Package 'hpa'

April 14, 2026

Type Package

Title Distributions Hermite Polynomial Approximation

Version 1.3.4

Date 2026-04-15

Description Multivariate conditional and marginal densities, moments, cumulative distribution functions as well as binary choice and sample selection models based on the Hermite polynomial approximation which was proposed and described by A. Gallant and D. W. Nychka (1987) <doi:10.2307/1913241>.

License GPL-3

Imports Rcpp (>= 1.1.1), RcppParallel (>= 5.1.11-2)

LinkingTo Rcpp, RcppArmadillo, RcppParallel

RoxygenNote 7.3.2

Encoding UTF-8

Suggests ggplot2, mvtnorm, titanic, sampleSelection, GA (>= 3.2)

NeedsCompilation yes

SystemRequirements GNU make

Author Bogdan Potanin [aut, cre, ctb],
Sofiia Dolgikh [ctb]

Maintainer Bogdan Potanin <bogdanpotanin@gmail.com>

Repository CRAN

Date/Publication 2026-04-14 09:50:02 UTC

Contents

bspline *B-splines generation, estimation and combination*

Description

Function bsplineGenerate generates a list of all basis splines with appropriate knots vector and degree. Function bsplineComb allows to get linear combinations of these b-splines with particular weights. Function bsplineEstimate estimates the spline at points x. The structure of this spline should be provided via m and knots arguments.

Usage

```
bsplineGenerate(knots, degree, is_names = TRUE)

bsplineEstimate(x, m, knots)

bsplineComb(splines, weights)
```

Arguments

knots	a numeric vector sorted in ascending order representing the knots of the spline.
degree	positive integer representing degree of the spline.
is_names	logical; if TRUE (default) then rows and columns of the spline matrices will have a names. Set it to FALSE in order to get notable speed boost.
x	numeric vector representing the points at which the spline should be estimated.
m	numeric matrix whose rows correspond to the spline intervals while columns represent the variables powers. Therefore the element in i-th row and j-th column represents the coefficient associated with the variable that 1) belongs to the i-th interval i.e. between the i-th and (i + 1)-th knots 2) raised to the power of (j - 1).
splines	list being returned by the bsplineGenerate function or a manually constructed list with b-splines knots and matrices entries.
weights	numeric vector of the same length as splines.

Details

In contrast to bs function bsplineGenerate generates a splines basis in a form of a list containing information concerning these b-splines structure. In order to evaluate one of these b-splines at particular points bsplineEstimate function should be applied.

Value

Function bsplineGenerate returns a list. Each element of this list is a list containing the following information concerning b-spline structure:

- knots - knots vector of the b-spline.

- `m` - matrix representing polynomial coefficients for each interval of the spline in the same manner as for `m` argument (see this argument description above).
- `ind` - index of the b-spline.

Function `bsplineComb` returns a list with the following arguments:

- `knots` - knots vector of the `splines`.
- `m` - linear combination of the `splines` matrices; coefficients of this linear combination are given via `weights` argument.

Function `bsplineGenerate` returns a numeric vector of values being calculated at points `x` via splines with `knots` vector and matrix `m`.

Examples

```
# Let's generate all b-splines of degree 3 with knot
# vector (-2.1, 1.5, 1.5, 2.2, 3.7, 4.2, 5)
b <- bsplineGenerate(knots = c(-2.1, 1.5, 1.5, 2.2, 3.7, 4.2, 5),
                     degree = 3)

# Get the first of these b-splines
b[[1]]

# Take a linear combination of these splines with
# weights 1.6, -1.2, and 3.2.
b_comb <- bsplineComb(splines = b, weights = c(1.6, -1.2, 3.2))

# Estimate these spline values at points (-3, 0.7, 2.5, 3.8, 10)
b_values <- bsplineEstimate(x = c(-3, 0.7, 2.5, 3.8, 10),
                            knots = b_comb$knots,
                            m = b_comb$m)

# Visualize the spline
s <- seq(from = 0, to = 5, length = 1000)
b_values_s <- bsplineEstimate(x = s,
                              knots = b_comb$knots,
                              m = b_comb$m)
plot(s, b_values_s)
```

---

coef.hpaBinary *Extract coefficients from a hpaBinary object*

Description

Extract coefficients from a hpaBinary object

Usage

```
## S3 method for class 'hpaBinary'
coef(object, ...)
```

Arguments

object	Object of class "hpaBinary"
...	further arguments (currently ignored)

---

coef.hpaML *Extract coefficients from a hpaML object*

Description

Extract coefficients from a hpaML object

Usage

```
## S3 method for class 'hpaML'
coef(object, ...)
```

Arguments

object	Object of class "hpaML"
...	further arguments (currently ignored)

---

coef.hpaSelection *Extract coefficients from a hpaSelection object*

Description

Extract coefficients from a hpaSelection object

Usage

```
## S3 method for class 'hpaSelection'
coef(object, ..., type = "all")
```

Arguments

object	Object of class "hpaSelection"
...	further arguments (currently ignored)
type	character; if "all" (default) then all estimated parameter values will be returned. If "selection" then selection equation coefficient estimates will be provided. If "outcome" then outcome equation coefficient estimates will be returned.

dnorm_parallel *Calculate normal pdf in parallel*

Description

Calculate in parallel for each value from vector x density function of normal distribution with mean equal to mean and standard deviation equal to sd.

Usage

```
dnorm_parallel(x, mean = 0, sd = 1, is_parallel = FALSE)
```

Arguments

x	numeric vector of quantiles.
mean	double value.
sd	double positive value.
is_parallel	if TRUE then multiple cores will be used for some calculations. It usually provides speed advantage for large enough samples (more than 1000 observations).

Examples

```
## Consider normal distribution with mean 3 and standard deviation 5.
## Calculate its density function at points 2 and 3.

# Create vector of points
my_points <- c(2, 3)

# Calculate pdf at these points
# (set is_parallel = TRUE in order
# to turn on parallel computations)
dnorm_parallel(my_points, 3, 5,
               is_parallel = FALSE)
```

hpaBinary *Semi-nonparametric single index binary choice model estimation*

Description

This function performs semi-nonparametric (SNP) maximum likelihood estimation of single index binary choice model using Hermite polynomial based approximating function proposed by Gallant and Nychka in 1987. Please, see dhpa 'Details' section to get more information concerning this approximating function.

Usage

```
hpaBinary(
  formula,
  data,
  K = 1L,
  mean_fixed = NA_real_,
  sd_fixed = NA_real_,
  constant_fixed = 0,
  coef_fixed = TRUE,
  is_x0_probit = TRUE,
  is_sequence = FALSE,
  x0 = numeric(0),
  cov_type = "sandwich",
  boot_iter = 100L,
  is_parallel = FALSE,
  opt_type = "optim",
  opt_control = NULL,
  is_validation = TRUE
)
```

Arguments

`formula`	an object of class "formula" (or one that can be coerced to that class): a symbolic description of the model to be fitted. All variables in `formula` should be numeric vectors of the same length.
`data`	data frame containing the variables in the model.
`K`	non-negative integer representing the polynomial degree (order).
`mean_fixed`	numeric value for the binary choice equation random error density mean parameter. Set it to `NA` (default) if this parameter should be estimated rather than fixed.
`sd_fixed`	numeric value for the binary choice equation random error density `sd` parameter. Set it to `NA` (default) if this parameter should be estimated rather than fixed.
`constant_fixed`	numeric value for the binary choice equation constant parameter. Set it to `NA` (default) if this parameter should be estimated rather than fixed.
`coef_fixed`	logical value indicating whether the binary equation first independent variable coefficient should be fixed (`TRUE`) or estimated (`FALSE`).
`is_x0_probit`	logical; if `TRUE` (default) then initial points for the optimization routine will be obtained by the probit model estimated via the glm function.
`is_sequence`	if TRUE, then function calculates models with polynomial degrees from 0 to K each time using initial values obtained from the previous step. In this case function will return the list of models where i-th list element corresponds to model calculated under K=(i-1).
`x0`	numeric vector of the optimization routine initial values. Note that `x0 = c(pol_coefficients[-1], mean, sd, coefficients)`.

`cov_type` character determining the type of covariance matrix to be returned and used for summary. If `cov_type = "hessian"` then negative inverse of the Hessian matrix will be applied. If `cov_type = "gop"` then inverse of the Jacobian outer product will be used. If `cov_type = "sandwich"` (default) then the sandwich covariance matrix estimator will be applied. If `cov_type = "bootstrap"` then bootstrap with `boot_iter` iterations will be used. If `cov_type = "hessianFD"` or `cov_type = "sandwichFD"` then (probably) more accurate but computationally demanding central difference Hessian approximation will be calculated for the inverse Hessian and sandwich estimators, respectively. Central differences are computed via analytically provided gradient. This Hessian matrix estimation approach seems to be less accurate than the BFGS approximation if polynomial order is high (usually greater than 5).

