Package: bssm (via r-universe)

October 29, 2026

Type Package

Title Bayesian Inference of Non-Linear and Non-Gaussian State Space Models

Version 2.0.3

Description Efficient methods for Bayesian inference of state space models via Markov chain Monte Carlo (MCMC) based on parallel importance sampling type weighted estimators (Vihola, Helske, and Franks, 2020, <doi:10.1111/sjos.12492>), particle MCMC, and its delayed acceptance version. Gaussian, Poisson, binomial, negative binomial, and Gamma observation densities and basic stochastic volatility models with linear-Gaussian state dynamics, as well as general non-linear Gaussian models and discretised diffusion models are supported. See Helske and Vihola (2021, <doi:10.32614/RJ-2021-103>) for details.

License GPL (>= 2)

Depends R (>= 4.1.0)

Suggests covr, ggplot2 (>= 2.0.0), KFAS (>= 1.2.1), knitr (>= 1.11), MASS, rmarkdown (>= 0.8.1), ramcmc, sde, sitmo, testthat

Imports bayesplot, checkmate, coda (>= 0.18-1), diagis, dplyr, posterior, Rcpp (>= 0.12.3), rlang, tidyr

LinkingTo ramcmc, Rcpp, RcppArmadillo, sitmo

SystemRequirements pandoc (>= 1.12.3, needed for vignettes)

VignetteBuilder knitr

BugReports https://github.com/helske/bssm/issues

URL https://github.com/helske/bssm

ByteCompile true

Encoding UTF-8

NeedsCompilation yes

RoxygenNote 7.2.3

Author Jouni Helske [aut, cre] (ORCID:
<https://orcid.org/0000-0001-7130-793X>), Matti Vihola [aut]
(ORCID: <https://orcid.org/0000-0002-8041-7222>)

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

Repository https://cran.r-universe.dev

Date/Publication 2025-09-24 13:30:03 UTC

RemoteUrl https://github.com/cran/bssm

RemoteRef HEAD

RemoteSha c8d331788927f0fd870b16d0d57fe803bd2a3786

Contents

ar1_lg . . . 3
ar1_ng . . . 4
as.data.frame.mcmc_output . . . 6
as_bssm . . . 7
as_draws_df.mcmc_output . . . 8
asymptotic_var . . . 9
bootstrap_filter . . . 10
bsm_lg . . . 12
bsm_ng . . . 14
bssm . . . 17
check_diagnostics . . . 19
cpp_example_model . . . 20
drownings . . . 20
ekf . . . 21
ekf_smoother . . . 22
ekpf_filter . . . 23
estimate_ess . . . 25
exchange . . . 26
expand_sample . . . 26
fast_smoother . . . 27
fitted.mcmc_output . . . 28
gaussian_approx . . . 29
iact . . . 30
importance_sample . . . 31
kfilter . . . 32
logLik.lineargaussian . . . 33
negbin_model . . . 35
negbin_series . . . 36
particle_smoother . . . 37
plot.mcmc_output . . . 39
poisson_series . . . 40
post_correct . . . 41
predict.mcmc_output . . . 43
print.mcmc_output . . . 46

run_mcmc 47
sim_smoother 53
ssm_mlg 55
ssm_mng 57
ssm_nlg 59
ssm_sde 61
ssm_ulg 63
ssm_ung 67
suggest_N 69
summary.mcmc_output 71
svm 72
ukf 74
uniform_prior 75

Index **78**

`ar1_lg` *Univariate Gaussian model with AR(1) latent process*

Description

Constructs a simple Gaussian model where the state dynamics follow an AR(1) process.

Usage

```
ar1_lg(y, rho, sigma, mu, sd_y, beta, xreg = NULL)
```

Arguments

`y`	A vector or a `ts` object of observations.
`rho`	A prior for autoregressive coefficient. Should be an object of class `bssm_prior`.
`sigma`	A prior for the standard deviation of noise of the AR-process. Should be an object of class `bssm_prior`
`mu`	A fixed value or a prior for the stationary mean of the latent AR(1) process. Should be an object of class `bssm_prior` or scalar value defining a fixed mean such as 0.
`sd_y`	A prior for the standard deviation of observation equation.
`beta`	A prior for the regression coefficients. Should be an object of class `bssm_prior` or `bssm_prior_list` (in case of multiple coefficients) or missing in case of no covariates.
`xreg`	A matrix containing covariates with number of rows matching the length of `y`. Can also be `ts`, `mts` or similar object convertible to matrix.

Value

An object of class `ar1_lg`.

Examples

```
set.seed(1)
mu <- 2
rho <- 0.7
sd_y <- 0.1
sigma <- 0.5
beta <- -1
x <- rnorm(30)
z <- y <- numeric(30)
z[1] <- rnorm(1, mu, sigma / sqrt(1 - rho^2))
y[1] <- rnorm(1, beta * x[1] + z[1], sd_y)
for(i in 2:30) {
  z[i] <- rnorm(1, mu * (1 - rho) + rho * z[i - 1], sigma)
  y[i] <- rnorm(1, beta * x[i] + z[i], sd_y)
}
model <- ar1_lg(y, rho = uniform(0.5, -1, 1),
  sigma = halfnormal(1, 10), mu = normal(0, 0, 1),
  sd_y = halfnormal(1, 10),
  xreg = x,  beta = normal(0, 0, 1))
out <- run_mcmc(model, iter = 2e4)
summary(out, return_se = TRUE)
```

---

ar1_ng *Non-Gaussian model with AR(1) latent process*

Description

Constructs a simple non-Gaussian model where the state dynamics follow an AR(1) process.

Usage

```
ar1_ng(y, rho, sigma, mu, distribution, phi, u, beta, xreg = NULL)
```

Arguments

y	A vector or a ts object of observations.
rho	A prior for autoregressive coefficient. Should be an object of class bssm_prior.
sigma	A prior for the standard deviation of noise of the AR-process. Should be an object of class bssm_prior
mu	A fixed value or a prior for the stationary mean of the latent AR(1) process. Should be an object of class bssm_prior or scalar value defining a fixed mean such as 0.
distribution	Distribution of the observed time series. Possible choices are "poisson", "binomial", "gamma", and "negative binomial".

phi	Additional parameter relating to the non-Gaussian distribution. For negative binomial distribution this is the dispersion term, for gamma distribution this is the shape parameter, and for other distributions this is ignored. Should an object of class bssm_prior or a positive scalar.
u	A vector of positive constants for non-Gaussian models. For Poisson, gamma, and negative binomial distribution, this corresponds to the offset term. For binomial, this is the number of trials.
beta	A prior for the regression coefficients. Should be an object of class bssm_prior or bssm_prior_list (in case of multiple coefficients) or missing in case of no covariates.
xreg	A matrix containing covariates with number of rows matching the length of y. Can also be ts, mts or similar object convertible to matrix.

Value

An object of class ar1_ng.

Examples

```
model <- ar1_ng(discoveries, rho = uniform(0.5,-1,1),
  sigma = halfnormal(0.1, 1), mu = normal(0, 0, 1),
  distribution = "poisson")
out <- run_mcmc(model, iter = 1e4, mcmc_type = "approx",
  output_type = "summary")

ts.plot(cbind(discoveries, exp(out$alphahat)), col = 1:2)

set.seed(1)
n <- 30
phi <- 2
rho <- 0.9
sigma <- 0.1
beta <- 0.5
u <- rexp(n, 0.1)
x <- rnorm(n)
z <- y <- numeric(n)
z[1] <- rnorm(1, 0, sigma / sqrt(1 - rho^2))
y[1] <- rnbinom(1, mu = u * exp(beta * x[1] + z[1]), size = phi)
for(i in 2:n) {
  z[i] <- rnorm(1, rho * z[i - 1], sigma)
  y[i] <- rnbinom(1, mu = u * exp(beta * x[i] + z[i]), size = phi)
}

model <- ar1_ng(y, rho = uniform_prior(0.9, 0, 1),
  sigma = gamma_prior(0.1, 2, 10), mu = 0.,
  phi = gamma_prior(2, 2, 1), distribution = "negative binomial",
  xreg = x, beta = normal_prior(0.5, 0, 1), u = u)
```

`as.data.frame.mcmc_output` *Convert MCMC Output to data.frame*

Description

Converts the MCMC output of `run_mcmc` to `data.frame`.

Usage

```
## S3 method for class 'mcmc_output'
as.data.frame(
  x,
  row.names,
  optional,
  variable = c("theta", "states"),
  times,
  states,
  expand = TRUE,
  use_times = TRUE,
  ...
)
```

Arguments

`x`	Object of class `mcmc_output` from `run_mcmc`.
`row.names`	Ignored.
`optional`	Ignored.
`variable`	Return samples of `"theta"` (default) or `"states"`?
`times`	A vector of indices. In case of states, what time points to return? Default is all.
`states`	A vector of indices. In case of states, what states to return? Default is all.
`expand`	Should the jump-chain be expanded? Defaults to `TRUE`. For `expand = FALSE` and always for IS-MCMC, the resulting data.frame contains variable weight (= counts * IS-weights).
`use_times`	If `TRUE` (default), transforms the values of the time variable to match the ts attribute of the input to define. If `FALSE`, time is based on the indexing starting from 1.
`...`	Ignored.

See Also

`as_draws` which converts the output for `as_draws` object.

Examples

```
data("poisson_series")
model <- bsm_ng(y = poisson_series,
sd_slope = halfnormal(0.1, 0.1),
sd_level = halfnormal(0.1, 1),
  distribution = "poisson")

out <- run_mcmc(model, iter = 2000, particles = 10)
head(as.data.frame(out, variable = "theta"))
head(as.data.frame(out, variable = "state"))

# don't expand the jump chain:
head(as.data.frame(out, variable = "theta", expand = FALSE))

# IS-weighted version:
out_is <- run_mcmc(model, iter = 2000, particles = 10,
  mcmc_type  = "is2")
head(as.data.frame(out_is, variable = "theta"))
```

---

as_bssm *Convert KFAS Model to bssm Model*

Description

Converts `SSModel` object of `KFAS` package to general `bssm` model of type `ssm_ulg`, `ssm_mlg`, `ssm_ung` or `ssm_mng`. As `KFAS` supports formula syntax for defining e.g. regression and cyclic components it maybe sometimes easier to define the model with `KFAS::SSModel` and then convert for the bssm style with `as_bssm`.

Usage

```
as_bssm(model, kappa = 100, ...)
```

Arguments

`model`	Object of class `SSModel`.
`kappa`	For `SSModel` object, a prior variance for initial state used to replace exact diffuse elements of the original model.
`...`	Additional arguments to model building functions of `bssm` (such as prior and updating functions, C, and D).

Value

An object of class `ssm_ulg`, `ssm_mlg`, `ssm_ung` or `ssm_mng`.

Examples

```
library("KFAS")
  model_KFAS <- SSModel(Nile ~
    SSMtrend(1, Q = 2, P1 = 1e4), H = 2)
  model_bssm <- as_bssm(model_KFAS)
  logLik(model_KFAS)
  logLik(model_bssm)
```

`as_draws_df.mcmc_output` *Convert* `run_mcmc` *Output to* `draws_df` *Format*

Description

Converts MCMC output from `run_mcmc` call to a `draws_df` format of the `posterior` package. This enables the use of diagnostics and plotting methods of `posterior` and `bayesplot` packages.

Usage

```
## S3 method for class 'mcmc_output'
as_draws_df(x, times, states, ...)

## S3 method for class 'mcmc_output'
as_draws(x, times, states, ...)
```

Arguments

`x`	An object of class `mcmc_output`.
`times`	A vector of indices defining which time points to return? Default is all. If 0, no samples for the states are extracted.
`states`	A vector of indices defining which states to return. Default is all. If 0, no samples for the states are extracted.
`...`	Ignored.

Value

A `draws_df` object.

Note

The jump chain representation is automatically expanded by `as_draws`, but if `run_mcmc` used IS-MCMC method, the output contains additional `weight` column corresponding to the IS-weights (without counts), which is ignored by `posterior` and `bayesplot`, i.e. those results correspond to approximate MCMC.

Examples

```
model <- bsm_lg(Nile,
  sd_y = tnormal(init = 100, mean = 100, sd = 100, min = 0),
  sd_level = tnormal(init = 50, mean = 50, sd = 100, min = 0),
  a1 = 1000, P1 = 500^2)

fit1 <- run_mcmc(model, iter = 2000)
draws <- as_draws(fit1)
head(draws, 4)
estimate_ess(draws$sd_y)
summary(fit1, return_se = TRUE)

# More chains:
model$theta[] <- c(50, 150) # change initial value
fit2 <- run_mcmc(model, iter = 2000, verbose = FALSE)
model$theta[] <- c(150, 50) # change initial value
fit3 <- run_mcmc(model, iter = 2000, verbose = FALSE)

# it is actually enough to transform first mcmc_output to draws object,
# rest are transformed automatically inside bind_draws
draws <- posterior::bind_draws(as_draws(fit1),
  as_draws(fit2), as_draws(fit3), along = "chain")

posterior::rhat(draws$sd_y)
```

---

asymptotic_var *Asymptotic Variance of IS-type Estimators*

Description

The asymptotic variance MCMCSE^2 is based on Corollary 1 of Vihola et al. (2020) from weighted samples from IS-MCMC. The default method is based on the integrated autocorrelation time (IACT) by Sokal (1997) which seem to work well for reasonable problems, but it is also possible to use the Geyer's method as implemented in `ess_mean` of the `posterior` package.

Usage

```
asymptotic_var(x, w, method = "sokal")
```

Arguments

`x`	A numeric vector of samples.
`w`	A numeric vector of weights. If missing, set to 1 (i.e. no weighting is assumed).
`method`	Method for computing IACT. Default is `"sokal"`, other option `"geyer"`.

Value

A single numeric value of asymptotic variance estimate.

References

Vihola M, Helske J, Franks J. (2020). Importance sampling type estimators based on approximate marginal Markov chain Monte Carlo. Scand J Statist. 1-38. https://doi.org/10.1111/sjos.12492

Sokal A. (1997). Monte Carlo Methods in Statistical Mechanics: Foundations and New Algorithms. In: DeWitt-Morette C, Cartier P, Folacci A (eds) Functional Integration. NATO ASI Series (Series B: Physics), vol 361. Springer, Boston, MA. https://doi.org/10.1007/978-1-4899-0319-8_6

Gelman, A, Carlin J B, Stern H S, Dunson, D B, Vehtari A, Rubin D B. (2013). Bayesian Data Analysis, Third Edition. Chapman and Hall/CRC.

Vehtari A, Gelman A, Simpson D, Carpenter B, Bürkner P-C. (2021). Rank-normalization, folding, and localization: An improved Rhat for assessing convergence of MCMC. Bayesian analysis, 16(2):667-718. https://doi.org/10.1214/20-BA1221

Examples

```
set.seed(1)
n <- 1e4
x <- numeric(n)
phi <- 0.7
for(t in 2:n) x[t] <- phi * x[t-1] + rnorm(1)
w <- rexp(n, 0.5 * exp(0.001 * x^2))
# different methods:
asymptotic_var(x, w, method = "sokal")
asymptotic_var(x, w, method = "geyer")

data("negbin_model")
# can be obtained directly with summary method
d <- suppressWarnings(as_draws(negbin_model))
sqrt(asymptotic_var(d$sd_level, d$weight))
```

---

bootstrap_filter *Bootstrap Filtering*

Description

Function `bootstrap_filter` performs a bootstrap filtering with stratification resampling.

Usage

```
bootstrap_filter(model, particles, ...)

## S3 method for class 'lineargaussian'
bootstrap_filter(
  model,
  particles,
  seed = sample(.Machine$integer.max, size = 1),
  ...
```

```
)

## S3 method for class 'nongaussian'
bootstrap_filter(
  model,
  particles,
  seed = sample(.Machine$integer.max, size = 1),
  ...
)

## S3 method for class 'ssm_nlg'
bootstrap_filter(
  model,
  particles,
  seed = sample(.Machine$integer.max, size = 1),
  ...
)

## S3 method for class 'ssm_sde'
bootstrap_filter(
  model,
  particles,
  L,
  seed = sample(.Machine$integer.max, size = 1),
  ...
)
```

Arguments

model	A model object of class bssm_model.
particles	Number of particles as a positive integer. Suitable values depend on the model and the data, and while larger values provide more accurate estimates, the run time also increases with respect to the number of particles, so it is generally a good idea to test the filter first with a small number of particles, e.g., less than 100.
...	Ignored.
seed	Seed for the C++ RNG (positive integer).
L	Positive integer defining the discretization level for SDE models.

Value

List with samples (alpha) from the filtering distribution and corresponding weights (weights), as well as filtered and predicted states and corresponding covariances (at, att, Pt, Ptt), and estimated log-likelihood (logLik).

References

Gordon, NJ, Salmond, DJ, Smith, AFM (1993) Novel approach to nonlinear/non-Gaussian Bayesian state estimation. IEE Proceedings F, 140(2), p. 107-113.

Examples

```
set.seed(1)
x <- cumsum(rnorm(50))
y <- rnorm(50, x, 0.5)
model <- bsm_lg(y, sd_y = 0.5, sd_level = 1, P1 = 1)

out <- bootstrap_filter(model, particles = 1000)
ts.plot(cbind(y, x, out$att), col = 1:3)
ts.plot(cbind(kfilter(model)$att, out$att), col = 1:3)

data("poisson_series")
model <- bsm_ng(poisson_series, sd_level = 0.1, sd_slope = 0.01,
  P1 = diag(1, 2), distribution = "poisson")

out <- bootstrap_filter(model, particles = 100)
ts.plot(cbind(poisson_series, exp(out$att[, 1])), col = 1:2)
```

---

bsm_lg *Basic Structural (Time Series) Model*

Description

Constructs a basic structural model with local level or local trend component and seasonal component.

Usage

```
bsm_lg(
  y,
  sd_y,
  sd_level,
  sd_slope,
  sd_seasonal,
  beta,
  xreg = NULL,
  period,
  a1 = NULL,
  P1 = NULL,
  D = NULL,
  C = NULL
)
```

Arguments

y	A vector or a `ts` object of observations.
sd_y	Standard deviation of the noise of observation equation. Should be an object of class `bssm_prior` or scalar value defining a known value such as 0.

sd_level	Standard deviation of the noise of level equation. Should be an object of class bssm_prior or scalar value defining a known value such as 0.
sd_slope	Standard deviation of the noise of slope equation. Should be an object of class bssm_prior, scalar value defining a known value such as 0, or missing, in which case the slope term is omitted from the model.
sd_seasonal	Standard deviation of the noise of seasonal equation. Should be an object of class bssm_prior, scalar value defining a known value such as 0, or missing, in which case the seasonal term is omitted from the model.
beta	A prior for the regression coefficients. Should be an object of class bssm_prior or bssm_prior_list (in case of multiple coefficients) or missing in case of no covariates.
xreg	A matrix containing covariates with number of rows matching the length of y. Can also be ts, mts or similar object convertible to matrix.
period	Length of the seasonal pattern. Must be a positive value greater than 2 and less than the length of the input time series. Default is frequency(y), which can also return non-integer value (in which case error is given).
a1	Prior means for the initial states (level, slope, seasonals). Defaults to vector of zeros.
P1	Prior covariance matrix for the initial states (level, slope, seasonals).Default is diagonal matrix with 100 on the diagonal.
D	Intercept terms for observation equation, given as a length n numeric vector or a scalar in case of time-invariant intercept.
C	Intercept terms for state equation, given as a m times n matrix or m times 1 matrix in case of time-invariant intercept.

Value

An object of class bsm_lg.

Examples

```
set.seed(1)
n <- 50
x <- rnorm(n)
level <- numeric(n)
level[1] <- rnorm(1)
for (i in 2:n) level[i] <- rnorm(1, -0.2 + level[i-1], sd = 0.1)
y <- rnorm(n, 2.1 + x + level)
model <- bsm_lg(y, sd_y = halfnormal(1, 5), sd_level = 0.1, a1 = level[1],
  P1 = matrix(0, 1, 1), xreg = x, beta = normal(1, 0, 1),
  D = 2.1, C = matrix(-0.2, 1, 1))

ts.plot(cbind(fast_smoother(model), level), col = 1:2)

prior <- uniform(0.1 * sd(log10(UKgas)), 0, 1)
# period here is redundant as frequency(UKgas) = 4
model_UKgas <- bsm_lg(log10(UKgas), sd_y = prior, sd_level =  prior,
  sd_slope =  prior, sd_seasonal =  prior, period = 4)
```

```
# Note small number of iterations for CRAN checks
mcmc_out <- run_mcmc(model_UKgas, iter = 5000)
summary(mcmc_out, return_se = TRUE)
# Use the summary method from coda:
summary(expand_sample(mcmc_out, "theta"))$stat
mcmc_out$theta[which.max(mcmc_out$posterior), ]
sqrt((fit <- StructTS(log10(UKgas), type = "BSM"))$coef)[c(4, 1:3)]
```

---

bsm_ng *Non-Gaussian Basic Structural (Time Series) Model*

Description

Constructs a non-Gaussian basic structural model with local level or local trend component, a seasonal component, and regression component (or subset of these components).

Usage

```
bsm_ng(
  y,
  sd_level,
  sd_slope,
  sd_seasonal,
  sd_noise,
  distribution,
  phi,
  u,
  beta,
  xreg = NULL,
  period,
  a1 = NULL,
  P1 = NULL,
  C = NULL
)
```

Arguments

y	A vector or a ts object of observations.
sd_level	Standard deviation of the noise of level equation. Should be an object of class bssm_prior or scalar value defining a known value such as 0.
sd_slope	Standard deviation of the noise of slope equation. Should be an object of class bssm_prior, scalar value defining a known value such as 0, or missing, in which case the slope term is omitted from the model.
sd_seasonal	Standard deviation of the noise of seasonal equation. Should be an object of class bssm_prior, scalar value defining a known value such as 0, or missing, in which case the seasonal term is omitted from the model.

sd_noise	A prior for the standard deviation of the additional noise term to be added to linear predictor, defined as an object of class bssm_prior. If missing, no additional noise term is used.
distribution	Distribution of the observed time series. Possible choices are "poisson", "binomial", "gamma", and "negative binomial".
phi	Additional parameter relating to the non-Gaussian distribution. For negative binomial distribution this is the dispersion term, for gamma distribution this is the shape parameter, and for other distributions this is ignored. Should an object of class bssm_prior or a positive scalar.
u	A vector of positive constants for non-Gaussian models. For Poisson, gamma, and negative binomial distribution, this corresponds to the offset term. For binomial, this is the number of trials.
beta	A prior for the regression coefficients. Should be an object of class bssm_prior or bssm_prior_list (in case of multiple coefficients) or missing in case of no covariates.
xreg	A matrix containing covariates with number of rows matching the length of y. Can also be ts, mts or similar object convertible to matrix.
period	Length of the seasonal pattern. Must be a positive value greater than 2 and less than the length of the input time series. Default is frequency(y), which can also return non-integer value (in which case error is given).
a1	Prior means for the initial states (level, slope, seasonals). Defaults to vector of zeros.
P1	Prior covariance matrix for the initial states (level, slope, seasonals).Default is diagonal matrix with 100 on the diagonal.
C	Intercept terms for state equation, given as a m x n or m x 1 matrix.

Value

An object of class bsm_ng.

Examples

```
# Same data as in Vihola, Helske, Franks (2020)
data(poisson_series)
s <- sd(log(pmax(0.1, poisson_series)))
model <- bsm_ng(poisson_series, sd_level = uniform(0.115, 0, 2 * s),
 sd_slope = uniform(0.004, 0, 2 * s), P1 = diag(0.1, 2),
 distribution = "poisson")


out <- run_mcmc(model, iter = 1e5, particles = 10)
summary(out, variable = "theta", return_se = TRUE)
# should be about 0.093 and 0.016
summary(out, variable = "states", return_se = TRUE,
 states = 1, times = c(1, 100))
# should be about -0.075, 2.618
```

```
model <- bsm_ng(Seatbelts[, "VanKilled"], distribution = "poisson",
  sd_level = halfnormal(0.01, 1),
  sd_seasonal = halfnormal(0.01, 1),
  beta = normal(0, 0, 10),
  xreg = Seatbelts[, "law"],
  # default values, just for illustration
  period = 12L,
  a1 = rep(0, 1 + 11), # level + period - 1 seasonal states
  P1 = diag(1, 12),
  C = matrix(0, 12, 1),
  u = rep(1, nrow(Seatbelts)))


set.seed(123)
mcmc_out <- run_mcmc(model, iter = 5000, particles = 10, mcmc_type = "da")
mcmc_out$acceptance_rate
theta <- expand_sample(mcmc_out, "theta")
plot(theta)
summary(theta)

library("ggplot2")
ggplot(as.data.frame(theta[,1:2]), aes(x = sd_level, y = sd_seasonal)) +
  geom_point() + stat_density2d(aes(fill = ..level.., alpha = ..level..),
  geom = "polygon") + scale_fill_continuous(low = "green", high = "blue") +
  guides(alpha = "none")

# Traceplot using as.data.frame method for MCMC output
library("dplyr")
as.data.frame(mcmc_out) |>
  filter(variable == "sd_level") |>
  ggplot(aes(y = value, x = iter)) + geom_line()


# Model with slope term and additional noise to linear predictor to capture
# excess variation
model2 <- bsm_ng(Seatbelts[, "VanKilled"], distribution = "poisson",
  sd_level = halfnormal(0.01, 1),
  sd_seasonal = halfnormal(0.01, 1),
  beta = normal(0, 0, 10),
  xreg = Seatbelts[, "law"],
  sd_slope = halfnormal(0.01, 0.1),
  sd_noise = halfnormal(0.01, 1))

# instead of extra noise term, model using negative binomial distribution:
model3 <- bsm_ng(Seatbelts[, "VanKilled"],
  distribution = "negative binomial",
  sd_level = halfnormal(0.01, 1),
  sd_seasonal = halfnormal(0.01, 1),
  beta = normal(0, 0, 10),
  xreg = Seatbelts[, "law"],
  sd_slope = halfnormal(0.01, 0.1),
  phi = gamma_prior(1, 5, 5))
```

bssm *Bayesian Inference of State Space Models*

Description

This package contains functions for efficient Bayesian inference of state space models (SSMs). For details, see the package vignette and the R Journal paper.

Details

The model is assumed to be either

- Exponential family state space model, where the state equation is linear Gaussian, and the conditional observation density is either Gaussian, Poisson, binomial, negative binomial or Gamma density.
- Basic stochastic volatility model.
- General non-linear model with Gaussian noise terms.
- Model with continuous SDE dynamics.

Missing values in response series are allowed as per SSM theory and can be automatically predicted, but there can be no missing values in the system matrices of the model.

The package contains multiple functions for building the model:

- `bsm_lg` for basic univariate structural time series model (BSM), `ar1` for univariate noisy AR(1) process, and `ssm_ulg` and `ssm_mlg` for arbitrary linear gaussian model with univariate/multivariate observations.
- The non-Gaussian versions (where observations are non-Gaussian) of the above models can be constructed using the functions `bsm_ng`, `ar1_ng`, `ssm_ung` and `ssm_mng`.
- An univariate stochastic volatility model can be defined using a function `svm`.
- For non-linear models, user must define the model using C++ snippets and the the function `ssm_nlg`. See details in the `growth_model` vignette.
- Diffusion models can be defined with the function `ssm_sde`, again using the C++ snippets. See `sde_model` vignette for details.

See the corresponding functions for some examples and details.

After building the model, the model can be estimated via `run_mcmc` function. The documentation of this function gives some examples. The `bssm` package includes several MCMC sampling and sequential Monte Carlo methods for models outside classic linear-Gaussian framework. For definitions of the currently supported models and methods, usage of the package as well as some theory behind the novel IS-MCMC and ψ-APF algorithms, see Helske and Vihola (2021), Vihola, Helske, Franks (2020), and the package vignettes.

The output of the `run_mcmc` can be analysed by extracting the posterior samples of the latent states and hyperparameters using `as.data.frame`, `as_draws`, `expand_sample`, and `summary` methods, as well as `fitted` and `predict` methods. Some MCMC diagnostics checks are available via `check_diagnostics` function, some of which are also provided via the print method of the

run_mcmc output. Functionality of the ggplot2 and bayesplot, can be used to visualize the posterior draws or their summary statistics, and further diagnostics checks can be performed with the help of the posterior and coda packages.

References

Helske J, Vihola M (2021). bssm: Bayesian Inference of Non-linear and Non-Gaussian State Space Models in R. The R Journal (2021) 13:2, 578-589. https://doi.org/10.32614/RJ-2021-103

Vihola, M, Helske, J, Franks, J. (2020). Importance sampling type estimators based on approximate marginal Markov chain Monte Carlo. Scand J Statist. 1-38. https://doi.org/10.1111/sjos.12492

H. Wickham. ggplot2: Elegant Graphics for Data Analysis. Springer-Verlag New York, 2016.

Gabry J, Mahr T (2022). "bayesplot: Plotting for Bayesian Models." R package version 1.9.0, https://mc-stan.org/bayesplot.

Bürkner P, Gabry J, Kay M, Vehtari A (2022). "posterior: Tools for Working with Posterior Distributions." R package version 1.2.1, https://mc-stan.org/posterior.

Martyn Plummer, Nicky Best, Kate Cowles and Karen Vines (2006). CODA: Convergence Diagnosis and Output Analysis for MCMC, R News, vol 6, 7-11.

Examples

```
# Create a local level model (latent random walk + noise) to the Nile
# dataset using the bsm_lg function:
model <- bsm_lg(Nile,
  sd_y = tnormal(init = 100, mean = 100, sd = 100, min = 0),
  sd_level = tnormal(init = 50, mean = 50, sd = 100, min = 0),
  a1 = 1000, P1 = 500^2)

# the priors for the unknown paramters sd_y and sd_level were defined
# as trunctated normal distributions, see ?bssm_prior for details

# Run the MCMC for 2000 iterations (notice the small number of iterations to
# comply with the CRAN's check requirements)
fit <- run_mcmc(model, iter = 2000)

# Some diagnostics checks:
check_diagnostics(fit)

# print some summary information:
fit

# traceplots:
plot(fit)

# extract the summary statistics for state variable
sumr <- summary(fit,variable = "states")

# visualize
library("ggplot2")
ggplot(sumr, aes(time, Mean)) +
    geom_ribbon(aes(ymin = `2.5%`, ymax = `97.5%`),alpha = 0.25) +
```

```
  geom_line() +
  theme_bw()
```

---

check_diagnostics *Quick Diagnostics Checks for* run_mcmc *Output*

Description

Prints out the acceptance rate, smallest effective sample sizes (ESS) and largest Rhat values for a quick first check that the sampling worked. For further checks, see e.g. bayesplot and coda packages.

Usage

```
check_diagnostics(x)
```

Arguments

x	Results object of class mcmc_output from run_mcmc.

Details

For methods other than IS-MCMC, the estimates are based on the improved diagnostics from the posterior package.For IS-MCMC, these Rhat, bulk-ESS, and tail-ESS estimates are based on the approximate posterior which should look reasonable, otherwise the IS-correction does not make much sense. For IS-MCMC, ESS estimates based on a weighted posterior are also computed.

Examples

```
set.seed(1)
n <- 30
phi <- 2
rho <- 0.9
sigma <- 0.1
beta <- 0.5
u <- rexp(n, 0.1)
x <- rnorm(n)
z <- y <- numeric(n)
z[1] <- rnorm(1, 0, sigma / sqrt(1 - rho^2))
y[1] <- rnbinom(1, mu = u * exp(beta * x[1] + z[1]), size = phi)
for(i in 2:n) {
  z[i] <- rnorm(1, rho * z[i - 1], sigma)
  y[i] <- rnbinom(1, mu = u * exp(beta * x[i] + z[i]), size = phi)
}

model <- ar1_ng(y, rho = uniform_prior(0.9, 0, 1),
  sigma = gamma_prior(0.1, 2, 10), mu = 0.,
  phi = gamma_prior(2, 2, 1), distribution = "negative binomial",
```

```
  xreg = x, beta = normal_prior(0.5, 0, 1), u = u)

out <- run_mcmc(model, iter = 1000, particles = 10)
check_diagnostics(out)
```

`cpp_example_model` *Example C++ Codes for Non-Linear and SDE Models*

Description

Example C++ Codes for Non-Linear and SDE Models

Usage

```
cpp_example_model(example, return_code = FALSE)
```

Arguments

`example`	Name of the example model. Run `cpp_example_model("abc")` to get the names of possible models.
`return_code`	If TRUE, will not compile the model but only returns the corresponding code.

Value

Returns pointers to the C++ snippets defining the model, or in case of `return_code = TRUE`, returns the example code without compiling.

Examples

```
cpp_example_model("sde_poisson_OU", return_code = TRUE)
```

`drownings` *Deaths by drowning in Finland in 1969-2019*

Description

Dataset containing number of deaths by drowning in Finland in 1969-2019, corresponding population sizes (in hundreds of thousands), and yearly average summer temperatures (June to August), based on simple unweighted average of three weather stations: Helsinki (Southern Finland), Jyvaskyla (Central Finland), and Sodankyla (Northern Finland).

Format

A time series object containing 51 observations.

Source

Statistics Finland https://stat.fi/tup/tilastotietokannat/index_en.html.

Examples

```
data("drownings")
model <- bsm_ng(drownings[, "deaths"], u = drownings[, "population"],
  xreg = drownings[, "summer_temp"], distribution = "poisson",
  beta = normal(0, 0, 1),
  sd_level = gamma_prior(0.1,2, 10), sd_slope = gamma_prior(0, 2, 10))

fit <- run_mcmc(model, iter = 5000,
  output_type = "summary", mcmc_type = "approx")
fit
ts.plot(model$y/model$u, exp(fit$alphahat[, 1]), col = 1:2)
```

---

ekf *(Iterated) Extended Kalman Filtering*

Description

Function `ekf` runs the (iterated) extended Kalman filter for the given non-linear Gaussian model of class `ssm_nlg`, and returns the filtered estimates and one-step-ahead predictions of the states α_t given the data up to time t.

Usage

```
ekf(model, iekf_iter = 0)
```

Arguments

`model`	Model of class `ssm_nlg`.
`iekf_iter`	Non-negative integer. The default zero corresponds to normal EKF, whereas `iekf_iter` > `0` corresponds to iterated EKF with `iekf_iter` iterations.

Value

List containing the log-likelihood, one-step-ahead predictions `at` and filtered estimates `att` of states, and the corresponding variances `Pt` and `Ptt`.

Examples

```
 # Takes a while on CRAN
set.seed(1)
mu <- -0.2
rho <- 0.7
sigma_y <- 0.1
sigma_x <- 1
```

```
x <- numeric(50)
x[1] <- rnorm(1, mu, sigma_x / sqrt(1 - rho^2))
for(i in 2:length(x)) {
  x[i] <- rnorm(1, mu * (1 - rho) + rho * x[i - 1], sigma_x)
}
y <- rnorm(50, exp(x), sigma_y)

pntrs <- cpp_example_model("nlg_ar_exp")

model_nlg <- ssm_nlg(y = y, a1 = pntrs$a1, P1 = pntrs$P1,
  Z = pntrs$Z_fn, H = pntrs$H_fn, T = pntrs$T_fn, R = pntrs$R_fn,
  Z_gn = pntrs$Z_gn, T_gn = pntrs$T_gn,
  theta = c(mu= mu, rho = rho,
    log_sigma_x = log(sigma_x), log_sigma_y = log(sigma_y)),
  log_prior_pdf = pntrs$log_prior_pdf,
  n_states = 1, n_etas = 1, state_names = "state")

out_ekf <- ekf(model_nlg, iekf_iter = 0)
out_iekf <- ekf(model_nlg, iekf_iter = 5)
ts.plot(cbind(x, out_ekf$att, out_iekf$att), col = 1:3)
```

`ekf_smoother` *Extended Kalman Smoothing*

Description

Function `ekf_smoother` runs the (iterated) extended Kalman smoother for the given non-linear Gaussian model of class `ssm_nlg`, and returns the smoothed estimates of the states and the corresponding variances. Function `ekf_fast_smoother` computes only smoothed estimates of the states.

Usage

```
ekf_smoother(model, iekf_iter = 0)

ekf_fast_smoother(model, iekf_iter = 0)
```

Arguments

`model`	Model of class `ssm_nlg`.
`iekf_iter`	Non-negative integer. The default zero corresponds to normal EKF, whereas `iekf_iter` > `0` corresponds to iterated EKF with `iekf_iter` iterations.

Value

List containing the log-likelihood, smoothed state estimates `alphahat`, and the corresponding variances `Vt` and `Ptt`.

Examples

```
 # Takes a while on CRAN
set.seed(1)
mu <- -0.2
rho <- 0.7
sigma_y <- 0.1
sigma_x <- 1
x <- numeric(50)
x[1] <- rnorm(1, mu, sigma_x / sqrt(1 - rho^2))
for(i in 2:length(x)) {
  x[i] <- rnorm(1, mu * (1 - rho) + rho * x[i - 1], sigma_x)
}
y <- rnorm(length(x), exp(x), sigma_y)

pntrs <- cpp_example_model("nlg_ar_exp")

model_nlg <- ssm_nlg(y = y, a1 = pntrs$a1, P1 = pntrs$P1,
  Z = pntrs$Z_fn, H = pntrs$H_fn, T = pntrs$T_fn, R = pntrs$R_fn,
  Z_gn = pntrs$Z_gn, T_gn = pntrs$T_gn,
  theta = c(mu= mu, rho = rho,
    log_sigma_x = log(sigma_x), log_sigma_y = log(sigma_y)),
  log_prior_pdf = pntrs$log_prior_pdf,
  n_states = 1, n_etas = 1, state_names = "state")

out_ekf <- ekf_smoother(model_nlg, iekf_iter = 0)
out_iekf <- ekf_smoother(model_nlg, iekf_iter = 1)
ts.plot(cbind(x, out_ekf$alphahat, out_iekf$alphahat), col = 1:3)
```

`ekpf_filter` *Extended Kalman Particle Filtering*

Description

Function `ekpf_filter` performs a extended Kalman particle filtering with stratification resampling, based on Van Der Merwe et al (2001).

Usage

```
ekpf_filter(model, particles, ...)

## S3 method for class 'ssm_nlg'
ekpf_filter(
  model,
  particles,
  seed = sample(.Machine$integer.max, size = 1),
  ...
)
```

Arguments

model	Model of class ssm_nlg.
particles	Number of particles as a positive integer. Suitable values depend on the model and the data, and while larger values provide more accurate estimates, the run time also increases with respect to the number of particles, so it is generally a good idea to test the filter first with a small number of particles, e.g., less than 100.
...	Ignored.
seed	Seed for the C++ RNG (positive integer).

Value

A list containing samples, filtered estimates and the corresponding covariances, weights, and an estimate of log-likelihood.

References

Van Der Merwe, R., Doucet, A., De Freitas, N., & Wan, E. A. (2001). The unscented particle filter. In Advances in neural information processing systems (pp. 584-590).

Examples

```
 # Takes a while
set.seed(1)
n <- 50
x <- y <- numeric(n)
y[1] <- rnorm(1, exp(x[1]), 0.1)
for(i in 1:(n-1)) {
 x[i+1] <- rnorm(1, sin(x[i]), 0.1)
 y[i+1] <- rnorm(1, exp(x[i+1]), 0.1)
}

pntrs <- cpp_example_model("nlg_sin_exp")

model_nlg <- ssm_nlg(y = y, a1 = pntrs$a1, P1 = pntrs$P1,
  Z = pntrs$Z_fn, H = pntrs$H_fn, T = pntrs$T_fn, R = pntrs$R_fn,
  Z_gn = pntrs$Z_gn, T_gn = pntrs$T_gn,
  theta = c(log_H = log(0.1), log_R = log(0.1)),
  log_prior_pdf = pntrs$log_prior_pdf,
  n_states = 1, n_etas = 1, state_names = "state")

out <- ekpf_filter(model_nlg, particles = 100)
ts.plot(cbind(x, out$at[1:n], out$att[1:n]), col = 1:3)
```

`estimate_ess` *Effective Sample Size for IS-type Estimators*

Description

Computes the effective sample size (ESS) based on weighted posterior samples.

Usage

```
estimate_ess(x, w, method = "sokal")
```

Arguments

`x`	A numeric vector of samples.
`w`	A numeric vector of weights. If missing, set to 1 (i.e. no weighting is assumed).
`method`	Method for computing the ESS. Default is `"sokal"`, other option are `"geyer"` (see also `asymptotic_var`).

Details

The asymptotic variance MCMCSE^2 is based on Corollary 1 of Vihola et al. (2020) which is used to compute an estimate for the ESS using the identity ESS(x) = var(x) / MCMCSE^2 where var(x) is the posterior variance of x assuming independent samples.

Value

A single numeric value of effective sample size estimate.

References

Vihola, M, Helske, J, Franks, J. (2020). Importance sampling type estimators based on approximate marginal Markov chain Monte Carlo. Scand J Statist. 1-38. https://doi.org/10.1111/sjos.12492

Sokal A. (1997). Monte Carlo Methods in Statistical Mechanics: Foundations and New Algorithms. In: DeWitt-Morette C, Cartier P, Folacci A (eds) Functional Integration. NATO ASI Series (Series B: Physics), vol 361. Springer, Boston, MA. https://doi.org/10.1007/978-1-4899-0319-8_6

Gelman, A, Carlin J B, Stern H S, Dunson, D B, Vehtari A, Rubin D B. (2013). Bayesian Data Analysis, Third Edition. Chapman and Hall/CRC.

Examples

```
set.seed(1)
n <- 1e4
x <- numeric(n)
phi <- 0.7
for(t in 2:n) x[t] <- phi * x[t-1] + rnorm(1)
w <- rexp(n, 0.5 * exp(0.001 * x^2))
# different methods:
```

```
estimate_ess(x, w, method = "sokal")
estimate_ess(x, w, method = "geyer")
```

---

exchange *Pound/Dollar daily exchange rates*

Description

Dataset containing daily log-returns from 1/10/81-28/6/85 as in Durbin and Koopman (2012).

Format

A vector of length 945.

Source

The data used to be available on the www.ssfpack.com/DKbook.html but this page is does not seem to be available anymore.

References

James Durbin, Siem Jan Koopman (2012). Time Series Analysis by State Space Methods. Oxford University Press. https://doi.org/10.1093/acprof:oso/9780199641178.001.0001

Examples

```
 # Don't test on CRAN as complains about parallelisation
data("exchange")
model <- svm(exchange, rho = uniform(0.97,-0.999,0.999),
 sd_ar = halfnormal(0.175, 2), mu = normal(-0.87, 0, 2))

out <- particle_smoother(model, particles = 500)
plot.ts(cbind(model$y, exp(out$alphahat)))
```

---

expand_sample *Expand the Jump Chain representation*

Description

The MCMC algorithms of bssm use a jump chain representation where we store the accepted values and the number of times we stayed in the current value. Although this saves bit memory and is especially convenient for IS-corrected MCMC, sometimes we want to have the usual sample paths (for example for drawing traceplots). Function expand_sample returns the expanded sample based on the counts (in form of coda::mcmc object. Note that for the IS-MCMC the expanded sample corresponds to the approximate posterior, i.e., the weights are ignored.

Usage

```
expand_sample(x, variable = "theta", times, states, by_states = TRUE)
```

Arguments

x	Output from run_mcmc.
variable	Expand parameters "theta" or states "states".
times	A vector of indices. In case of states, what time points to expand? Default is all.
states	A vector of indices. In case of states, what states to expand? Default is all.
by_states	If TRUE (default), return list by states. Otherwise by time.

Details

This functions is mostly for backwards compatibility, methods as.data.frame and as_draws produce likely more convenient output.

Value

An object of class "mcmc" of the coda package.

See Also

as.data.frame.mcmc_output and as_draws.mcmc_output.

Examples

```
set.seed(1)
n <- 50
x <- cumsum(rnorm(n))
y <- rnorm(n, x)
model <- bsm_lg(y, sd_y = gamma_prior(1, 2, 2),
  sd_level = gamma_prior(1, 2, 2))
fit <- run_mcmc(model, iter = 1e4)
# Traceplots for theta
plot.ts(expand_sample(fit, variable = "theta"))
# Traceplot for x_5
plot.ts(expand_sample(fit, variable = "states", times = 5,
  states = 1)$level)
```

---

fast_smoother *Kalman Smoothing*

Description

Methods for Kalman smoothing of the states. Function fast_smoother computes only smoothed estimates of the states, and function smoother computes also smoothed variances.

Usage

```
fast_smoother(model, ...)

## S3 method for class 'lineargaussian'
fast_smoother(model, ...)

smoother(model, ...)

## S3 method for class 'lineargaussian'
smoother(model, ...)
```

Arguments

model	Model to be approximated. Should be of class bsm_ng, ar1_ng svm, ssm_ung, or ssm_mng, or ssm_nlg, i.e. non-gaussian or non-linear bssm_model.
...	Ignored.

Details

For non-Gaussian models, the smoothing is based on the approximate Gaussian model.

Value

Matrix containing the smoothed estimates of states, or a list with the smoothed states and the variances.

Examples

```
model <- bsm_lg(Nile,
  sd_level = tnormal(120, 100, 20, min = 0),
  sd_y = tnormal(50, 50, 25, min = 0),
  a1 = 1000, P1 = 200)
ts.plot(cbind(Nile, fast_smoother(model)), col = 1:2)
model <- bsm_lg(Nile,
  sd_y = tnormal(120, 100, 20, min = 0),
  sd_level = tnormal(50, 50, 25, min = 0),
  a1 = 1000, P1 = 500^2)

out <- smoother(model)
ts.plot(cbind(Nile, out$alphahat), col = 1:2)
ts.plot(sqrt(out$Vt[1, 1, ]))
```

---

fitted.mcmc_output *Fitted for State Space Model*

Description

Returns summary statistics from the posterior predictive distribution of the mean.

Usage

```
## S3 method for class 'mcmc_output'
fitted(object, model, probs = c(0.025, 0.975), ...)
```

Arguments

object	Results object of class mcmc_output from run_mcmc based on the input model.
model	A bssm_model object.
probs	Numeric vector defining the quantiles of interest. Default is c(0.025, 0.975).
...	Ignored.

Examples

```
prior <- uniform(0.1 * sd(log10(UKgas)), 0, 1)
model <- bsm_lg(log10(UKgas), sd_y = prior, sd_level =  prior,
  sd_slope =  prior, sd_seasonal =  prior, period = 4)
fit <- run_mcmc(model, iter = 1e4)
res <- fitted(fit, model)
head(res)
```

---

gaussian_approx *Gaussian Approximation of Non-Gaussian/Non-linear State Space Model*

Description

Returns the approximating Gaussian model which has the same conditional mode of p(alphaly, theta) as the original model. This function is rarely needed itself, and is mainly available for testing and debugging purposes.

Usage

```
gaussian_approx(model, max_iter, conv_tol, ...)

## S3 method for class 'nongaussian'
gaussian_approx(model, max_iter = 100, conv_tol = 1e-08, ...)

## S3 method for class 'ssm_nlg'
gaussian_approx(model, max_iter = 100, conv_tol = 1e-08, iekf_iter = 0, ...)
```

Arguments

model	Model to be approximated. Should be of class bsm_ng, ar1_ng svm, ssm_ung, or ssm_mng, or ssm_nlg, i.e. non-gaussian or non-linear bssm_model.
max_iter	Maximum number of iterations as a positive integer. Default is 100 (although typically only few iterations are needed).

conv_tol	Positive tolerance parameter. Default is 1e-8. Approximation is claimed to be converged when the mean squared difference of the modes of is less than conv_tol.
...	Ignored.
iekf_iter	For non-linear models, non-negative number of iterations in iterated EKF (defaults to 0, i.e. normal EKF). Used only for models of class ssm_nlg.

Value

Returns linear-Gaussian SSM of class ssm_ulg or ssm_mlg which has the same conditional mode of p(alphaly, theta) as the original model.

References

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

Vihola, M, Helske, J, Franks, J. (2020). Importance sampling type estimators based on approximate marginal Markov chain Monte Carlo. Scand J Statist. 1-38. https://doi.org/10.1111/sjos.12492

Examples

```
data("poisson_series")
model <- bsm_ng(y = poisson_series, sd_slope = 0.01, sd_level = 0.1,
  distribution = "poisson")
out <- gaussian_approx(model)
for(i in 1:7)
 cat("Number of iterations used: ", i, ", y[1] = ",
   gaussian_approx(model, max_iter = i, conv_tol = 0)$y[1], "\n", sep ="")
```

---

iact *Integrated Autocorrelation Time*

Description

Estimates the integrated autocorrelation time (IACT) based on Sokal (1997). Note that the estimator is not particularly good for very short series x (say < 100), but that is not very practical for MCMC applications anyway.

Usage

```
iact(x)
```

Arguments

x	A numeric vector.

Value

A single numeric value of IACT estimate.

References

Sokal A. (1997) Monte Carlo Methods in Statistical Mechanics: Foundations and New Algorithms. In: DeWitt-Morette C., Cartier P., Folacci A. (eds) Functional Integration. NATO ASI Series (Series B: Physics), vol 361. Springer, Boston, MA. https://doi.org/10.1007/978-1-4899-0319-8_6

Examples

```
set.seed(1)
n <- 1000
x <- numeric(n)
phi <- 0.8
for(t in 2:n) x[t] <- phi * x[t-1] + rnorm(1)
iact(x)
```

---

importance_sample *Importance Sampling from non-Gaussian State Space Model*

Description

Returns `nsim` samples from the approximating Gaussian model with corresponding (scaled) importance weights. Probably mostly useful for comparing KFAS and bssm packages.

Usage

```
importance_sample(model, nsim, use_antithetic, max_iter, conv_tol, seed, ...)

## S3 method for class 'nongaussian'
importance_sample(
  model,
  nsim,
  use_antithetic = TRUE,
  max_iter = 100,
  conv_tol = 1e-08,
  seed = sample(.Machine$integer.max, size = 1),
  ...
)
```

Arguments

`model`	Model of class `bsm_ng`, `ar1_ng` `svm`, `ssm_ung`, or `ssm_mng`.
`nsim`	Number of samples (positive integer). Suitable values depend on the model and the data, and while larger values provide more accurate estimates, the run time also increases with respect to to the number of samples, so it is generally a good idea to test the filter first with a small number of samples, e.g., less than 100.

use_antithetic	Logical. If TRUE (default), use antithetic variable for location in simulation smoothing. Ignored for ssm_mng models.
max_iter	Maximum number of iterations as a positive integer. Default is 100 (although typically only few iterations are needed).
conv_tol	Positive tolerance parameter. Default is 1e-8. Approximation is claimed to be converged when the mean squared difference of the modes of is less than conv_tol.
seed	Seed for the C++ RNG (positive integer).
...	Ignored.

Examples

```
data("sexratio", package = "KFAS")
model <- bsm_ng(sexratio[, "Male"], sd_level = 0.001,
  u = sexratio[, "Total"],
  distribution = "binomial")

imp <- importance_sample(model, nsim = 1000)

est <- matrix(NA, 3, nrow(sexratio))
for(i in 1:ncol(est)) {
  est[, i] <- diagis::weighted_quantile(exp(imp$alpha[i, 1, ]), imp$weights,
    prob = c(0.05,0.5,0.95))
}

ts.plot(t(est),lty = c(2,1,2))
```

---

kfilter *Kalman Filtering*

Description

Function kfilter runs the Kalman filter for the given model, and returns the filtered estimates and one-step-ahead predictions of the states α_t given the data up to time t.

Usage

```
kfilter(model, ...)

## S3 method for class 'lineargaussian'
kfilter(model, ...)

## S3 method for class 'nongaussian'
kfilter(model, ...)
```

Arguments

model	Model of class lineargaussian, nongaussian or ssm_nlg.
...	Ignored.

Details

For non-Gaussian models, the filtering is based on the approximate Gaussian model.

Value

List containing the log-likelihood (approximate in non-Gaussian case), one-step-ahead predictions at and filtered estimates att of states, and the corresponding variances Pt and Ptt up to the time point n+1 where n is the length of the input time series.

See Also

bootstrap_filter

Examples

```
x <- cumsum(rnorm(20))
y <- x + rnorm(20, sd = 0.1)
model <- bsm_lg(y, sd_level = 1, sd_y = 0.1)
ts.plot(cbind(y, x, kfilter(model)$att), col = 1:3)
```

---

logLik.lineargaussian *Extract Log-likelihood of a State Space Model of class* bssm_model

---

Description

Computes the log-likelihood of a state space model defined by bssm package.

Usage

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

## S3 method for class 'nongaussian'
logLik(
  object,
  particles,
  method = "psi",
  max_iter = 100,
  conv_tol = 1e-08,
  seed = sample(.Machine$integer.max, size = 1),
  ...
)
```

```
## S3 method for class 'ssm_nlg'
logLik(
  object,
  particles,
  method = "bsf",
  max_iter = 100,
  conv_tol = 1e-08,
  iekf_iter = 0,
  seed = sample(.Machine$integer.max, size = 1),
  ...
)

## S3 method for class 'ssm_sde'
logLik(
  object,
  particles,
  L,
  seed = sample(.Machine$integer.max, size = 1),
  ...
)
```

Arguments

object	Model of class bssm_model.
...	Ignored.
particles	Number of samples for particle filter (non-negative integer). If 0, approximate log-likelihood is returned either based on the Gaussian approximation or EKF, depending on the method argument.
method	Sampling method. For Gaussian and non-Gaussian models with linear dynamics,options are "bsf" (bootstrap particle filter, default for non-linear models) and "psi" (ψ-APF, the default for other models). For-nonlinear models option "ekf" uses EKF/IEKF-based particle filter (or just EKF/IEKF approximation in the case of particles = 0).
max_iter	Maximum number of iterations used in Gaussian approximation, as a positive integer. Default is 100 (although typically only few iterations are needed).
conv_tol	Positive tolerance parameter used in Gaussian approximation. Default is 1e-8.
seed	Seed for the C++ RNG (positive integer).
iekf_iter	Non-negative integer. If zero (default), first approximation for non-linear Gaussian models is obtained from extended Kalman filter. If iekf_iter > 0, iterated extended Kalman filter is used with iekf_iter iterations.
L	Integer defining the discretization level defined as (2^L).

Value

A numeric value.

References

Durbin, J., & Koopman, S. (2002). A Simple and Efficient Simulation Smoother for State Space Time Series Analysis. Biometrika, 89(3), 603-615.

Shephard, N., & Pitt, M. (1997). Likelihood Analysis of Non-Gaussian Measurement Time Series. Biometrika, 84(3), 653-667.

Gordon, NJ, Salmond, DJ, Smith, AFM (1993). Novel approach to nonlinear/non-Gaussian Bayesian state estimation. IEE Proceedings-F, 140, 107-113.

Vihola, M, Helske, J, Franks, J. Importance sampling type estimators based on approximate marginal Markov chain Monte Carlo. Scand J Statist. 2020; 1-38. https://doi.org/10.1111/sjos.12492

Van Der Merwe, R, Doucet, A, De Freitas, N, Wan, EA (2001). The unscented particle filter. In Advances in neural information processing systems, p 584-590.

Jazwinski, A 1970. Stochastic Processes and Filtering Theory. Academic Press.

Kitagawa, G (1996). Monte Carlo filter and smoother for non-Gaussian nonlinear state space models. Journal of Computational and Graphical Statistics, 5, 1-25.

See Also

particle_smoother

Examples

```
model <- ssm_ulg(y = c(1,4,3), Z = 1, H = 1, T = 1, R = 1)
logLik(model)
model <- ssm_ung(y = c(1,4,3), Z = 1, T = 1, R = 0.5, P1 = 2,
  distribution = "poisson")

model2 <- bsm_ng(y = c(1,4,3), sd_level = 0.5, P1 = 2,
  distribution = "poisson")

logLik(model, particles = 0)
logLik(model2, particles = 0)
logLik(model, particles = 10, seed = 1)
logLik(model2, particles = 10, seed = 1)
```

---

negbin_model *Estimated Negative Binomial Model of Helske and Vihola (2021)*

Description

This model was used in Helske and Vihola (2021), but with larger number of iterations. Here only 2000 iterations were used in order to reduce the size of the model object in CRAN.

Format

A object of class `mcmc_output`.

References

Helske J, Vihola M (2021). bssm: Bayesian Inference of Non-linear and Non-Gaussian State Space Models in R. The R Journal (2021) 13:2, 578-589. https://doi.org/10.32614/RJ-2021-103

Examples

```
# reproducing the model:
data("negbin_series")
# Construct model for bssm
bssm_model <- bsm_ng(negbin_series[, "y"],
  xreg = negbin_series[, "x"],
  beta = normal(0, 0, 10),
  phi = halfnormal(1, 10),
  sd_level = halfnormal(0.1, 1),
  sd_slope = halfnormal(0.01, 0.1),
  a1 = c(0, 0), P1 = diag(c(10, 0.1)^2),
  distribution = "negative binomial")


# In the paper we used 60000 iterations with first 10000 as burnin
fit_bssm <- run_mcmc(bssm_model, iter = 2000, particles = 10, seed = 1)
fit_bssm
```

---

negbin_series *Simulated Negative Binomial Time Series Data*

Description

See example for code for reproducing the data. This was used in Helske and Vihola (2021).

Format

A time series `mts` object with 200 time points and two series.

References

Helske J, Vihola M (2021). bssm: Bayesian Inference of Non-linear and Non-Gaussian State Space Models in R. The R Journal (2021) 13:2, 578-589. https://doi.org/10.32614/RJ-2021-103

See Also

`negbin_model`

Examples

```
# The data was generated as follows:
set.seed(123)
n <- 200
sd_level <- 0.1
drift <- 0.01
beta <- -0.9
phi <- 5

level <- cumsum(c(5, drift + rnorm(n - 1, sd = sd_level)))
x <- 3 + (1:n) * drift + sin(1:n + runif(n, -1, 1))
y <- rnbinom(n, size = phi, mu = exp(beta * x + level))
```

`particle_smoother` *Particle Smoothing*

Description

Function `particle_smoother` performs particle smoothing based on either bootstrap particle filter (Gordon et al. 1993), ψ-auxiliary particle filter (ψ-APF) (Vihola et al. 2020), extended Kalman particle filter (Van Der Merwe et al. 2001), or its version based on iterated EKF (Jazwinski, 1970). The smoothing phase is based on the filter-smoother algorithm by Kitagawa (1996).

Usage

```
particle_smoother(model, particles, ...)

## S3 method for class 'lineargaussian'
particle_smoother(
  model,
  particles,
  method = "psi",
  seed = sample(.Machine$integer.max, size = 1),
  ...
)

## S3 method for class 'nongaussian'
particle_smoother(
  model,
  particles,
  method = "psi",
  seed = sample(.Machine$integer.max, size = 1),
  max_iter = 100,
  conv_tol = 1e-08,
  ...
)
```

```
## S3 method for class 'ssm_nlg'
particle_smoother(
  model,
  particles,
  method = "bsf",
  seed = sample(.Machine$integer.max, size = 1),
  max_iter = 100,
  conv_tol = 1e-08,
  iekf_iter = 0,
  ...
)

## S3 method for class 'ssm_sde'
particle_smoother(
  model,
  particles,
  L,
  seed = sample(.Machine$integer.max, size = 1),
  ...
)
```

Arguments

model	A model object of class bssm_model.
particles	Number of particles as a positive integer. Suitable values depend on the model, the data, and the chosen algorithm. While larger values provide more accurate estimates, the run time also increases with respect to the number of particles, so it is generally a good idea to test the filter first with a small number of particles, e.g., less than 100.
...	Ignored.
method	Choice of particle filter algorithm. For Gaussian and non-Gaussian models with linear dynamics, options are "bsf" (bootstrap particle filter, default for non-linear models) and "psi" (ψ-APF, the default for other models), and for non-linear models option "ekf" (extended Kalman particle filter) is also available.
seed	Seed for the C++ RNG (positive integer).
max_iter	Maximum number of iterations used in Gaussian approximation, as a positive integer. Default is 100 (although typically only few iterations are needed).
conv_tol	Positive tolerance parameter used in Gaussian approximation. Default is 1e-8.
iekf_iter	Non-negative integer. If zero (default), first approximation for non-linear Gaussian models is obtained from extended Kalman filter. If iekf_iter > 0, iterated extended Kalman filter is used with iekf_iter iterations.
L	Positive integer defining the discretization level for SDE model.

Details

See one of the vignettes for ψ-APF in case of nonlinear models.

Value

List with samples (alpha) from the smoothing distribution and corresponding weights (weights), as well as smoothed means and covariances (alphahat and Vt) of the states and estimated log-likelihood (logLik).

References

Gordon, NJ, Salmond, DJ, Smith, AFM (1993). Novel approach to nonlinear/non-Gaussian Bayesian state estimation. IEE Proceedings-F, 140, 107-113. https://doi.org/10.1049/ip-f-2.1993.0015

Vihola, M, Helske, J, Franks, J. Importance sampling type estimators based on approximate marginal Markov chain Monte Carlo. Scand J Statist. 2020; 1-38. https://doi.org/10.1111/sjos.12492

Van Der Merwe, R, Doucet, A, De Freitas, N, Wan, EA (2001). The unscented particle filter. In Advances in neural information processing systems, p 584-590.

Jazwinski, A 1970. Stochastic Processes and Filtering Theory. Academic Press.

Kitagawa, G (1996). Monte Carlo filter and smoother for non-Gaussian nonlinear state space models. Journal of Computational and Graphical Statistics, 5, 1-25. https://doi.org/10.2307/1390750

Examples

```
set.seed(1)
x <- cumsum(rnorm(100))
y <- rnorm(100, x)
model <- ssm_ulg(y, Z = 1, T = 1, R = 1, H = 1, P1 = 1)
system.time(out <- particle_smoother(model, particles = 1000))
# same with simulation smoother:
system.time(out2 <- sim_smoother(model, particles = 1000,
  use_antithetic = TRUE))
ts.plot(out$alphahat, rowMeans(out2), col = 1:2)
```

---

plot.mcmc_output *Trace and Density Plots for* mcmc_output

Description

Plots the trace and density plots of the hyperparameters theta from the MCMC run by run_mcmc.

Usage

```
## S3 method for class 'mcmc_output'
plot(x, ...)
```

Arguments

x	Object of class mcmc_output from run_mcmc.
...	Further arguments to bayesplot::mcmc_combo.

Details

For further visualization (of the states), you can extract the posterior samples with `as.data.frame` and `as_draws` methods to be used for example with the `bayesplot` or `ggplot2` packages.

Value

The output object from bayesplot::mcmc_combo.

Note

For IS-MCMC, these plots correspond to the approximate (non-weighted) samples .

See Also

`check_diagnostics` for a quick diagnostics statistics of the model.

Examples

```
data("negbin_model")
# Note the very small number of iterations, so the plots look bad
plot(negbin_model)
```

---

poisson_series *Simulated Poisson Time Series Data*

Description

See example for code for reproducing the data. This was used in Vihola, Helske, Franks (2020).

Format

A vector of length 100.

References

Vihola, M, Helske, J, Franks, J (2020). Importance sampling type estimators based on approximate marginal Markov chain Monte Carlo. Scand J Statist. 1-38. https://doi.org/10.1111/sjos.12492

Examples

```
# The data was generated as follows:
set.seed(321)
slope <- cumsum(c(0, rnorm(99, sd = 0.01)))
y <- rpois(100, exp(cumsum(slope + c(0, rnorm(99, sd = 0.1)))))
```

post_correct *Run Post-correction for Approximate MCMC using ψ-APF*

Description

Function post_correct updates previously obtained approximate MCMC output with post-correction weights leading to asymptotically exact weighted posterior, and returns updated MCMC output where components weights, posterior, alpha, alphahat, and Vt are updated (depending on the original output type).

Usage

```
post_correct(
  model,
  mcmc_output,
  particles,
  threads = 1L,
  is_type = "is2",
  seed = sample(.Machine$integer.max, size = 1)
)
```

Arguments

model	Model of class nongaussian or ssm_nlg.
mcmc_output	An output from run_mcmc used to compute the MAP estimate of theta. While the intended use assumes this is from approximate MCMC, it is not actually checked, i.e., it is also possible to input previous (asymptotically) exact output.
particles	Number of particles for ψ-APF (positive integer). Suitable values depend on the model and the data, but often relatively small value less than say 50 is enough. See also suggest_N
threads	Number of parallel threads (positive integer, default is 1).
is_type	Type of IS-correction. Possible choices are "is3" for simple importance sampling (weight is computed for each MCMC iteration independently), "is2" for jump chain importance sampling type weighting (default), or "is1" for importance sampling type weighting where the number of particles used forweight computations is proportional to the length of the jump chain block.
seed	Seed for the C++ RNG (positive integer).

Value

The original object of class mcmc_output with updated weights, log-posterior values and state samples or summaries (depending on the mcmc_output$mcmc_type).

References

Doucet A, Pitt M K, Deligiannidis G, Kohn R (2018). Efficient implementation of Markov chain Monte Carlo when using an unbiased likelihood estimator. Biometrika, 102, 2, 295-313, https://doi.org/10.1093/biomet/asu07

Vihola M, Helske J, Franks J (2020). Importance sampling type estimators based on approximate marginal Markov chain Monte Carlo. Scand J Statist. 1-38. https://doi.org/10.1111/sjos.12492

Examples

```
set.seed(1)
n <- 300
x1 <- sin((2 * pi / 12) * 1:n)
x2 <- cos((2 * pi / 12) * 1:n)
alpha <- numeric(n)
alpha[1] <- 0
rho <- 0.7
sigma <- 2
mu <- 1
for(i in 2:n) {
  alpha[i] <- rnorm(1, mu * (1 - rho) + rho * alpha[i-1], sigma)
}
u <- rpois(n, 50)
y <- rbinom(n, size = u, plogis(0.5 * x1 + x2 + alpha))

ts.plot(y / u)

model <- ar1_ng(y, distribution = "binomial",
  rho = uniform(0.5, -1, 1), sigma = gamma_prior(1, 2, 0.001),
  mu = normal(0, 0, 10),
  xreg = cbind(x1,x2), beta = normal(c(0, 0), 0, 5),
  u = u)

out_approx <- run_mcmc(model, mcmc_type = "approx",
  local_approx = FALSE, iter = 50000)

out_is2 <- post_correct(model, out_approx, particles = 30,
  threads = 2)
out_is2$time

summary(out_approx, return_se = TRUE)
summary(out_is2, return_se = TRUE)

# latent state
library("dplyr")
library("ggplot2")
state_approx <- as.data.frame(out_approx, variable = "states") |>
  group_by(time) |>
  summarise(mean = mean(value))

state_exact <- as.data.frame(out_is2, variable = "states") |>
  group_by(time) |>
  summarise(mean = weighted.mean(value, weight))
```

```
dplyr::bind_rows(approx = state_approx,
  exact = state_exact, .id = "method") |>
  filter(time > 200) |>
ggplot(aes(time, mean, colour = method)) +
  geom_line() +
  theme_bw()

# posterior means
p_approx <- predict(out_approx, model, type = "mean",
  nsim = 1000, future = FALSE) |>
  group_by(time) |>
  summarise(mean = mean(value))
p_exact <- predict(out_is2, model, type = "mean",
  nsim = 1000, future = FALSE) |>
  group_by(time) |>
  summarise(mean = mean(value))

dplyr::bind_rows(approx = p_approx,
  exact = p_exact, .id = "method") |>
  filter(time > 200) |>
ggplot(aes(time, mean, colour = method)) +
  geom_line() +
  theme_bw()
```

`predict.mcmc_output` *Predictions for State Space Models*

Description

Draw samples from the posterior predictive distribution for future time points given the posterior draws of hyperparameters θ and latent state $alpha_{n+1}$ returned by `run_mcmc`. Function can also be used to draw samples from the posterior predictive distribution $p(\tilde{y}_1, \ldots, \tilde{y}_n | y_1, \ldots, y_n)$.

Usage

```
## S3 method for class 'mcmc_output'
predict(
  object,
  model,
  nsim,
  type = "response",
  future = TRUE,
  seed = sample(.Machine$integer.max, size = 1),
  ...
)
```

Arguments

object	Results object of class mcmc_output from run_mcmc.
model	A bssm_model object. Should have same structure and class as the original model which was used in run_mcmc, in order to plug the posterior samples of the model parameters to the right places. It is also possible to input the original model for obtaining predictions for past time points. In this case, set argument future to FALSE.
nsim	Positive integer defining number of samples to draw. Should be less than or equal to sum(object$counts) i.e. the number of samples in the MCMC output. Default is to use all the samples.
type	Type of predictions. Possible choices are "mean" "response", or "state" level.
future	Default is TRUE, in which case predictions are for the future, using posterior samples of (theta, alpha_T+1) i.e. the posterior samples of hyperparameters and latest states. Otherwise it is assumed that model corresponds to the original model.
seed	Seed for the C++ RNG (positive integer). Note that this affects only the C++ side, and predict also uses R side RNG for subsampling, so for replicable results you should call set.seed before predict.
...	Ignored.

Value

A data.frame consisting of samples from the predictive posterior distribution.

See Also

fitted for in-sample predictions.

Examples

```
library("graphics")
y <- log10(JohnsonJohnson)
prior <- uniform(0.01, 0, 1)
model <- bsm_lg(window(y, end = c(1974, 4)), sd_y = prior,
  sd_level = prior, sd_slope = prior, sd_seasonal = prior)

mcmc_results <- run_mcmc(model, iter = 5000)
future_model <- model
future_model$y <- ts(rep(NA, 25),
  start = tsp(model$y)[2] + 2 * deltat(model$y),
  frequency = frequency(model$y))
# use "state" for illustrative purposes, we could use type = "mean" directly
pred <- predict(mcmc_results, model = future_model, type = "state",
  nsim = 1000)

library("dplyr")
sumr_fit <- as.data.frame(mcmc_results, variable = "states") |>
```

```
  group_by(time, iter) |>
  mutate(signal =
      value[variable == "level"] +
      value[variable == "seasonal_1"]) |>
  group_by(time) |>
  summarise(mean = mean(signal),
    lwr = quantile(signal, 0.025),
    upr = quantile(signal, 0.975))

sumr_pred <- pred |>
  group_by(time, sample) |>
  mutate(signal =
      value[variable == "level"] +
      value[variable == "seasonal_1"]) |>
  group_by(time) |>
  summarise(mean = mean(signal),
    lwr = quantile(signal, 0.025),
    upr = quantile(signal, 0.975))

# If we used type = "mean", we could do
# sumr_pred <- pred |>
#   group_by(time) |>
#   summarise(mean = mean(value),
#     lwr = quantile(value, 0.025),
#     upr = quantile(value, 0.975))

library("ggplot2")
rbind(sumr_fit, sumr_pred) |>
  ggplot(aes(x = time, y = mean)) +
  geom_ribbon(aes(ymin = lwr, ymax = upr),
   fill = "#92f0a8", alpha = 0.25) +
  geom_line(colour = "#92f0a8") +
  theme_bw() +
  geom_point(data = data.frame(
    mean = log10(JohnsonJohnson),
    time = time(JohnsonJohnson)))

# Posterior predictions for past observations:
yrep <- predict(mcmc_results, model = model, type = "response",
  future = FALSE, nsim = 1000)
meanrep <- predict(mcmc_results, model = model, type = "mean",
  future = FALSE, nsim = 1000)

sumr_yrep <- yrep |>
  group_by(time) |>
  summarise(earnings = mean(value),
    lwr = quantile(value, 0.025),
    upr = quantile(value, 0.975)) |>
  mutate(interval = "Observations")

sumr_meanrep <- meanrep |>
  group_by(time) |>
  summarise(earnings = mean(value),
```

```
    lwr = quantile(value, 0.025),
    upr = quantile(value, 0.975)) |>
  mutate(interval = "Mean")

rbind(sumr_meanrep, sumr_yrep) |>
  mutate(interval =
    factor(interval, levels = c("Observations", "Mean"))) |>
  ggplot(aes(x = time, y = earnings)) +
  geom_ribbon(aes(ymin = lwr, ymax = upr, fill = interval),
   alpha = 0.75) +
  theme_bw() +
  geom_point(data = data.frame(
    earnings = model$y,
    time = time(model$y)))
```

`print.mcmc_output` *Print Results from MCMC Run*

Description

Prints some basic summaries from the MCMC run by `run_mcmc`.

Usage

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

Arguments

`x`	Object of class `mcmc_output` from `run_mcmc`.
`...`	Ignored.

Examples

```
data("negbin_model")
print(negbin_model)
```

run_mcmc *Bayesian Inference of State Space Models*

Description

Adaptive Markov chain Monte Carlo simulation for SSMs using Robust Adaptive Metropolis algorithm by Vihola (2012). Several different MCMC sampling schemes are implemented, see parameter arguments, package vignette, Vihola, Helske, Franks (2020) and Helske and Vihola (2021) for details.

Usage

```
run_mcmc(model, ...)

## S3 method for class 'lineargaussian'
run_mcmc(
  model,
  iter,
  output_type = "full",
  burnin = floor(iter/2),
  thin = 1,
  gamma = 2/3,
  target_acceptance = 0.234,
  S,
  end_adaptive_phase = FALSE,
  threads = 1,
  seed = sample(.Machine$integer.max, size = 1),
  verbose,
  ...
)

## S3 method for class 'nongaussian'
run_mcmc(
  model,
  iter,
  particles,
  output_type = "full",
  mcmc_type = "is2",
  sampling_method = "psi",
  burnin = floor(iter/2),
  thin = 1,
  gamma = 2/3,
  target_acceptance = 0.234,
  S,
  end_adaptive_phase = FALSE,
  local_approx = TRUE,
  threads = 1,
```

```
  seed = sample(.Machine$integer.max, size = 1),
  max_iter = 100,
  conv_tol = 1e-08,
  verbose,
  ...
)

## S3 method for class 'ssm_nlg'
run_mcmc(
  model,
  iter,
  particles,
  output_type = "full",
  mcmc_type = "is2",
  sampling_method = "bsf",
  burnin = floor(iter/2),
  thin = 1,
  gamma = 2/3,
  target_acceptance = 0.234,
  S,
  end_adaptive_phase = FALSE,
  threads = 1,
  seed = sample(.Machine$integer.max, size = 1),
  max_iter = 100,
  conv_tol = 1e-08,
  iekf_iter = 0,
  verbose,
  ...
)

## S3 method for class 'ssm_sde'
run_mcmc(
  model,
  iter,
  particles,
  output_type = "full",
  mcmc_type = "is2",
  L_c,
  L_f,
  burnin = floor(iter/2),
  thin = 1,
  gamma = 2/3,
  target_acceptance = 0.234,
  S,
  end_adaptive_phase = FALSE,
  threads = 1,
  seed = sample(.Machine$integer.max, size = 1),
  verbose,
```

```
  ...
)
```

Arguments

model	Model of class bssm_model.
...	Ignored.
iter	A positive integer defining the total number of MCMC iterations. Suitable value depends on the model, data, and the choice of specific algorithms (mcmc_type and sampling_method). As increasing iter also increases run time, it is is generally good idea to first test the performance with a small values, e.g., less than 10000.
output_type	Either "full" (default, returns posterior samples from the posterior $p(\alpha, \theta|y)$), "theta" (for marginal posterior of theta), or "summary" (return the mean and variance estimates of the states and posterior samples of theta). See details.
burnin	A positive integer defining the length of the burn-in period which is disregarded from the results. Defaults to iter / 2. Note that all MCMC algorithms of bssm use adaptive MCMC during the burn-in period in order to find good proposal distribution.
thin	A positive integer defining the thinning rate. All the MCMC algorithms in bssm use the jump chain representation (see refs), and the thinning is applied to these blocks. Defaults to 1. For IS-corrected methods, larger value can also be statistically more effective. Note: With output_type = "summary", the thinning does not affect the computations of the summary statistics in case of pseudo-marginal methods.
gamma	Tuning parameter for the adaptation of RAM algorithm. Must be between 0 and 1.
target_acceptance	Target acceptance rate for MCMC. Defaults to 0.234. Must be between 0 and 1.
S	Matrix defining the initial value for the lower triangular matrix of the RAM algorithm, so that the covariance matrix of the Gaussian proposal distribution is SS'. Note that for some parameters (currently the standard deviation, dispersion, and autoregressive parameters of the BSM and AR(1) models) the sampling is done in unconstrained parameter space, i.e. internal_theta = log(theta) (and logit(rho) or AR coefficient).
end_adaptive_phase	Logical, if TRUE, S is held fixed after the burnin period. Default is FALSE.
threads	Number of threads for state simulation. Positive integer (default is 1). Note that parallel computing is only used in the post-correction phase of IS-MCMC and when sampling the states in case of (approximate) Gaussian models.
seed	Seed for the C++ RNG (positive integer).
verbose	If TRUE, prints a progress bar to the console. If missing, defined by rlang::is_interactive. Set to FALSE if number of iterations is less than 50.
particles	A positive integer defining the number of state samples per MCMC iteration for models other than linear-Gaussian models. Ignored if mcmc_type is "approx" or

"ekf". Suitable values depend on the model, the data, mcmc_type and sampling_method. While large values provide more accurate estimates, the run time also increases with respect to to the number of particles, so it is generally a good idea to test the run time firstwith a small number of particles, e.g., less than 100.

mcmc_type — What type of MCMC algorithm should be used for models other than linear-Gaussian models? Possible choices are "pm" for pseudo-marginal MCMC, "da" for delayed acceptance version of PMCMC , "approx" for approximate inference based on the Gaussian approximation of the model, "ekf" for approximate inference using extended Kalman filter (for ssm_nlg), or one of the three importance sampling type weighting schemes: "is3" for simple importance sampling (weight is computed for each MCMC iteration independently), "is2" for jump chain importance sampling type weighting (default), or "is1" for importance sampling type weighting where the number of particles used for weight computations is proportional to the length of the jump chain block.

sampling_method — Method for state sampling when for models other than linear-Gaussian models. If "psi", ψ-APF is used (default). If "spdk", non-sequential importance sampling based on Gaussian approximation is used. If "bsf", bootstrap filter is used. If "ekf", particle filter based on EKF-proposals are used (only for ssm_nlg models).

local_approx — If TRUE (default), Gaussian approximation needed for some of the methods is performed at each iteration. If FALSE, approximation is updated only once at the start of the MCMC using the initial model.

max_iter — Maximum number of iterations used in Gaussian approximation, as a positive integer. Default is 100 (although typically only few iterations are needed).

conv_tol — Positive tolerance parameter used in Gaussian approximation.

iekf_iter — Non-negative integer. The default zero corresponds to normal EKF, whereas iekf_iter > 0 corresponds to iterated EKF with iekf_iter iterations. Used only for models of class ssm_nlg.

L_c, L_f — For ssm_sde models, Positive integer values defining the discretization levels for first and second stages (defined as 2^L). For pseudo-marginal methods ("pm"), maximum of these is used.

Details

For linear-Gaussian models, option "summary" does not simulate states directly but computes the posterior means and variances of states using fast Kalman smoothing. This is slightly faster, more memory efficient and more accurate than calculations based on simulation smoother. In other cases, the means and covariances are computed using the full output of particle filter instead of subsampling one of these as in case of output_type = "full". The states are sampled up to the time point n+1 where n is the length of the input time series i.e. the last values are one-step-ahead predictions. (for predicting further, see ?predict.mcmc_output).

Initial values for the sampling are taken from the model object (model$theta). If you want to continue from previous run, you can reconstruct your original model by plugging in the previously obtained parameters to model$theta, providing the S matrix for the RAM algorithm and setting burnin = 0. See example. Note however, that this is not identical as running all the iterations once,

due to the RNG "discontinuity" and because even without burnin bssm does include "theta_0" i.e. the initial theta in the final chain (even with `burnin=0`).

Value

An object of class `mcmc_output`.

References

Vihola M (2012). Robust adaptive Metropolis algorithm with coerced acceptance rate. Statistics and Computing, 22(5), p 997-1008. https://doi.org/10.1007/s11222-011-9269-5

Vihola, M, Helske, J, Franks, J (2020). Importance sampling type estimators based on approximate marginal Markov chain Monte Carlo. Scand J Statist. 1-38. https://doi.org/10.1111/sjos.12492

Helske J, Vihola M (2021). bssm: Bayesian Inference of Non-linear and Non-Gaussian State Space Models in R. The R Journal (2021) 13:2, 578-589. https://doi.org/10.32614/RJ-2021-103

Vihola, M, Helske, J, Franks, J. Importance sampling type estimators based on approximate marginal Markov chain Monte Carlo. Scand J Statist. 2020; 1-38. https://doi.org/10.1111/sjos.12492

Examples

```
model <- ar1_lg(LakeHuron, rho = uniform(0.5,-1,1),
  sigma = halfnormal(1, 10), mu = normal(500, 500, 500),
  sd_y = halfnormal(1, 10))

mcmc_results <- run_mcmc(model, iter = 2e4)
summary(mcmc_results, return_se = TRUE)

sumr <- summary(mcmc_results, variable = "states")
library("ggplot2")
ggplot(sumr, aes(time, Mean)) +
  geom_ribbon(aes(ymin = `2.5%`, ymax = `97.5%`), alpha = 0.25) +
  geom_line() + theme_bw() +
  geom_point(data = data.frame(Mean = LakeHuron, time = time(LakeHuron)),
    col = 2)

# Continue from the previous run
model$theta[] <- mcmc_results$theta[nrow(mcmc_results$theta), ]
run_more <- run_mcmc(model, S = mcmc_results$S, iter = 1000, burnin = 0)

set.seed(1)
n <- 50
slope <- cumsum(c(0, rnorm(n - 1, sd = 0.001)))
level <- cumsum(slope + c(0, rnorm(n - 1, sd = 0.2)))
y <- rpois(n, exp(level))
poisson_model <- bsm_ng(y,
  sd_level = halfnormal(0.01, 1),
  sd_slope = halfnormal(0.01, 0.1),
  P1 = diag(c(10, 0.1)), distribution = "poisson")

# Note small number of iterations for CRAN checks
mcmc_out <- run_mcmc(poisson_model, iter = 1000, particles = 10,
```

```
  mcmc_type = "da")
summary(mcmc_out, what = "theta", return_se = TRUE)

set.seed(123)
n <- 50
sd_level <- 0.1
drift <- 0.01
beta <- -0.9
phi <- 5

level <- cumsum(c(5, drift + rnorm(n - 1, sd = sd_level)))
x <- 3 + (1:n) * drift + sin(1:n + runif(n, -1, 1))
y <- rnbinom(n, size = phi, mu = exp(beta * x + level))

model <- bsm_ng(y, xreg = x,
  beta = normal(0, 0, 10),
  phi = halfnormal(1, 10),
  sd_level = halfnormal(0.1, 1),
  sd_slope = halfnormal(0.01, 0.1),
  a1 = c(0, 0), P1 = diag(c(10, 0.1)^2),
  distribution = "negative binomial")

# run IS-MCMC
# Note small number of iterations for CRAN checks
fit <- run_mcmc(model, iter = 4000,
  particles = 10, mcmc_type = "is2", seed = 1)

# extract states
d_states <- as.data.frame(fit, variable = "states", time = 1:n)

library("dplyr")
library("ggplot2")

 # compute summary statistics
level_sumr <- d_states |>
  filter(variable == "level") |>
  group_by(time) |>
  summarise(mean = diagis::weighted_mean(value, weight),
    lwr = diagis::weighted_quantile(value, weight,
      0.025),
    upr = diagis::weighted_quantile(value, weight,
      0.975))

# visualize
level_sumr |> ggplot(aes(x = time, y = mean)) +
  geom_line() +
  geom_line(aes(y = lwr), linetype = "dashed", na.rm = TRUE) +
  geom_line(aes(y = upr), linetype = "dashed", na.rm = TRUE) +
  theme_bw() +
  theme(legend.title = element_blank()) +
  xlab("Time") + ylab("Level")

# theta
```

```
d_theta <- as.data.frame(fit, variable = "theta")
ggplot(d_theta, aes(x = value)) +
 geom_density(aes(weight = weight), adjust = 2, fill = "#92f0a8") +
 facet_wrap(~ variable, scales = "free") +
 theme_bw()


# Bivariate Poisson model:

set.seed(1)
x <- cumsum(c(3, rnorm(19, sd = 0.5)))
y <- cbind(
  rpois(20, exp(x)),
  rpois(20, exp(x)))

prior_fn <- function(theta) {
  # half-normal prior using transformation
  dnorm(exp(theta), 0, 1, log = TRUE) + theta # plus jacobian term
}

update_fn <- function(theta) {
  list(R = array(exp(theta), c(1, 1, 1)))
}

model <- ssm_mng(y = y, Z = matrix(1,2,1), T = 1,
  R = 0.1, P1 = 1, distribution = "poisson",
  init_theta = log(0.1),
  prior_fn = prior_fn, update_fn = update_fn)

# Note small number of iterations for CRAN checks
out <- run_mcmc(model, iter = 4000, mcmc_type = "approx")

sumr <- as.data.frame(out, variable = "states") |>
  group_by(time) |> mutate(value = exp(value)) |>
  summarise(mean = mean(value),
    ymin = quantile(value, 0.05), ymax = quantile(value, 0.95))
ggplot(sumr, aes(time, mean)) +
geom_ribbon(aes(ymin = ymin, ymax = ymax),alpha = 0.25) +
geom_line() +
geom_line(data = data.frame(mean = y[, 1], time = 1:20),
  colour = "tomato") +
geom_line(data = data.frame(mean = y[, 2], time = 1:20),
  colour = "tomato") +
theme_bw()
```

---

sim_smoother *Simulation Smoothing*

Description

Function sim_smoother performs simulation smoothing i.e. simulates the states from the conditional distribution $p(\alpha|y, \theta)$ for linear-Gaussian models.

Usage

```
sim_smoother(model, nsim, seed, use_antithetic = TRUE, ...)

## S3 method for class 'lineargaussian'
sim_smoother(
  model,
  nsim = 1,
  seed = sample(.Machine$integer.max, size = 1),
  use_antithetic = TRUE,
  ...
)

## S3 method for class 'nongaussian'
sim_smoother(
  model,
  nsim = 1,
  seed = sample(.Machine$integer.max, size = 1),
  use_antithetic = TRUE,
  ...
)
```

Arguments

model	Model of class bsm_lg, ar1_lg ssm_ulg, or ssm_mlg, or one of the non-gaussian models bsm_ng, ar1_ng svm, ssm_ung, or ssm_mng.
nsim	Number of samples (positive integer). Suitable values depend on the model and the data, and while larger values provide more accurate estimates, the run time also increases with respect to to the number of samples, so it is generally a good idea to test the filter first with a small number of samples, e.g., less than 100.
seed	Seed for the C++ RNG (positive integer).
use_antithetic	Logical. If TRUE (default), use antithetic variable for location in simulation smoothing. Ignored for ssm_mng models.
...	Ignored.

Details

For non-Gaussian/non-linear models, the simulation is based on the approximating Gaussian model.

Value

An array containing the generated samples.

Examples

```
# only missing data, simulates from prior
model <- bsm_lg(rep(NA, 25), sd_level = 1,
  sd_y = 1)
# use antithetic variable for location
sim <- sim_smoother(model, nsim = 4, use_antithetic = TRUE, seed = 1)
ts.plot(sim[, 1, ])
cor(sim[, 1, ])
```

---

ssm_mlg *General multivariate linear Gaussian state space models*

Description

Construct an object of class `ssm_mlg` by directly defining the corresponding terms of the model.

Usage

```
ssm_mlg(
  y,
  Z,
  H,
  T,
  R,
  a1 = NULL,
  P1 = NULL,
  init_theta = numeric(0),
  D = NULL,
  C = NULL,
  state_names,
  update_fn = default_update_fn,
  prior_fn = default_prior_fn
)
```

Arguments

`y`	Observations as multivariate time series or matrix with dimensions n x p.
`Z`	System matrix Z of the observation equation as p x m matrix or p x m x n array.
`H`	Lower triangular matrix H of the observation. Either a scalar or a vector of length n.
`T`	System matrix T of the state equation. Either a m x m matrix or a m x m x n array.
`R`	Lower triangular matrix R the state equation. Either a m x k matrix or a m x k x n array.
`a1`	Prior mean for the initial state as a vector of length m.

P1	Prior covariance matrix for the initial state as m x m matrix.
init_theta	Initial values for the unknown hyperparameters theta (i.e. unknown variables excluding latent state variables).
D	Intercept terms for observation equation, given as a p x n matrix.
C	Intercept terms for state equation, given as m x n matrix.
state_names	A character vector defining the names of the states.
update_fn	A function which returns list of updated model components given input vector theta. This function should take only one vector argument which is used to create list with elements named as Z, H, T, R, a1, P1, D, and C, where each element matches the dimensions of the original model It's best to check the internal dimensions with str(model_object) as the dimensions of input arguments can differ from the final dimensions. If any of these components is missing, it is assumed to be constant wrt. theta.
prior_fn	A function which returns log of prior density given input vector theta.

Details

The general multivariate linear-Gaussian model is defined using the following observational and state equations:

$$y_t = D_t + Z_t\alpha_t + H_t\epsilon_t, (\text{observation equation})$$
$$\alpha_{t+1} = C_t + T_t\alpha_t + R_t\eta_t, (\text{transition equation})$$

where $\epsilon_t \sim N(0, I_p)$, $\eta_t \sim N(0, I_k)$ and $\alpha_1 \sim N(a_1, P_1)$ independently of each other. Here p is the number of time series and k is the number of disturbance terms (which can be less than m, the number of states).

The update_fn function should take only one vector argument which is used to create list with elements named as Z, H T, R, a1, P1, D, and C, where each element matches the dimensions of the original model. If any of these components is missing, it is assumed to be constant wrt. theta. Note that while you can input say R as m x k matrix for ssm_mlg, update_fn should return R as m x k x 1 in this case. It might be useful to first construct the model without updating function

Value

An object of class ssm_mlg.

Examples

```
data("GlobalTemp", package = "KFAS")
model_temp <- ssm_mlg(GlobalTemp, H = matrix(c(0.15,0.05,0, 0.05), 2, 2),
  R = 0.05, Z = matrix(1, 2, 1), T = 1, P1 = 10,
  state_names = "temperature",
  # using default values, but being explicit for testing purposes
  D = matrix(0, 2, 1), C = matrix(0, 1, 1))
ts.plot(cbind(model_temp$y, smoother(model_temp)$alphahat), col = 1:3)
```

ssm_mng *General Non-Gaussian State Space Model*

Description

Construct an object of class ssm_mng by directly defining the corresponding terms of the model.

Usage

```
ssm_mng(
  y,
  Z,
  T,
  R,
  a1 = NULL,
  P1 = NULL,
  distribution,
  phi = 1,
  u,
  init_theta = numeric(0),
  D = NULL,
  C = NULL,
  state_names,
  update_fn = default_update_fn,
  prior_fn = default_prior_fn
)
```

Arguments

y	Observations as multivariate time series or matrix with dimensions n x p.
Z	System matrix Z of the observation equation as p x m matrix or p x m x n array.
T	System matrix T of the state equation. Either a m x m matrix or a m x m x n array.
R	Lower triangular matrix R the state equation. Either a m x k matrix or a m x k x n array.
a1	Prior mean for the initial state as a vector of length m.
P1	Prior covariance matrix for the initial state as m x m matrix.
distribution	A vector of distributions of the observed series. Possible choices are "poisson", "binomial", "negative binomial", "gamma", and "gaussian".
phi	Additional parameters relating to the non-Gaussian distributions. For negative binomial distribution this is the dispersion term, for gamma distribution this is the shape parameter, for Gaussian this is standard deviation, and for other distributions this is ignored.

u	A matrix of positive constants for non-Gaussian models (of same dimensions as y). For Poisson, gamma, and negative binomial distribution, this corresponds to the offset term. For binomial, this is the number of trials (and as such should be integer(ish)).
init_theta	Initial values for the unknown hyperparameters theta (i.e. unknown variables excluding latent state variables).
D	Intercept terms for observation equation, given as p x n matrix.
C	Intercept terms for state equation, given as m x n matrix.
state_names	A character vector defining the names of the states.
update_fn	A function which returns list of updated model components given input vector theta. This function should take only one vector argument which is used to create list with elements named as Z, T, R, a1, P1, D, C, and phi, where each element matches the dimensions of the original model. If any of these components is missing, it is assumed to be constant wrt. theta. It's best to check the internal dimensions with str(model_object) as the dimensions of input arguments can differ from the final dimensions.
prior_fn	A function which returns log of prior density given input vector theta.

Details

The general multivariate non-Gaussian model is defined using the following observational and state equations:

$$p^i(y_t^i|D_t + Z_t\alpha_t), \text{(observation equation)}$$
$$\alpha_{t+1} = C_t + T_t\alpha_t + R_t\eta_t, \text{(transition equation)}$$

where $\eta_t \sim N(0, I_k)$ and $\alpha_1 \sim N(a_1, P_1)$ independently of each other, and $p^i(y_t|.)$ is either Poisson, binomial, gamma, Gaussian, or negative binomial distribution for each observation series $i = 1, ..., p$. Here k is the number of disturbance terms (which can be less than m, the number of states).

Value

An object of class ssm_mng.

Examples

```
set.seed(1)
n <- 20
x <- cumsum(rnorm(n, sd = 0.5))
phi <- 2
y <- cbind(
  rgamma(n, shape = phi, scale = exp(x) / phi),
  rbinom(n, 10, plogis(x)))

Z <- matrix(1, 2, 1)
T <- 1
R <- 0.5
```

```
a1 <- 0
P1 <- 1

update_fn <- function(theta) {
  list(R = array(theta[1], c(1, 1, 1)), phi = c(theta[2], 1))
}

prior_fn <- function(theta) {
  ifelse(all(theta > 0), sum(dnorm(theta, 0, 1, log = TRUE)), -Inf)
}

model <- ssm_mng(y, Z, T, R, a1, P1, phi = c(2, 1),
  init_theta = c(0.5, 2),
  distribution = c("gamma", "binomial"),
  u = cbind(1, rep(10, n)),
  update_fn = update_fn, prior_fn = prior_fn,
  state_names = "random_walk",
  # using default values, but being explicit for testing purposes
  D = matrix(0, 2, 1), C = matrix(0, 1, 1))

# smoothing based on approximating gaussian model
ts.plot(cbind(y, fast_smoother(model)),
  col = 1:3, lty = c(1, 1, 2))
```

---

ssm_nlg *General multivariate nonlinear Gaussian state space models*

Description

Constructs an object of class ssm_nlg by defining the corresponding terms of the observation and state equation.

Usage

```
ssm_nlg(
  y,
  Z,
  H,
  T,
  R,
  Z_gn,
  T_gn,
  a1,
  P1,
  theta,
  known_params = NA,
  known_tv_params = matrix(NA),
  n_states,
```

```
  n_etas,
  log_prior_pdf,
  time_varying = rep(TRUE, 4),
  state_names = paste0("state", 1:n_states)
)
```

Arguments

y	Observations as multivariate time series (or matrix) of length n.
Z, H, T, R	An external pointers (object of class externalptr) for the C++ functions which define the corresponding model functions.
Z_gn, T_gn	An external pointers (object of class externalptr) for the C++ functions which define the gradients of the corresponding model functions.
a1	Prior mean for the initial state as object of class externalptr
P1	Prior covariance matrix for the initial state as object of class externalptr
theta	Parameter vector passed to all model functions.
known_params	A vector of known parameters passed to all model functions.
known_tv_params	A matrix of known parameters passed to all model functions.
n_states	Number of states in the model (positive integer).
n_etas	Dimension of the noise term of the transition equation (positive integer).
log_prior_pdf	An external pointer (object of class externalptr) for the C++ function which computes the log-prior density given theta.
time_varying	Optional logical vector of length 4, denoting whether the values of Z, H, T, and R vary with respect to time variable (given identical states). If used, this can speed up some computations.
state_names	A character vector containing names for the states.

Details

The nonlinear Gaussian model is defined as

$$y_t = Z(t, \alpha_t, \theta) + H(t, \theta)\epsilon_t, \text{(observation equation)}$$
$$\alpha_{t+1} = T(t, \alpha_t, \theta) + R(t, \theta)\eta_t, \text{(transition equation)}$$

where $\epsilon_t \sim N(0, I_p)$, $\eta_t \sim N(0, I_m)$ and $\alpha_1 \sim N(a_1, P_1)$ independently of each other, and functions Z, H, T, R can depend on α_t and parameter vector θ.

Compared to other models, these general models need a bit more effort from the user, as you must provide the several small C++ snippets which define the model structure. See examples in the vignette and cpp_example_model.

Value

An object of class ssm_nlg.

Examples

```
 # Takes a while on CRAN
set.seed(1)
n <- 50
x <- y <- numeric(n)
y[1] <- rnorm(1, exp(x[1]), 0.1)
for(i in 1:(n-1)) {
 x[i+1] <- rnorm(1, sin(x[i]), 0.1)
 y[i+1] <- rnorm(1, exp(x[i+1]), 0.1)
}

pntrs <- cpp_example_model("nlg_sin_exp")

model_nlg <- ssm_nlg(y = y, a1 = pntrs$a1, P1 = pntrs$P1,
  Z = pntrs$Z_fn, H = pntrs$H_fn, T = pntrs$T_fn, R = pntrs$R_fn,
  Z_gn = pntrs$Z_gn, T_gn = pntrs$T_gn,
  theta = c(log_H = log(0.1), log_R = log(0.1)),
  log_prior_pdf = pntrs$log_prior_pdf,
  n_states = 1, n_etas = 1, state_names = "state")

out <- ekf(model_nlg, iekf_iter = 100)
ts.plot(cbind(x, out$at[1:n], out$att[1:n]), col = 1:3)
```

---

ssm_sde *Univariate state space model with continuous SDE dynamics*

Description

Constructs an object of class `ssm_sde` by defining the functions for the drift, diffusion and derivative of diffusion terms of univariate SDE, as well as the log-density of observation equation. We assume that the observations are measured at integer times (missing values are allowed).

Usage

```
ssm_sde(
  y,
  drift,
  diffusion,
  ddiffusion,
  obs_pdf,
  prior_pdf,
  theta,
  x0,
  positive
)
```

Arguments

y	Observations as univariate time series (or vector) of length n.
drift, diffusion, ddiffusion	An external pointers for the C++ functions which define the drift, diffusion and derivative of diffusion functions of SDE.
obs_pdf	An external pointer for the C++ function which computes the observational log-density given the the states and parameter vector theta.
prior_pdf	An external pointer for the C++ function which computes the prior log-density given the parameter vector theta.
theta	Parameter vector passed to all model functions.
x0	Fixed initial value for SDE at time 0.
positive	If TRUE, positivity constraint is forced by abs in Milstein scheme.

Details

As in case of ssm_nlg models, these general models need a bit more effort from the user, as you must provide the several small C++ snippets which define the model structure. See vignettes for an example and cpp_example_model.

Value

An object of class ssm_sde.

Examples

```
 # Takes a while on CRAN
library("sde")
set.seed(1)
# theta_0 = rho = 0.5
# theta_1 = nu = 2
# theta_2 = sigma = 0.3
x <- sde.sim(t0 = 0, T = 50, X0 = 1, N = 50,
       drift = expression(0.5 * (2 - x)),
       sigma = expression(0.3),
       sigma.x = expression(0))
y <- rpois(50, exp(x[-1]))

# source c++ snippets
pntrs <- cpp_example_model("sde_poisson_OU")

sde_model <- ssm_sde(y, pntrs$drift, pntrs$diffusion,
 pntrs$ddiffusion, pntrs$obs_density, pntrs$prior,
 c(rho = 0.5, nu = 2, sigma = 0.3), 1, positive = FALSE)

est <- particle_smoother(sde_model, L = 12, particles = 500)

ts.plot(cbind(x, est$alphahat,
  est$alphahat - 2*sqrt(c(est$Vt)),
  est$alphahat + 2*sqrt(c(est$Vt))),
```

```
  col = c(2, 1, 1, 1), lty = c(1, 1, 2, 2))


# Takes time with finer mesh, parallelization with IS-MCMC helps a lot
out <- run_mcmc(sde_model, L_c = 4, L_f = 8,
  particles = 50, iter = 2e4,
  threads = 4L)
```

---

ssm_ulg *General univariate linear-Gaussian state space models*

Description

Construct an object of class ssm_ulg by directly defining the corresponding terms of the model.

Usage

```
ssm_ulg(
  y,
  Z,
  H,
  T,
  R,
  a1 = NULL,
  P1 = NULL,
  init_theta = numeric(0),
  D = NULL,
  C = NULL,
  state_names,
  update_fn = default_update_fn,
  prior_fn = default_prior_fn
)
```

Arguments

y	Observations as time series (or vector) of length n.
Z	System matrix Z of the observation equation. Either a vector of length m, a m x n matrix, or object which can be coerced to such.
H	A vector of standard deviations. Either a scalar or a vector of length n.
T	System matrix T of the state equation. Either a m x m matrix or a m x m x n array, or object which can be coerced to such.
R	Lower triangular matrix R the state equation. Either a m x k matrix or a m x k x n array, or object which can be coerced to such.
a1	Prior mean for the initial state as a vector of length m.

P1	Prior covariance matrix for the initial state as m x m matrix.
init_theta	Initial values for the unknown hyperparameters theta (i.e. unknown variables excluding latent state variables).
D	Intercept terms D_t for the observations equation, given as a scalar or vector of length n.
C	Intercept terms C_t for the state equation, given as a m times 1 or m times n matrix.
state_names	A character vector defining the names of the states.
update_fn	A function which returns list of updated model components given input vector theta. This function should take only one vector argument which is used to create list with elements named as Z, H, T, R, a1, P1, D, and C, where each element matches the dimensions of the original model It's best to check the internal dimensions with str(model_object) as the dimensions of input arguments can differ from the final dimensions. If any of these components is missing, it is assumed to be constant wrt. theta.
prior_fn	A function which returns log of prior density given input vector theta.

Details

The general univariate linear-Gaussian model is defined using the following observational and state equations:

$$y_t = D_t + Z_t\alpha_t + H_t\epsilon_t, \text{(observation equation)}$$
$$\alpha_{t+1} = C_t + T_t\alpha_t + R_t\eta_t, \text{(transition equation)}$$

where $\epsilon_t \sim N(0, 1)$, $\eta_t \sim N(0, I_k)$ and $\alpha_1 \sim N(a_1, P_1)$ independently of each other. Here k is the number of disturbance terms which can be less than m, the number of states.

The update_fn function should take only one vector argument which is used to create list with elements named as Z, H T, R, a1, P1, D, and C, where each element matches the dimensions of the original model. If any of these components is missing, it is assumed to be constant wrt. theta. Note that while you can input say R as m x k matrix for ssm_ulg, update_fn should return R as m x k x 1 in this case. It might be useful to first construct the model without updating function and then check the expected structure of the model components from the output.

Value

An object of class ssm_ulg.

Examples

```
# Regression model with time-varying coefficients
set.seed(1)
n <- 100
x1 <- rnorm(n)
x2 <- rnorm(n)
b1 <- 1 + cumsum(rnorm(n, sd = 0.5))
b2 <- 2 + cumsum(rnorm(n, sd = 0.1))
```

```
y <- 1 + b1 * x1 + b2 * x2 + rnorm(n, sd = 0.1)

Z <- rbind(1, x1, x2)
H <- 0.1
T <- diag(3)
R <- diag(c(0, 1, 0.1))
a1 <- rep(0, 3)
P1 <- diag(10, 3)

# updates the model given the current values of the parameters
update_fn <- function(theta) {
  R <- diag(c(0, theta[1], theta[2]))
  dim(R) <- c(3, 3, 1)
  list(R = R, H = theta[3])
}
# prior for standard deviations as half-normal(1)
prior_fn <- function(theta) {
  if(any(theta < 0)) {
    log_p <- -Inf
  } else {
    log_p <- sum(dnorm(theta, 0, 1, log = TRUE))
  }
  log_p
}

model <- ssm_ulg(y, Z, H, T, R, a1, P1,
  init_theta = c(1, 0.1, 0.1),
  update_fn = update_fn, prior_fn = prior_fn,
  state_names = c("level", "b1", "b2"),
  # using default values, but being explicit for testing purposes
  C = matrix(0, 3, 1), D = numeric(1))

out <- run_mcmc(model, iter = 5000)
out
sumr <- summary(out, variable = "state", times = 1:n)
sumr$true <- c(b1, b2, rep(1, n))
library(ggplot2)
ggplot(sumr, aes(x = time, y = Mean)) +
geom_ribbon(aes(ymin = `2.5%`, ymax = `97.5%`), alpha = 0.5) +
geom_line() +
geom_line(aes(y = true), colour = "red") +
facet_wrap(~ variable, scales = "free") +
theme_bw()

# Perhaps easiest way to construct a general SSM for bssm is to use the
# model building functionality of KFAS:
library("KFAS")

model_kfas <- SSModel(log(drivers) ~ SSMtrend(1, Q = 5e-4)+
  SSMseasonal(period = 12, sea.type = "trigonometric", Q = 0) +
 log(PetrolPrice) + law, data = Seatbelts, H = 0.005)

# use as_bssm function for conversion, kappa defines the
```

```
# prior variance for diffuse states
model_bssm <- as_bssm(model_kfas, kappa = 100)

# define updating function for parameter estimation
# we can use SSModel and as_bssm functions here as well
# (for large model it is more efficient to do this
# "manually" by constructing only necessary matrices,
# i.e., in this case  a list with H and Q)

prior_fn <- function(theta) {
  if(any(theta < 0)) -Inf else sum(dnorm(theta, 0, 0.1, log = TRUE))
}

update_fn <- function(theta) {

  model_kfas <- SSModel(log(drivers) ~ SSMtrend(1, Q = theta[1]^2)+
    SSMseasonal(period = 12,
      sea.type = "trigonometric", Q = theta[2]^2) +
    log(PetrolPrice) + law, data = Seatbelts, H = theta[3]^2)

  # the bssm_model object is essentially list so this is fine
  as_bssm(model_kfas, kappa = 100, init_theta = init_theta,
    update_fn = update_fn, prior_fn = prior_fn)
}

init_theta <- rep(1e-2, 3)
names(init_theta) <- c("sd_level", "sd_seasonal", "sd_y")

model_bssm <- update_fn(init_theta)


out <- run_mcmc(model_bssm, iter = 10000, burnin = 5000)
out

# Above the regression coefficients are modelled as
# time-invariant latent states.
# Here is an alternative way where we use variable D so that the
# coefficients are part of parameter vector theta. Note however that the
# first option often preferable in order to keep the dimension of theta low.

updatefn2 <- function(theta) {
  # note no PetrolPrice or law variables here
  model_kfas2 <- SSModel(log(drivers) ~ SSMtrend(1, Q = theta[1]^2)+
    SSMseasonal(period = 12, sea.type = "trigonometric", Q = theta[2]^2),
    data = Seatbelts, H = theta[3]^2)

  X <- model.matrix(~ -1 + law + log(PetrolPrice), data = Seatbelts)
  D <- t(X %*% theta[4:5])
  as_bssm(model_kfas2, D = D, kappa = 100)
}
prior2 <- function(theta) {
 if(any(theta[1:3] < 0)) {
  -Inf
```

```
 } else {
   sum(dnorm(theta[1:3], 0, 0.1, log = TRUE)) +
   sum(dnorm(theta[4:5], 0, 10, log = TRUE))
 }
}
init_theta <- c(rep(1e-2, 3), 0, 0)
names(init_theta) <- c("sd_level", "sd_seasonal", "sd_y", "law", "Petrol")
model_bssm2 <- updatefn2(init_theta)
model_bssm2$theta <- init_theta
model_bssm2$prior_fn <- prior2
model_bssm2$update_fn <- updatefn2

out2 <- run_mcmc(model_bssm2, iter = 10000, burnin = 5000)
out2
```

---

ssm_ung *General univariate non-Gaussian state space model*

Description

Construct an object of class ssm_ung by directly defining the corresponding terms of the model.

Usage

```
ssm_ung(
  y,
  Z,
  T,
  R,
  a1 = NULL,
  P1 = NULL,
  distribution,
  phi = 1,
  u,
  init_theta = numeric(0),
  D = NULL,
  C = NULL,
  state_names,
  update_fn = default_update_fn,
  prior_fn = default_prior_fn
)
```

Arguments

y	Observations as time series (or vector) of length n.
Z	System matrix Z of the observation equation. Either a vector of length m, a m x n matrix, or object which can be coerced to such.

T	System matrix T of the state equation. Either a m x m matrix or a m x m x n array, or object which can be coerced to such.
R	Lower triangular matrix R the state equation. Either a m x k matrix or a m x k x n array, or object which can be coerced to such.
a1	Prior mean for the initial state as a vector of length m.
P1	Prior covariance matrix for the initial state as m x m matrix.
distribution	Distribution of the observed time series. Possible choices are "poisson", "binomial", "gamma", and "negative binomial".
phi	Additional parameter relating to the non-Gaussian distribution. For negative binomial distribution this is the dispersion term, for gamma distribution this is the shape parameter, and for other distributions this is ignored. Should an object of class bssm_prior or a positive scalar.
u	A vector of positive constants for non-Gaussian models. For Poisson, gamma, and negative binomial distribution, this corresponds to the offset term. For binomial, this is the number of trials.
init_theta	Initial values for the unknown hyperparameters theta (i.e. unknown variables excluding latent state variables).
D	Intercept terms D_t for the observations equation, given as a scalar or vector of length n.
C	Intercept terms C_t for the state equation, given as a m times 1 or m times n matrix.
state_names	A character vector defining the names of the states.
update_fn	A function which returns list of updated model components given input vector theta. This function should take only one vector argument which is used to create list with elements named as Z, T, R, a1, P1, D, C, and phi, where each element matches the dimensions of the original model. If any of these components is missing, it is assumed to be constant wrt. theta. It's best to check the internal dimensions with str(model_object) as the dimensions of input arguments can differ from the final dimensions.
prior_fn	A function which returns log of prior density given input vector theta.

Details

The general univariate non-Gaussian model is defined using the following observational and state equations:

$$p(y_t|D_t + Z_t\alpha_t), \text{(observation equation)}$$
$$\alpha_{t+1} = C_t + T_t\alpha_t + R_t\eta_t, \text{(transition equation)}$$

where $\eta_t \sim N(0, I_k)$ and $\alpha_1 \sim N(a_1, P_1)$ independently of each other, and $p(y_t|.)$ is either Poisson, binomial, gamma, or negative binomial distribution. Here k is the number of disturbance terms which can be less than m, the number of states.

The update_fn function should take only one vector argument which is used to create list with elements named as Z, phi T, R, a1, P1, D, and C, where each element matches the dimensions of the

original model. If any of these components is missing, it is assumed to be constant wrt. theta. Note that while you can input say R as m x k matrix for ssm_ung, update_fn should return R as m x k x 1 in this case. It might be useful to first construct the model without updating function and then check the expected structure of the model components from the output.

Value

An object of class ssm_ung.

Examples

```
data("drownings", package = "bssm")
model <- ssm_ung(drownings[, "deaths"], Z = 1, T = 1, R = 0.2,
  a1 = 0, P1 = 10, distribution = "poisson", u = drownings[, "population"])

# approximate results based on Gaussian approximation
out <- smoother(model)
ts.plot(cbind(model$y / model$u, exp(out$alphahat)), col = 1:2)
```

---

suggest_N *Suggest Number of Particles for ψ-APF Post-correction*

Description

Function estimate_N estimates suitable number particles needed for accurate post-correction of approximate MCMC.

Usage

```
suggest_N(
  model,
  theta,
  candidates = seq(10, 100, by = 10),
  replications = 100,
  seed = sample(.Machine$integer.max, size = 1)
)
```

Arguments

model	Model of class nongaussian or ssm_nlg.
theta	A vector of theta corresponding to the model, at which point the standard deviation of the log-likelihood is computed. Typically MAP estimate from the (approximate) MCMC run. Can also be an output from run_mcmc which is then used to compute the MAP estimate of theta.
candidates	A vector of positive integers containing the candidate number of particles to test. Default is seq(10, 100, by = 10).
replications	Positive integer, how many replications should be used for computing the standard deviations? Default is 100.
seed	Seed for the C++ RNG (positive integer).

Details

Function `suggest_N` estimates the standard deviation of the logarithm of the post-correction weights at approximate MAP of theta, using various particle sizes and suggest smallest number of particles which still leads standard deviation less than 1. Similar approach was suggested in the context of pseudo-marginal MCMC by Doucet et al. (2015), but see also Section 10.3 in Vihola et al (2020).

Value

List with suggested number of particles `N` and matrix containing estimated standard deviations of the log-weights and corresponding number of particles.

References

Doucet, A, Pitt, MK, Deligiannidis, G, Kohn, R (2015). Efficient implementation of Markov chain Monte Carlo when using an unbiased likelihood estimator, Biometrika, 102(2) p. 295-313, https://doi.org/10.1093/biomet/asu075

Vihola, M, Helske, J, Franks, J (2020). Importance sampling type estimators based on approximate marginal Markov chain Monte Carlo. Scand J Statist. 1-38. https://doi.org/10.1111/sjos.12492

Examples

```
set.seed(1)
n <- 300
x1 <- sin((2 * pi / 12) * 1:n)
x2 <- cos((2 * pi / 12) * 1:n)
alpha <- numeric(n)
alpha[1] <- 0
rho <- 0.7
sigma <- 1.2
mu <- 1
for(i in 2:n) {
  alpha[i] <- rnorm(1, mu * (1 - rho) + rho * alpha[i-1], sigma)
}
u <- rpois(n, 50)
y <- rbinom(n, size = u, plogis(0.5 * x1 + x2 + alpha))

ts.plot(y / u)

model <- ar1_ng(y, distribution = "binomial",
  rho = uniform(0.5, -1, 1), sigma = gamma_prior(1, 2, 0.001),
  mu = normal(0, 0, 10),
  xreg = cbind(x1,x2), beta = normal(c(0, 0), 0, 5),
  u = u)

# theta from earlier approximate MCMC run
# out_approx <- run_mcmc(model, mcmc_type = "approx",
#   iter = 5000)
# theta <- out_approx$theta[which.max(out_approx$posterior), ]

theta <- c(rho = 0.64, sigma = 1.16, mu = 1.1, x1 = 0.56, x2 = 1.28)
```

```
estN <- suggest_N(model, theta, candidates = seq(10, 50, by = 10),
  replications = 50, seed = 1)
plot(x = estN$results$N, y = estN$results$sd, type = "b")
estN$N
```

---

summary.mcmc_output *Summary Statistics of Posterior Samples*

Description

This functions returns a data frame containing mean, standard deviations, standard errors, and effective sample size estimates for parameters and states.

Usage

```
## S3 method for class 'mcmc_output'
summary(
  object,
  return_se = FALSE,
  variable = "theta",
  probs = c(0.025, 0.975),
  times,
  states,
  use_times = TRUE,
  method = "sokal",
  ...
)
```

Arguments

object	Output from run_mcmc
return_se	if FALSE (default), computation of standard errors and effective sample sizes is omitted (as they can take considerable time for models with large number of states and time points).
variable	Are the summary statistics computed for either "theta" (default), "states", or "both"?
probs	A numeric vector defining the quantiles of interest. Default is c(0.025, 0.975).
times	A vector of indices. For states, for what time points the summaries should be computed? Default is all, ignored if variable = "theta".
states	A vector of indices. For what states the summaries should be computed?. Default is all, ignored if variable = "theta".
use_times	If TRUE (default), transforms the values of the time variable to match the ts attribute of the input to define. If FALSE, time is based on the indexing starting from 1.

method	Method for computing integrated autocorrelation time. Default is "sokal", other option is "geyer".
...	Ignored.

Details

For IS-MCMC two types of standard errors are reported. SE-IS can be regarded as the square root of independent IS variance, whereas SE corresponds to the square root of total asymptotic variance (see Remark 3 of Vihola et al. (2020)).

Value

If variable is "theta" or "states", a data.frame object. If "both", a list of two data frames.

References

Vihola, M, Helske, J, Franks, J. Importance sampling type estimators based on approximate marginal Markov chain Monte Carlo. Scand J Statist. 2020; 1-38. https://doi.org/10.1111/sjos.12492

Examples

```
data("negbin_model")
summary(negbin_model, return_se = TRUE, method = "geyer")
summary(negbin_model, times = c(1, 200), prob = c(0.05, 0.5, 0.95))
```

---

svm *Stochastic Volatility Model*

Description

Constructs a simple stochastic volatility model with Gaussian errors and first order autoregressive signal. See the main vignette for details.

Usage

```
svm(y, mu, rho, sd_ar, sigma)
```

Arguments

y	A numeric vector or a ts object of observations.
mu	A prior for mu parameter of transition equation. Should be an object of class bssm_prior.
rho	A prior for autoregressive coefficient. Should be an object of class bssm_prior.
sd_ar	A prior for the standard deviation of noise of the AR-process. Should be an object of class bssm_prior.

sigma A prior for sigma parameter of observation equation, internally denoted as phi. Should be an object of class bssm_prior. Ignored if mu is provided. Note that typically parametrization using mu is preferred due to better numerical properties and availability of better Gaussian approximation. Most notably the global approximation approach does not work with sigma parameterization as sigma is not a parameter of the resulting approximate model.

Value

An object of class svm.

Examples

```
data("exchange")
y <- exchange[1:100] # for faster CRAN check
model <- svm(y, rho = uniform(0.98, -0.999, 0.999),
 sd_ar = halfnormal(0.15, 5), sigma = halfnormal(0.6, 2))

obj <- function(pars) {
   -logLik(svm(y,
     rho = uniform(pars[1], -0.999, 0.999),
     sd_ar = halfnormal(pars[2], 5),
     sigma = halfnormal(pars[3], 2)), particles = 0)
}
opt <- optim(c(0.98, 0.15, 0.6), obj,
  lower = c(-0.999, 1e-4, 1e-4),
  upper = c(0.999, 10, 10), method = "L-BFGS-B")
pars <- opt$par
model <- svm(y,
  rho = uniform(pars[1],-0.999,0.999),
  sd_ar = halfnormal(pars[2], 5),
  sigma = halfnormal(pars[3], 2))

# alternative parameterization
model2 <- svm(y, rho = uniform(0.98,-0.999, 0.999),
 sd_ar = halfnormal(0.15, 5), mu = normal(0, 0, 1))

obj2 <- function(pars) {
   -logLik(svm(y,
     rho = uniform(pars[1], -0.999, 0.999),
     sd_ar = halfnormal(pars[2], 5),
     mu = normal(pars[3], 0, 1)), particles = 0)
}
opt2 <- optim(c(0.98, 0.15, 0), obj2, lower = c(-0.999, 1e-4, -Inf),
  upper = c(0.999, 10, Inf), method = "L-BFGS-B")
pars2 <- opt2$par
model2 <- svm(y,
  rho = uniform(pars2[1],-0.999,0.999),
  sd_ar = halfnormal(pars2[2], 5),
  mu = normal(pars2[3], 0, 1))

# sigma is internally stored in phi
```

```
ts.plot(cbind(model$phi * exp(0.5 * fast_smoother(model)),
  exp(0.5 * fast_smoother(model2))), col = 1:2)
```

---

ukf *Unscented Kalman Filtering*

Description

Function ukf runs the unscented Kalman filter for the given non-linear Gaussian model of class ssm_nlg, and returns the filtered estimates and one-step-ahead predictions of the states α_t given the data up to time t.

Usage

```
ukf(model, alpha = 0.001, beta = 2, kappa = 0)
```

Arguments

model	Model of class ssm_nlg.
alpha	Positive tuning parameter of the UKF. Default is 0.001. Smaller the value, closer the sigma point are to the mean of the state.
beta	Non-negative tuning parameter of the UKF. The default value is 2, which is optimal for Gaussian states.
kappa	Non-negative tuning parameter of the UKF, which also affects the spread of sigma points. Default value is 0.

Value

List containing the log-likelihood, one-step-ahead predictions at and filtered estimates att of states, and the corresponding variances Pt and Ptt.

Examples

```
 # Takes a while on CRAN
set.seed(1)
mu <- -0.2
rho <- 0.7
sigma_y <- 0.1
sigma_x <- 1
x <- numeric(50)
x[1] <- rnorm(1, mu, sigma_x / sqrt(1 - rho^2))
for(i in 2:length(x)) {
  x[i] <- rnorm(1, mu * (1 - rho) + rho * x[i - 1], sigma_x)
}
y <- rnorm(50, exp(x), sigma_y)

pntrs <- cpp_example_model("nlg_ar_exp")
```

```
model_nlg <- ssm_nlg(y = y, a1 = pntrs$a1, P1 = pntrs$P1,
  Z = pntrs$Z_fn, H = pntrs$H_fn, T = pntrs$T_fn, R = pntrs$R_fn,
  Z_gn = pntrs$Z_gn, T_gn = pntrs$T_gn,
  theta = c(mu= mu, rho = rho,
    log_sigma_x = log(sigma_x), log_sigma_y = log(sigma_y)),
  log_prior_pdf = pntrs$log_prior_pdf,
  n_states = 1, n_etas = 1, state_names = "state")

out_iekf <- ekf(model_nlg, iekf_iter = 5)
out_ukf <- ukf(model_nlg, alpha = 0.01, beta = 2, kappa = 1)
ts.plot(cbind(x, out_iekf$att, out_ukf$att), col = 1:3)
```

`uniform_prior` *Prior objects for bssm models*

Description

These simple objects of class `bssm_prior` are used to construct a prior distributions for the hyperparameters theta for some of the model objects of `bssm` package. Note that these priors do not include the constant terms as they do not affect the sampling.

Usage

```
uniform_prior(init, min, max)

uniform(init, min, max)

halfnormal_prior(init, sd)

halfnormal(init, sd)

normal_prior(init, mean, sd)

normal(init, mean, sd)

tnormal_prior(init, mean, sd, min = -Inf, max = Inf)

tnormal(init, mean, sd, min = -Inf, max = Inf)

gamma_prior(init, shape, rate)

gamma(init, shape, rate)
```

Arguments

`init`	Initial value for the parameter, used in initializing the model components and as a starting values in MCMC.

min	Lower bound of the uniform and truncated normal prior.
max	Upper bound of the uniform and truncated normal prior.
sd	Positive value defining the standard deviation of the (underlying i.e. non-truncated) Normal distribution.
mean	Mean of the Normal prior.
shape	Positive shape parameter of the Gamma prior.
rate	Positive rate parameter of the Gamma prior.

Details

Currently supported priors are

- uniform prior (`uniform()`) with a probability density function (pdf) defined as $\frac{1}{max-min}$ for $min < theta < max$.
- normal (`normal()`), a normal distribution parameterized via mean and standard deviation, i.e. N(mean, sd^2).
- truncated normal distribution (`tnormal()`), a normal distribution with known truncation points (from below and/or above). Ignoring the scaling factors, this corresponds to the pdf of N(mean, sd^2) when $min < theta < max$ and zero otherwise.
- half-normal (`halfnormal()`) with a pdf matching the pdf of the truncated normal distribution with min=0 and max=inf.
- gamma (`gamma`), a gamma distribution with shape and rate parameterization.

All parameters are vectorized so for regression coefficient vector beta you can define prior for example as `normal(0, 0, c(10, 20))`.

For the general exponential models, i.e. models built with the `ssm_ulg`, `ssm_ung`, `ssm_mlg`, and `ssm_mng`, you can define arbitrary priors by defining the `prior_fn` function, which takes the one argument, `theta`, corresponding to the hyperparameter vector of the model, and returns a log-density of the (joint) prior (see the R Journal paper and e.g. `ssm_ulg` for examples). Similarly, the priors for the non-linear models (`ssm_nlg`) and SDE models (`ssm_sde`) are constructed via C++ snippets (see the vignettes for details).

The longer name versions of the prior functions with `_prior` ending are identical with shorter versions and they are available only to avoid clash with R's primitive function `gamma` (other long prior names are just for consistent naming).

Value

object of class `bssm_prior` or `bssm_prior_list` in case of multiple priors (i.e. multiple regression coefficients).

Examples

```
# create uniform prior on [-1, 1] for one parameter with initial value 0.2:
uniform(init = 0.2, min = -1.0, max = 1.0)
# two normal priors at once i.e. for coefficients beta:
normal(init = c(0.1, 2.5), mean = 0.1, sd = c(1.5, 2.8))
# Gamma prior (not run because autotest tests complain)
```

```
# gamma(init = 0.1, shape = 2.5, rate = 1.1)
# Same as
gamma_prior(init = 0.1, shape = 2.5, rate = 1.1)
# Half-normal
halfnormal(init = 0.01, sd = 0.1)
# Truncated normal
tnormal(init = 5.2, mean = 5.0, sd = 3.0, min = 0.5, max = 9.5)


# Further examples for diagnostic purposes:
uniform(c(0, 0.2), c(-1.0, 0.001), c(1.0, 1.2))
normal(c(0, 0.2), c(-1.0, 0.001), c(1.0, 1.2))
tnormal(c(2, 2.2), c(-1.0, 0.001), c(1.0, 1.2), c(1.2, 2), 3.3)
halfnormal(c(0, 0.2), c(1.0, 1.2))
# not run because autotest bug
# gamma(c(0.1, 0.2), c(1.2, 2), c(3.3, 3.3))

# longer versions:
uniform_prior(init = c(0, 0.2), min = c(-1.0, 0.001), max = c(1.0, 1.2))
normal_prior(init = c(0, 0.2), mean = c(-1.0, 0.001), sd = c(1.0, 1.2))
tnormal_prior(init = c(2, 2.2), mean = c(-1.0, 0.001), sd = c(1.0, 1.2),
  min = c(1.2, 2), max = 3.3)
halfnormal_prior(init = c(0, 0.2), sd = c(1.0, 1.2))
gamma_prior(init = c(0.1, 0.2), shape = c(1.2, 2), rate = c(3.3, 3.3))
```

Index

∗ **datasets**
drownings, 20
exchange, 26
negbin_model, 35
negbin_series, 36
poisson_series, 40

ar1_lg, 3
ar1_ng, 4
as.data.frame.mcmc_output, 6
as_bssm, 7
as_draws *(*as_draws_df.mcmc_output*)*, 8
as_draws.mcmc_output *(*as_draws_df.mcmc_output*)*, 8
as_draws_df *(*as_draws_df.mcmc_output*)*, 8
as_draws_df.mcmc_output, 8
asymptotic_var, 9

bayesplot::mcmc_combo, *39, 40*
bootstrap_filter, 10, *33*
bsm_lg, 12
bsm_ng, 14
bssm, 17
bssm-package *(*bssm*)*, 17
bssm_prior *(*uniform_prior*)*, 75
bssm_prior_list *(*uniform_prior*)*, 75

check_diagnostics, 19, *40*
cpp_example_model, 20

drownings, 20

ekf, 21
ekf_fast_smoother *(*ekf_smoother*)*, 22
ekf_smoother, 22
ekpf_filter, 23
estimate_ess, 25
exchange, 26
expand_sample, 26

fast_smoother, 27
fitted.mcmc_output, 28

gamma *(*uniform_prior*)*, 75
gamma_prior *(*uniform_prior*)*, 75
gaussian_approx, 29

halfnormal *(*uniform_prior*)*, 75
halfnormal_prior *(*uniform_prior*)*, 75

iact, 30
importance_sample, 31

kfilter, 32

logLik.lineargaussian, 33
logLik.nongaussian *(*logLik.lineargaussian*)*, 33
logLik.ssm_nlg *(*logLik.lineargaussian*)*, 33
logLik.ssm_sde *(*logLik.lineargaussian*)*, 33

negbin_model, 35
negbin_series, 36
normal *(*uniform_prior*)*, 75
normal_prior *(*uniform_prior*)*, 75

particle_smoother, 37
plot.mcmc_output, 39
poisson_series, 40
post_correct, 41
predict *(*predict.mcmc_output*)*, 43
predict.mcmc_output, 43
print.mcmc_output, 46

run_mcmc, *6*, *19*, *27*, *29*, *39*, *44*, *46*, 47

sim_smoother, 53
smoother *(*fast_smoother*)*, 27
ssm_mlg, 55
ssm_mng, 57

ssm_nlg, 59
ssm_sde, 61
ssm_ulg, 63
ssm_ung, 67
suggest_N, 69
summary.mcmc_output, 71
svm, 72

tnormal *(*uniform_prior*)*, 75
tnormal_prior *(*uniform_prior*)*, 75
ts, *72*

ukf, 74
uniform *(*uniform_prior*)*, 75
uniform_prior, 75