# Package: KFAS (via r-universe)

September 20, 2024

**Version** 1.5.1

**Title** Kalman Filter and Smoother for Exponential Family State Space Models

**Depends** R (>= 3.1.0)

**Imports** stats

**Suggests** knitr, lme4, MASS, Matrix, testthat

**Description** State space modelling is an efficient and flexible framework for statistical inference of a broad class of time series and other data. KFAS includes computationally efficient functions for Kalman filtering, smoothing, forecasting, and simulation of multivariate exponential family state space models, with observations from Gaussian, Poisson, binomial, negative binomial, and gamma distributions. See the paper by Helske (2017) <doi:10.18637/jss.v078.i10> for details.

**License** GPL (>= 2)

**BugReports** https://github.com/helske/KFAS/issues

**VignetteBuilder** knitr

**RoxygenNote** 7.2.3

**Encoding** UTF-8

**ByteCompile** true

**URL** https://github.com/helske/KFAS

**NeedsCompilation** yes

**Author** Jouni Helske [aut, cre] (<https://orcid.org/0000-0001-7130-793X>)

**Maintainer** Jouni Helske <jouni.helske@iki.fi>

**Repository** CRAN

**Date/Publication** 2023-09-05 08:20:02 UTC

# Contents



---

`alcohol` *Alcohol related deaths in Finland 1969–2013*

---

### Description

A multivariate time series object containing the number of alcohol related deaths and population sizes (divided by 100000) of Finland in four age groups. See JSS paper for examples.

### Format

A multivariate time series object with 45 times 8 observations.

## Source

Statistics Finland https://statfin.stat.fi/PxWeb/pxweb/en/StatFin/.

---

`approxSSM` *Linear Gaussian Approximation for Exponential Family State Space Model*

---

## Description

Function `approxSMM` performs a linear Gaussian approximation of an exponential family state space model.

## Usage

```
approxSSM(
  model,
  theta,
  maxiter = 50,
  tol = 1e-08,
  expected = FALSE,
  H_tol = 1e+15
)
```

## Arguments

| | |
|---|---|
| `model` | A non-Gaussian state space model object of class `SSModel`. |
| `theta` | Initial values for conditional mode theta. |
| `maxiter` | The maximum number of iterations used in approximation. Default is 50. |
| `tol` | Tolerance parameter for convergence checks. |
| `expected` | Logical value defining the approximation of H_t in case of Gamma and negative binomial distribution. Default is `FALSE` which matches the algorithm of Durbin & Koopman (1997), whereas `TRUE` uses the expected value of observations in the equations, leading to results which match with `glm` (where applicable). The latter case was the default behaviour of KFAS before version 1.3.8. Essentially this is the difference between observed and expected information. |
| `H_tol` | Tolerance parameter for check `max(H)` > `tol_H`, which suggests that the approximation converged to degenerate case with near zero signal-to-noise ratio. Default is very generous 1e15. |

## Details

This function is rarely needed itself, it is mainly available for illustrative and debugging purposes. The underlying Fortran code is used by other functions of KFAS for non-Gaussian state space modelling.

The linear Gaussian approximating model is defined by

$$\tilde{y}_t = Z_t\alpha_t + \epsilon_t, \quad \epsilon_t \sim N(0, \tilde{H}_t),$$

$$\alpha_{t+1} = T_t\alpha_t + R_t\eta_t, \quad \eta_t \sim N(0, Q_t),$$

and $\alpha_1 \sim N(a_1, P_1)$,

where $\tilde{y}$ and $\tilde{H}$ are chosen in a way that the linear Gaussian approximating model has the same conditional mode of $\theta = Z\alpha$ given the observations $y$ as the original non-Gaussian model. Models also have a same curvature at the mode.

The approximation of the exponential family state space model is based on iterative weighted least squares method, see McCullagh and Nelder (1983) p.31 and Durbin Koopman (2012) p. 243.

## Value

An object of class `SSModel` which contains the approximating Gaussian state space model with following additional components:

| | |
|---|---|
| `thetahat` | Mode of $p(\theta\|y)$. |
| `iterations` | Number of iterations used. |
| `difference` | Relative difference in the last step of approximation algorithm. |

## References

- McCullagh, P. and Nelder, J. A. (1983). Generalized linear models. Chapman and Hall.
- Koopman, S.J. and Durbin, J. (2012). Time Series Analysis by State Space Methods. Second edition. Oxford University Press.

## See Also

`importanceSSM`, `SSModel`, `KFS`, `KFAS`.

## Examples

```
# A Gamma example modified from ?glm (with log-link)
clotting <- data.frame(
  u = c(5,10,15,20,30,40,60,80,100),
  lot1 = c(118,58,42,35,27,25,21,19,18),
  lot2 = c(69,35,26,21,18,16,13,12,12))

glmfit1 <- glm(lot1 ~ log(u), data = clotting, family = Gamma(link = "log"))
glmfit2 <- glm(lot2 ~ log(u), data = clotting, family = Gamma(link = "log"))

# Model lot1 and lot2 together (they are still assumed independent)
# Note that Gamma distribution is parameterized by 1/dispersion i.e. shape parameter
model <- SSModel(cbind(lot1, lot2) ~ log(u),
                u = 1/c(summary(glmfit1)$dispersion, summary(glmfit2)$dispersion),
                data = clotting, distribution = "gamma")
approxmodel <- approxSSM(model)
```

```
# Conditional modes of linear predictor:
approxmodel$thetahat
cbind(glmfit1$linear.predictor, glmfit2$linear.predictor)

KFS(approxmodel)
summary(glmfit1)
summary(glmfit2)

# approxSSM uses modified step-halving for more robust convergence than glm:
y <- rep (0:1, c(15, 10))
suppressWarnings(glm(formula = y ~ 1, family = binomial(link = "logit"), start = 2))
model <- SSModel(y~1, dist = "binomial")
KFS(model, theta = 2)
KFS(model, theta = 7)
```

---

`artransform` *Mapping real valued parameters to stationary region*

---

## Description

Function `artransform` transforms $p$ real valued parameters to stationary region of $p$th order autoregressive process using parametrization suggested by Jones (1980). Fortran code is a converted from `stats` package's C-function `partrans`.

## Usage

```
artransform(param)
```

## Arguments

| | |
|---|---|
| `param` | Real valued parameters for the transformation. |

## Value

transformed The parameters satisfying the stationary constrains.

## Note

This should in theory always work, but in practice the initial transformation by `tanh` can produce values numerically identical to 1, leading to AR coefficients which do not satisfy the stationarity constraints. See example in `logLik.SSModel` on how to scope with those issues.

## References

Jones, R. H (1980). Maximum likelihood fitting of ARMA models to time series with missing observations, Technometrics Vol 22. p. 389–395.

## Examples

```
artransform(1:3)
```

boat *Oxford-Cambridge boat race results 1829-2011*

### Description

Results of the annual boat race between universities of Oxford (0) and Cambridge (1).

### Format

A time series object containing 183 observations (including 28 missing observations).

### Source

http://www.ssfpack.com/DKbook.html

### References

Koopman, S.J. and Durbin J. (2012). Time Series Analysis by State Space Methods. Oxford: Oxford University Press.

### Examples

```
data("boat")

# Model from DK2012, bernoulli response based on random walk
model <- SSModel(boat ~ SSMtrend(1, Q = NA), distribution = "binomial")

fit_nosim <- fitSSM(model, inits = log(0.25), method = "BFGS", hessian = TRUE)
# nsim set to small for faster execution of example
# doesn't matter here as the model/data is so poor anyway
fit_sim <- fitSSM(model, inits = log(0.25), method = "BFGS", hessian = TRUE, nsim = 100)

# Compare with the results from DK2012
model_DK <- SSModel(boat ~ SSMtrend(1, Q = 0.33), distribution = "binomial")

# Big difference in variance parameters:
fit_nosim$model["Q"]
fit_sim$model["Q"]

# approximate 95% confidence intervals for variance parameter:
# very wide, there really isn't enough information in the data
# as a comparison, a fully Bayesian approach (using BUGS) with [0, 10] uniform prior for sigma
# gives posterior mode for Q as 0.18, and 95% credible interval [0.036, 3.083]

exp(fit_nosim$optim.out$par + c(-1, 1)*qnorm(0.975)*sqrt(1/fit_nosim$optim.out$hessian[1]))
exp(fit_sim$optim.out$par + c(-1, 1)*qnorm(0.975)*sqrt(1/fit_sim$optim.out$hessian[1]))

# 95% confidence intervals for probability that Cambridge wins
pred_nosim <- predict(fit_nosim$model, interval = "confidence")
pred_sim <- predict(fit_sim$model, interval = "confidence")
```

```
ts.plot(pred_nosim, pred_sim, col = c(1, 2, 2, 3, 4, 4), lty = c(1, 2, 2), ylim = c(0, 1))
points(x = time(boat), y = boat, pch = 15, cex = 0.5)

# if we trust the approximation, fit_nosim gives largest log-likelihood:
logLik(fit_nosim$model)
logLik(fit_sim$model)
logLik(model_DK)

# and using importance sampling fit_sim is the best:
logLik(fit_nosim$model, nsim = 100)
logLik(fit_sim$model, nsim = 100)
logLik(model_DK, nsim = 100)

## Not run:
# only one unknown parameter, easy to check the shape of likelihood:
# very flat, as was expected based on Hessian
ll_nosim <- Vectorize(function(x) {
  model["Q"] <- x
  logLik(model)
})
ll_sim <- Vectorize(function(x) {
  model["Q"] <- x
  logLik(model, nsim = 100)
})
curve(ll_nosim(x), from = 0.1, to = 0.5, ylim = c(-106, -104.5))
curve(ll_sim(x), from = 0.1, to = 0.5, add = TRUE, col = "red")

## End(Not run)
```

---

`coef.SSModel` *Smoothed Estimates or One-step-ahead Predictions of States*

---

## Description

Compute smoothed estimates or one-step-ahead predictions of states of `SSModel` object or extract them from output of `KFS`. For non-Gaussian models without simulation (`nsim = 0`), these are the estimates of conditional modes of states. For Gaussian models and non-Gaussian models with importance sampling, these are the estimates of conditional means of states.

## Usage

```
## S3 method for class 'KFS'
coef(
  object,
  start = NULL,
  end = NULL,
  filtered = FALSE,
  states = "all",
  last = FALSE,
```

```
  ...
)

## S3 method for class 'SSModel'
coef(
  object,
  start = NULL,
  end = NULL,
  filtered = FALSE,
  states = "all",
  last = FALSE,
  nsim = 0,
  ...
)
```

## Arguments

| | |
|---|---|
| object | An object of class KFS or SSModel. |
| start | The start time of the period of interest. Defaults to first time point of the object. |
| end | The end time of the period of interest. Defaults to the last time point of the object. |
| filtered | Logical, return filtered instead of smoothed estimates of state vector. Default is FALSE. |
| states | Which states to extract? Either a numeric vector containing the indices of the corresponding states, or a character vector defining the types of the corresponding states. Possible choices are "all", "level", "slope", "trend", "regression", "arima", "custom", "cycle" or "seasonal", where "trend" extracts all states relating to trend. These can be combined. Default is "all". |
| last | If TRUE, extract only the last time point as numeric vector (ignoring start and end). Default is FALSE. |
| ... | Additional arguments to KFS. Ignored in method for object of class KFS. |
| nsim | Only for method for for non-Gaussian model of class SSModel. The number of independent samples used in importance sampling. Default is 0, which computes the approximating Gaussian model by approxSSM and performs the usual Gaussian filtering/smoothing so that the smoothed state estimates equals to the conditional mode of $p(\alpha_t\|y)$. In case of nsim = 0, the mean estimates and their variances are computed using the Delta method (ignoring the covariance terms). |

## Value

Multivariate time series containing estimates states.

## Examples

```
model <- SSModel(log(drivers) ~ SSMtrend(1, Q = list(1)) +
 SSMseasonal(period = 12, sea.type = "trigonometric") +
 log(PetrolPrice) + law, data = Seatbelts, H = 1)
```

```
coef(model, states = "regression", last = TRUE)
coef(model, start = c(1983, 12), end = c(1984, 2))
out <- KFS(model)
coef(out, states = "regression", last = TRUE)
coef(out, start = c(1983, 12), end = c(1984, 2))
```

---

confint.KFS *Confidence Intervals of Smoothed States*

---

## Description

Extract confidence intervals of the smoothed estimates of states from the output of KFS.

## Usage

```
## S3 method for class 'KFS'
confint(object, parm = "all", level = 0.95, ...)
```

## Arguments

| | |
|---|---|
| object | An object of class KFS. |
| parm | Which states to extract? Either a numeric vector containing the indices of the corresponding states, or a character vector defining the types of the corresponding states. Possible choices are "all", "level", "slope", "trend", "regression", "arima", "custom", "cycle" or "seasonal", where "trend" extracts all states relating to trend. These can be combined. Default is "all". |
| level | The confidence level required. Defaults to 0.95. |
| ... | Ignored. |

## Value

A list of confidence intervals for each state

## Examples

```
model <- SSModel(log(drivers) ~ SSMtrend(1, Q = list(1)) +
 SSMseasonal(period = 12, sea.type = "trigonometric") +
 log(PetrolPrice) + law, data = Seatbelts, H = 1)
out <- KFS(model)

confint(out, parm = "regression")
```

fitSSM *Maximum Likelihood Estimation of a State Space Model*

### Description

Function fitSSM finds the maximum likelihood estimates for unknown parameters of an arbitary state space model, given the user-defined model updating function.

### Usage

```
fitSSM(model, inits, updatefn, checkfn, update_args = NULL, ...)
```

### Arguments

| | |
|---|---|
| model | Model object of class SSModel. |
| inits | Initial values for optim. |
| updatefn | User defined function which updates the model given the parameters. Must be of form updatefn(pars, model, ...), where ... correspond to optional additional arguments. Function should return the original model with updated parameters. See details for description of the default updatefn. |
| checkfn | Optional function of form checkfn(model) for checking the validity of the model. Should return TRUE if the model is valid, and FALSE otherwise. See details. |
| update_args | Optional list containing additional arguments to updatefn. |
| ... | Further arguments for functions optim and logLik.SSModel, such as nsim = 1000, marginal = TRUE, and method = "BFGS". |

### Details

Note that fitSSM actually minimizes -logLik(model), so for example the Hessian matrix returned by hessian = TRUE has an opposite sign than expected.