`boot_iter` the number of bootstrap iterations for the covariance matrix estimator when `cov_type = "bootstrap"`.

`is_parallel` if `TRUE` then multiple cores will be used for some calculations. It usually provides speed advantage for large enough samples (more than 1000 observations).

`opt_type` string value determining the type of the optimization routine to be applied. The default is `"optim"` meaning that the BFGS method from the `optim` function will be applied. If `opt_type = "GA"` then the `ga` function will be additionally applied.

`opt_control` a list containing arguments to be passed to the optimization routine depending on the `opt_type` argument value. Please see details to get additional information.

`is_validation` logical value indicating whether function input arguments should be validated. Set it to `FALSE` for a slight performance boost (default value is `TRUE`).

Details

The Hermite polynomial approximation approach for densities has been proposed by A. Gallant and D. W. Nychka in 1987. The main idea is to approximate unknown distribution density with a scaled Hermite polynomial. For more information please refer to the literature listed below.

Let's use notations introduced in `dhpa` 'Details' section. Function `hpaBinary` maximizes the following quasi log-likelihood function:

$$\ln L(\gamma_0, \gamma, \alpha, \mu, \sigma; x) = \sum_{i:z_i=1} \ln\left(\overline{F}_\xi(-(\gamma_0 + \gamma x_i), \infty; \alpha, \mu, \sigma)\right) +$$

$$+ \sum_{i:z_i=0} \ln\left(\overline{F}_\xi(-\infty, -(\gamma_0 + x_i\gamma); \alpha, \mu, \sigma)\right),$$

where (in addition to previously defined notations):

x_i - is row vector of regressors derived from `data` according to `formula`.

γ - is column vector of regression coefficients.

γ_0 - constant.

z_i - binary (0 or 1) dependent variable defined in `formula`.

Note that ξ is one-dimensional and `K` corresponds to $K = K_1$.

The first polynomial coefficient (zero powers) is set to 1 for identification purposes i.e. $\alpha_0 = 1$.

If coef_fixed is TRUE then the coefficient for the first independent variable in formula will be fixed to 1 i.e. $\gamma_1 = 1$.

If mean_fixed is not NA then μ is fixed to mean_fixed.

If sd_fixed is not NA then σ is fixed to sd_fixed. However, if is_x0_probit = TRUE then parameter σ will be scale-adjusted in order to provide better initial point for optimization routine. Please extract the adjusted value of σ from the function's output list. The same applies to mean_fixed.

Rows in data corresponding to variables mentioned in formula which have at least one NA value will be ignored.

All variables mentioned in formula should be numeric vectors.

The function calculates standard errors via the sandwich estimator and significance levels are reported taking into account quasi maximum likelihood estimator (QMLE) asymptotic normality. If one wants to switch from QMLE to semi-nonparametric estimator (SNPE) during hypothesis testing then covariance matrix should be estimated again using the bootstrap.

This function maximizes (quasi) log-likelihood function via the optim function setting its method argument to "BFGS". If opt_type = "GA" then the genetic algorithm from the ga function will be additionally (after optim putting its solution (par) into the suggestions matrix) applied in order to perform global optimization. Note that global optimization takes much more time (usually minutes but sometimes hours or even days). The number of iterations and population size of the genetic algorithm will grow linearly along with the number of estimated parameters. If it seems that global maximum has not been found then it is possible to continue the search restarting the function setting its input argument x0 to x1 output value. Note that if cov_type = "bootstrap" then the ga function will not be used for bootstrap iterations since it may be extremely time consuming.

If opt_type = "GA" then opt_control should be a list containing the values to be passed to the ga function. It is possible to pass arguments lower, upper, popSize, pcrossover, pmutation, elitism, maxiter, suggestions, optim, optimArgs, seed and monitor. Note that it is possible to set population, selection, crossover and mutation arguments changing ga default parameters via gaControl function. These arguments information is reported in ga. In order to provide manual values for lower and upper bounds please follow the parameters ordering mentioned above for the x0 argument. If these bounds are not provided manually then they (except those related to the polynomial coefficients) will depend on the estimates obtained by local optimization via optim function (these estimates will be in the middle between lower and upper). Specifically for each sd parameter lower (upper) bound is 5 times lower (higher) than this parameter's optim estimate. For each mean and regression coefficient parameter its lower and upper bounds deviate from corresponding optim estimate by two absolute values of this estimate. Finally, lower and upper bounds for each polynomial coefficient are -10 and 10, respectively (do not depend on their optim estimates).

The following arguments differ from their defaults in ga:

- pmutation = 0.2.
- optim = TRUE.
- optimArgs = list("method" = "Nelder-Mead", "poptim" = 0.2, "pressel" = 0.5).
- seed = 8.
- elitism = 2 + round(popSize * 0.1).

Let's denote by n_reg the number of regressors included in the formula. The arguments popSize and maxiter of the ga function have been set proportional to the number of estimated polynomial coefficients and independent variables:

- `popSize = 10 + 5 * (K + 1) + 2 * n_reg`
- `maxiter = 50 * (1 + K) + 10 * n_reg`

Value

This function returns an object of class "hpaBinary".

An object of class "hpaBinary" is a list containing the following components:

- `optim` - `optim` function output. If `opt_type = "GA"`, then it is the list containing the outputs of the `optim` and `ga` functions.
- `x1` - numeric vector of distribution parameter estimates.
- `mean` - mean (mu) parameter of density function estimate.
- `sd` - sd (sigma) parameter of density function estimate.
- `pol_coefficients` - polynomial coefficients estimates.
- `pol_degrees` - the same as the `K` input parameter.
- `coefficients` - regression (single index) coefficients estimates.
- `cov_mat` - covariance matrix estimate.
- `marginal_effects` - marginal effects matrix where columns are variables and rows are observations.
- `results` - numeric matrix representing estimation results.
- `log-likelihood` - value of Log-Likelihood function.
- `AIC` - AIC value.
- `errors_exp` - random error expectation estimate.
- `errors_var` - random error variance estimate.
- `dataframe` - data frame containing variables mentioned in `formula` without `NA` values.
- `model_Lists` - lists containing information about fixed parameters and parameters indexes in `x1`.
- `n_obs` - number of observations.
- `z_latent` - latent variable (single index) estimates.
- `z_prob` - probabilities of positive outcome (i.e., 1) estimates.

References

A. Gallant and D. W. Nychka (1987) <doi:10.2307/1913241>

See Also

summary.hpaBinary, predict.hpaBinary, plot.hpaBinary, logLik.hpaBinary

Examples

```
## Estimate survival probability on Titanic

library("titanic")

# Prepare data set converting
# all variables to numeric vectors
h <- data.frame("male" = as.numeric(titanic_train$Sex == "male"))
h$class_1 <- as.numeric(titanic_train$Pclass == 1)
h$class_2 <- as.numeric(titanic_train$Pclass == 2)
h$class_3 <- as.numeric(titanic_train$Pclass == 3)
h$sibl <- titanic_train$SibSp
h$survived <- titanic_train$Survived
h$age <- titanic_train$Age
h$parch <- titanic_train$Parch
h$fare <- titanic_train$Fare

# Estimate model parameters
model_hpa_1 <- hpaBinary(survived ~class_1 + class_2 +
male + age + sibl + parch + fare,
K = 3, data = h)
# get summary
summary(model_hpa_1)

# Get predicted probabilities
pred_hpa_1 <- predict(model_hpa_1)

# Calculate number of correct predictions
hpa_1_correct_0 <- sum((pred_hpa_1 < 0.5) &
                       (model_hpa_1$dataframe$survived == 0))
hpa_1_correct_1 <- sum((pred_hpa_1 >= 0.5) &
                       (model_hpa_1$dataframe$survived == 1))
hpa_1_correct <- hpa_1_correct_1 + hpa_1_correct_0

# Plot the random errors density approximation
plot(model_hpa_1)



## Estimate parameters on data simulated from Student distribution

library("mvtnorm")
set.seed(123)

# Simulate independent variables from normal distribution
n <- 5000
X <- rmvnorm(n=n, mean = c(0,0),
sigma = matrix(c(1,0.5,0.5,1), ncol=2))

# Simulate random errors from Student distribution
epsilon <- rt(n, 5) * (3 / sqrt(5))
```

```
# Calculate latent and observable variables values
z_star <- 1 + X[, 1] + X[, 2] + epsilon
z <- as.numeric((z_star > 0))

# Store the results into a data frame
h <- as.data.frame(cbind(z,X))
names(h) <- c("z", "x1", "x2")

# Estimate model parameters
model <- hpaBinary(formula = z ~ x1 + x2, data=h, K = 3)
summary(model)

# Get predicted probabilities of 1 values
predict(model)

# Plot the density function approximation
plot(model)
```

---

hpaDist *Probabilities and Moments Hermite Polynomial Approximation*

Description

Approximation of truncated, marginal and conditional densities, moments and cumulative probabilities of multivariate distributions via a Hermite polynomial based approach proposed by Gallant and Nychka in 1987.

