Package 'SVDNF'

January 14, 2023

Type Package

Title Discrete Nonlinear Filtering for Stochastic Volatility Models

Version 0.1.3

Author Louis Arsenault-Mahjoubi [aut, cre],
Jean-François Bégin [aut],
Mathieu Boudreault [aut]

Maintainer Louis Arsenault-Mahjoubi <larsenau@sfu.ca>

Description Generates simulated paths from various financial stochastic volatility models with jumps and applies the discrete nonlinear filter (DNF) of Kitagawa (1987) <doi:10.1080/01621459.1987.10478534> to compute likelihood evaluations, filtering distribution estimates, and maximum likelihood parameter estimates.
The algorithm is implemented following the work of Bégin and Boudreault (2021) <doi:10.1080/10618600.2020.1840995>.

License GPL-3

Encoding UTF-8

Imports Rcpp (>= 1.0.9), methods

LinkingTo Rcpp

NeedsCompilation yes

Repository CRAN

Date/Publication 2023-01-14 22:20:02 UTC

R topics documented:

DNF.dynamicsSVM *Discrete Nonlinear Filtering Algorithm for Stochastic Volatility Models*

Description

The DNF function applies the discrete nonlinear filter (DNF) of Kitagawa (1987) as per the implementation of Bégin & Boudreault (2020) to obtain likelihood evaluations and filtering distribution estimates for a wide class of stochastic volatility models.

Usage

```
## S3 method for class 'dynamicsSVM'
DNF(dynamics, data, N = 50, K = 20, R = 1, grids, ...)
```

Arguments

dynamics	A dynamicsSVM object representing the model dynamics to be used by the DNF.
data	A series of asset returns for which we want to run the DNF.
N	Number of nodes in the variance grid.
K	Number of nodes in the jump size grid.
R	Maximum number of jumps used in the numerical integration at each timestep.
grids	Grids to be used for numerical integration by the DNF function. The DNF function creates grids for built-in models. However, this arguments must be provided for custom models. It should contain a list of three sequences: var_mid_points (variance mid-point sequence), j_nums (sequence for the number of jumps), and jump_mid_points (jump mid-point sequence). If there are no variance jumps in the model, set jump_mid_points equal to zero. If there are no jumps in the model, both j_nums and jump_mid_points should be set to zero.
...	Further arguments passed to or from other methods.

Value

log_likelihood	Log-likelihood evaluation based on the DNF.
filter_grid	Grid of dimensions N by T+1 that stores each time-step's filtering distributions (we assume the filtering distribution is uniform at $t = 0$).
likelihoods	Likelihood contribution at each time-step throughout the series.
grids	List of grids used for numerical integration by the DNF.
dynamics	The model dynamics used by the DNF.

References

Bégin, J.F., Boudreault, M. (2021) Likelihood evaluation of jump-diffusion models using deterministic nonlinear filters. *Journal of Computational and Graphical Statistics*, 30(2), 452–466.

Kitagawa, G. (1987) Non-Gaussian state-space modeling of nonstationary time series. *Journal of the American Statistical Association*, 82(400), 1032–1041.