This function is simple wrapper around optim. For optimal performance in complicated problems, it is more efficient to use problem specific codes with calls to logLik method directly.

In fitSSM, the objective function for optim first updates the model based on the current values of the parameters under optimization, using function updatefn. Then function checkfn is used for checking that the resulting model is valid (the default checkfn checks for non-finite values and overly large (>1e7) values in covariance matrices). If checkfn returns TRUE, the log-likelihood is computed using a call -logLik(model,check.model = FALSE). Otherwise objective function returns value corresponding to .Machine$double.xmax^0.75.

The default updatefn can be used to estimate the values marked as NA in unconstrained time-invariant covariance matrices Q and H. Note that the default updatefn function cannot be used with trigonometric seasonal components as its covariance structure is of form $\sigma$I, i.e. not all NA's correspond to unique value.

The code for the default updatefn can be found in the examples. As can be seen from the function definition, it is assumed that unconstrained optimization method such as BFGS is used.

Note that for non-Gaussian models derivative-free optimization methods such as Nelder-Mead might be more reliable than methods which use finite difference approximations. This is due to noise caused by the relative stopping criterion used for finding approximating Gaussian model. In most cases this does not seem to cause any problems though.

## Value

A list with elements

| | |
|---|---|
| optim.out | Output from function optim. |
| model | Model with estimated parameters. |

## See Also

logLik, KFAS, boat, sexratio, GlobalTemp, SSModel, importanceSSM, approxSSM for more examples.

## Examples

```
# Example function for updating covariance matrices H and Q
# (also used as a default function in fitSSM)

updatefn <- function(pars, model){
  if(any(is.na(model$Q))){
    Q <- as.matrix(model$Q[,,1])
    naQd  <- which(is.na(diag(Q)))
    naQnd <- which(upper.tri(Q[naQd,naQd]) & is.na(Q[naQd,naQd]))
    Q[naQd,naQd][lower.tri(Q[naQd,naQd])] <- 0
    diag(Q)[naQd] <- exp(0.5 * pars[1:length(naQd)])
    Q[naQd,naQd][naQnd] <- pars[length(naQd)+1:length(naQnd)]
    model$Q[naQd,naQd,1] <- crossprod(Q[naQd,naQd])
  }
 if(!identical(model$H,'Omitted') && any(is.na(model$H))){#'
   H<-as.matrix(model$H[,,1])
   naHd  <- which(is.na(diag(H)))
   naHnd <- which(upper.tri(H[naHd,naHd]) & is.na(H[naHd,naHd]))
   H[naHd,naHd][lower.tri(H[naHd,naHd])] <- 0
   diag(H)[naHd] <-
     exp(0.5 * pars[length(naQd)+length(naQnd)+1:length(naHd)])
   H[naHd,naHd][naHnd] <-
     pars[length(naQd)+length(naQnd)+length(naHd)+1:length(naHnd)]
   model$H[naHd,naHd,1] <- crossprod(H[naHd,naHd])
   }
 model
}

# Example function for checking the validity of covariance matrices.

checkfn <- function(model){
  #test positive semidefiniteness of H and Q
  !inherits(try(ldl(model$H[,,1]),TRUE),'try-error') &&
  !inherits(try(ldl(model$Q[,,1]),TRUE),'try-error')
```

```
}

model <- SSModel(Nile ~ SSMtrend(1, Q = list(matrix(NA))), H = matrix(NA))

#function for updating the model
update_model <- function(pars, model) {
  model["H"] <- pars[1]
  model["Q"] <- pars[2]
  model
}

#check that variances are non-negative
check_model <- function(model) {
  (model["H"] > 0 && model["Q"] > 0)
}

fit <- fitSSM(inits = rep(var(Nile)/5, 2), model = model,
                 updatefn = update_model, checkfn = check_model)

# More complex model

set.seed(1)

n <- 1000

x1 <- rnorm(n)
x2 <- rnorm(n)
beta1 <- 1 + cumsum(rnorm(n, sd = 0.1)) # time-varying regression effect
beta2 <- -0.3 # time-invariant effect

# ARMA(2, 1) errors
z <- arima.sim(model = list(ar = c(0.7, -0.4), ma = 0.5), n = n, sd = 0.5)

# generate data, regression part + ARMA errors
y <- beta1 * x1 + beta2 * x2 + z
ts.plot(y)

# build the model using just zeros for now
# But note no extra white noise term so H is fixed to zero
model <- SSModel(y ~ SSMregression(~ x1 + x2, Q = 0, R = matrix(c(1, 0), 2, 1)) +
  SSMarima(rep(0, 2), 0, Q = 0), H = 0)

# update function for fitSSM

update_function <- function(pars, model){

  ## separate calls for model components, use exp to ensure positive variances
  tmp_reg <- SSMregression(~ x1 + x2, Q = exp(pars[1]), R = matrix(c(1, 0), 2, 1))
  tmp_arima <- try(SSMarima(artransform(pars[2:3]),
    artransform(pars[4]), Q = exp(pars[5])), silent = TRUE)

  # stationary check, see note in artransform docs
```

```
  if(inherits(tmp_arima, "try-error")) {
    model$Q[] <- NA # set something to NA just in case original model is ok
    return(model) # this goes to checkfn and causes rejection due to NA values
  }

  model["Q", etas = "regression"] <- tmp_reg$Q
  model["Q", etas = "arima"] <- tmp_arima$Q

  model["T", "arima"] <- tmp_arima$T
  model["R", states = "arima", etas = "arima"] <- tmp_arima$R
  model["P1", "arima"] <- tmp_arima$P1

  # you could also directly build the whole model here again, i.e.
  # model <- SSModel(y ~
  #   SSMregression(~ x1 + x2, Q = exp(pars[1]), R = matrix(c(1, 0), 2, 1)) +
  #   SSMarima(artransform(pars[2:3]), artransform(pars[4]), Q = exp(pars[5])),
  #   H = 0)

  model

}
fit  <- fitSSM(model = model,
  inits = rep(0.1, 5),
  updatefn = update_function, method = "BFGS")

ts.plot(cbind(beta1, KFS(fit$model)$alphahat[, "x1"]), col = 1:2)
```

---

`fitted.SSModel` *Smoothed Estimates or One-step-ahead Predictions of Fitted Values*

---

## Description

Computes fitted values from output of `KFS` (or using the `SSModel` object), i.e. one-step-ahead predictions $f(\theta_t|y_{t-1},\ldots,y_1)$ (`m`) or smoothed estimates $f(\theta_t|y_n,\ldots,y_1)$ (`muhat`), where $f$ is the inverse of the link function (identity in Gaussian case), except in case of Poisson distribution where $f$ is multiplied with the exposure $u_t$.

## Usage

```
## S3 method for class 'KFS'
fitted(object, start = NULL, end = NULL, filtered = FALSE, ...)

## S3 method for class 'SSModel'
fitted(object, start = NULL, end = NULL, filtered = FALSE, nsim = 0, ...)
```

## Arguments

`object` An object of class `KFS` or `SSModel`.

| | |
|---|---|
| `start` | The start time of the period of interest. Defaults to first time point of the object. |
| `end` | The end time of the period of interest. Defaults to the last time point of the object. |
| `filtered` | Logical, return filtered instead of smoothed estimates of state vector. Default is `FALSE`. |
| `...` | Additional arguments to `KFS`. Ignored in method for object of class `KFS`. |
| `nsim` | Only for method for for non-Gaussian model of class `SSModel`. The number of independent samples used in importance sampling. Default is 0, which computes the approximating Gaussian model by `approxSSM` and performs the usual Gaussian filtering/smoothing so that the smoothed state estimates equals to the conditional mode of $p(\alpha_t \vert y)$. In case of `nsim = 0`, the mean estimates and their variances are computed using the Delta method (ignoring the covariance terms). |

### Value

Multivariate time series containing fitted values.

### See Also

`signal` for partial signals and their covariances.

### Examples

```
data("sexratio")
model <- SSModel(Male ~ SSMtrend(1,Q = list(NA)),u = sexratio[, "Total"],
  data = sexratio, distribution = "binomial")
model <- fitSSM(model,inits = -15, method = "BFGS")$model
out <- KFS(model)
identical(drop(out$muhat), fitted(out))

fitted(model)
```

---

## GlobalTemp — *Two series of average global temperature deviations for years 1880-1987*

---

### Description

This data set contains two series of average global temperature deviations for years 1880-1987. These series are same as used in Shumway and Stoffer (2006), where they are known as HL and Folland series. For more details, see Shumway and Stoffer (2006, p. 327).

### Format

A time series object containing 108 times 2 observations.

### Source

http://lib.stat.cmu.edu/general/stoffer/tsa2/

### References

Shumway, Robert H. and Stoffer, David S. (2006). Time Series Analysis and Its Applications: With R examples.

### Examples

```
# Example of multivariate local level model with only one state
# Two series of average global temperature deviations for years 1880-1987
# See Shumway and Stoffer (2006), p. 327 for details

data("GlobalTemp")

model_temp <- SSModel(GlobalTemp ~ SSMtrend(1, Q = NA, type = "common"),
  H = matrix(NA, 2, 2))

# Estimating the variance parameters
inits <- chol(cov(GlobalTemp))[c(1, 4, 3)]
inits[1:2] <- log(inits[1:2])
fit_temp <- fitSSM(model_temp, c(0.5*log(.1), inits), method = "BFGS")

out_temp <- KFS(fit_temp$model)

ts.plot(cbind(model_temp$y, coef(out_temp)), col = 1:3)
legend("bottomright",
  legend = c(colnames(GlobalTemp), "Smoothed signal"), col = 1:3, lty = 1)
```

---

## hatvalues.KFS *Extract Hat Values from KFS Output*

---

### Description

Extract hat values from KFS output, when KFS was run with signal (non-Gaussian case) or mean smoothing (Gaussian case).

### Usage

```
## S3 method for class 'KFS'
hatvalues(model, ...)
```

### Arguments

| | |
|---|---|
| model | An object of class KFS. |
| ... | Additional arguments to approxSSM. |

## Details

Hat values in KFAS are defined as the diagonal elements of V_t/H_t where V_t is the covariance matrix of signal/mean at time t and H_t is the covariance matrix of disturbance vector $\epsilon$ of (approximating) Gaussian model at time t. This definition gives identical results with the standard definition in case of GLMs. Note that it is possible to construct a state space model where this definition is not meaningful (for example the covariance matrix H_t can contain zeros on diagonal).

## Value

Multivariate time series containing hat values.

## Examples

```
model <- SSModel(sr ~ pop15 + pop75 + dpi + ddpi, data = LifeCycleSavings)
out <- KFS(model, filtering = "state", smoothing = "none")
# estimate sigma2
model["H"] <- mean(c(out$v[1:out$d][out$Finf==0]^2/out$F[1:out$d][out$Finf==0],
                     out$v[-(1:out$d)]^2/out$F[-(1:out$d)]))
c(hatvalues(KFS(model)))
```

---

importanceSSM *Importance Sampling of Exponential Family State Space Model*

---

## Description

Function importanceSSM simulates states or signals of the exponential family state space model conditioned with the observations, returning the simulated samples of the states/signals with the corresponding importance weights.

## Usage

```
importanceSSM(
  model,
  type = c("states", "signals"),
  filtered = FALSE,
  nsim = 1000,
  save.model = FALSE,
  theta,
  antithetics = FALSE,
  maxiter = 50,
  expected = FALSE,
  H_tol = 1e+15
)
```

### Arguments

| | |
|---|---|
| model | Exponential family state space model of class SSModel. |
| type | What to simulate, "states" or "signals". Default is "states" |
| filtered | Simulate from $p(\alpha_t \mid y_{t-1}, ..., y_1)$ instead of $p(\alpha \mid y)$. Note that for large models this can be very slow. Default is FALSE. |
| nsim | Number of independent samples. Default is 1000. |
| save.model | Return the original model with the samples. Default is FALSE. |
| theta | Initial values for the conditional mode theta. |
| antithetics | Logical. If TRUE, two antithetic variables are used in simulations, one for location and another for scale. Default is FALSE. |
| maxiter | Maximum number of iterations used in linearisation. Default is 50. |
| expected | Logical value defining the approximation of H_t in case of Gamma and negative binomial distribution. Default is FALSE which matches the algorithm of Durbin & Koopman (1997), whereas TRUE uses the expected value of observations in the equations, leading to results which match with glm (where applicable). The latter case was the default behaviour of KFAS before version 1.3.8. Essentially this is the difference between observed and expected information. |
| H_tol | Tolerance parameter for check max(H) > H_tol, which suggests that the approximation converged to degenerate case with near zero signal-to-noise ratio. Default is very generous 1e15. |

### Details

Function can use two antithetic variables, one for location and other for scale, so output contains four blocks of simulated values which correlate which each other (ith block correlates negatively with (i+1)th block, and positively with (i+2)th block etc.).

### Value

A list containing elements

| | |
|---|---|
| samples | Simulated samples. |
| weights | Importance weights. |
| model | Original model in case of save.model==TRUE. |

### Examples

```
data("sexratio")
model <- SSModel(Male ~ SSMtrend(1, Q = list(NA)), u = sexratio[,"Total"], data = sexratio,
                 distribution = "binomial")
fit <- fitSSM(model, inits = -15, method = "BFGS")
fit$model$Q #1.107652e-06
# Computing confidence intervals for sex ratio
# Uses importance sampling on response scale (1000 samples with antithetics)
set.seed(1)
imp <- importanceSSM(fit$model, nsim = 250, antithetics = TRUE)
```

```
sexratio.smooth <- numeric(length(model$y))
sexratio.ci <- matrix(0, length(model$y), 2)
w <- imp$w/sum(imp$w)
for(i in 1:length(model$y)){
  sexr <- exp(imp$sample[i,1,])
  sexratio.smooth[i]<-sum(sexr*w)
  oo <- order(sexr)
  sexratio.ci[i,] <- c(sexr[oo][which.min(abs(cumsum(w[oo]) - 0.05))],
                   sexr[oo][which.min(abs(cumsum(w[oo]) - 0.95))])
}

## Not run:
# Filtered estimates
impf <- importanceSSM(fit$model, nsim = 250, antithetics = TRUE,filtered=TRUE)
sexratio.filter <- rep(NA,length(model$y))
sexratio.fci <- matrix(NA, length(model$y), 2)
w <- impf$w/rowSums(impf$w)
for(i in 2:length(model$y)){
  sexr <- exp(impf$sample[i,1,])
  sexratio.filter[i] <- sum(sexr*w[i,])
  oo<-order(sexr)
  sexratio.fci[i,] <- c(sexr[oo][which.min(abs(cumsum(w[i,oo]) - 0.05))],
                    sexr[oo][which.min(abs(cumsum(w[i,oo]) - 0.95))])
}

ts.plot(cbind(sexratio.smooth,sexratio.ci,sexratio.filter,sexratio.fci),
        col=c(1,1,1,2,2,2),lty=c(1,2,2,1,2,2))

## End(Not run)
```

---

`is.SSModel` *Test whether object is a valid* `SSModel` *object*

---

## Description

Function `is.SSModel` tests whether the object is a valid `SSModel` object.

## Usage

```
is.SSModel(object, na.check = FALSE, return.logical = TRUE)
```

## Arguments

| | |
|---|---|
| `object` | An object to be tested. |
| `na.check` | Test the system matrices for `NA` and infinite values. Also checks for large values (> 1e7) in covariance matrices `H` and `Q` which could cause large rounding errors in filtering. Positive semidefiniteness of these matrices is not checked. Default is `FALSE`. |
| `return.logical` | If `FALSE` (default), an error is given if the the model is not a valid `SSModel` object. Otherwise logical value is returned. |

### Details

Note that the validity of the values in `y` and `Z` are not tested. These can contain NA values (but not infinite values), with condition that when `Z[i,,t]` contains NA value, the corresponding `y[t,i]` must also have NA value. In this case `Z[i,,t]` is not referenced in filtering and smoothing, and algorithms works properly.

### Value

Logical value or nothing, depending on the value of `return.logical`.

### Examples

```
model <- SSModel(rnorm(10) ~ 1)
is.SSModel(model)
model['H'] <- 1
is.SSModel(model)
model$H[] <- 1
is.SSModel(model)
model$H[,,1] <- 1
is.SSModel(model)
model$H <- 1
is.SSModel(model)
```

---

## KFAS — *KFAS: Functions for Exponential Family State Space Models*

---

### Description

Package KFAS contains functions for Kalman filtering, smoothing and simulation of linear state space models with exact diffuse initialization.

### Details

Note, this help page might be more readable in pdf format due to the mathematical formulas containing subscripts.

The linear Gaussian state space model is given by

$$y_t = Z_t \alpha_t + \epsilon_t, \text{(observation equation)}$$

$$\alpha_{t+1} = T_t \alpha_t + R_t \eta_t, \text{(transition equation)}$$

where $\epsilon_t \sim N(0, H_t)$, $\eta_t \sim N(0, Q_t)$ and $\alpha_1 \sim N(a_1, P_1)$ independently of each other.

All system and covariance matrices `Z`, `H`, `T`, `R` and `Q` can be time-varying, and partially or totally missing observations $y_t$ are allowed.

Covariance matrices H and Q has to be positive semidefinite (although this is not checked).

Model components in `KFAS` are defined as

**y** A n x p matrix containing the observations.

**Z** A p x m x 1 or p x m x n array corresponding to the system matrix of observation equation.

**H** A p x p x 1 or p x p x n array corresponding to the covariance matrix of observational disturbances epsilon.

**T** A m x m x 1 or m x m x n array corresponding to the first system matrix of state equation.

**R** A m x k x 1 or m x k x n array corresponding to the second system matrix of state equation.

**Q** A k x k x 1 or k x k x n array corresponding to the covariance matrix of state disturbances eta

**a1** A m x 1 matrix containing the expected values of the initial states.

**P1** A m x m matrix containing the covariance matrix of the nondiffuse part of the initial state vector.

**P1inf** A m x m matrix containing the covariance matrix of the diffuse part of the initial state vector.

**u** A n x p matrix of an additional parameters in case of non-Gaussian model.

In case of any of the series in model is defined as non-Gaussian, the observation equation is of form

$$\prod_i^p p_i(y_{t,p}|\theta_t)$$

with $\theta_{t,i} = Z_{i,t}\alpha_t$ being one of the following:

- $y_t \sim N(\mu_t, u_t)$, with identity link $\theta_t = \mu_t$. Note that now variances are defined using $u_t$, not $H_t$. If the correlation between Gaussian observation equations is needed, one can use $u_t = 0$ and add correlating disturbances into state equation (although care is then needed when making inferences about signal which contains the error terms also).
- $y_t \sim$ Poisson$(u_t\lambda_t)$, where $u_t$ is an offset term, with $\theta_t = log(\lambda_t)$.
- $y_t \sim$ binomial$(u_t, \pi_t)$, with $\theta_t = log[\pi_t/(1-\pi_t)]$, where $\pi_t$ is the probability of success at time $t$.
- $y_t \sim$ gamma$(u_t, \mu_t)$, with $\theta_t = log(\mu_t)$, where $\mu_t$ is the mean parameter and $u_t$ is the shape parameter.
- $y_t \sim$ negative binomial$(u_t, \mu_t)$, with expected value $\mu_t$ and variance $\mu_t + \mu_t^2/u_t$ (see `dnbinom`), then $\theta_t = log(\mu_t)$.

For exponential family models $u_t = 1$ as a default. For completely Gaussian models, parameter is omitted. Note that series can have different distributions in case of multivariate models.

For the unknown elements of initial state vector $a_1$, KFAS uses exact diffuse initialization by Koopman and Durbin (2000, 2012, 2003), where the unknown initial states are set to have a zero mean and infinite variance, so

$$P_1 = P_{*,1} + \kappa P_{\infty,1},$$

with $\kappa$ going to infinity and $P_{\infty,1}$ being diagonal matrix with ones on diagonal elements corresponding to unknown initial states.

This method is basically a equivalent of setting uninformative priors for the initial states in a Bayesian framework.

Diffuse phase is continued until rank of $P_{\infty,t}$ becomes zero. Rank of $P_{\infty,t}$ decreases by 1, if $F_{\infty,t} > \xi_t > 0$, where $\xi_t$ is by default `.Machine$double.eps^0.5*min(X)^2)`, where X is absolute values of non-zero elements of array Z. Usually the number of diffuse time points equals the number unknown elements of initial state vector, but missing observations or time-varying system matrices can affect this. See Koopman and Durbin (2000, 2012, 2003) for details for exact diffuse and non-diffuse filtering. If the number of diffuse states is large compared to the data, it is possible that the model is degenerate in a sense that not enough information is available for leaving the diffuse phase.

To lessen the notation and storage space, KFAS uses letters P, F and K for non-diffuse part of the corresponding matrices, omitting the asterisk in diffuse phase.

All functions of KFAS use the univariate approach (also known as sequential processing, see Anderson and Moore (1979)) which is from Koopman and Durbin (2000, 2012). In univariate approach the observations are introduced one element at the time. Therefore the prediction error variance matrices F and Finf do not need to be non-singular, as there is no matrix inversions in univariate approach algorithm. This provides possibly more faster filtering and smoothing than normal multivariate Kalman filter algorithm, and simplifies the formulas for diffuse filtering and smoothing. If covariance matrix H is not diagonal, it is possible to transform the model by either using LDL decomposition on H, or augmenting the state vector with $\epsilon$ disturbances (this is done automatically in KFAS if needed). See `transformSSM` for more details.

## References

Helske J. (2017). KFAS: Exponential Family State Space Models in R, Journal of Statistical Software, 78(10), 1-39. doi:10.18637/jss.v078.i10

Koopman, S.J. and Durbin J. (2000). Fast filtering and smoothing for non-stationary time series models, Journal of American Statistical Assosiation, 92, 1630-38.

Koopman, S.J. and Durbin J. (2012). Time Series Analysis by State Space Methods. Second edition. Oxford: Oxford University Press.

Koopman, S.J. and Durbin J. (2003). Filtering and smoothing of state vector for diffuse state space models, Journal of Time Series Analysis, Vol. 24, No. 1.

Shumway, Robert H. and Stoffer, David S. (2006). Time Series Analysis and Its Applications: With R examples.

## See Also

See also `logLik`, `fitSSM`, `boat`, `sexratio`, `GlobalTemp`, `SSModel`, `importanceSSM`, `approxSSM` for more examples.