Density approximating function is scale-adjusted product of two terms. The first one is squared multivariate polynomial of degree `pol_degrees` with a vector of coefficients `pol_coefficients`. The second is product of independent normal random variables' densities with expected values and standard deviations given by `mean` and `sd` vectors, respectively. Approximating function satisfies properties of density function thus generating a broad family of distributions. Characteristics of these distributions (moments, quantiles, probabilities and so on) may provide accurate approximations to the characteristic of other distributions. Moreover it is usually possible to provide arbitrary close approximation by means of polynomial degrees increasing.

Usage

```
dhpa(
  x,
  pol_coefficients = numeric(0),
  pol_degrees = numeric(0),
  given_ind = numeric(0),
  omit_ind = numeric(0),
  mean = numeric(0),
  sd = numeric(0),
```

```
  is_parallel = FALSE,
  log = FALSE,
  is_validation = TRUE
)

phpa(
  x,
  pol_coefficients = numeric(0),
  pol_degrees = numeric(0),
  given_ind = numeric(0),
  omit_ind = numeric(0),
  mean = numeric(0),
  sd = numeric(0),
  is_parallel = FALSE,
  log = FALSE,
  is_validation = TRUE
)

ihpa(
  x_lower = numeric(0),
  x_upper = numeric(0),
  pol_coefficients = numeric(0),
  pol_degrees = numeric(0),
  given_ind = numeric(0),
  omit_ind = numeric(0),
  mean = numeric(0),
  sd = numeric(0),
  is_parallel = FALSE,
  log = FALSE,
  is_validation = TRUE
)

ehpa(
  x = numeric(0),
  pol_coefficients = numeric(0),
  pol_degrees = numeric(0),
  given_ind = numeric(0),
  omit_ind = numeric(0),
  mean = numeric(0),
  sd = numeric(0),
  expectation_powers = numeric(0),
  is_parallel = FALSE,
  is_validation = TRUE
)

etrhpa(
  tr_left = numeric(0),
  tr_right = numeric(0),
```

```
  pol_coefficients = numeric(0),
  pol_degrees = numeric(0),
  mean = numeric(0),
  sd = numeric(0),
  expectation_powers = numeric(0),
  is_parallel = FALSE,
  is_validation = TRUE
)

dtrhpa(
  x,
  tr_left = numeric(0),
  tr_right = numeric(0),
  pol_coefficients = numeric(0),
  pol_degrees = numeric(0),
  given_ind = numeric(0),
  omit_ind = numeric(0),
  mean = numeric(0),
  sd = numeric(0),
  is_parallel = FALSE,
  log = FALSE,
  is_validation = TRUE
)

itrhpa(
  x_lower = numeric(0),
  x_upper = numeric(0),
  tr_left = numeric(0),
  tr_right = numeric(0),
  pol_coefficients = numeric(0),
  pol_degrees = numeric(0),
  given_ind = numeric(0),
  omit_ind = numeric(0),
  mean = numeric(0),
  sd = numeric(0),
  is_parallel = FALSE,
  log = FALSE,
  is_validation = TRUE
)

dhpaDiff(
  x,
  pol_coefficients = numeric(0),
  pol_degrees = numeric(0),
  given_ind = numeric(0),
  omit_ind = numeric(0),
  mean = numeric(0),
  sd = numeric(0),
```

```
  type = "pol_coefficients",
  is_parallel = FALSE,
  log = FALSE,
  is_validation = TRUE
)

ehpaDiff(
  x = numeric(0),
  pol_coefficients = numeric(0),
  pol_degrees = numeric(0),
  given_ind = numeric(0),
  omit_ind = numeric(0),
  mean = numeric(0),
  sd = numeric(0),
  expectation_powers = numeric(0),
  type = "pol_coefficients",
  is_parallel = FALSE,
  log = FALSE,
  is_validation = TRUE
)

ihpaDiff(
  x_lower = numeric(0),
  x_upper = numeric(0),
  pol_coefficients = numeric(0),
  pol_degrees = numeric(0),
  given_ind = numeric(0),
  omit_ind = numeric(0),
  mean = numeric(0),
  sd = numeric(0),
  type = "pol_coefficients",
  is_parallel = FALSE,
  log = FALSE,
  is_validation = TRUE
)

qhpa(
  p,
  x = matrix(1, 1),
  pol_coefficients = numeric(0),
  pol_degrees = numeric(0),
  given_ind = numeric(0),
  omit_ind = numeric(0),
  mean = numeric(0),
  sd = numeric(0)
)

rhpa(
```

```
  n,
  pol_coefficients = numeric(0),
  pol_degrees = numeric(0),
  mean = numeric(0),
  sd = numeric(0)
)
```

Arguments

x	numeric matrix of function arguments and conditional values. Note that the rows of x are points (observations) while the random vector components (the variables) are columns.
pol_coefficients	numeric vector of polynomial coefficients.
pol_degrees	a non-negative integer vector of polynomial degrees (orders).
given_ind	logical or numeric vector indicating whether corresponding random vector component is conditioned. By default it is a logical vector of FALSE values. If given_ind[i] equals TRUE or i then the i-th column of x matrix will contain conditional values.
omit_ind	logical or numeric vector indicating whether the corresponding random vector component is omitted. By default it is a logical vector of FALSE values. If omit_ind[i] equals TRUE or i then the values in the i-th column of the x matrix will be ignored.
mean	numeric vector of expected values.
sd	positive numeric vector of standard deviations.
is_parallel	if TRUE then multiple cores will be used for some calculations. It usually provides speed advantage for large enough samples (more than 1000 observations).
log	logical; if TRUE then probabilities p are given as log(p) or derivatives will be given with respect to log(p).
is_validation	logical value indicating whether function input arguments should be validated. Set it to FALSE for a slight performance boost (default value is TRUE).
x_lower	numeric matrix of lower integration limits. Note that the rows of x_lower are observations while the variables are columns.
x_upper	numeric matrix of upper integration limits. Note that the rows of x_upper are observations while the variables are columns.
expectation_powers	integer vector of random vector components' powers.
tr_left	numeric matrix of left (lower) truncation limits. Note that the rows of tr_left are observations while the columns are variables. If tr_left and tr_right are single row matrices then the same truncation limits will be applied to all observations that are determined by the first row of these matrices.
tr_right	numeric matrix of right (upper) truncation limits. Note that the rows of tr_right are observations while the columns are variables. If tr_left and tr_right are single row matrices then the same truncation limits will be applied to all observations that are determined by the first row of these matrices.

type	determines the partial derivatives to be included in the gradient. If type="pol_coefficients" then gradient will contain partial derivatives with respect to polynomial coefficients listed in the same order as pol_coefficients. Other available types are type = "mean" and type = "sd". For function dhpaDiff it is possible to take gradient respect to the x points setting type="x". For function ihpaDiff it is possible to take gradient with respect to the x_lower and x_upper points setting type = "x_lower" or type = "x_upper", respectively. In order to get full gradient please set type="all".
p	numeric vector of probabilities
n	positive integer representing the number of observations

Details

It is possible to approximate densities dhpa, cumulative probabilities phpa, interval probabilities ihpa, moments ehpa as well as their truncated dtrhpa, itrhpa, etrhpa forms and gradients dhpaDiff, ihpaDiff. Note that phpa is a special case of ihpa where x corresponds to x_upper while x_lower is a matrix of negative infinity values. So phpa is intended to approximate cumulative distribution functions while ihpa approximates probabilities that random vector components will be between values determined by rows of x_lower and x_upper matrices. Further details are given below.

Since density approximating function is non-negative and integrates to 1, it is a density function for some m-variate random vector ξ. Approximating function $f_\xi(x)$ has the following form:

$$f_\xi(x) = f_\xi(x;\mu,\sigma,\alpha) = \frac{1}{\psi}\prod_{t=1}^{m}\phi(x_t;\mu_t,\sigma_t)\left(\sum_{i_1=0}^{K_1}\cdots\sum_{i_m=0}^{K_m}\alpha_{(i_1,\ldots,i_m)}\prod_{r=1}^{m}x_r^{i_r}\right)^2$$

$$\psi = \sum_{i_1=0}^{K_1}\cdots\sum_{i_m=0}^{K_m}\sum_{j_1=0}^{K_1}\cdots\sum_{j_m=0}^{K_m}\alpha_{(i_1,\cdots,i_m)}\alpha_{(j_1,\cdots,j_m)}\prod_{r=1}^{m}\mathcal{M}(i_r+j_r;\mu_r,\sigma_r),$$

where:

$x = (x_1, ..., x_m)$ - is vector of arguments i.e. rows of x matrix in dhpa.

$\alpha_{(i_1,\cdots,i_m)}$ - is polynomial coefficient corresponding to pol_coefficients[k] element. In order to investigate correspondence between k and $(i_1, \cdots, i_m)$ values, please see 'Examples' section below or polynomialIndex function 'Details', 'Value' and 'Examples' sections. Note that if $m = 1$ then pol_coefficients[k] simply corresponds to α_{k-1}.

$(K_1, ..., K_m)$ - are polynomial degrees (orders) provided via pol_degrees argument so pol_degrees[i] determines K_i.

$\phi(.;\mu_t,\sigma_t)$ - is a normal random variable density function where μ_t and σ_t are mean and standard deviation determined by mean[t] and sd[t] arguments values.

$\mathcal{M}(q;\mu_r,\sigma_r)$ - is q-th order moment of a normal random variable with mean μ_r and standard deviation σ_r. Note that function normalMoment allows to calculate and differentiate normal random variable's moments.

ψ - constant term insuring that $f_\xi(x)$ is a density function.

Therefore dhpa allows to calculate $f_{\xi}(x)$ values at points determined by rows of x matrix given polynomial degrees pol_degrees (K) as well as mean (μ), sd (σ) and pol_coefficients (α) parameter values. Note that mean, sd and pol_degrees are m-variate vectors while pol_coefficients has prod(pol_degrees + 1) elements.

Cumulative probabilities could be approximated as follows:

$$P\left(\underline{x}_1 \leq \xi_1 \leq \overline{x}_1, ..., \underline{x}_m \leq \xi_m \leq \overline{x}_m\right) =$$

$$= \bar{F}_{\xi}(\underline{x}, \bar{x}) = \bar{F}_{\xi}(\underline{x}, \bar{x}; \mu, \sigma, \alpha) = \frac{1}{\psi} \prod_{t=1}^{m} (\Phi(\bar{x}_t; \mu_t, \sigma_t) - \Phi(\underline{x}_t; \mu_t, \sigma_t)) *$$

$$* \sum_{i_1=0}^{K_1} ... \sum_{i_m=0}^{K_m} \sum_{j_1=0}^{K_1} ... \sum_{j_m=0}^{K_m} \alpha_{(i_1,...,i_m)} \alpha_{(j_1,...,j_m)} \prod_{r=1}^{m} \mathcal{M}_{TR}\left(i_r + j_r; \underline{x}_r, \overline{x}_r, \mu_r, \sigma_r\right)$$

where:

$\Phi(.; \mu_t, \sigma_t)$ - is normal random variable's cumulative distribution function where μ_t and σ_t are mean and standard deviation determined by mean[t] and sd[t] arguments values.

$\mathcal{M}_{TR}(q; \underline{x}_r, \overline{x}_r, \mu_r, \sigma_r)$ - is q-th order moment of a truncated (from above by $\overline{x}_r$ and from below by $\underline{x}_r$) normal random variable with mean μ_r and standard deviation σ_r. Note that function truncatedNormalMoment allows to calculate and differentiate truncated normal random variable's moments.

$\overline{x} = (\overline{x}_1, ..., \overline{x}_m)$ - vector of upper integration limits, i.e., rows of the x_upper matrix in ihpa.

$\underline{x} = (\underline{x}_1, ..., \underline{x}_m)$ - vector of lower integration limits, i.e., rows of the x_lower matrix in ihpa.

Therefore ihpa allows to calculate interval distribution function $\bar{F}_{\xi}(\underline{x}, \bar{x})$ values at points determined by rows of x_lower ($\underline{x}$) and x_upper ($\overline{x}$) matrices. The rest of the arguments are similar to dhpa.

Expected value powered product approximation is as follows:

$$E\left(\prod_{t=1}^{m} \xi_t^{k_t}\right) = \frac{1}{\psi} \sum_{i_1=0}^{K_1} ... \sum_{i_m=0}^{K_m} \sum_{j_1=0}^{K_1} ... \sum_{j_m=0}^{K_m} \alpha_{(i_1,...,i_m)} \alpha_{(j_1,...,j_m)} \prod_{r=1}^{m} \mathcal{M}(i_r + j_r + k_t; \mu_r, \sigma_r)$$

where $(k_1, ..., k_m)$ are integer powers determined by expectation_powers argument of ehpa so expectation_powers[t] assigns k_t. Note that argument x in ehpa allows to determine conditional values.

Expanding polynomial degrees $(K_1, ..., K_m)$ it is possible to provide arbitrarily close approximation to a density of some m-variate random vector $\xi^{\star}$. So actually $f_{\xi}(x)$ approximates $f_{\xi^{\star}}(x)$. Accurate approximation requires selection of appropriate mean, sd and pol_coefficients values. In order to get sample estimates of these parameters, apply hpaML function.

In order to perform calculation for marginal distribution of some ξ components, provide omitted components via omit_ind argument. For examples if one assumes $m = 5$-variate distribution and wants to deal with 1-st, 3-rd, and 5-th components only i.e. (ξ_1, ξ_3, ξ_5) then set omit_ind = c(FALSE, TRUE, FALSE, TRUE, FALSE) indicating that ξ_2 and ξ_4 should be 'omitted' from ξ since 2-nd and 4-th values of omit_ind are TRUE. Then x still should be a 5 column matrix but values in 2-nd and 4-th columns will not affect calculation results. Meanwhile note that marginal distribution of

t components of ξ usually does not coincide with any marginal distribution generated by t-variate density approximating function.

In order to perform calculation for conditional distribution, i.e., given fixed values for some ξ components, provide these components via given_ind argument. For example, if one assumes $m = 5$-variate distribution and wants to deal with 1-st, 3-rd, and 5-th components given fixed values (suppose 8 and 10) for the other two components, i.e., $(\xi|\xi_2 = 8, \xi_4 = 10)$, then set given_ind = c(FALSE, TRUE, FALSE, TRUE, FALSE) and x[2] = 8, x[4] = 10 where for simplicity it is assumed that x is a single row 5 column matrix; it is possible to provide different conditional values for the same components by setting different values to different x rows.

Note that it is possible to combine given_ind and omit_ind arguments. However, it is wrong to set both given_ind[i] and omit_ind[i] to TRUE. Also at least one value should be FALSE for both given_ind and omit_ind.

In order to consider truncated distribution of ξ, i.e., $\left(\xi|\overline{a}_1 \le \xi_1 \le \overline{b}_1, \cdots, \overline{a}_m \le \xi_m \le \overline{b}_m\right)$ please set lower (left) truncation points $\overline{a}$ and upper (right) truncation points $\overline{b}$ via tr_left and tr_right arguments, respectively. Note that if lower truncation points are negative infinity and upper truncation points are positive infinity, then dtrhpa, itrhpa and etrhpa are similar to dhpa, ihpa and ehpa correspondingly.

In order to calculate Jacobian of $f_\xi(x; \mu, \sigma, \alpha)$ and $\bar{F}_\xi(\underline{x}, \bar{x}; \mu, \sigma, \alpha)$ w.r.t all or some particular parameters, apply dhpaDiff and ihpaDiff functions, respectively, specifying parameters of interest via type argument. If x or x_lower and x_upper are single row matrices then gradients will be calculated.

For further information please see 'Examples' section. Note that examples are given separately for each function.

If given_ind and/or omit_ind are numeric vectors then they are insensitive to the order of elements. For example given_ind = c(5, 2, 3) is similar to given_ind = c(2, 3, 5).

The Hermite polynomial approximation approach for densities has been proposed by A. Gallant and D. W. Nychka in 1987. The main idea is to approximate unknown distribution density with a scaled Hermite polynomial. For more information please refer to the literature listed below.

Value

Functions dhpa, phpa and dtrhpa return vector of probabilities of length nrow(x).

Functions ihpa and itrhpa return vector of probabilities of length nrow(x_upper).

If x argument has not been provided or is a single row matrix then function ehpa returns a moment value. Otherwise it returns a vector of length nrow(x) containing moment values.

If tr_left and tr_right arguments are single row matrices then function etrhpa returns a moment value. Otherwise it returns vector of length max(nrow(tr_left), nrow(tr_right)) containing moments values.

Functions dhpaDiff and ihpaDiff return Jacobian matrix. The number of columns depends on the type argument. The number of rows is nrow(x) for dhpaDiff and nrow(x_upper) for ihpaDiff

If mean or sd are not specified, they default to the default values of m-dimensional vectors of 0 and 1, respectively. If x_lower is not specified then it is a matrix of the same size as x_upper containing negative infinity values only. If expectation_powers is not specified then it is an m-dimensional vector of 0 values.

Please see 'Details' section for additional information.

References

A. Gallant and D. W. Nychka (1987) <doi:10.2307/1913241>

Examples

```
## Example demonstrating dhpa function application.
## Let's approximate three random variables (i.e. X1, X2 and X3)
## the joint density function at points  x = (0,1, 0.2, 0.3) and
## y = (0.5, 0.8, 0.6) with Hermite polynomial of (1, 2, 3) degrees which
## polynomial coefficients equal 1 except the coefficient related to
## x1*(x3^2)
## polynomial element which equals 2. Also suppose that the normal density's
## related mean vector equals (1.1, 1.2, 1.3) while the standard deviation
## vector is (2.1, 2.2, 2.3).

# Prepare initial values
x <- matrix(c(0.1, 0.2, 0.3), nrow = 1)   # x point as a single row matrix
y <- matrix(c(0.5, 0.8, 0.6), nrow = 1)   # y point as a single row matrix
x_y <- rbind(x, y)                        # matrix whose rows are x and y
mean <- c(1.1, 1.2, 1.3)
sd <- c(2.1, 2.2, 2.3)
pol_degrees <- c(1, 2, 3)

# Create polynomial powers and indexes correspondence matrix
pol_ind <- polynomialIndex(pol_degrees)
# Set all polynomial coefficients to 1
pol_coefficients <- rep(1, ncol(pol_ind))
pol_degrees_n <- length(pol_degrees)

# Assign coefficient 2 to the polynomial element (x1 ^ 1)*(x2 ^ 0)*(x3 ^ 2)
pol_coefficients[apply(pol_ind, 2, function(x) all(x == c(1, 0, 2)))] <- 2

# Visualize correspondence between polynomial
# elements and their coefficients
as.data.frame(rbind(pol_ind, pol_coefficients),
   row.names = c("x1 power", "x2 power", "x3 power", "coefficients"),
   optional = TRUE)
printPolynomial(pol_degrees, pol_coefficients)

# Calculate density approximation
   # at point x (note that x should be a matrix)
dhpa(x = x,
     pol_coefficients = pol_coefficients,
     pol_degrees = pol_degrees,
     mean = mean, sd = sd)
   # at points x and y
dhpa(x = x_y,
     pol_coefficients = pol_coefficients,
     pol_degrees = pol_degrees,
     mean = mean, sd = sd)

# Condition second component to be 0.5 i.e. X2 = 0.5.
```

```
# Substitute x and y second components with conditional value 0.5
x <- matrix(c(0.1, 0.5, 0.3), nrow = 1) # or simply x[2] <- 0.5
y <- matrix(c(0.4, 0.5, 0.6), nrow = 1) # or simply y[2] <- 0.5
x_y <- rbind(x, y)
# Set TRUE to the second component indicating that it is conditioned
given_ind <- c(FALSE, TRUE, FALSE)

# Calculate conditional (on X2 = 0.5) density approximation
   # at point x
dhpa(x = x,
     pol_coefficients = pol_coefficients,
     pol_degrees = pol_degrees,
     mean = mean, sd = sd,
     given_ind = given_ind)
   # at points x and y
dhpa(x = x_y,
     pol_coefficients = pol_coefficients,
     pol_degrees = pol_degrees,
     mean = mean, sd = sd,
     given_ind = given_ind)

# Consider third component marginal distribution conditioned on the
# second component 0.5 value i.e. (X3 | X2 = 0.5).
# Set TRUE to the first component indicating that it is omitted
omit_ind <- c(TRUE, FALSE, FALSE)

# Calculate conditional (on x2 = 0.5) marginal (for x3) density approximation
   # at point x
dhpa(x = x,
     pol_coefficients = pol_coefficients,
     pol_degrees = pol_degrees,
     mean = mean, sd = sd,
     given_ind = given_ind,
     omit_ind = omit_ind)
   # at points x and y
dhpa(x = x_y,
     pol_coefficients = pol_coefficients,
     pol_degrees = pol_degrees,
     mean = mean, sd = sd,
     given_ind = given_ind,
     omit_ind = omit_ind)

## Example demonstrating phpa function application.
## Let's approximate three random variables (X1, X2, X3) the
## joint cumulative distribution function (cdf) at point (0.1, 0.2, 0.3)
## with a Hermite polynomial of (1, 2, 3) degrees whose polynomial
## coefficients equal 1 except the coefficient related to x1*(x3^2)
## polynomial element which equals 2. Also suppose that the normal density's
## mean vector equals (1.1, 1.2, 1.3) while the standard deviation
## vector is (2.1, 2.2, 2.3).

## Prepare initial values
x <- matrix(c(0.1, 0.2, 0.3), nrow = 1)
```

```
mean <- c(1.1, 1.2, 1.3)
sd <- c(2.1, 2.2, 2.3)
pol_degrees <- c(1, 2, 3)

# Create polynomial powers and indexes correspondence matrix
pol_ind <- polynomialIndex(pol_degrees)

# Set all polynomial coefficients to 1
pol_coefficients <- rep(1, ncol(pol_ind))
pol_degrees_n <- length(pol_degrees)

# Assign coefficient 2 to the polynomial element (x1 ^ 1)*(x2 ^ 0)*(x3 ^ 2)
pol_coefficients[apply(pol_ind, 2, function(x) all(x == c(1, 0, 2)))] <- 2

# Visualize correspondence between polynomial
# elements and their coefficients
as.data.frame(rbind(pol_ind, pol_coefficients),
           row.names = c("x1 power", "x2 power",
                         "x3 power", "coefficients"),
            optional = TRUE)
printPolynomial(pol_degrees, pol_coefficients)

# Calculate cdf approximation at point x
phpa(x = x,
     pol_coefficients = pol_coefficients,
     pol_degrees = pol_degrees,
     mean = mean, sd = sd)

# Condition second component to be 0.5
# Substitute x second component with the conditional value 0.5
x <- matrix(c(0.1, 0.5, 0.3), nrow = 1) # or simply x[2] <- 0.5

# Set the second component to TRUE indicating that it is conditioned
given_ind <- c(FALSE, TRUE, FALSE)

# Calculate conditional (on X2 = 0.5) cdf approximation at point x
phpa(x = x,
     pol_coefficients = pol_coefficients,
     pol_degrees = pol_degrees,
     mean = mean, sd = sd,
     given_ind = given_ind)

# Consider third component marginal distribution
# conditioned on the second component 0.5 value

# Set the first component to TRUE indicating that it is omitted
omit_ind <- c(TRUE, FALSE, FALSE)

# Calculate conditional (on X2 = 0.5) marginal (for X3) cdf
# approximation at point x
phpa(x = x,
      pol_coefficients = pol_coefficients,
      pol_degrees = pol_degrees,
```

```
      mean = mean, sd = sd,
      given_ind = given_ind,
      omit_ind = omit_ind)

## Example demonstrating ihpa function application.
## Let's approximate three random variables (X1, X2, X3) the joint interval
## distribution function (intdf) at lower and upper points (0.1, 0.2, 0.3)
## and (0,4, 0.5, 0.6), respectively, with a Hermite polynomial of (1, 2, 3)
## degrees whose polynomial coefficients equal 1 except the coefficient
## related to x1*(x3^2) polynomial element which equals 2. Also suppose that
## the normal density's mean vector equals (1.1, 1.2, 1.3) while the standard
## deviation vector is (2.1, 2.2, 2.3).

## Prepare initial values
x_lower <- matrix(c(0.1, 0.2, 0.3), nrow=1)
x_upper <- matrix(c(0.4, 0.5, 0.6), nrow=1)
mean <- c(1.1, 1.2, 1.3)
sd <- c(2.1, 2.2, 2.3)
pol_degrees <- c(1, 2, 3)

# Create polynomial powers and indexes correspondence matrix
pol_ind <- polynomialIndex(pol_degrees)

# Set all polynomial coefficients to 1
pol_coefficients <- rep(1, ncol(pol_ind))
pol_degrees_n <- length(pol_degrees)

# Assign coefficient 2 to the polynomial element (x1 ^ 1)*(x2 ^ 0)*(x3 ^ 2)
pol_coefficients[apply(pol_ind, 2, function(x) all(x == c(1, 0, 2)))] <- 2

# Visualize correspondence between polynomial
# elements and their coefficients
as.data.frame(rbind(pol_ind, pol_coefficients),
           row.names = c("x1 power", "x2 power",
                         "x3 power", "coefficients"),
              optional = TRUE)
printPolynomial(pol_degrees, pol_coefficients)

# Calculate intdf approximation at points x_lower and x_upper
ihpa(x_lower = x_lower, x_upper = x_upper,
     pol_coefficients = pol_coefficients,
     pol_degrees = pol_degrees,
     mean = mean, sd = sd)

# Condition second component to be 0.7
# Substitute x second component with the conditional value 0.7
x_upper <- matrix(c(0.4, 0.7, 0.6), nrow = 1)

# Set the second component to TRUE indicating that it is conditioned
given_ind <- c(FALSE, TRUE, FALSE)

# Calculate conditional (on X2 = 0.7) the intdf approximation
# at points x_lower and x_upper
```

```
ihpa(x_lower = x_lower, x_upper = x_upper,
     pol_coefficients = pol_coefficients,
     pol_degrees = pol_degrees,
     mean = mean, sd = sd,
     given_ind = given_ind)

# Consider third component marginal distribution
# conditioned on the second component 0.7 value
# Set the first component to TRUE indicating that it is omitted
omit_ind <- c(TRUE, FALSE, FALSE)

# Calculate conditional (on X2 = 0.7) marginal (for X3)
# the intdf approximation at points x_lower and x_upper
ihpa(x_lower = x_lower, x_upper = x_upper,
     pol_coefficients = pol_coefficients,
     pol_degrees = pol_degrees,
     mean = mean, sd = sd,
     given_ind = given_ind, omit_ind = omit_ind)

## Example demonstrating ehpa function application.
## Let's approximate some random variables (X1, X2, X3) the powered product
## expectation for powers (3, 2, 1) with Hermite polynomial of (1, 2, 3)
## degrees whose polynomial coefficients equal 1 except the coefficient
## related to x1*(x3^2) polynomial element which equals 2.
## Also suppose that the normal density's mean vector equals
## (1.1, 1.2, 1.3) while the standard deviation vector is (2.1, 2.2, 2.3).

# Prepare initial values
expectation_powers = c(3,2,1)
mean <- c(1.1, 1.2, 1.3)
sd <- c(2.1, 2.2, 2.3)
pol_degrees <- c(1, 2, 3)

# Create polynomial powers and indexes correspondence matrix
pol_ind <- polynomialIndex(pol_degrees)

# Set all polynomial coefficients to 1
pol_coefficients <- rep(1, ncol(pol_ind))
pol_degrees_n <- length(pol_degrees)

# Assign coefficient 2 to the polynomial element (x1 ^ 1)*(x2 ^ 0)*(x3 ^ 2)
pol_coefficients[apply(pol_ind, 2, function(x) all(x == c(1, 0, 2)))] <- 2

# Visualize correspondence between polynomial elements and their coefficients
as.data.frame(rbind(pol_ind, pol_coefficients),
              row.names = c("x1 power", "x2 power",
                            "x3 power", "coefficients"),
              optional = TRUE)
printPolynomial(pol_degrees, pol_coefficients)

# Calculate expected powered product approximation
ehpa(pol_coefficients = pol_coefficients,
     pol_degrees = pol_degrees,
```

```
     mean = mean, sd = sd,
     expectation_powers = expectation_powers)

# Condition second component to be 0.5
# Substitute x second component with the conditional value 0.5
x <- matrix(c(NA, 0.5, NA), nrow = 1)
# Set the second component to TRUE indicating that it is conditioned
given_ind <- c(FALSE, TRUE, FALSE)

# Calculate conditional (on X2 = 0.5) expected powered product approximation
ehpa(x = x,
     pol_coefficients = pol_coefficients,
     pol_degrees = pol_degrees,
     mean = mean, sd = sd,
     expectation_powers = expectation_powers,
     given_ind = given_ind)

# Consider third component marginal distribution
# conditioned on the second component 0.5 value
# Set the first component to TRUE indicating that it is omitted
omit_ind <- c(TRUE, FALSE, FALSE)

# Calculate conditional (on X2 = 0.5) marginal (for X3) expected powered
# product approximation at points x_lower and x_upper
ehpa(x = x,
     pol_coefficients = pol_coefficients,
     pol_degrees = pol_degrees,
     mean = mean, sd = sd,
     expectation_powers = expectation_powers,
     given_ind = given_ind,
     omit_ind = omit_ind)

## Example demonstrating etrhpa function application.
## Let's approximate three truncated random variables (X1, X2, X3) the
## powered product expectation for powers (3, 2, 1) with a Hermite polynomial
## of (1,2,3) degrees whose polynomial coefficients equal 1 except the
## coefficient related to x1*(x3^2) polynomial element which equals 2. Also
## suppose that the normal density's mean vector equals (1.1, 1.2, 1.3)
## while the standard deviation vector is (2.1, 2.2, 2.3). Suppose that lower
## and upper truncation points are (-1.1,-1.2,-1.3) and (1.1,1.2,1.3),
## respectively.

# Prepare initial values
expectation_powers = c(3,2,1)
tr_left = matrix(c(-1.1,-1.2,-1.3), nrow = 1)
tr_right = matrix(c(1.1,1.2,1.3), nrow = 1)
mean <- c(1.1, 1.2, 1.3)
sd <- c(2.1, 2.2, 2.3)
pol_degrees <- c(1, 2, 3)

# Create polynomial powers and indexes correspondence matrix
pol_ind <- polynomialIndex(pol_degrees)
# Set all polynomial coefficients to 1
```

```
pol_coefficients <- rep(1, ncol(pol_ind))
pol_degrees_n <- length(pol_degrees)

# Assign coefficient 2 to the polynomial element (x1 ^ 1)*(x2 ^ 0)*(x3 ^ 2)
pol_coefficients[apply(pol_ind, 2, function(x) all(x == c(1, 0, 2)))] <- 2

# Visualize correspondence between polynomial elements and their coefficients
as.data.frame(rbind(pol_ind, pol_coefficients),
              row.names = c("x1 power", "x2 power",
                            "x3 power", "coefficients"),
              optional = TRUE)
printPolynomial(pol_degrees, pol_coefficients)

# Calculate expected powered product approximation for truncated distribution
etrhpa(pol_coefficients = pol_coefficients,
       pol_degrees = pol_degrees,
       mean = mean, sd = sd,
       expectation_powers = expectation_powers,
       tr_left = tr_left, tr_right = tr_right)

## Example demonstrating dtrhpa function application.
## Let's approximate three random variables (X1, X2, X3) the joint density
## function at point (0.1, 0.2, 0.3) with Hermite polynomial of (1,2,3)
## degrees whose polynomial coefficients equal 1 except the coefficient
## related to x1*(x3^2) polynomial element which equals 2. Also suppose that
## the normal density's mean vector equals (1.1, 1.2, 1.3) while the standard
## deviations vector is (2.1, 2.2, 2.3). Suppose that lower and upper
## truncation points are (-1.1,-1.2,-1.3) and (1.1,1.2,1.3), respectively.

# Prepare initial values
x <- matrix(c(0.1, 0.2, 0.3), nrow=1)
tr_left = matrix(c(-1.1,-1.2,-1.3), nrow = 1)
tr_right = matrix(c(1.1,1.2,1.3), nrow = 1)
mean <- c(1.1, 1.2, 1.3)
sd <- c(2.1, 2.2, 2.3)
pol_degrees <- c(1, 2, 3)

# Create polynomial powers and indexes correspondence matrix
pol_ind <- polynomialIndex(pol_degrees)

# Set all polynomial coefficients to 1
pol_coefficients <- rep(1, ncol(pol_ind))
pol_degrees_n <- length(pol_degrees)

# Assign coefficient 2 to the polynomial element (x1 ^ 1)*(x2 ^ 0)*(x3 ^ 2)
pol_coefficients[apply(pol_ind, 2, function(x) all(x == c(1, 0, 2)))] <- 2

# Visualize correspondence between polynomial elements and their coefficients
as.data.frame(rbind(pol_ind, pol_coefficients),
              row.names = c("x1 power", "x2 power",
                            "x3 power", "coefficients"),
              optional = TRUE)
printPolynomial(pol_degrees, pol_coefficients)
```

```
# Calculate density approximation at point x
dtrhpa(x = x,
       pol_coefficients = pol_coefficients,
       pol_degrees = pol_degrees,
       mean = mean, sd = sd,
       tr_left = tr_left,
       tr_right = tr_right)

# Condition second component to be 0.5
# Substitute x second component with the conditional value 0.5
x <- matrix(c(0.1, 0.5, 0.3), nrow = 1)
# Set the second component to TRUE indicating that it is conditioned
given_ind <- c(FALSE, TRUE, FALSE)
# Calculate conditional (on x2 = 0.5) density approximation at point x
dtrhpa(x = x,
       pol_coefficients = pol_coefficients,
       pol_degrees = pol_degrees,
       mean = mean, sd = sd,
       given_ind = given_ind,
       tr_left = tr_left, tr_right = tr_right)

# Consider third component marginal distribution
# conditioned on the second component 0.5 value
# Set the first component to TRUE indicating that it is omitted
omit_ind <- c(TRUE, FALSE, FALSE)

# Calculate conditional (on X2 = 0.5) marginal (for X3)
# density approximation at point x
dtrhpa(x = x,
       pol_coefficients = pol_coefficients,
       pol_degrees = pol_degrees,
       mean = mean, sd = sd,
       given_ind = given_ind, omit_ind = omit_ind,
       tr_left = tr_left, tr_right = tr_right)

## Example demonstrating itrhpa function application.
## Let's approximate three truncated random variables (X1, X2, X3) the joint
## interval distribution function at lower and upper points (0,1, 0.2, 0.3)
## and (0.4, 0.5, 0.6), respectively, with a Hermite polynomial of (1 ,2, 3)
## degrees whose polynomial coefficients equal 1 except the coefficient
## related to x1*(x3^2) polynomial element which equals 2. Also suppose
## that the normal density's mean vector equals (1.1, 1.2, 1.3) while
## the standard deviation vector is (2.1, 2.2, 2.3). Suppose that lower and
## upper truncation are (-1.1,-1.2,-1.3) and (1.1,1.2,1.3), respectively.

# Prepare initial values
x_lower <- matrix(c(0.1, 0.2, 0.3), nrow=1)
x_upper <- matrix(c(0.4, 0.5, 0.6), nrow=1)
tr_left = matrix(c(-1.1,-1.2,-1.3), nrow = 1)
tr_right = matrix(c(1.1,1.2,1.3), nrow = 1)
mean <- c(1.1, 1.2, 1.3)
sd <- c(2.1, 2.2, 2.3)
```

```
pol_degrees <- c(1, 2, 3)

# Create polynomial powers and indexes correspondence matrix
pol_ind <- polynomialIndex(pol_degrees)
# Set all polynomial coefficients to 1
pol_coefficients <- rep(1, ncol(pol_ind))
pol_degrees_n <- length(pol_degrees)

# Assign coefficient 2 to the polynomial element (x1 ^ 1)*(x2 ^ 0)*(x3 ^ 2)
pol_coefficients[apply(pol_ind, 2, function(x) all(x == c(1, 0, 2)))] <- 2
# Visualize correspondence between polynomial
# elements and their coefficients
as.data.frame(rbind(pol_ind, pol_coefficients),
              row.names = c("x1 power", "x2 power",
                            "x3 power", "coefficients"),
              optional = TRUE)
printPolynomial(pol_degrees, pol_coefficients)

# Calculate intdf approximation at points x_lower and x_upper
itrhpa(x_lower = x_lower, x_upper = x_upper,
       pol_coefficients = pol_coefficients,
       pol_degrees = pol_degrees,
       mean = mean, sd = sd,
       tr_left = tr_left, tr_right = tr_right)

# Condition second component to be 0.7
# Substitute x second component with the conditional value 0.7
x_upper <- matrix(c(0.4, 0.7, 0.6), nrow = 1)
# Set the second component to TRUE indicating that it is conditioned
given_ind <- c(FALSE, TRUE, FALSE)

# Calculate conditional (on X2 = 0.7) the intdf
# approximation at points x_lower and x_upper
itrhpa(x_lower = x_lower, x_upper = x_upper,
       pol_coefficients = pol_coefficients,
       pol_degrees = pol_degrees,
       mean = mean, sd = sd,
       given_ind = given_ind,
       tr_left = tr_left, tr_right = tr_right)

# Consider third component marginal distribution
# conditioned on the second component 0.7 value
# Set the first component to TRUE indicating that it is omitted
omit_ind <- c(TRUE, FALSE, FALSE)

# Calculate conditional (on X2 = 0.7) marginal (for X3) the intdf
# approximation at points x_lower and x_upper
itrhpa(x_lower = x_lower, x_upper = x_upper,
       pol_coefficients = pol_coefficients,
       pol_degrees = pol_degrees,
       mean = mean, sd = sd,
       given_ind = given_ind, omit_ind = omit_ind,
       tr_left = tr_left, tr_right = tr_right)
```

```
## Example demonstrating dhpaDiff function application.
## Let's approximate three random variables (X1, X2, X3) the joint density
## function at point (0.1, 0.2, 0.3) with a Hermite polynomial of (1,2,3)
## degrees whose polynomial coefficients are 1 except the coefficient related
## to x1*(x3^2) polynomial element which equals 2. Also suppose that the
## normal density related mean vector equals (1.1, 1.2, 1.3) while the
## standard deviations vector is (2.1, 2.2, 2.3). In this example let's
## calculate the density approximating function's gradient with respect
## to various parameters

# Prepare initial values
x <- matrix(c(0.1, 0.2, 0.3), nrow = 1)
mean <- c(1.1, 1.2, 1.3)
sd <- c(2.1, 2.2, 2.3)
pol_degrees <- c(1, 2, 3)

# Create polynomial powers and indexes correspondence matrix
pol_ind <- polynomialIndex(pol_degrees)
# Set all polynomial coefficients to 1
pol_coefficients <- rep(1, ncol(pol_ind))
pol_degrees_n <- length(pol_degrees)

# Assign coefficient 2 to the polynomial element (x1 ^ 1)*(x2 ^ 0)*(x3 ^ 2)
pol_coefficients[apply(pol_ind, 2, function(x) all(x == c(1, 0, 2)))] <- 2

# Visualize correspondence between polynomial
# elements and their coefficients
as.data.frame(rbind(pol_ind, pol_coefficients),
              row.names = c("x1 power", "x2 power",
                            "x3 power", "coefficients"),
              optional = TRUE)
printPolynomial(pol_degrees, pol_coefficients)

# Calculate density approximation gradient with
# respect to the polynomial coefficients at point x
dhpaDiff(x = x,
         pol_coefficients = pol_coefficients,
         pol_degrees = pol_degrees,
         mean = mean, sd = sd)

# Condition second component to be 0.5
# Substitute x second component with the conditional value 0.5
x <- matrix(c(0.1, 0.5, 0.3), nrow = 1)
# Set TRUE to the second component indicating that it is conditioned
given_ind <- c(FALSE, TRUE, FALSE)

# Calculate conditional (on x2 = 0.5) density approximation's
# gradient with respect to polynomial coefficients at point x
dhpaDiff(x = x,
         pol_coefficients = pol_coefficients,
         pol_degrees = pol_degrees,
         mean = mean, sd = sd,
```

```
         given_ind = given_ind)

# Consider third component marginal distribution
# conditioned on the second component 0.5 value
# Set TRUE to the first component indicating that it is omitted
omit_ind <- c(TRUE, FALSE, FALSE)

# Calculate conditional (on X2 = 0.5) marginal (for X3) density
# approximation's gradient with respect to:
  # polynomial coefficients
dhpaDiff(x = x,
         pol_coefficients = pol_coefficients,
         pol_degrees = pol_degrees,
         mean = mean, sd = sd,
         given_ind = given_ind,
         omit_ind = omit_ind)
  # mean
dhpaDiff(x = x,
         pol_coefficients = pol_coefficients,
         pol_degrees = pol_degrees,
         mean = mean, sd = sd,
         given_ind = given_ind,
         omit_ind = omit_ind,
         type = "mean")
  # sd
dhpaDiff(x = x,
         pol_coefficients = pol_coefficients,
         pol_degrees = pol_degrees,
         mean = mean, sd = sd,
         given_ind = given_ind,
         omit_ind = omit_ind,
         type = "sd")
 # x
dhpaDiff(x = x,
         pol_coefficients = pol_coefficients,
         pol_degrees = pol_degrees,
         mean = mean, sd = sd,
         given_ind = given_ind,
         omit_ind = omit_ind,
         type = "x")

## Example demonstrating ehpaDiff function application.
## Let's approximate three random variables (X1, X2, X3) the expectation
## of the form E((X1 ^ 3) * (X2 ^ 1) * (X3 ^ 2)) and calculate the gradient

# Distribution parameters
mean <- c(1.1, 1.2, 1.3)
sd <- c(2.1, 2.2, 2.3)
pol_degrees <- c(1, 2, 3)
pol_coefficients_n <- prod(pol_degrees + 1)
pol_coefficients <- rep(1, pol_coefficients_n)

# Set powers for expectation
```

```
expectation_powers <- c(3, 1, 2)

# Calculate expectation approximation gradient
# with respect to all parameters
ehpaDiff(pol_coefficients = pol_coefficients,
         pol_degrees = pol_degrees,
         mean = mean, sd = sd,
         expectation_powers = expectation_powers,
         type = "all")

# Let's calculate gradient of E(X1 ^ 3 | (X2 = 1, X3 = 2))
x <- c(0, 1, 2)                  # x[1] may be arbitrary (not NA) values
expectation_powers <- c(3, 0, 0) # expectation_powers[2:3] may be
                                 # arbitrary (not NA) values
given_ind <- c(2, 3)
ehpaDiff(x = x,
         pol_coefficients = pol_coefficients,
         pol_degrees = pol_degrees,
         mean = mean, sd = sd,
         given_ind = given_ind,
         expectation_powers = expectation_powers,
         type = "all")
## Example demonstrating ihpaDiff function application.
## Let's approximate three random variables (X1, X2, X3 ) the joint interval
## distribution function (intdf) at lower and upper points (0,1, 0.2, 0.3)
## and (0.4, 0.5, 0.6), respectively, with a Hermite polynomial of (1, 2, 3)
## degrees whose polynomial coefficients equal 1 except the coefficient
## related to x1*(x3^2) polynomial element which equals 2.
## Also suppose that the normal density's mean vector equals
## (1.1, 1.2, 1.3) while the standard deviation vector is (2.1, 2.2, 2.3).
## In this example let's calculate the interval distribution approximating
## function gradient with respect to the polynomial coefficients.

# Prepare initial values
x_lower <- matrix(c(0.1, 0.2, 0.3), nrow=1)
x_upper <- matrix(c(0.4, 0.5, 0.6), nrow=1)
mean <- c(1.1, 1.2, 1.3)
sd <- c(2.1, 2.2, 2.3)
pol_degrees <- c(1, 2, 3)

# Create polynomial powers and indexes correspondence matrix
pol_ind <- polynomialIndex(pol_degrees)

# Set all polynomial coefficients to 1
pol_coefficients <- rep(1, ncol(pol_ind))
pol_degrees_n <- length(pol_degrees)

# Assign coefficient 2 to the polynomial element (x1 ^ 1)*(x2 ^ 0)*(x3 ^ 2)
pol_coefficients[apply(pol_ind, 2, function(x) all(x == c(1, 0, 2)))] <- 2

# Visualize correspondence between polynomial
# elements and their coefficients
as.data.frame(rbind(pol_ind, pol_coefficients),
```

```
              row.names = c("x1 power", "x2 power",
                            "x3 power", "coefficients"),
              optional = TRUE)
printPolynomial(pol_degrees, pol_coefficients)

# Calculate intdf approximation gradient with respect to the
# polynomial coefficients at points x_lower and x_upper
ihpaDiff(x_lower = x_lower, x_upper = x_upper,
         pol_coefficients = pol_coefficients,
         pol_degrees = pol_degrees,
         mean = mean, sd = sd)

# Condition second component to be 0.7
# Substitute x second component with the conditional value 0.7
x_upper <- matrix(c(0.4, 0.7, 0.6), nrow = 1)

# Set the second component to TRUE indicating that it is conditioned
given_ind <- c(FALSE, TRUE, FALSE)

# Calculate conditional (on X2 = 0.7) the intdf approximation
# with respect to the polynomial coefficients at points x_lower and x_upper
ihpaDiff(x_lower = x_lower, x_upper = x_upper,
         pol_coefficients = pol_coefficients,
         pol_degrees = pol_degrees,
         mean = mean, sd = sd,
         given_ind = given_ind)

# Consider third component marginal distribution
# conditioned on the second component 0.7 value
# Set the first component to TRUE indicating that it is omitted
omit_ind <- c(TRUE, FALSE, FALSE)

# Calculate conditional (on X2 = 0.7) marginal (for X3) the intdf
# approximation with respect to:
  # polynomial coefficients
ihpaDiff(x_lower = x_lower, x_upper = x_upper,
         pol_coefficients = pol_coefficients,
         pol_degrees = pol_degrees,
         mean = mean, sd = sd,
         given_ind = given_ind, omit_ind = omit_ind)
  # mean
ihpaDiff(x_lower = x_lower, x_upper = x_upper,
         pol_coefficients = pol_coefficients,
         pol_degrees = pol_degrees,
         mean = mean, sd = sd,
         given_ind = given_ind, omit_ind = omit_ind,
         type = "mean")
  # sd
ihpaDiff(x_lower = x_lower, x_upper = x_upper,
         pol_coefficients = pol_coefficients,
         pol_degrees = pol_degrees,
         mean = mean, sd = sd,
         given_ind = given_ind, omit_ind = omit_ind,
```

```
         type = "sd")
  # x_lower
ihpaDiff(x_lower = x_lower, x_upper = x_upper,
         pol_coefficients = pol_coefficients,
         pol_degrees = pol_degrees,
         mean = mean, sd = sd,
         given_ind = given_ind, omit_ind = omit_ind,
         type = "x_lower")
  # x_upper
ihpaDiff(x_lower = x_lower, x_upper = x_upper,
          pol_coefficients = pol_coefficients,
          pol_degrees = pol_degrees,
          mean = mean, sd = sd,
          given_ind = given_ind, omit_ind = omit_ind,
          type = "x_upper")

## Examples demonstrating qhpa function application.

## Sub-example 1 - univariate distribution
## Consider random variable X

# Distribution parameters
mean <- 1
sd <- 2
pol_degrees <- 2
pol_coefficients <- c(1, 0.1, -0.01)

# The level of quantile
p <- 0.7

# Calculate quantile of X
qhpa(p = p,
     pol_coefficients = pol_coefficients,
     pol_degrees = pol_degrees,
     mean = mean, sd = sd)

## Sub-example 2 - marginal distribution
## Consider random vector (X1, X2) and the quantile of X1

# Distribution parameters
mean <- c(1, 1.2)
sd <- c(2, 3)
pol_degrees <- c(2, 2)
pol_coefficients <- c(1, 0.1, -0.01, 0.2, 0.012,
                      0.0013, 0.0042, 0.00025, 0)

# The level of quantile
p <- 0.7

# Calculate quantile of X1
qhpa(p = p,
     pol_coefficients = pol_coefficients,
     pol_degrees = pol_degrees,
```

```
     mean = mean, sd = sd,
     omit_ind = 2)                          # set the omitted variable index

## Sub-example 3 - marginal and conditional distribution
## Consider random vector (X1, X2, X3) and
## the quantiles of X1|X3 and X1|(X2,X3)
mean <- c(1, 1.2, 0.9)
sd <- c(2, 3, 2.5)
pol_degrees <- c(1, 1, 1)
pol_coefficients <- c(1, 0.1, -0.01, 0.2, 0.012,
                      0.0013, 0.0042, 0.00025)

# The level of quantile
p <- 0.7

# Calculate quantile of X1|X3 = 0.2
qhpa(p = p,
     x = matrix(c(NA, NA, 0.2), nrow = 1),  # set any values to
                                            # the unconditioned and
                                            # omitted components
     pol_coefficients = pol_coefficients,
     pol_degrees = pol_degrees,
     mean = mean, sd = sd,
     omit_ind = 2,                          # set the omitted variable index
     given_ind = 3)                         # set the conditioned variable index

# Calculate quantile of X1|(X2 = 0.5, X3 = 0.2)
qhpa(p = p,
     x = matrix(c(NA, 0.5, 0.2), nrow = 1), # set any values to the
                                            # unconditioned and
                                            # omitted components
     pol_coefficients = pol_coefficients,
     pol_degrees = pol_degrees,
     mean = mean, sd = sd,
     given_ind = c(2, 3))                   # set the conditioned
                                            # variables indexes

## Examples demonstrating rhpa function application.

# Set seed for reproducibility
set.seed(123)

# Distribution parameters
mean <- 1
sd <- 2
pol_degrees <- 2
pol_coefficients <- c(1, 0.1, -0.01)

# Simulate two observations from this distribution
rhpa(n = 2,
     pol_coefficients = pol_coefficients,
     pol_degrees = pol_degrees,
     mean = mean, sd = sd)
```

hpaDist0 *Fast pdf and cdf for standardized univariate PGN distribution*

Description

These functions use fast algorithms to calculate densities and probabilities (along with their derivatives) related to the standardized PGN distribution.

Usage

```
dhpa0(
  x,
  pc,
  mean = 0,
  sd = 1,
  is_parallel = FALSE,
  log = FALSE,
  is_validation = TRUE,
  is_grad = FALSE
)

phpa0(
  x,
  pc,
  mean = 0,
  sd = 1,
  is_parallel = FALSE,
  log = FALSE,
  is_validation = TRUE,
  is_grad = FALSE
)
```

Arguments

x	numeric vector of function arguments.
pc	polynomial coefficients without the first term.
mean	expected value (mean) of the distribution.
sd	standard deviation of the distribution.
is_parallel	logical; if TRUE then multiple cores will be used for some calculations. Currently unavailable.
log	logical; if TRUE then probabilities p are given as log(p) or derivatives will be given with respect to log(p).
is_validation	logical value indicating whether function input arguments should be validated. