Also

dlmodeler, dlmodeler.forecast

Examples

```
require(dlmodeler)

# generate some data
N <- 365*5
t <- c(1:N,rep(NA,365))
a <- rnorm(N+365,0,.5)
y <- pi + cos(2*pi*t/365.25) + .25*sin(2*pi*t/365.25*3) +
     exp(1)*a + rnorm(N+365,0,.5)

# build a model for this data
m1 <- dlmodeler.build.polynomial(0,sigmaH=.5,name='level')
m2 <- dlmodeler.build.dseasonal(7,sigmaH=0,name='week')
m3 <- dlmodeler.build.tseasonal(365.25,3,sigmaH=0,name='year')
m4 <- dlmodeler.build.regression(a,sigmaH=0,name='reg')
m <- dlmodeler.add(m1,dlmodeler.add(m2,dlmodeler.add(m3,m4)),
                   name='mymodel')

system.time(f <- dlmodeler.filter(y, m, raw.result=TRUE))

# extract all the components
m.state.mean <- dlmodeler.extract(f,m,type="state",
                                  value="mean")
m.state.cov <- dlmodeler.extract(f,m,type="state",
                                 value="covariance")
m.obs.mean <- dlmodeler.extract(f,m,type="observation",
                                value="mean")
m.obs.cov <- dlmodeler.extract(f,m,type="observation",
                               value="covariance")
m.obs.int <- dlmodeler.extract(f,m,type="observation",
                               value="interval",prob=.99)

par(mfrow=c(2,1))

# show the one step ahead forecasts & 99% prediction intervals
plot(y,xlim=c(N-10,N+30))
lines(m.obs.int$mymodel$upper[1,],col='light grey')
lines(m.obs.int$mymodel$lower[1,],col='light grey')
lines(m.obs.int$mymodel$mean[1,],col=2)

# see to which values the filter has converged:
m.state.mean$level[,N] # should be close to pi
mean(abs(m.state.mean$week[,N])) # should be close to 0
m.state.mean$year[1,N] # should be close to 1
m.state.mean$year[6,N] # should be close to .25
m.state.mean$reg[,N] # should be close to e

# show the filtered level+year components
plot(m.obs.mean$level[1,]+m.obs.mean$year[1,],
type='l',ylim=c(pi-2,pi+2),col='light green',
ylab="smoothed & filtered level+year")
```

```
system.time(s <- dlmodeler.smooth(f,m))

# show the smoothed level+year components
s.obs.mean <- dlmodeler.extract(s,m,type="observation",
                                value="mean")
lines(s.obs.mean$level[1,]+s.obs.mean$year[1,],type='l',
ylim=c(pi-2,pi+2),col='dark green')
```

`dlmodeler.fit` *Fitting function for a model (MLE, MSE, MAD, sigma)*

Description

Fits a DLM by maximum likelihood (MLE), minimum squared errror (MSE), minimum average deviation (MAD) or minimum standard deviation (sigma) methods.

Usage

```
dlmodeler.fit(yt, model=NULL,
              method=c("MLE","MSE","MAD","MAPE","sigma"), ...)

dlmodeler.fit.MLE(yt, build.fun, par,
                  backend = c('KFAS','FKF','dlm'), method = "L-BFGS-B",
                  verbose = FALSE, silent = FALSE, filter = TRUE,
                  smooth = FALSE, raw.result = FALSE, ...)

dlmodeler.fit.MSE(yt, build.fun, par,
                  ahead, iters = NCOL(yt)-ahead-start-1,
                  step = 1, start = 1,
                  backend = c('KFAS','FKF','dlm'), method = "L-BFGS-B",
                  verbose = FALSE, silent = FALSE,
                  filter = TRUE, smooth = FALSE,
                  raw.result=FALSE, ...)

dlmodeler.fit.MAD(yt, build.fun, par,
                  ahead, iters = NCOL(yt)-ahead-start-1,
                  step = 1, start = 1,
                  backend = c('KFAS','FKF','dlm'), method = "L-BFGS-B",
                  verbose = FALSE, silent = FALSE,
                  filter = TRUE, smooth = FALSE,
                  raw.result=FALSE, ...)

dlmodeler.fit.MAPE(yt, build.fun, par,
                  ahead, iters = NCOL(yt)-ahead-start-1,
                  step = 1, start = 1,
                  backend = c('KFAS','FKF','dlm'), method = "L-BFGS-B",
                  verbose = FALSE, silent = FALSE,
```

```
                  filter = TRUE, smooth = FALSE,
                  raw.result=FALSE, ...)

dlmodeler.fit.sigma(yt, build.fun, par,
                  backend = c('KFAS','FKF','dlm'), method = "L-BFGS-B",
                  verbose = FALSE, silent = FALSE,
                  filter = TRUE, smooth = FALSE,
                  raw.result=FALSE, ...)
```

Arguments

`yt`	matrix of observed values (one column per time step).
`model`	object of class `dlmodeler` with `NA` values to be fitted.
`build.fun`	function taking parameter vector `p` as first argument and returning a DLM.
`par`	initial value of the parameter vector `p`.
`backend`	an optional argument which specifies the back-end to use for the computations.
`method`	optimization method passed to function `optim`.
`verbose`	if `TRUE`, then write one line per iteration giving the parameter vector `p` and the value of the objective function.
`silent`	if `TRUE`, then do not write anything.
`filter`	if `TRUE`, then return the filtered optimal model.
`smooth`	if `TRUE`, the return the smoothed optimal model.
`raw.result`	if `TRUE`, the raw results from the back-end will be stored in `raw.result`.
`ahead`	in case of MSE fitting, the number of predictions to make for each iteration.
`iters`	in case of MSE fitting, the number of iterations.
`step`	in case of MSE fitting, the step between iterations.
`start`	in case of MSE fitting, the index of the first prediction.
`...`	additional arguments passed to `build.fun`.

Details

`dlmodeler.fit.MLE` is designed to find parameter values which maximize the log-likelihood for the given data. This is called Maximum Likelihood Estimation.

`dlmodeler.fit.MSE` is designed to find parameter values which minimize the average n-step ahead prediction squared error $(predicted - actual)^2$ for the given data. This is called Minimum Squared Error fitting. The squared error is averaged over `ahead` prediction steps. Note that having `ahead==1` is roughly equivalent to MLE fitting as long as only the mean is concerned.

`dlmodeler.fit.MAD` is designed to find parameter values which minimize the average n-step ahead prediction absolute error $|predicted - actual|$ for the given data. This is called Minimum Average Deviation fitting. The absolute error is averaged over `ahead` prediction steps.

`dlmodeler.fit.MAPE` is designed to find parameter values which minimize the average n-step ahead prediction absolute percentage error $|predicted - actual|$ for the given data. This is called Minimum Average Percentage Error fitting. The absolute percentage error is averaged over `ahead` prediction steps.

dlmodeler.fit.sigma is designed to find parameter values which minimize the one-step ahead prediction variance for the given data.

Value

An object of class dlmodeler.fit with the following values:

par	optimal parameter returned by the optimization function optim()
message	message returned by the optimization function optim()
convergence	convergence code returned by the optimization function optim()
model	optimal model found: build.fun(par)
logLik	value of the log-likelihood or NA
par0	initial value of par
filtered	optionally, the filtered model: dlmodeler.filter(yt,build.fun(par))

Note

dlmodeler.fit automatically fits models which contain NA values.

Author(s)

Cyrille Szymanski <cnszym@gmail.com>

See Also

dlmodeler, dlmodeler.filter, dlmodeler.smooth, dlmodeler.forecast

Examples

```
require(dlmodeler)

# analysis from Durbin & Koopman book page 32

# load and show the data
y <- matrix(Nile,nrow=1)
plot(y[1,],type='l')

# y(t)   = a(t) + eta(t)
# a(t+1) = a(t) + eps(t)
mod <- dlmodeler.build.polynomial(0,sigmaH=NA,sigmaQ=NA,name='p32')

# fit the model by maximum likelihood estimation
fit <- dlmodeler.fit(y, mod, method="MLE")

# compare the fitted parameters with those reported by the authors
fit$par[2]         # psi = -2.33
fit$model$Ht[1,1] # H   = 15099
fit$model$Qt[1,1] # Q   = 1469.1

# compute the filtered and smoothed values
```

```
f <- dlmodeler.filter(y, fit$mod, smooth=TRUE)

# f.ce represents the filtered one steap ahead observation
# prediction expectations E[y(t) | y(1), y(2), ..., y(t-1)]
f.ce <- dlmodeler.extract(f, fit$model,
                          type="observation", value="mean")

# s.ce represents the smoothed observation expectations
# E[y(t) | y(1), y(2), ..., y(n)]
s.ce <- dlmodeler.extract(f$smooth, fit$model,
                          type="observation", value="mean")

# plot the components
plot(y[1,],type='l')
lines(f.ce$p32[1,],col='light blue',lty=2)
lines(s.ce$p32[1,],col='dark blue')
```

dlmodeler.forecast — *Forecast function*

Description

Simulates forecasting for a DLM, with the specified horizon, step and number of iterations.

Usage

```
dlmodeler.forecast(yt, model,
                   ahead = 1,
                   iters = 1, step = 1, start = 1,
                   prob = .90,
                   backend = c('KFAS','FKF','dlm'),
                   debug = FALSE)
```

Arguments

yt	matrix of observed values (one column per time-step).
model	an instance of dlmodeler.
ahead	in case of MSE fitting, the number of predictions to make for each iteration.
iters	in case of MSE fitting, the number of iterations.
step	in case of MSE fitting, the step between iterations.
start	in case of MSE fitting, the index of the first prediction.
prob	probability to use for the computation of prediction intervals.
backend	an optional argument which specifies the back-end to use for the computations.
debug	use slow but more robust code.

Details

This function simulates forecasts for the specified serie yt and model by generating iters forecasts every step points. The procedure starts at position start, and each iteration, ahead values are predicted.

Value

A data.frame with the following variables:

index	the index of the forecasted value, index==i means that the i-th element of the serie is forecasted
distance	the forecasting distance, distance==k means that this value is a k-step ahead forecast
lower	the lower bound for yhat computed with probability prob
yhat	the forecasted value for the specified index and distance
upper	the upper bound for yhat computed with probability prob
y	the observed serie for the specified yt[index]

Note

Currently, the function only works for univariate time-series.

Currently the implementation is very slow, but its speed will be increased in future versions of this package.

Author(s)

Cyrille Szymanski <cnszym@gmail.com>

See Also

dlmodeler, dlmodeler.filter, dlmodeler.smooth, dlmodeler.forecast

Examples

```
require(dlmodeler)

# generate some quarterly data
n <- 80
level <- 12
sigma <- .75
season <- c(5,6,8,2)
y <- level + 3*sin((1:n)/10) + rep(season,n/4) + rnorm(n, 0, sigma)
y <- matrix(y,nrow=1)

# fit a stochastic level + quarterly seasonal model to the data by
# maximum likelihood estimation
build.fun <- function(p) {
sigmaH <- exp(p[1])
sigmaQ <- exp(p[2])*sigmaH
```

```
mod1 <- dlmodeler.build.polynomial(0,sigmaH=sigmaH,sigmaQ=sigmaQ)
mod2 <- dlmodeler.build.dseasonal(4,sigmaH=0)
mod <- dlmodeler.add(mod1, mod2)
return(mod)
}
fit <- dlmodeler.fit.MLE(y, build.fun, c(0,0))

# generate forecasts for observations 81 to 100
f <- dlmodeler.forecast(y, fit$model, start=80, ahead=20)
plot(y[1,],type='l',xlim=c(60,100),ylim=c(10,30))
lines(f$index,f$yhat,col='dark blue')
lines(f$index,f$lower,col='light blue')
lines(f$index,f$upper,col='light blue')

# simulate forecasts post-ex.
f <- dlmodeler.forecast(y, fit$model, ahead=20, start=20, iters=40)
plot(y[1,],type='p')

# show the one step ahead forecasts
with(f[f$distance==1,], lines(index,yhat,col='dark blue'))
# show the 10 step ahead forecasts
with(f[f$distance==10,], lines(index,yhat,col='blue'))
# show the 20 step ahead forecasts
with(f[f$distance==20,], lines(index,yhat,col='light blue'))
```

`dlmodeler.yeardays` *Return the number of days and weekdays in a given year*

Description

TODO

Usage

```
dlmodeler.yeardays(year)
```

Arguments

`year`	year.

Details

TODO

Value

TODO

Note

TODO

Author(s)

Cyrille Szymanski <cnszym@gmail.com>

See Also

`dlmodeler.build`

Examples

```
## TODO
```

`print.dlmodeler` *Print a model*

Description

Prints a short message describing a DLM.

Usage

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

Arguments

`x`	model to be printed.
`...`	unused.

Details

This function will print the dimensions of the DLM, the time-varying terms, and the names of the components.

Value

No value.

Author(s)

Cyrille Szymanski <cnszym@gmail.com>

See Also

`dlmodeler.build`

Examples

```
require(dlmodeler)

# a stochastic level+trend DLM
mod <- dlmodeler.build(
a0 = c(0,0), # initial state: (level, trend)
P0 = diag(c(0,0)), # initial state variance set to...
P0inf = diag(2), # ...use exact diffuse initialization
matrix(c(1,0,1,1),2,2), # state transition matrix
diag(c(1,1)), # state disturbance selection matrix
diag(c(.5,.05)), # state disturbance variance matrix
matrix(c(1,0),1,2), # observation design matrix
matrix(1,1,1) # observation disturbance variance matrix
)
# print the model
mod
# check if it is valid
dlmodeler.check(mod)[1]==1

# an empty DLM with 4 state variables (3 of which are stocastic)
# and bi-variate observations
mod <- dlmodeler.build(dimensions=c(4,3,2))
# print the model
mod
# check if it is valid
dlmodeler.check(mod)[1]==1
```

Index