## Examples

```
################################################
# Example of local level model for Nile series #
################################################
# See Durbin and Koopman (2012) and also ?SSModel for another Nile example

model_Nile <- SSModel(Nile ~
  SSMtrend(1, Q = list(matrix(NA))), H = matrix(NA))
model_Nile
model_Nile <- fitSSM(model_Nile, c(log(var(Nile)), log(var(Nile))),
  method = "BFGS")$model
```

```
# Filtering and state smoothing
out_Nile <- KFS(model_Nile, filtering = "state", smoothing = "state")
out_Nile

# Confidence and prediction intervals for the expected value and the observations.
# Note that predict uses original model object, not the output from KFS.
conf_Nile <- predict(model_Nile, interval = "confidence", level = 0.9)
pred_Nile <- predict(model_Nile, interval = "prediction", level = 0.9)

ts.plot(cbind(Nile, pred_Nile, conf_Nile[, -1]), col = c(1:2, 3, 3, 4, 4),
        ylab = "Predicted Annual flow", main = "River Nile")


# Missing observations, using the same parameter estimates

NileNA <- Nile
NileNA[c(21:40, 61:80)] <- NA
model_NileNA <- SSModel(NileNA ~ SSMtrend(1, Q = list(model_Nile$Q)),
H = model_Nile$H)

out_NileNA <- KFS(model_NileNA, "mean", "mean")

# Filtered and smoothed states
ts.plot(NileNA, fitted(out_NileNA, filtered = TRUE), fitted(out_NileNA),
  col = 1:3, ylab = "Predicted Annual flow",
  main = "River Nile")


## Not run:
##################
# Seatbelts data #
##################
# See Durbin and Koopman (2012)

model_drivers <- SSModel(log(drivers) ~ SSMtrend(1, Q = list(NA))+
   SSMseasonal(period = 12, sea.type = "trigonometric", Q = NA) +
   log(PetrolPrice) + law, data = Seatbelts, H = NA)

# As trigonometric seasonal contains several disturbances which are all
# identically distributed, default behaviour of fitSSM is not enough,
# as we have constrained Q. We can either provide our own
# model updating function with fitSSM, or just use optim directly:

# option 1:
ownupdatefn <- function(pars, model){
  model$H[] <- exp(pars[1])
  diag(model$Q[, , 1]) <- exp(c(pars[2], rep(pars[3], 11)))
  model #for optim, replace this with -logLik(model) and call optim directly
}

fit_drivers <- fitSSM(model_drivers,
  log(c(var(log(Seatbelts[, "drivers"])), 0.001, 0.0001)),
```

```
  ownupdatefn, method = "BFGS")

out_drivers <- KFS(fit_drivers$model, smoothing = c("state", "mean"))
out_drivers
ts.plot(out_drivers$model$y, fitted(out_drivers), lty = 1:2, col = 1:2,
  main = "Observations and smoothed signal with and without seasonal component")
lines(signal(out_drivers, states = c("regression", "trend"))$signal,
  col = 4, lty = 1)
legend("bottomleft", col = c(1, 2, 4), lty = c(1, 2, 1),
  legend = c("Observations", "Smoothed signal", "Smoothed level"))

# Multivariate model with constant seasonal pattern,
# using the the seat belt law dummy only for the front seat passangers,
# and restricting the rank of the level component by using custom component

model_drivers2 <- SSModel(log(cbind(front, rear)) ~ -1 +
    log(PetrolPrice) + log(kms) +
    SSMregression(~law, data = Seatbelts, index = 1) +
    SSMcustom(Z = diag(2), T = diag(2), R = matrix(1, 2, 1),
      Q = matrix(1), P1inf = diag(2)) +
    SSMseasonal(period = 12, sea.type = "trigonometric"),
  data = Seatbelts, H = matrix(NA, 2, 2))

# An alternative way for defining the rank deficient trend component:

# model_drivers2 <- SSModel(log(cbind(front, rear)) ~ -1 +
#     log(PetrolPrice) + log(kms) +
#     SSMregression(~law, data = Seatbelts, index = 1) +
#     SSMtrend(degree = 1, Q = list(matrix(0, 2, 2))) +
#     SSMseasonal(period = 12, sea.type = "trigonometric"),
#   data = Seatbelts, H = matrix(NA, 2, 2))
#
# Modify model manually:
# model_drivers2$Q <- array(1, c(1, 1, 1))
# model_drivers2$R <- model_drivers2$R[, -2, , drop = FALSE]
# attr(model_drivers2, "k") <- 1L
# attr(model_drivers2, "eta_types") <- attr(model_drivers2, "eta_types")[1]


likfn <- function(pars, model, estimate = TRUE){
  diag(model$H[, , 1]) <- exp(0.5 * pars[1:2])
  model$H[1, 2, 1] <- model$H[2, 1, 1] <-
    tanh(pars[3]) * prod(sqrt(exp(0.5 * pars[1:2])))
  model$R[28:29] <- exp(pars[4:5])
  if(estimate) return(-logLik(model))
  model
}

fit_drivers2 <- optim(f = likfn, p = c(-8, -8, 1, -1, -3), method = "BFGS",
  model = model_drivers2)
model_drivers2 <- likfn(fit_drivers2$p, model_drivers2, estimate = FALSE)
model_drivers2$R[28:29, , 1]%*%t(model_drivers2$R[28:29, , 1])
model_drivers2$H
```

```
out_drivers2 <- KFS(model_drivers2)
out_drivers2
ts.plot(signal(out_drivers2, states = c("custom", "regression"))$signal,
  model_drivers2$y, col = 1:4)

# For confidence or prediction intervals, use predict on the original model
pred <- predict(model_drivers2,
  states = c("custom", "regression"), interval = "prediction")

# Note that even though the intervals were computed without seasonal pattern,
# PetrolPrice induces seasonal pattern to predictions
ts.plot(pred$front, pred$rear, model_drivers2$y,
  col = c(1, 2, 2, 3, 4, 4, 5, 6), lty = c(1, 2, 2, 1, 2, 2, 1, 1))

## End(Not run)

######################
# ARMA(2, 2) process #
######################
set.seed(1)
y <- arima.sim(n = 1000, list(ar = c(0.8897, -0.4858), ma = c(-0.2279, 0.2488)),
               innov = rnorm(1000) * sqrt(0.5))


model_arima <- SSModel(y ~ SSMarima(ar = c(0, 0), ma = c(0, 0), Q = 1), H = 0)

likfn <- function(pars, model, estimate = TRUE){
  tmp <- try(SSMarima(artransform(pars[1:2]), artransform(pars[3:4]),
    Q = exp(pars[5])), silent = TRUE)
  if(!inherits(tmp, "try-error")){
    model["T", "arima"] <- tmp$T
    model["R", states = "arima", etas = "arima"] <- tmp$R
    model["P1", "arima"] <- tmp$P1
    model["Q", etas = "arima"] <- tmp$Q
    if(estimate){
      -logLik(model)
    } else model
  } else {
    if(estimate){
      1e100
    } else model
  }
}

fit_arima <- optim(par = c(rep(0, 4), log(1)), fn = likfn, method = "BFGS",
  model = model_arima)
model_arima <- likfn(fit_arima$par, model_arima, FALSE)

# AR coefficients:
model_arima$T[2:3, 2, 1]
# MA coefficients:
model_arima$R[3:4]
```

```
# sigma2:
model_arima$Q[1]
# intercept
KFS(model_arima)
# same with arima:
arima(y, c(2, 0, 2))
# small differences because the intercept is handled differently in arima

## Not run:
#################
# Poisson model #
#################
# See Durbin and Koopman (2012)
model_van <- SSModel(VanKilled ~ law + SSMtrend(1, Q = list(matrix(NA)))+
               SSMseasonal(period = 12, sea.type = "dummy", Q = NA),
               data = Seatbelts, distribution = "poisson")

# Estimate variance parameters
fit_van <- fitSSM(model_van, c(-4, -7), method = "BFGS")

model_van <- fit_van$model

# use approximating model, gives posterior modes
out_nosim <- KFS(model_van, nsim = 0)
# State smoothing via importance sampling
out_sim <- KFS(model_van, nsim = 1000)

out_nosim
out_sim

## End(Not run)

## using deterministic inputs in observation and state equations
model_Nile <- SSModel(Nile ~
  SSMcustom(Z=1, T = 1, R = 0, a1 = 100, P1inf = 0, P1 = 0, Q = 0, state_names = "d_t") +
  SSMcustom(Z=0, T = 1, R = 0, a1 = 100, P1inf = 0, P1 = 0, Q = 0, state_names = "c_t") +
  SSMtrend(1, Q = 1500), H = 15000)
model_Nile$T
model_Nile$T[1, 3, 1] <- 1 # add c_t to level
model_Nile0 <- SSModel(Nile ~
  SSMtrend(2, Q = list(1500, 0), a1 = c(0, 100), P1inf = diag(c(1, 0))),
  H = 15000)

ts.plot(KFS(model_Nile0)$mu, KFS(model_Nile)$mu, col = 1:2)

##########################################################
### Examples of generalized linear modelling with KFAS ###
##########################################################

# Same example as in ?glm
counts <- c(18, 17, 15, 20, 10, 20, 25, 13, 12)
outcome <- gl(3, 1, 9)
treatment <- gl(3, 3)
```

```
glm_D93 <- glm(counts ~ outcome + treatment, family = poisson())

model_D93 <- SSModel(counts ~ outcome + treatment,
  distribution = "poisson")

out_D93 <- KFS(model_D93)
coef(out_D93, last = TRUE)
coef(glm_D93)

summary(glm_D93)$cov.s
out_D93$V[, , 1]

# approximating model as in GLM
out_D93_nosim <- KFS(model_D93, smoothing = c("state", "signal", "mean"),
  expected = TRUE)

# with importance sampling. Number of simulations is too small here,
# with large enough nsim the importance sampling actually gives
# very similar results as the approximating model in this case
set.seed(1)
out_D93_sim <- KFS(model_D93,
  smoothing = c("state", "signal", "mean"), nsim = 1000)


## linear predictor
# GLM
glm_D93$linear.predictor
# approximate model, this is the posterior mode of p(theta|y)
c(out_D93_nosim$thetahat)
# importance sampling on theta, gives E(theta|y)
c(out_D93_sim$thetahat)


## predictions on response scale
# GLM
fitted(glm_D93)
# approximate model with backtransform, equals GLM
fitted(out_D93_nosim)
# importance sampling on exp(theta)
fitted(out_D93_sim)

# prediction variances on link scale
# GLM
as.numeric(predict(glm_D93, type = "link", se.fit = TRUE)$se.fit^2)
# approx, equals to GLM results
c(out_D93_nosim$V_theta)
# importance sampling on theta
c(out_D93_sim$V_theta)


# prediction variances on response scale
# GLM
as.numeric(predict(glm_D93, type = "response", se.fit = TRUE)$se.fit^2)
```

```
# approx, equals to GLM results
c(out_D93_nosim$V_mu)
# importance sampling on theta
c(out_D93_sim$V_mu)

# A Gamma example modified from ?glm
# Now with log-link, and identical intercept terms
clotting <- data.frame(
u = c(5,10,15,20,30,40,60,80,100),
lot1 = c(118,58,42,35,27,25,21,19,18),
lot2 = c(69,35,26,21,18,16,13,12,12))

model_gamma <- SSModel(cbind(lot1, lot2) ~ -1 + log(u) +
    SSMregression(~ 1, type = "common", remove.intercept = FALSE),
  data = clotting, distribution = "gamma")

update_shapes <- function(pars, model) {
  model$u[, 1] <- pars[1]
  model$u[, 2] <- pars[2]
  model
}
fit_gamma <- fitSSM(model_gamma, inits = c(1, 1), updatefn = update_shapes,
method = "L-BFGS-B", lower = 0, upper = 100)
logLik(fit_gamma$model)
KFS(fit_gamma$model)
fit_gamma$model["u", times = 1]



## Not run:
####################################
### Linear mixed model with KFAS ###
####################################

# example from ?lmer of lme4 package
data("sleepstudy", package = "lme4")

model_lmm <- SSModel(Reaction ~ Days +
    SSMregression(~ Days, Q = array(0, c(2, 2, 180)),
       P1 = matrix(NA, 2, 2), remove.intercept = FALSE), sleepstudy, H = NA)

# The first 10 time points the third and fouth state
# defined with SSMregression correspond to the first subject, and next 10 time points
# are related to second subject and so on.

# need to use ordinary $ assignment as [ assignment operator for SSModel
# object guards against dimension altering
model_lmm$T <- array(model_lmm["T"], c(4, 4, 180))
attr(model_lmm, "tv")[3] <- 1L #needs to be integer type!

# "cut the connection" between the subjects
times <- seq(10, 180, by = 10)
model_lmm["T",states = 3:4, times = times] <- 0
```

```
# for the first subject the variance of the random effect is defined via P1
# for others, we use Q
model_lmm["Q", times = times] <- NA

update_lmm <- function(pars = init, model){
  P1 <- diag(exp(pars[1:2]))
  P1[1, 2] <- pars[3]
  model["P1", states = 3:4] <- model["Q", times = times] <-
    crossprod(P1)
  model["H"] <- exp(pars[4])
  model
}

inits <- c(0, 0, 0, 3)

fit_lmm <- fitSSM(model_lmm, inits, update_lmm, method = "BFGS")
out_lmm <- KFS(fit_lmm$model)
# unconditional covariance matrix of random effects
fit_lmm$model["P1", states = 3:4]

# conditional covariance matrix of random effects
# same for each subject and time point due to model structure
# these differ from the ones obtained from lmer as these are not conditioned
# on the fixed effects
out_lmm$V[3:4,3:4,1]

## End(Not run)
## Not run:
#########################################
### Example of cubic spline smoothing ###
#########################################
# See Durbin and Koopman (2012)
require("MASS")
data("mcycle")

model <- SSModel(accel ~ -1 +
    SSMcustom(Z = matrix(c(1, 0), 1, 2),
      T = array(diag(2), c(2, 2, nrow(mcycle))),
      Q = array(0, c(2, 2, nrow(mcycle))),
      P1inf = diag(2), P1 = diag(0, 2)), data = mcycle)

model$T[1, 2, ] <- c(diff(mcycle$times), 1)
model$Q[1, 1, ] <- c(diff(mcycle$times), 1)^3/3
model$Q[1, 2, ] <- model$Q[2, 1, ] <- c(diff(mcycle$times), 1)^2/2
model$Q[2, 2, ] <- c(diff(mcycle$times), 1)


updatefn <- function(pars, model, ...){
  model["H"] <- exp(pars[1])
  model["Q"] <- model["Q"] * exp(pars[2])
  model
}
```

```
fit <- fitSSM(model, inits = c(4, 4), updatefn = updatefn, method = "BFGS")

pred <- predict(fit$model, interval = "conf", level = 0.95)
plot(x = mcycle$times, y = mcycle$accel, pch = 19)
lines(x = mcycle$times, y = pred[, 1])
lines(x = mcycle$times, y = pred[, 2], lty = 2)
lines(x = mcycle$times, y = pred[, 3], lty = 2)

## End(Not run)

## Not run:
###################################################################
# Example of multivariate model with common parameters           #
# and unknown intercept terms in state and observation equations #
###################################################################
set.seed(1)
n1 <- 20
n2 <- 30
z1 <- sin(1:n1)
z2 <- cos(1:n2)

C <- 0.6
D <- -0.4
# random walk with drift D
x1 <- cumsum(rnorm(n1) + D)
x2 <- cumsum(rnorm(n2) + D)

y1 <- rnorm(n1, z1 * x1 + C * 1)
y2 <- rnorm(n2, z2 * x2 + C * 2)

n <- max(n1, n2)
Y <- matrix(NA, n, 2)
Y[1:n1, 1] <- y1
Y[1:n2, 2] <- y2

Z <- array(0, c(2, 4, n))
Z[1, 1, 1:n1] <- z1
Z[2, 2, 1:n2] <- z2 # trailing zeros are ok, as corresponding y is NA
Z[1, 3, ] <- 1 # x = 1
Z[2, 3, ] <- 2 # x = 2
# last state is only used in state equation so zeros in Z

T <- diag(4) # a1_t for y1, a2_t for y2, C, D
T[1, 4] <- 1 # D affects a_t
T[2, 4] <- 1 # D affects a_t
Q <- diag(c(NA, NA, 0, 0))
P1inf <- diag(4)
model <- SSModel(Y ~ -1 + SSMcustom(Z = Z, T = T, Q = Q, P1inf = P1inf,
  state_names = c("a1", "a2", "C", "D")), H = diag(NA, 2))

updatefn <- function(pars, model) {
  model$Q[] <- diag(c(exp(pars[1]), exp(pars[1]), 0, 0))
```

```
  model$H[] <- diag(exp(pars[2]), 2)
  model
}

fit <- fitSSM(model, inits = rep(-1, 2), updatefn = updatefn)

fit$model$H[1]
fit$model$Q[1]
KFS(fit$model)


## End(Not run)
```

---

KFS *Kalman Filter and Smoother with Exact Diffuse Initialization for Exponential Family State Space Models*

---

## Description

Performs Kalman filtering and smoothing with exact diffuse initialization using univariate approach for exponential family state space models.

## Usage

```
KFS(
  model,
  filtering,
  smoothing,
  simplify = TRUE,
  transform = c("ldl", "augment"),
  nsim = 0,
  theta,
  maxiter = 50,
  convtol = 1e-08,
  return_model = TRUE,
  expected = FALSE,
  H_tol = 1e+15,
  transform_tol
)
```

## Arguments

model Object of class SSModel.

filtering Types of filtering. Possible choices are "state", "signal", "mean", and "none". Default is "state" for Gaussian and "none" for non-Gaussian models. Multiple values are allowed. For Gaussian models, the signal is the mean. Note that filtering for non-Gaussian models with importance sampling can be very slow with large models.

smoothing
: Types of smoothing. Possible choices are "state", "signal", "mean", "disturbance", and "none". Default is "state" and "mean". For non-Gaussian models, option "disturbance" is not supported, and for Gaussian models option "mean" is identical to "signal". Multiple values are allowed.

simplify
: If FALSE and the model is completely Gaussian, KFS returns some generally not so interesting variables from filtering and smoothing. Default is TRUE.

transform
: How to transform the model in case of non-diagonal covariance matrix H. Defaults to "ldl". See function transformSSM for details.

nsim
: The number of independent samples used in importance sampling. Only used for non-Gaussian models. Default is 0, which computes the approximating Gaussian model by approxSSM and performs the usual Gaussian filtering/smoothing so that the smoothed state estimates equals to the conditional mode of $p(\alpha_t|y)$. In case of nsim = 0, the mean estimates and their variances are computed using the Delta method (ignoring the covariance terms).

theta
: Initial values for conditional mode theta. Only used for non-Gaussian models.

maxiter
: The maximum number of iterations used in Gaussian approximation. Default is 50. Only used for non-Gaussian models.

convtol
: Tolerance parameter for convergence checks for Gaussian approximation. Only used for non-Gaussian models.

return_model
: Logical, indicating whether the original input model should be returned as part of the output. Defaults to TRUE, but for large models can be set to FALSE in order to save memory. However, many of the methods operating on the output of KFS use this model so this will not work if return_model=FALSE.

expected
: Logical value defining the approximation of H_t in case of Gamma and negative binomial distribution. Default is FALSE which matches the algorithm of Durbin & Koopman (1997), whereas TRUE uses the expected value of observations in the equations, leading to results which match with glm (where applicable). The latter case was the default behaviour of KFAS before version 1.3.8. Essentially this is the difference between observed and expected information in the GLM context. Only used for non-Gaussian model.

H_tol
: Tolerance parameter for check max(H) > tol_H, which suggests that the approximation converged to degenerate case with near zero signal-to-noise ratio. Default is very generous 1e15. Only used for non-Gaussian model.

transform_tol
: Tolerance parameter for LDL decomposition in case of a non-diagonal H and transform = "ldl". See transformSSM and ldl for details.

## Details

Notice that in case of multivariate Gaussian observations, v, F, Finf, K and Kinf are usually not the same as those calculated in usual multivariate Kalman filter. As filtering is done one observation element at the time, the elements of the prediction error $v_t$ are uncorrelated, and F, Finf, K and Kinf contain only the diagonal elemens of the corresponding covariance matrices. The usual multivariate versions of F and v can be obtained from the output of KFS using the function mvInnovations.

In rare cases (typically with regression components with high multicollinearity or long cyclic patterns), the cumulative rounding errors in Kalman filtering and smoothing can cause the diffuse phase end too early, or the backward smoothing gives negative variances (in diffuse and nondiffuse cases). Since version 1.4.0, filtering and smoothing algorithms truncate these values to zero during the recursions, but this can still leads some numerical problems. In these cases, redefining the prior state variances more informative is often helpful. Changing the tolerance parameter `tol` of the model (see `SSModel`) to smaller (or larger), or scaling the model input can sometimes help as well. These numerical issues are well known in Kalman filtering/smoothing in general (there are other numerically more stable versions available, but these are in general slower).

Fon non-Gaussian models the components corresponding to diffuse filtering (`Finf`, `Pinf`, `d`, `Kinf`) are not returned even when `filtering` is used. Results based on approximating Gaussian model can be obtained by running `KFS` using the output of `approxSSM`.

In case of non-Gaussian models with `nsim = 0`, the smoothed estimates relate to the conditional mode of $p(\alpha|y)$. When using importance sampling (`nsim>0`), results correspond to the conditional mean of $p(\alpha|y)$.

## Value

What `KFS` returns depends on the arguments `filtering`, `smoothing` and `simplify`, and whether the model is Gaussian or not:

- `model` Original state space model.
- `KFS_transform` How the non-diagonal `H` was handled.
- `logLik` Value of the log-likelihood function. Only returned for fully Gaussian models.
- `a` One-step-ahead predictions of states, $a_t = E(\alpha_t|y_{t-1},\ldots,y_1)$.
- `P` Non-diffuse parts of the error covariance matrix of predicted states, $P_t = Var(\alpha_t|y_{t-1},\ldots,y_1)$.
- `Pinf` Diffuse part of the error covariance matrix of predicted states. Only returned for Gaussian models.
- `att` Filtered estimates of states, $a_tt = E(\alpha_t|y_t,\ldots,y_1)$.
- `Ptt` Non-diffuse parts of the error covariance matrix of filtered states, $P_tt = Var(\alpha_t|y_t,\ldots,y_1)$.
- `t` One-step-ahead predictions of signals, $E(Z_t\alpha_t|y_{t-1},\ldots,y_1)$.
- `P_theta` Non-diffuse part of $Var(Z_t\alpha_t|y_{t-1},\ldots,y_1)$.
- `m` One-step-ahead predictions $f(\theta_t)|y_{t-1},\ldots,y_1)$, where $f$ is the inverse link function. In case of Poisson distribution these predictions are multiplied with exposure $u_t$.
- `P_mu` Non-diffuse part of $Var(f(\theta_t)|y_{t-1},\ldots,y_1)$. In case of Poisson distribution this is $Var(u_tf(\theta_t)|y_{t-1},\ldots,y_1)$. If `nsim = 0`, only diagonal elements (variances) are computed, using the Delta method.
- `alphahat` Smoothed estimates of states, $E(\alpha_t|y_1,\ldots,y_n)$.
- `V` Error covariance matrices of smoothed states, $Var(\alpha_t|y_1,\ldots,y_n)$.
- `thetahat` Smoothed estimates of signals, $E(Z_t\alpha_t|y_1,\ldots,y_n)$.
- `V_theta` Error covariance matrices of smoothed signals $Var(Z[t]\alpha_t|y_1,\ldots,y_n)$..
- `muhat` Smoothed estimates of $f(\theta_t)|y_1,\ldots,y_n)$, where $f$ is the inverse link function, or in Poisson case $u_tf(\theta_t)|y_1,\ldots,y_n)$, where $u$ is the exposure term.

V_mu
: Error covariances $Cov(f(\theta_t)|y_1, \ldots, y_n)$ (or the covariances of $u_t f(\theta_t)$ given the data in case of Poisson distribution). If `nsim = 0`, only diagonal elements (variances) are computed, using the Delta method.

etahat
: Smoothed disturbance terms $E(\eta_t|y_1, \ldots, y_n)$. Only for Gaussian models.

V_eta
: Error covariances $Var(\eta_t|y_1, \ldots, y_n)$. Note that for computing auxiliary residuals you shoud use method `rstandard.KFS`.

epshat
: Smoothed disturbance terms $E(\epsilon_{t,i}|y_1, \ldots, y_n)$. Note that due to the possible diagonalization these are on transformed scale. Only for Gaussian models.

V_eps
: Diagonal elements of $Var(\epsilon_t|y_1, \ldots, y_n)$. Note that due to the diagonalization the off-diagonal elements are zero. Only for Gaussian models. Note that for computing auxiliary residuals you shoud use method `rstandard.KFS`.

iterations
: The number of iterations used in linearization of non-Gaussian model.

v
: Prediction errors $v_{t,i} = y_{t,i} - Z_{i,t}a_{t,i}, i = 1, \ldots, p$, where

$$a_{t,i} = E(\alpha_t|y_{t,i-1}, \ldots, y_{t,1}, \ldots, y_{1,1})$$

. Only returned for Gaussian models.

F
: Prediction error variances $Var(v_{t,i})$. Only returned for Gaussian models.

Finf
: Diffuse part of prediction error variances. Only returned for Gaussian models.

d
: The last time index of diffuse phase, i.e. the non-diffuse phase began at time $d + 1$. Only returned for Gaussian models.

j
: The last observation index $i$ of $y_{i,t}$ of the diffuse phase. Only returned for Gaussian models.

In addition, if argument `simplify = FALSE`, list contains following components:

K
: Covariances $Cov(\alpha_{t,i}, y_{t,i}|y_{t,i-1}, \ldots, y_{t,1}, y_{t-1}, \ldots, y_1), \quad i = 1, \ldots, p$.

Kinf
: Diffuse part of $K_t$.

r
: Weighted sums of innovations $v_{t+1}, \ldots, v_n$. Notice that in literature $t$ in $r_t$ goes from $0, \ldots, n$. Here $t = 1, \ldots, n + 1$. Same applies to all $r$ and $N$ variables.

r0, r1
: Diffuse phase decomposition of $r_t$.

N
: Covariances $Var(r_t)$.

N0, N1, N2
: Diffuse phase decomposition of $N_t$.

## References


Koopman, S.J. and Durbin J. (2000). Fast filtering and smoothing for non-stationary time series models, Journal of American Statistical Assosiation, 92, 1630-38.

Koopman, S.J. and Durbin J. (2001). Time Series Analysis by State Space Methods. Oxford: Oxford University Press.

Koopman, S.J. and Durbin J. (2003). Filtering and smoothing of state vector for diffuse state space models, Journal of Time Series Analysis, Vol. 24, No. 1.

### See Also

KFAS for examples

logLik, KFAS, fitSSM, boat, sexratio, GlobalTemp, SSModel, importanceSSM, approxSSM for examples.

### Examples

```
set.seed(1)
x <- cumsum(rnorm(100, 0, 0.1))
y <- rnorm(100, x, 0.1)
model <- SSModel(y ~ SSMtrend(1, Q = 0.01), H = 0.01)
out <- KFS(model)
ts.plot(ts(x), out$a, out$att, out$alpha, col = 1:4)
```

---

## ldl *LDL Decomposition of a Matrix*

---

### Description

Function `ldl` computes the LDL decomposition of a positive semidefinite matrix.

### Usage

```
ldl(x, tol)
```

### Arguments

| | |
|---|---|
| `x` | Symmetrix matrix. |
| `tol` | Tolerance parameter for LDL decomposition, determines which diagonal values are counted as zero. Same value is used in isSymmetric function. Default is `max(100, max(abs(diag(as.matrix(x))))) * .Machine$double.eps`. |

### Value

Transformed matrix with D in diagonal, L in strictly lower diagonal and zeros on upper diagonal.

### Examples

```
# Positive semidefinite matrix, example matrix taken from ?chol
x <- matrix(c(1:5, (1:5)^2), 5, 2)
x <- cbind(x, x[, 1] + 3*x[, 2])
m <- crossprod(x)
l <- ldl(m, tol = 1e-8) # arm64 Mac setup in CRAN fails with default tolerance
d <- diag(diag(l))
diag(l) <- 1
all.equal(l %*% d %*% t(l), m, tol = 1e-15)
```

`logLik.SSModel` *Log-likelihood of the State Space Model.*

### Description

Function `logLik.SSmodel` computes the log-likelihood value of a state space model.

### Usage

```
## S3 method for class 'SSModel'
logLik(
  object,
  marginal = FALSE,
  nsim = 0,
  antithetics = TRUE,
  theta,
  check.model = TRUE,
  transform = c("ldl", "augment"),
  maxiter = 50,
  seed,
  convtol = 1e-08,
  expected = FALSE,
  H_tol = 1e+15,
  transform_tol,
  ...
)
```

### Arguments

| | |
|---|---|
| `object` | State space model of class `SSModel`. |
| `marginal` | Logical. Compute marginal instead of diffuse likelihood (see details). Default is `FALSE`. |
| `nsim` | Number of independent samples used in estimating the log-likelihood of the non-Gaussian state space model. Default is 0, which gives good starting value for optimization. Only used for non-Gaussian model. |
| `antithetics` | Logical. If TRUE, two antithetic variables are used in simulations, one for location and another for scale. Default is TRUE. Only used for non-Gaussian model. |
| `theta` | Initial values for conditional mode theta. Only used for non-Gaussian model. |
| `check.model` | Logical. If TRUE, function `is.SSModel` is called before computing the likelihood. Default is `TRUE`. |
| `transform` | How to transform the model in case of non-diagonal covariance matrix $H$. Defaults to `"ldl"`. See function `transformSSM` for details. |
| `maxiter` | The maximum number of iterations used in linearisation. Default is 50. Only used for non-Gaussian model. |

| | |
|---|---|
| seed | The value is used as a seed via `set.seed` function. Only used for non-Gaussian model. |
| convtol | Tolerance parameter for convergence checks for Gaussian approximation. Only used for non-Gaussian model. |
| expected | Logical value defining the approximation of H_t in case of Gamma and negative binomial distribution. Default is `FALSE` which matches the algorithm of Durbin & Koopman (1997), whereas `TRUE` uses the expected value of observations in the equations, leading to results which match with `glm` (where applicable). The latter case was the default behaviour of KFAS before version 1.3.8. Essentially this is the difference between observed and expected information. Only used for non-Gaussian model. |
| H_tol | Tolerance parameter for check `max(H) > tol_H`, which suggests that the approximation converged to degenerate case with near zero signal-to-noise ratio. Default is very generous 1e15. Only used for non-Gaussian model. |
| transform_tol | Tolerance parameter for LDL decomposition in case of a non-diagonal H and `transform = "ldl"`. See `transformSSM` and `ldl` for details. |
| ... | Ignored. |

## Details

As KFAS is based on diffuse initialization, the log-likelihood is also diffuse, which coincides with restricted likelihood (REML) in an appropriate (mixed) models. However, in typical REML estimation constant term $log|X'X|$ is omitted from the log-likelihood formula. Similar term is also missing in diffuse log-likelihood formulations of state space models, but unlike in simpler linear models this term is not necessarily constant. Therefore omitting this term can lead to suboptimal results in model estimation if there is unknown parameters in diffuse parts of Zt or Tt (Francke et al. 2011). Therefore so called marginal log-likelihood (diffuse likelihood + extra term) is recommended. See also Gurka (2006) for model comparison in mixed model settings using REML with and without the additional (constant) term. The marginal log-likelihood can be computed by setting `marginal = TRUE`.

Note that for non-Gaussian models with importance sampling derivative-free optimization methods such as Nelder-Mead might be more reliable than methods which use finite difference approximations. This is due to noise caused by the relative stopping criterion used for finding approximating Gaussian model.

## Value

Log-likelihood of the model.

## References

Francke, M. K., Koopman, S. J. and De Vos, A. F. (2010), Likelihood functions for state space models with diffuse initial conditions. Journal of Time Series Analysis, 31: 407–414.

Gurka, M. J (2006), Selecting the Best Linear Mixed Model Under REML. The American Statistician, Vol. 60.

Casals, J., Sotoca, S., Jerez, M. (2014), Minimally conditioned likelihood for a nonstationary state space model. Mathematics and Computers in Simulation, Vol. 100.

## Examples

```
# Example of estimating AR model with covariates, and how to deal with possible
# non-stationarity in optimization.

set.seed(1)
x <- rnorm(100)
y <- 2 * x + arima.sim(n = 100, model = list(ar = c(0.5, -0.3)))

model<- SSModel(y ~ SSMarima(ar = c(0.5, -0.3),  Q = 1) + x, H = 0)

obj <- function(pars, model, estimate = TRUE) {
  #guard against stationarity
  armamod <- try(SSMarima(ar = artransform(pars[1:2]), Q = exp(pars[3])), silent = TRUE)
  if(class(armamod) == "try-error") {
    return(Inf)
  } else {
    # use advanced subsetting method for SSModels, see ?`[.SSModel`
    model["T", states = "arima"] <- armamod$T
    model["Q", eta = "arima"]  <- armamod$Q
    model["P1", states = "arima"]  <- armamod$P1
    if(estimate) {
      -logLik(model)
    } else {
      model
    }
  }
}
fit <- optim(p = c(0.5,-0.5,1), fn = obj, model = model, method ="BFGS")

model <- obj(fit$par, model, FALSE)
model$T
model$Q
coef(KFS(model), last = TRUE)
```

---

mvInnovations *Multivariate Innovations*

---

## Description

Function `mvInnovations` computes the multivariate versions of one step-ahead prediction errors and their variances using the output of KFS.

## Usage

```
mvInnovations(x)
```

### Arguments

| | |
|---|---|
| x | Object of class KFS. |

### Value

| | |
|---|---|
| v | Multivariate prediction errors $v_t = y_t - Z_t a_t$ |
| F | Prediction error variances $Var(v_t)$. |
| Finf | Diffuse part of $F_t$. |

### Examples

```
# Compute the filtered estimates based on the KFS output

filtered <- function(x) {
  innov <- mvInnovations(x)
  att <- window(x$a, end = end(x$a) - 1)
  tvz <- attr(x$model,"tv")[1]

  for (i in 1:nrow(att)) {
    att[i,] <- att[i,] +
      x$P[,,i] %*%
      t(solve(innov$F[,,i], x$model$Z[, , tvz * (i - 1) + 1, drop = FALSE])) %*%
      innov$v[i, ]
  }
  att
}
```

---

plot.SSModel *Diagnostic Plots of State Space Models*

---

### Description

Diagnostic plots based on standardized residuals for objects of class SSModel.

### Usage

```
## S3 method for class 'SSModel'
plot(x, nsim = 0, zerotol = 0, expected = FALSE, ...)
```

### Arguments

| | |
|---|---|
| x | Object of class SSModel. |
| nsim | The number of independent samples used in importance sampling. Only used for non-Gaussian model. Default is 0, which computes the approximating Gaussian model by approxSSM and performs the usual Gaussian filtering/smoothing so that the smoothed state estimates equals to the conditional mode of $p(\alpha_t\|y)$. In case of nsim = 0, the mean estimates and their variances are computed using the Delta method (ignoring the covariance terms). |

`zerotol` Tolerance parameter for positivity checking in standardization. Default is zero. The values of D <= zerotol * max(D, 0) are deemed to zero.

`expected` Logical value defining the approximation of H_t in case of Gamma and negative binomial distribution. Default is `FALSE` which matches the algorithm of Durbin & Koopman (1997), whereas `TRUE` uses the expected value of observations in the equations, leading to results which match with `glm` (where applicable). The latter case was the default behaviour of KFAS before version 1.3.8. Essentially this is the difference between observed and expected information.

`...` Ignored.

## Examples

```
modelNile <- SSModel(Nile ~ SSMtrend(1, Q = list(matrix(NA))), H = matrix(NA))
modelNile <- fitSSM(inits = c(log(var(Nile)),log(var(Nile))), model = modelNile,
 method = "BFGS")$model

if (interactive()) {
  plot(modelNile)
}
```

---

`predict.SSModel` *State Space Model Predictions*

---

## Description

Function `predict.SSModel` predicts the future observations of a state space model of class `SSModel`.

## Usage

```
## S3 method for class 'SSModel'
predict(
  object,
  newdata,
  n.ahead,
  interval = c("none", "confidence", "prediction"),
  level = 0.95,
  type = c("response", "link"),
  states = NULL,
  se.fit = FALSE,
  nsim = 0,
  prob = TRUE,
  maxiter = 50,
  filtered = FALSE,
  expected = FALSE,
  ...
)
```

## Arguments

| | |
|---|---|
| object | Object of class SSModel. |
| newdata | A compatible SSModel object to be added in the end of the old object for which the predictions are required. If omitted, predictions are either for the past data points, or if argument n.ahead is given, n.ahead time steps ahead. |
| n.ahead | Number of steps ahead at which to predict. Only used if newdata is omitted. Note that when using n.ahead, object cannot contain time varying system matrices. |
| interval | Type of interval calculation. |
| level | Confidence level for intervals. |
| type | Scale of the prediction, "response" or "link". |
| states | Which states are used in computing the predictions. Either a numeric vector containing the indices of the corresponding states, or a character vector defining the types of the corresponding states. Possible choices are "all", "level", "slope" (which does not make sense as the corresponding Z is zero.), "trend", "regression", "arima", "custom", "cycle" or "seasonal", where "trend" extracts all states relating to trend. These can be combined. Default is "all". |
| se.fit | If TRUE, standard errors of fitted values are computed. Default is FALSE. |
| nsim | Number of independent samples used in importance sampling. Used only for non-Gaussian models. |
| prob | if TRUE (default), the predictions in binomial case are probabilities instead of counts. |
| maxiter | The maximum number of iterations used in approximation Default is 50. Only used for non-Gaussian model. |
| filtered | If TRUE, compute predictions based on filtered (one-step-ahead) estimates. Default is FALSE i.e. predictions are based on all available observations given by user. For diffuse phase, interval bounds and standard errors of fitted values are set to -Inf/Inf (If the interest is in the first time points it might be useful to use non-exact diffuse initialization.). |
| expected | Logical value defining the approximation of H_t in case of Gamma and negative binomial distribution. Default is FALSE which matches the algorithm of Durbin & Koopman (1997), whereas TRUE uses the expected value of observations in the equations, leading to results which match with glm (where applicable). The latter case was the default behaviour of KFAS before version 1.3.8. Essentially this is the difference between observed and expected information in GLM context. |
| ... | Ignored. |

## Details

For non-Gaussian models, the results depend whether importance sampling is used (nsim>0). without simulations, the confidence intervals are based on the Gaussian approximation of $p(\alpha|y)$. Confidence intervals in response scale are computed in linear predictor scale, and then transformed to response scale. The prediction intervals are not supported. With importance sampling, the confidence intervals are computed as the empirical quantiles from the weighted sample, whereas the prediction intervals contain additional step of simulating the response variables from the sampling distribution $p(y|\theta^i)$.

Predictions take account the uncertainty in state estimation (given the prior distribution for the initial states), but not the uncertainty of estimating the parameters in the system matrices (i.e. $Z$, $Q$ etc.). Thus the obtained confidence/prediction intervals can underestimate the true uncertainty for short time series and/or complex models.

If no simulations are used, the standard errors in response scale are computed using the Delta method.

## Value

A matrix or list of matrices containing the predictions, and optionally standard errors.

## Examples

```
set.seed(1)
x <- runif(n=100,min=1,max=3)
y <- rpois(n=100,lambda=exp(x-1))
model <- SSModel(y~x,distribution="poisson")
xnew <- seq(0.5,3.5,by=0.1)
newdata <- SSModel(rep(NA,length(xnew))~xnew,distribution="poisson")
pred <- predict(model,newdata=newdata,interval="prediction",level=0.9,nsim=100)
plot(x=x,y=y,pch=19,ylim=c(0,25),xlim=c(0.5,3.5))
matlines(x=xnew,y=pred,col=c(2,2,2),lty=c(1,2,2),type="l")

model <- SSModel(Nile~SSMtrend(1,Q=1469),H=15099)
pred <- predict(model,n.ahead=10,interval="prediction",level=0.9)
pred
```

---

# print.KFS — *Print Ouput of Kalman Filter and Smoother*

---

## Description

Print Ouput of Kalman Filter and Smoother

## Usage

```
## S3 method for class 'KFS'
print(x, type = "state", digits = max(3L, getOption("digits") - 3L), ...)
```

## Arguments

| | |
|---|---|
| `x` | output object from function KFS. |
| `type` | What to print. Possible values are `"state"` (default), `"signal"`, and `"mean"`. Multiple choices are allowed. |
| `digits` | minimum number of digits to be printed. |
| `...` | Ignored. |

`print.SSModel` *Print SSModel Object*

### Description

Print SSModel Object

### Usage

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

### Arguments

| | |
|---|---|
| `x` | SSModel object |
| `...` | Ignored. |

`rename_states` *Rename the States of SSModel Object*

### Description

A simple function for renaming the states of `SSModel` object. Note that since KFAS version 1.2.3 the auxiliary functions such as `SSMtrend` have argument `state_names` which can be used to overwrite the default state names when building the model with `SSModel`.

### Usage

```
rename_states(model, state_names)
```

### Arguments

| | |
|---|---|
| `model` | Object of class SSModel |
| `state_names` | Character vector giving new names for the states. |

### Value

Original model with dimnames corresponding to states renamed.

### Examples

```
custom_model <- SSModel(1:10 ~ -1 +
SSMcustom(Z = 1, T = 1, R = 1, Q = 1, P1inf = 1), H = 1)
custom_model <- rename_states(custom_model, "level")
ll_model <- SSModel(1:10 ~ SSMtrend(1, Q = 1), H = 1)
test_these <- c("y", "Z", "H", "T", "R", "Q", "a1", "P1", "P1inf")
identical(custom_model[test_these], ll_model[test_these])
```

residuals.KFS *Extract Residuals of KFS output*

## Description

Extract Residuals of KFS output

## Usage

```
## S3 method for class 'KFS'
residuals(object, type = c("recursive", "pearson", "response", "state"), ...)
```

## Arguments

| | |
|---|---|
| `object` | KFS object |
| `type` | Character string defining the type of residuals. |
| `...` | Ignored. |

## Details

For object of class KFS, several types of residuals can be computed:

- `"recursive"`: One-step-ahead prediction residuals $v_{t,i} = y_{t,i} - Z_{t,i}a_{t,i}$. For non-Gaussian case recursive residuals are computed as $y_t - f(Z_t a_t)$, i.e. non-sequentially. Computing recursive residuals for large non-Gaussian models can be time consuming as filtering is needed.
- `"pearson"`:
$$(y_{t,i} - \theta_{t,i})/\sqrt{V(\mu_{t,i})}, \quad i = 1, \dots, p, t = 1, \dots, n,$$
where $V(\mu_{t,i})$ is the variance function of the series $i$
- `"response"`: Data minus fitted values, $y - E(y)$.
- `"state"`: Residuals based on the smoothed disturbance terms $\eta$ are defined as
$$\hat{\eta}_t, \quad t = 1, \dots, n.$$
Only defined for fully Gaussian models.

| rstandard.KFS | *Extract Standardized Residuals from KFS output* |
|---|---|

## Description

Extract Standardized Residuals from KFS output

## Usage

```
## S3 method for class 'KFS'
rstandard(
  model,
  type = c("recursive", "pearson", "state"),
  standardization_type = c("marginal", "cholesky"),
  zerotol = 0,
  expected = FALSE,
  ...
)
```

## Arguments

| | |
|---|---|
| `model` | KFS object |
| `type` | Type of residuals. See details. |
| `standardization_type` | Type of standardization. Either `"marginal"` (default) for marginal standardization or `"cholesky"` for Cholesky-type standardization. |
| `zerotol` | Tolerance parameter for positivity checking in standardization. Default is zero. The values of D <= zerotol * max(D, 0) are deemed to zero. |
| `expected` | Logical value defining the approximation of H_t in case of Gamma and negative binomial distribution. Default is `FALSE` which matches the algorithm of Durbin & Koopman (1997), whereas `TRUE` uses the expected value of observations in the equations, leading to results which match with `glm` (where applicable). The latter case was the default behaviour of KFAS before version 1.3.8. Essentially this is the difference between observed and expected information. |
| `...` | Ignored. |

## Details

For object of class KFS with fully Gaussian observations, several types of standardized residuals can be computed. Also the standardization for multivariate residuals can be done either by Cholesky decomposition $L_t^{-1} residual_t$, or component-wise $residual_t / sd(residual_t)$,.

- "recursive": For Gaussian models the vector standardized one-step-ahead prediction residuals are defined as

$$v_{t,i}/\sqrt{F_{i,t}},$$

with residuals being undefined in diffuse phase. Note that even in multivariate case these standardized residuals coincide with the ones obtained from the Kalman filter without the sequential processing (which is not true for the non-standardized innovations). For non-Gaussian models the vector standardized recursive residuals are obtained as

$$L_t^{-1}(y_t - \mu_t),$$

where $L_t$ is the lower triangular matrix from Cholesky decomposition of $Var(y_t|y_{t-1},\dots,y_1)$. Computing these for large non-Gaussian models can be time consuming as filtering is needed.

For Gaussian models the component-wise standardized one-step-ahead prediction residuals are defined as

$$v_t/\sqrt{diag(F_t)},$$

where $v_t$ and $F_t$ are based on the standard multivariate processing. For non-Gaussian models these are obtained as

$$(y_t - \mu_t)/\sqrt{diag(F_t)},$$

where $F_t = Var(y_t|y_{t-1},\dots,y_1)$.

- "state": Residuals based on the smoothed state disturbance terms $\eta$ are defined as

$$L_t^{-1}\hat{\eta}_t, \quad t = 1,\dots,n,$$

  where $L_t$ is either the lower triangular matrix from Cholesky decomposition of $Var(\hat{\eta}_t) = Q_t - V_{\eta,t}$, or the diagonal of the same matrix.

- "pearson": Standardized Pearson residuals

$$L_t^{-1}(y_t - \theta_i), \quad t = 1,\dots,n,$$

  where $L_t$ is the lower triangular matrix from Cholesky decomposition of $Var(\hat{\mu}_t) = H_t - V_{\mu,t}$, or the diagonal of the same matrix. For Gaussian models, these coincide with the standardized smoothed $\epsilon$ disturbance residuals (as $V_{\mu,t} = V_{\epsilon,t}$), and for generalized linear models these coincide with the standardized Pearson residuals (hence the name).

Note that the variance estimates from `KFS` are of form Var(x | y), e.g., `V_eps` from `KFS` is $Var(\epsilon_t|Y)$ and matches with equation 4.69 in Section 4.5.3 of Durbin and Koopman (2012). (in case of univariate Gaussian model). But for the standardization we need Var(E(x | y)) (e.g., Var(`epshat`) which we get with the law of total variance as $H_t - V_eps$ for example.

## Examples

```
# Replication of residual plot of Section 8.2 of Durbin and Koopman (2012)
model <- SSModel(log(drivers) ~ SSMtrend(1, Q = list(NA))+
    SSMseasonal(period = 12, sea.type = "trigonometric", Q = NA),
  data = Seatbelts, H = NA)

updatefn <- function(pars, model){
  model$H[] <- exp(pars[1])
  diag(model$Q[, , 1]) <- exp(c(pars[2], rep(pars[3], 11)))
  model
}

fit <- fitSSM(model = model,
```

```
  inits = log(c(var(log(Seatbelts[, "drivers"])), 0.001, 0.0001)),
  updatefn = updatefn)

# tiny differences due to different optimization algorithms
setNames(c(diag(fit$model$Q[,,1])[1:2], fit$model$H[1]),
  c("level", "seasonal", "irregular"))

out <- KFS(fit$model, smoothing = c("state", "mean", "disturbance"))

plot(cbind(
  recursive = rstandard(out),
  irregular = rstandard(out, "pearson"),
  state = rstandard(out, "state")[,1]),
  main = "recursive and state residuals", type = "h")

# Figure 2.8 in DK2012
model_Nile <- SSModel(Nile ~
    SSMtrend(1, Q = list(matrix(NA))), H = matrix(NA))
model_Nile <- fitSSM(model_Nile, c(log(var(Nile)), log(var(Nile))),
  method = "BFGS")$model

out_Nile <- KFS(model_Nile,  smoothing = c("state", "mean", "disturbance"))

par(mfrow = c(2, 2))
res_p <- rstandard(out_Nile, "pearson")
ts.plot(res_p)
abline(a = 0, b= 0, lty = 2)
hist(res_p, freq = FALSE)
lines(density(res_p))
res_s <- rstandard(out_Nile, "state")
ts.plot(res_s)
abline(a = 0, b= 0, lty = 2)
hist(res_s, freq = FALSE)
lines(density(res_s))
```

---

## sexratio — *Number of males and females born in Finland from 1751 to 2011*

---

### Description

A time series object containing the number of males and females born in Finland from 1751 to 2011.

### Format

A time series object containing the number of males and females born in Finland from 1751 to 2011.

## Source

Statistics Finland https://statfin.stat.fi/PxWeb/pxweb/en/StatFin/.

## Examples

```
data("sexratio")
model <- SSModel(Male ~ SSMtrend(1, Q = NA), u = sexratio[, "Total"],
  data = sexratio, distribution = "binomial")
fit <- fitSSM(model, inits = -15, method = "BFGS")
fit$model["Q"]

# Computing confidence intervals in response scale
# Uses importance sampling on response scale (400 samples with antithetics)

pred <- predict(fit$model, type = "response", interval = "conf", nsim = 100)

ts.plot(cbind(model$y/model$u, pred), col = c(1, 2, 3, 3), lty = c(1, 1, 2, 2))

## Not run:
# Now with sex ratio instead of the probabilities:
imp <- importanceSSM(fit$model, nsim = 1000, antithetics = TRUE)
sexratio.smooth <- numeric(length(model$y))
sexratio.ci <- matrix(0, length(model$y), 2)
w <- imp$w/sum(imp$w)
for(i in 1:length(model$y)){
 sexr <- exp(imp$sample[i, 1, ])
 sexratio.smooth[i] <- sum(sexr*w)
 oo <- order(sexr)
 sexratio.ci[i, ] <- c(sexr[oo][which.min(abs(cumsum(w[oo]) - 0.05))],
                      sexr[oo][which.min(abs(cumsum(w[oo]) - 0.95))])
}

# Same by direct transformation:
out <- KFS(fit$model, smoothing = "signal", nsim = 1000)
sexratio.smooth2 <- exp(out$thetahat)
sexratio.ci2 <- exp(c(out$thetahat) + qnorm(0.025) *
  sqrt(drop(out$V_theta))%o%c(1, -1))

ts.plot(cbind(sexratio.smooth, sexratio.ci, sexratio.smooth2, sexratio.ci2),
        col = c(1, 1, 1, 2, 2, 2), lty = c(1, 2, 2, 1, 2, 2))

## End(Not run)
```

---

signal *Extracting the Partial Signal Of a State Space Model*

---

## Description

Function signal returns the signal of a state space model using only subset of states.

## Usage

```
signal(object, states = "all", filtered = FALSE)
```

## Arguments

| | |
|---|---|
| `object` | Object of class KFS. |
| `states` | Which states are combined? Either a numeric vector containing the indices of the corresponding states, or a character vector defining the types of the corresponding states. Possible choices are "all", "level", "slope", "trend", "regression", "arima", "custom", "cycle" or "seasonal", where "trend" extracts states relating to trend. These can be combined. Default is "all". |
| `filtered` | If TRUE, filtered signal is used. Otherwise smoothed signal is used. |

## Value

| | |
|---|---|
| `signal` | Time series object of filtered signal $Z_t a_t$ or smoothed signal $Z_t \hat{\alpha}_t$ using only the defined states. |
| `variance` | Cov($Z_t a_t$) or Cov($Z_t \hat{\alpha}_t$) using only the defined states. For the covariance matrices of the filtered signal, only the non-diffuse part of P is used. |

## Examples

```
model <- SSModel(log(drivers) ~ SSMtrend(1, NA) +
    SSMseasonal(12, sea.type = 'trigonometric', Q = NA) +
    log(PetrolPrice) + law,data = Seatbelts, H = NA)

ownupdatefn <- function(pars,model,...){
  model$H[] <- exp(pars[1])
  diag(model$Q[,,1]) <- exp(c(pars[2], rep(pars[3], 11)))
  model
}

fit <- fitSSM(inits = log(c(var(log(Seatbelts[,'drivers'])), 0.001, 0.0001)),
  model = model, updatefn = ownupdatefn, method = 'BFGS')

out <- KFS(fit$model, smoothing = c('state', 'mean'))
ts.plot(cbind(out$model$y, fitted(out)),lty = 1:2, col = 1:2,
  main = 'Observations and smoothed signal with and without seasonal component')
lines(signal(out, states = c('regression', 'trend'))$signal, col = 4, lty = 1)
legend('bottomleft',
  legend = c('Observations', 'Smoothed signal','Smoothed level'),
  col = c(1, 2, 4), lty = c(1, 2, 1))
```

simulateSSM *Simulation of a Gaussian State Space Model*

## Description

Function simulateSMM simulates states, signals, disturbances or missing observations of the Gaussian state space model either conditional on the data (simulation smoother) or unconditionally.

## Usage

```
simulateSSM(
  object,
  type = c("states", "signals", "disturbances", "observations", "epsilon", "eta"),
  filtered = FALSE,
  nsim = 1,
  antithetics = FALSE,
  conditional = TRUE
)
```

## Arguments

| | |
|---|---|
| object | Gaussian state space object of class SSModel. |
| type | What to simulate. |
| filtered | Simulate from $p(\alpha_t \| y_{t-1}, ..., y_1)$ instead of $p(\alpha \| y)$. |
| nsim | Number of independent samples. Default is 1. |
| antithetics | Use antithetic variables in simulation. Default is FALSE. |
| conditional | Simulations are conditional to data. If FALSE, the states having exact diffuse initial distribution (as defined by P1inf are fixed to corresponding values of a1. See details. |

## Details

Simulation smoother algorithm is based on article by J. Durbin and S.J. Koopman (2002). The simulation filter (filtered = TRUE) is a straightforward modification of the simulations smoother, where only filtering steps are performed.

Function can use two antithetic variables, one for location and other for scale, so output contains four blocks of simulated values which correlate which each other (ith block correlates negatively with (i+1)th block, and positively with (i+2)th block etc.).

Note that KFAS versions 1.2.0 and older, for unconditional simulation the initial distribution of states was fixed so that a1 was set to the smoothed estimates of the first state and the initial variance was set to zero. Now original a1 and P1 are used, and P1inf is ignored (i.e. diffuse states are fixed to corresponding elements of a1).

## Value

An n x k x nsim array containing the simulated series, where k is number of observations, signals, states or disturbances.

## References

Durbin J. and Koopman, S.J. (2002). A simple and efficient simulation smoother for state space time series analysis, Biometrika, Volume 89, Issue 3

## Examples

```
set.seed(123)
# simulate new observations from the "fitted" model
model <- SSModel(Nile ~ SSMtrend(1, Q = 1469), H = 15099)
# signal conditional on the data i.e. samples from p(theta | y)
# unconditional simulation is not reasonable as the model is nonstationary
signal_sim <- simulateSSM(model, type = "signals", nsim = 10)
# and add unconditional noise term i.e samples from p(epsilon)
epsilon_sim <- simulateSSM(model, type = "epsilon", nsim = 10,
  conditional = FALSE)
observation_sim <- signal_sim + epsilon_sim

ts.plot(observation_sim[,1,], Nile, col = c(rep(2, 10), 1),
  lty = c(rep(2, 10), 1), lwd = c(rep(1, 10), 2))

# fully unconditional simulation:
observation_sim2 <- simulateSSM(model, type = "observations", nsim = 10,
  conditional = FALSE)
ts.plot(observation_sim[,1,], observation_sim2[,1,], Nile,
col = c(rep(2:3, each = 10), 1), lty = c(rep(2, 20), 1),
lwd = c(rep(1, 20), 2))

# illustrating use of antithetics
model <- SSModel(matrix(NA, 100, 1) ~ SSMtrend(1, 1, P1inf = 0), H = 1)

set.seed(123)
sim <- simulateSSM(model, "obs", nsim = 2, antithetics = TRUE)
# first time points
sim[1,,]
# correlation structure between simulations with two antithetics
cor(sim[,1,])

out_NA <- KFS(model, filtering = "none", smoothing = "state")
model["y"] <- sim[, 1, 1]
out_obs <- KFS(model, filtering = "none", smoothing = "state")

set.seed(40216)
# simulate states from the p(alpha | y)
sim_conditional <- simulateSSM(model, nsim = 10, antithetics = TRUE)

# mean of the simulated states is exactly correct due to antithetic variables
mean(sim_conditional[2, 1, ])
```

```
out_obs$alpha[2]
# for variances more simulations are needed
var(sim_conditional[2, 1, ])
out_obs$V[2]

set.seed(40216)
# no data, simulations from p(alpha)
sim_unconditional <- simulateSSM(model, nsim = 10, antithetics = TRUE,
  conditional = FALSE)
mean(sim_unconditional[2, 1, ])
out_NA$alpha[2]
var(sim_unconditional[2, 1, ])
out_NA$V[2]

ts.plot(cbind(sim_conditional[,1,1:5], sim_unconditional[,1,1:5]),
  col = rep(c(2,4), each = 5))
lines(out_obs$alpha, lwd=2)
```

---

SSMarima *Create a State Space Model Object of Class SSModel*

---

## Description

Function SSModel creates a state space object object of class SSModel which can be used as an input object for various functions of KFAS package.

## Usage

```
SSMarima(
  ar = NULL,
  ma = NULL,
  d = 0,
  Q,
  stationary = TRUE,
  index,
  n = 1,
  state_names = NULL,
  ynames
)

SSMcustom(Z, T, R, Q, a1, P1, P1inf, index, n = 1, state_names = NULL)

SSMcycle(
  period,
  Q,
  type,
  index,
```

```
  a1,
  P1,
  P1inf,
  damping = 1,
  n = 1,
  state_names = NULL,
  ynames
)

SSModel(formula, data, H, u, distribution, tol = .Machine$double.eps^0.5)

SSMregression(
  rformula,
  data,
  type,
  Q,
  index,
  R,
  a1,
  P1,
  P1inf,
  n = 1,
  remove.intercept = TRUE,
  state_names = NULL,
  ynames
)

SSMseasonal(
  period,
  Q,
  sea.type = c("dummy", "trigonometric"),
  type,
  index,
  a1,
  P1,
  P1inf,
  n = 1,
  state_names = NULL,
  ynames,
  harmonics
)

SSMtrend(
  degree = 1,
  Q,
  type,
  index,
  a1,
```

```
  P1,
  P1inf,
  n = 1,
  state_names = NULL,
  ynames
)
```

### Arguments

| | |
|---|---|
| ar | For arima component, a numeric vector containing the autoregressive coeffients. |
| ma | For arima component, a numericvector containing the moving average coeffients. |
| d | For arima component, a degree of differencing. |
| Q | For arima, cycle and seasonal component, a $p \times p$ covariance matrix of the disturbances (or in the time varying case $p \times p \times n$ array), where where $p$ = `length(index)`. For trend component, list of length `degree` containing the $p \times p$ or $p \times p \times n$ covariance matrices. For a custom component, arbitrary covariance matrix or array of disturbance terms $\eta_t$ |
| stationary | For arima component, logical value indicating whether a stationarity of the arima part is assumed. Defaults to TRUE. |
| index | A vector indicating for which series the corresponding components are constructed. |
| n | Length of the series, only used internally for dimensionality check. |
| state_names | A character vector giving the state names. |
| ynames | names of the times series, used internally. |
| Z | For a custom component, system matrix or array of observation equation. |
| T | For a custom component, system matrix or array of transition equation. |
| R | For a custom and regression components, optional $m \times k$ system matrix or array of transition equation. |
| a1 | Optional $m \times 1$ matrix giving the expected value of the initial state vector $\alpha_1$. |
| P1 | Optional $m \times m$ matrix giving the covariance matrix of $\alpha_1$. In the diffuse case the non-diffuse part of $P_1$. |
| P1inf | Optional $m \times m$ matrix giving the diffuse part of $P_1$. Diagonal matrix with ones on diagonal elements which correspond to the diffuse initial states. If `P1inf[i,i]>0`, corresponding row and column of `P1` should be zero. |
| period | For a cycle and seasonal components, the length of the cycle/seasonal pattern. |
| type | For cycle, seasonal, trend and regression components, character string defining if `"distinct"` or `"common"` states are used for different series. |
| damping | A damping factor for cycle component. Defaults to 1. Note that there are no checks for the range of the factor. |
| formula | An object of class `formula` containing the symbolic description of the model. The intercept term can be removed with `-1` as in `lm`. In case of trend or differenced arima component the intercept is removed automatically in order to keep the model identifiable. See package vignette and examples in `KFAS` for special functions used in model construction. |

data
: An optional data frame, list or environment containing the variables in the model.

H
: Covariance matrix or array of disturbance terms $\epsilon_t$ of observation equation. Defaults to an identity matrix. Omitted in case of non-Gaussian distributions (augment the state vector if you want to add additional noise).

u
: Additional parameters for non-Gaussian models. See details in KFAS.

distribution
: A vector of distributions of the observations. Default is `rep("gaussian", p)`, where `p` is the number of series.

tol
: A tolerance parameter used in checking whether `Finf` or `F` is numerically zero. Defaults to `.Machine$double.eps^0.5`. If `F < tol * max(abs(Z[Z > 0]))^2`, then F is deemed to be zero (i.e. differences are due to numerical precision). This has mostly effect only on determining when to end exact diffuse phase. Tweaking this and/or scaling model parameters/observations can sometimes help with numerical issues.

rformula
: For regression component, right hand side formula or list of of such formulas defining the custom regression part.

remove.intercept
: Remove intercept term from regression model. Default is `TRUE`. This tries to ensure that there are no extra intercept terms in the model.

sea.type
: For seasonal component, character string defining whether to use `"dummy"` or `"trigonometric"` form of the seasonal component.

harmonics
: For univariate trigonometric seasonal, argument `harmonics` can be used to specify which subharmonics are added to the model. Note that for multivariate model you can call `SSMseasonal` multiple times with different values of `index`.

degree
: For trend component, integer defining the degree of the polynomial trend. 1 corresponds to local level, 2 for local linear trend and so forth.

## Details

Formula of the model can contain the usual regression part and additional functions defining different types of components of the model, named as `SSMarima`, `SSMcustom`, `SSMcycle`, `SSMregression`, `SSMseasonal` and `SSMtrend`.

For more details, see package vignette (the mathematical notation is somewhat non-readable in ASCII).

## Value

Object of class `SSModel`, which is a list with the following components:

y
: A n x p matrix containing the observations.

Z
: A p x m x 1 or p x m x n array corresponding to the system matrix of observation equation.

H
: A p x p x 1 or p x p x n array corresponding to the covariance matrix of observational disturbances epsilon.

T
: A m x m x 1 or m x m x n array corresponding to the first system matrix of state equation.

R
: A m x k x 1 or m x k x n array corresponding to the second system matrix of state equation.

Q
: A k x k x 1 or k x k x n array corresponding to the covariance matrix of state disturbances eta

a1
: A m x 1 matrix containing the expected values of the initial states.

P1
: A m x m matrix containing the covariance matrix of the nondiffuse part of the initial state vector.

P1inf
: A m x m matrix containing the covariance matrix of the diffuse part of the initial state vector. If P1[i,i] is non-zero then P1inf[i,i] is automatically set to zero.

u
: A n x p matrix of an additional parameters in case of non-Gaussian model.

distribution
: A vector of length p giving the distributions of the observations.

tol
: A tolerance parameter for Kalman filtering.

call
: Original call to the function.

In addition, object of class SSModel contains following attributes:

names
: Names of the list components.

p, m, k, n
: Integer valued scalars defining the dimensions of the model components.

state_types
: Types of the states in the model.

eta_types
: Types of the state disturbances in the model.

tv
: Integer vector stating whether Z,H,T,R or Q is time-varying (indicated by 1 in tv and 0 otherwise). If you manually change the dimensions of the matrices you must change this attribute also.

## See Also

artransform

KFAS for more examples.

## Examples

```
# add intercept to state equation by augmenting the state vector:
# diffuse initialization for the intercept, gets estimated like other states:
# for known fixed intercept, just set P1 = P1inf = 0 (default in SSMcustom).
intercept <- 0
model_int <- SSModel(Nile ~ SSMtrend(1, Q = 1469) +
SSMcustom(Z = 0, T = 1, Q = 0, a1 = intercept, P1inf = 1), H = 15099)

model_int$T
model_int$T[1, 2, 1] <- 1 # add the intercept value to level
out <- KFS(model_int)

# An example of a time-varying variance

model_drivers <- SSModel(log(cbind(front, rear)) ~ SSMtrend(1, Q = list(diag(2))),
data = Seatbelts, H = array(NA, c(2, 2, 192)))
```

```
ownupdatefn <- function(pars, model){
  diag(model$Q[, , 1]) <- exp(pars[1:2])
  model$H[,,1:169] <- diag(exp(pars[3:4])) # break in variance
  model$H[,,170:192] <- diag(exp(pars[5:6]))
  model
}

fit_drivers <- fitSSM(model_drivers, inits = rep(-1, 6),
  updatefn = ownupdatefn, method = "BFGS")
fit_drivers$model$H[,,1]
fit_drivers$model$H[,,192]

# An example of shift in the level component

Tt <- array(diag(2), c(2, 2, 100))
Tt[1,2,28] <- 1
Z <- matrix(c(1,0), 1, 2)
Q <- diag(c(NA, 0), 2)
model <- SSModel(Nile ~ -1 + SSMcustom(Z, Tt, Q = Q, P1inf = diag(2)),
  H = matrix(NA))

model <- fitSSM(model, c(10,10), method = "BFGS")$model
model$Q
model$H

conf_Nile <- predict(model, interval = "confidence", level = 0.9)
pred_Nile <- predict(model, interval = "prediction", level = 0.9)

ts.plot(cbind(Nile, pred_Nile, conf_Nile[, -1]), col = c(1:2, 3, 3, 4, 4),
        ylab = "Predicted Annual flow", main = "River Nile")

# dynamic regression model

set.seed(1)
x1 <- rnorm(100)
x2 <- rnorm(100)
b1 <- 1 + cumsum(rnorm(100, sd = 1))
b2 <- 2 + cumsum(rnorm(100, sd = 0.1))
y <- 1 + b1 * x1 + b2 * x2 + rnorm(100, sd = 0.1)

model <- SSModel(y ~ SSMregression(~ x1 + x2, Q = diag(NA,2)), H = NA)

fit <- fitSSM(model, inits = c(0,0,0), method = "BFGS")

model <- fit$model
model$Q
model$H
out <- KFS(model)

ts.plot(out$alphahat[,-1], b1, b2, col = 1:4)

# SSMregression with multivariate observations
```

```
x <- matrix(rnorm(30), 10, 3) # one variable per each series
y <- x + rnorm(30)
model <- SSModel(y ~ SSMregression(list(~ X1, ~ X2, ~ X3), data = data.frame(x)))
# more generally SSMregression(sapply(1:3, function(i) formula(paste0("~ X",i))), ...)

# three covariates per series, with same coefficients:
y <- x[,1] + x[,2] + x[,3] + matrix(rnorm(30), 10, 3)
model <- SSModel(y ~ -1 + SSMregression(~ X1 + X2 + X3, remove.intercept = FALSE,
  type = "common", data = data.frame(x)))

# the above cases can be combined in various ways, you can call SSMregression multiple times:
model <- SSModel(y ~  SSMregression(~ X1 + X2, type = "common") +
  SSMregression(~ X2), data = data.frame(x))

# examples of using data argument
y <- x <- rep(1, 3)
data1 <- data.frame(x = rep(2, 3))
data2 <- data.frame(x = rep(3, 3))

f <- formula(~ -1 + x)
# With data missing the environment of formula is checked,
# and if not found in there a calling environment via parent.frame is checked.

c(SSModel(y ~ -1 + x)["Z"]) # 1
c(SSModel(y ~ -1 + x, data = data1)["Z"]) # 2

c(SSModel(y ~ -1 + SSMregression(~ -1 + x))["Z"]) # 1
c(SSModel(y ~ -1 + SSMregression(~ -1 + x, data = data1))["Z"]) # 2
c(SSModel(y ~ -1 + SSMregression(~ -1 + x), data = data1)["Z"]) # 2
SSModel(y ~ -1 + x + SSMregression(~ -1 + x, data = data1))["Z"] # 1 and 2
SSModel(y ~ -1 + x + SSMregression(~ -1 + x), data = data1)["Z"] # both are 2
SSModel(y ~ -1 + x + SSMregression(~ -1 + x, data = data1), data = data2)["Z"] # 3 and 2

SSModel(y ~ -1 + x + SSMregression(f))["Z"] # 1 and 1
SSModel(y ~ -1 + x + SSMregression(f), data = data1)["Z"] # 2 and 1
SSModel(y ~ -1 + x + SSMregression(f,data = data1))["Z"] # 1 and 2

rm(x)
c(SSModel(y ~ -1 + SSMregression(f, data = data1))$Z) # 2
## Not run:
# This fails as there is no x in the environment of f
try(c(SSModel(y ~ -1 + SSMregression(f), data = data1)$Z))

## End(Not run)
```

---

`transformSSM` *Transform Multivariate State Space Model for Sequential Processing*

---

### Description

transformSSM transforms the general multivariate Gaussian state space model to form suitable for sequential processing.

### Usage

```
transformSSM(object, type = c("ldl", "augment"), tol)
```

### Arguments

| | |
|---|---|
| object | State space model object from function SSModel. |
| type | Option "ldl" performs LDL decomposition for covariance matrix $H_t$, and multiplies the observation equation with the $L_t^{-1}$, so $\epsilon_t^* \sim N(0, D_t)$. Option "augment" adds $\epsilon_t$ to the state vector, so $Q_t$ becomes block diagonal with blocks $Q_t$ and $H_t$. |
| tol | Tolerance parameter for LDL decomposition (see ldl). Default is max(100, max(abs(apply(object$H, 3, diag)))) * .Machine$double.eps. |

### Details

As all the functions in KFAS use univariate approach i.e. sequential processing, the covariance matrix $H_t$ of the observation equation needs to be either diagonal or zero matrix. Function transformSSM performs either the LDL decomposition of $H_t$, or augments the state vector with the disturbances of the observation equation.

In case of a LDL decomposition, the new $H_t$ contains the diagonal part of the decomposition, whereas observations $y_t$ and system matrices $Z_t$ are multiplied with the inverse of $L_t$. Note that although the state estimates and their error covariances obtained by Kalman filtering and smoothing are identical with those obtained from ordinary multivariate filtering, the one-step-ahead errors $v_t$ and their variances $F_t$ do differ. The typical multivariate versions can be obtained from output of KFS using mvInnovations function.

In case of augmentation of the state vector, some care is needed interpreting the subsequent filtering/smoothing results: For example the muhat from the output of KFS now contains also the smoothed observational level noise as that is part of the state vector.

### Value

| | |
|---|---|
| model | Transformed model. |

---

## [<-.SSModel *Extract or Replace Parts of a State Space Model*

---

### Description

S3 methods for getting and setting parts of object of class SSModel. These methods ensure that dimensions of system matrices are not altered.

### Usage

```
## S3 replacement method for class 'SSModel'
x[element, states, etas, series, times, ...] <- value

## S3 method for class 'SSModel'
x[element, states, etas, series, times, drop = TRUE, ...]
```

### Arguments

| | |
|---|---|
| `x` | Object of class `SSModel`. |
| `element` | Which element(s) is chosen. Typical values are `"y"`, `"Z"`, `"H"`, `"T"`, `"R"`, `"Q"`, `"a1"`, `"P1"`, `"P1inf"`, and `"u"`. See details. |
| `states` | Which states are chosen. Either a numeric vector containing the indices of the states, or a character vector defining the types of the states. Possible choices are `"all"`, `"level"`, `"slope"`, `"trend"`, `"regression"`, `"arima"`, `"custom"`, `"cycle"` or `"seasonal"`, where `"trend"` extracts all states relating to trend. These can be combined. Default is `"all"`. |
| `etas` | Which disturbances eta are chosen. Used for elements `"R"` and `"Q"`. Either a numeric vector containing the indices of the etas, or a character vector defining the types of the etas. Possible choices are `"all"`, `"level"`, `"slope"`, `"trend"`, `"regression"`, `"arima"`, `"custom"`, `"cycle"` or `"seasonal"`, where `"trend"` extracts all etas relating to trend. These can be combined. Default is `"all"`. |
| `series` | Numeric. Which series are chosen. Used for elements `"y"`, `"Z"`, and `"u"`. |
| `times` | Numeric. Which time points are chosen. |
| `...` | Ignored. |
| `value` | A value to be assigned to x. |
| `drop` | Logical. If `TRUE` (default) the result is coerced to the lowest possible dimension. |

### Details

If `element` is not one of `"y"`, `"Z"`, `"H"`, `"T"`, `"R"`, `"Q"`, `"a1"`, `"P1"`, `"P1inf"`, `"u"`, the default single bracket list extraction and assignments (`x[element]` and `x[element] <- value`) are used (and other arguments are ignored).

If `element` is one of `"y"`, `"Z"`, `"H"`, `"T"`, `"R"`, `"Q"`, `"a1"`, `"P1"`, `"P1inf"`, `"u"` and if the arguments `states`, `etas`, `times` and `series` are all missing, the double bracket list extraction `x[[element]]` and modified double bracket list assignment `x[[element]][] <- value` are used.

If neither of above holds, then for example in case of `element = Z` the extraction is of form `x$Z[series, states, times, drop]`.

### Value

A selected subset of the chosen element or a value.

## Examples

```
set.seed(1)
model <- SSModel(rnorm(10) ~ 1)
model["H"]
model["H"] <- 10
# H is still an array:
model["H"]
logLik(model)
model$H <- 1
# model["H"] throws an error as H is now scalar:
model$H
logLik(model, check.model = TRUE) #with check.model = FALSE R crashes!
```

# Index