Examples

```
set.seed(1)
# Generate 200 returns from the DuffiePanSingleton model
DuffiePanSingleton_mod <- dynamicsSVM(model = "DuffiePanSingleton")
DuffiePanSingleton_sim <- modelSim(t = 200, dynamics = DuffiePanSingleton_mod)

# Run DNF on the data
dnf_filter <- DNF(data = DuffiePanSingleton_sim$returns,
  dynamics = DuffiePanSingleton_mod)

# Print log-likelihood evaluation.
dnf_filter$log_likelihood

# Using a custom model.
# Here, we define the DuffiePanSingleton model as a custom model
# to get the same log-likelihood found using the built-in option

# Daily observations
h <- 1/252

# Parameter values
mu <- 0.038; kappa <- 3.689; theta <- 0.032
sigma <- 0.446; rho <- -0.745; omega <- 5.125
delta <- 0.03; alpha <- -0.014; rho_z <- -1.809; nu <- 0.004

# Jump compensator
alpha_bar <- exp(alpha + 0.5 * delta^2) / (1 - rho_z * nu) - 1

# Returns drift and diffusion
mu_y <- function(x, mu, alpha_bar, omega, h) {
  return(h * (mu - x / 2 - alpha_bar * omega))
}
mu_y_params <- list(mu, alpha_bar, omega, h)
sigma_y <- function(x, h) {
  return(sqrt(h * pmax(x, 0)))
}
sigma_y_params <- list(h)

# Volatility factor drift and diffusion
mu_x <- function(x, kappa, theta, h) {
  return(x + h * kappa * (theta - pmax(0, x)))
}
mu_x_params <- list(kappa, theta, h)
```

```
sigma_x <- function(x, sigma, h) {
  return(sigma * sqrt(h * pmax(x, 0)))
}
sigma_x_params <- list(sigma, h)

# Jump distribution for the DuffiePanSingleton Model
jump_density <- dpois
jump_dist <- rpois
jump_params <- c(h * omega)

# Create the custom model
custom_mod <- dynamicsSVM(model = 'Custom',
  mu_x = mu_x, mu_y = mu_y, sigma_x = sigma_x, sigma_y = sigma_y,
  mu_x_params = mu_x_params, mu_y_params = mu_y_params,
  sigma_x_params = sigma_x_params, sigma_y_params = sigma_y_params,
  jump_params = jump_params, jump_dist = jump_dist, jump_density = jump_density,
  nu = nu, rho_z = rho_z, rho = rho)
# Define the grid for DNF
N <- 50; R <- 1; K <- 20
var_mid_points <- seq(from = sqrt(0.0000001),
  to = sqrt(theta + (3 + log(N)) * sqrt(0.5 * theta * sigma^2 / kappa)), length = N)^2

j_nums <- seq(from = 0, to = R, by = 1)

jump_mid_points <- seq(from = 0.000001, to = (3 + log(K)) * sqrt(R) * nu, length = K)

grids <- list(var_mid_points = var_mid_points,
  j_nums = j_nums, jump_mid_points = jump_mid_points)

# Run the DNF function with the custom model
dnf_custom <- DNF(data = DuffiePanSingleton_sim$returns, grids = grids,
  dynamics = custom_mod)

# Check if we get the same log-likelihoods
dnf_custom$log_likelihood; dnf_filter$log_likelihood
```

DNFOptim *Discrete Nonlinear Filter Maximum Likelihood Estimation Function*

Description

The DNFOptim function finds maximum likelihood estimates for stochastic volatility models parameters using the DNF function.

Usage

```
DNFOptim(dynamics, data, N = 50, K = 20, R = 1,
  grids = 'Default',
  rho = 0, delta = 0, alpha = 0, rho_z = 0, nu = 0, jump_params_list = "dummy",
  ...)
```

Arguments

dynamics	A dynamicsSVM object representing the model dynamics to be used by the optimizer to find maximum likelihood parameter estimates.
data	A series of asset returns for which we want to find maximum likelihood estimates.
N	Number of nodes in the variance grid.
K	Number of nodes in the jump size grid.
grids	Grids to be used for numerical integration by the DNF function. The DNF function creates grids for built-in models. However, this arguments must be provided for custom models. It should contain a list of three sequences: var_mid_points (variance mid-point sequence), j_nums (sequence for the number of jumps), and jump_mid_points (jump mid-point sequence). If there are no variance jumps in the model, set jump_mid_points equal to zero. If there are no jumps in the model, both j_nums and jump_mid_points should be set to zero.
R	Maximum number of jumps used in the numerical integration at each timestep.
rho, delta, alpha, rho_z, nu	See help(dynamicsSVM) for a description of each of these arguments individually. These arguments should be used only for custom models and can be fixed to a certain value (e.g., rho = -0.75). If they are estimated, they should be set to 'var' (e.g., to estimate rho set rho = 'var') and include it in the vector par to be passed to the optim function. See Note for more details on the order in which custom models should receive parameters.
jump_params_list	List of the names of the arguments in the jump parameter distribution is the order that thye are used by the jump_dist function. This is used by DNFOptim to check for parameters that occur both in the jump_dist function and as arguments in drift or diffusion functions.
...	Further arguments to be passed to the optim function. See Note.

Value

Returns a list obtained from R's optim function. See help(optim) for details about the output.

Note

When passing the initial parameter vector par to the optim function (via ...), the parameters should follow a specific order.

For the PittMalikDoucet model, the parameters should be in the following order: phi, theta, sigma, rho, p, delta, and alpha.

For the DuffiePanSingleton model, the parameters should be in the following order: mu, kappa, theta, sigma, rho, omega, delta, alpha, rho_z, and nu.

All other built-in models can be seen as being nested within these two models (i.e., Heston and Bates models are nested in the DuffiePanSingleton model, while Taylor and TaylorWithLeverage are nested in the PittMalikDoucet model). Their parameters should be

passed in the same order as those in the more general models, minus the parameters not found in these nested models.

For example, the `Taylor` model contains neither jumps nor correlation between volatility and returns innovations. Thus, its three parameters are passed in the order: `phi`, `theta`, and `sigma`.

When `models = "Custom"`, parameters should be passed in the following order: `mu_y_params`, `sigma_y_params`, `mu_x_params`, `sigma_x_params`, `rho`, `delta`, `alpha`, `rho_z`, `nu`, and `jump_params`. If an argument is repeated (e.g., both `mu_y_params` and `sigma_y_params` use the same parameter), write it only when it first appears in the custom model order.

References

R Core Team (2019). R: A language and environment for statistical computing. R Foundation for Statistical Computing, Vienna, Austria. URL https://www.R-project.org/.

Examples

```
set.seed(1)

# Generating return data
Taylor_mod <- dynamicsSVM(model = "Taylor", phi = 0.9,
  theta = -7.36, sigma = 0.363)
Taylor_sim <- modelSim(t = 30, dynamics = Taylor_mod, init_vol = -7.36)
plot(Taylor_sim$volatility_factor, type = 'l')
plot(Taylor_sim$returns, type = 'l')

# Initial values and optimization bounds
init_par <- c( 0.7, -5, 0.3)
lower <- c(0.01, -20, 0.01); upper <- c(0.99, 0, 1)

# Running DNFOptim to get MLEs
optim_test <- DNFOptim(data = Taylor_sim$returns,
  dynamics = Taylor_mod,
  par = init_par, lower = lower, upper = upper, method = "L-BFGS-B")

# Parameter estimates
optim_test$par
```

`dynamicsSVM` *Stochastic Volatility Models Dynamics*

Description

`dynamicsSVM` creates stochastic volatility model dynamics by either choosing from a set of built-in model dynamics or using custom drift and diffusion functions, as well as custom jump distributions. See Note for information about how to define custom functions.

Usage

```
dynamicsSVM(mu = 0.038, kappa = 3.689, theta = 0.032, sigma = 0.446,
rho = -0.745, omega = 5.125, delta = 0.03, alpha = -0.014,
rho_z = -1.809, nu = 0.004, p = 0.01, phi = 0.965, h = 1/252,
model = "Heston", mu_x, mu_y, sigma_x, sigma_y,
jump_dist = rpois, jump_density = dpois, jump_params = 0,
mu_x_params, mu_y_params, sigma_x_params, sigma_y_params)
```

Arguments

mu	Annual expected rate of return.
kappa	Variance rate of mean reversion.
theta	Unconditional mean variance.
sigma	Volatility of the variance.
rho	Correlation between the return and the variance noise terms.
omega	Jump arrival intensity for models with Poisson jumps.
delta	Standard deviation of return jumps.
alpha	Mean of return jumps.
rho_z	Pseudo-correlation parameter between return and variance jumps.
nu	Mean for variance jumps.
p	Jump probability for models with Bernoulli jumps.
phi	Volatility persistence parameter.
h	Time interval between observations (e.g., h = 1/252 for daily data).
model	Model used by the discrete nonlinear filter. The options are "Heston", "Bates", "DuffiePanSingleton", "Taylor", "TaylorWithLeverage", "PittMalikDoucet", and "Custom". If model = "Custom", users should pass the drift functions (i.e., mu_x and mu_y), the diffusion functions (i.e., sigma_x and sigma_y), and the jump distribution, (i.e., jump_dist) as well as their parameters to the DNF function. See Examples.
mu_x	Function for variance drift (to be used with a custom model).
mu_y	Function for returns drift (to be used with a custom model).
sigma_x	Function for variance diffusion (to be used with a custom model).
sigma_y	Function for returns diffusion (to be used with a custom model).
jump_dist	Distribution used to generate return or volatility jumps at each timestep (if both types of jumps are in the model, they are assumed to occur simulaneously).
jump_density	Probability mass function used to compute the probability of return or volatility jumps at each timestep (if both types of jumps are in the model, they are assumed to occur simulaneously).
jump_params	List of parameters to be used as arguments in the jump_dist and jump_density function (parameters should be listed in the order that jump_dist uses them).
mu_x_params	List of parameters to be used as arguments in the mu_x function (parameters should be listed in the order that mu_x uses them).

mu_y_params	List of parameters to be used as arguments in the mu_y function (parameters should be listed in the order that mu_y uses them).
sigma_x_params	List of parameters to be used as arguments in the sigma_x function (parameters should be listed in the order that sigma_x uses them).
sigma_y_params	List of parameters to be used as arguments in the sigma_y function (parameters should be listed in the order that sigma_y uses them).

Value

Returns an object of type dynamicsSVM.

Note

Custom functions should have x (the volatility factor) as well as the function's other parameters as arguments.

If the custom function does not use any parameters, one should include an argument called dummy and its parameters as a `list(0)`. For example, for the Taylor model,

```
sigma_y_taylor <- function(x, dummy) { return(exp(x / 2)) }
sigma_y_params <- list(0)
```

It should also be noted that the custom function is a vector for x. This means that users should use vectorized version of functions. For example, `pmax(0,x)` instead of `max(0,x)` as code seen in the Example section below.

Examples

```
# Create a dynamicsSVM object with model DuffiePanSingleton and default parameters
DuffiePanSingleton_mod <- dynamicsSVM(model = "DuffiePanSingleton")

# Here, we define the same DuffiePanSingleton model
# using the custom model option.

# Daily observations
h <- 1/252

# Parameter values
mu <- 0.038; kappa <- 3.689; theta <- 0.032
sigma <- 0.446; rho <- -0.745; omega <- 5.125
delta <- 0.03; alpha <- -0.014; rho_z <- -1.809; nu <- 0.004

# Jump compensator
alpha_bar <- exp(alpha + 0.5 * delta^2) / (1 - rho_z * nu) - 1

# Returns drift and diffusion
mu_y <- function(x, mu, alpha_bar, omega, h) {
  return(h * (mu - x / 2 - alpha_bar * omega))
}
mu_y_params <- list(mu, alpha_bar, omega, h)
sigma_y <- function(x, h) {
  return(sqrt(h * pmax(x, 0)))
}
```

```
sigma_y_params <- list(h)

# Volatility factor drift and diffusion
mu_x <- function(x, kappa, theta, h) {
  return(x + h * kappa * (theta - pmax(0, x)))
}
mu_x_params <- list(kappa, theta, h)

sigma_x <- function(x, sigma, h) {
  return(sigma * sqrt(h * pmax(x, 0)))
}
sigma_x_params <- list(sigma, h)

# Jump distribution for the DuffiePanSingleton Model
jump_density <- dpois
jump_dist <- rpois
jump_params <- c(h * omega)

# Create the custom model
custom_DPS <- dynamicsSVM(model = 'Custom',
  mu_x = mu_x, mu_y = mu_y, sigma_x = sigma_x, sigma_y = sigma_y,
  mu_x_params = mu_x_params, mu_y_params = mu_y_params,
  sigma_x_params = sigma_x_params, sigma_y_params = sigma_y_params,
  jump_params = jump_params, jump_dist = jump_dist, jump_density = jump_density,
  nu = nu, rho_z = rho_z)
```

`modelSim.dynamicsSVM` *Simulation from Stochastic Volatility Models with Jumps*

Description

The `modelSim` function generates returns and variances for a wide class of stochastic volatility models.

Usage

```
## S3 method for class 'dynamicsSVM'
modelSim(dynamics, t, init_vol = 0.032, ...)
```

Arguments

`dynamics`	A dynamicsSVM object representing the model dynamics to be used for simulating data.
`t`	Number of observations to be simulated.
`init_vol`	Initial value of the volatility factor (e.i., value of `x_0`).
`...`	Further arguments passed to or from other methods.

Value

`volatility_factor`
: Vector of the instantaneous volatility factor values generated by the `modelSim` function.

`returns`
: Vector of the returns generated by the `modelSim` function.

Examples

```
set.seed(1)
# Generate 250 returns from the DuffiePanSingleton model
DuffiePanSingleton_mod <- dynamicsSVM(model = "DuffiePanSingleton")
DuffiePanSingleton_sim <- modelSim(t = 200, dynamics = DuffiePanSingleton_mod)

# Plot the volatility factor and returns that were generated
plot(DuffiePanSingleton_sim$volatility_factor, type = 'l',
 main = 'DuffiePanSingleton Model Simulated Volatility Factor', ylab = 'Volatility Factor')

plot(DuffiePanSingleton_sim$returns, type = 'l',
  main = 'DuffiePanSingleton Model Simulated Returns', ylab = 'Returns')

# Generate 250 steps from a custom model
# Set parameters
kappa <- 100; theta <- 0.05; sigma <- 2.3; h <- 1/252 ; mu <- 0.04
rho <- -0.8; omega <- 5; alpha <- -0.025; nu <- 0.01; rho_z <- -1; delta <- 0.025
# Jump compensator
alpha_bar <- exp(alpha + 0.5 * delta^2) / (1 - rho_z * nu) - 1

# Define returns drift and diffusion functions
# Returns drift and diffusion
mu_y <- function(x, mu, alpha_bar, omega, h){
  return(h * (mu - x / 2 - alpha_bar * omega))
}
mu_y_params <- list(mu, alpha_bar, omega , h)
sigma_y <- function(x, h, sigma){
  return(sigma * sqrt(h) * pmax(x,0))
}
sigma_y_params <- list(h, sigma)

# Volatility factor drift and diffusion functions
mu_x <- function(x, h, kappa, theta){
  return(x + h * kappa * pmax(0,x) * (theta - pmax(0,x)))
}
mu_x_params <- list(h, kappa, theta)

sigma_x <- function(x, sigma, h){
  return(sigma * sqrt(h) * pmax(0,x))
}
sigma_x_params <- list(sigma, h)

# Include simultaneous return and volatility factor jumps
# based on the Poisson distribution for jump times
jump_dist <- rpois
```

```
jump_params <- list(omega * h)
custom_mod <- dynamicsSVM(model = "Custom", mu_x = mu_x, mu_y = mu_y,
  sigma_x = sigma_x, sigma_y = sigma_y,
  mu_x_params = mu_x_params, mu_y_params = mu_y_params,
  sigma_x_params = sigma_x_params, sigma_y_params = sigma_y_params,
  jump_dist = jump_dist, jump_params = jump_params,
  nu = nu, rho_z = rho_z, omega = omega, alpha = alpha, delta = delta)
custom <- modelSim(t = 250, dynamics = custom_mod)

plot(custom$volatility_factor, type = 'l',
  main = 'Custom Model Simulated Volatility Factor', ylab = 'Volatility Factor')
plot(custom$returns, type = 'l',
  main = 'Custom Model Simulated Returns', ylab = 'Returns')
```

`plot.SVDNF` *DNF Filtering Distribution Plot Function*

Description

This function plots the median of the filtering and prediction distributions estimated from the `DNF` function along with user-selected upper and lower percentiles.

Usage

```
## S3 method for class 'SVDNF'
plot(x, lower_p = 0.05, upper_p = 0.95, tlim = 'default',
  location = 'topright', ...)
```

Arguments

`x`	An `SVDNF` object. The `plot` plots the median and selected percentiles from the filtering distribution.
`lower_p`	Lower percentile of the filtering distribution to plot.
`upper_p`	Upper percentile of the filtering distribution to plot.
`tlim`	The `plot` function plots the filtering and prediction distributions over the interval `tlim`. For example to plot the first `500` steps, set `tlim = c(1, 500)`. By default, filtering and prediction distribution estimates for every step in the time-series are generated. If `tlim` is set to a single number (e.g., `tlim = c(5)`), `plot` graphs the estimated probability density functions of the filtering (in magenta) and prediction (in blue) distributions at that timestep.
`location`	Location keyword passed to the `legend` function to determine the location of the legend. The keyword should be selected from the list `"bottomright"`, `"bottom"`, `"bottomleft"`, `"left"`, `"topleft"`, `"top"`, `"topright"`, `"right"`, and `"center"`.
`...`	Other parameters to be passed through to function.

Value

No return value; this function generates a plot with the median of the volatility factor obtained from the filtering distribution as well as its upper and lower percentiles from `lower_p` and `upper_p`.

Examples

```
set.seed(1)
# Generate 500 returns from the Bates model.
Bates_mod <- dynamicsSVM(model = "Bates")
Bates_sim <- modelSim(t = 500, dynamics = Bates_mod)

# Runs DNF on the data.
dnf_filter <- DNF(data = Bates_sim$returns, dynamics = Bates_mod)

# Plot whole interval (default)
plot(dnf_filter, ylim = c(0, 0.15), type = 'l',
  ylab = "Volatility Factor", xlab = 'Time')

# Plot specific interval
tlim <- c(100,350)
plot(dnf_filter, ylim = c(0, 0.15), type = 'l',
  ylab = "Volatility Factor", xlab = 'Time', tlim = tlim)

# Plot specific point
tlim <- c(100)
plot(dnf_filter, ylim = c(0, 0.15), type = 'l',
  ylab = "Volatility Factor", xlab = 'Time', tlim = tlim)
```

Index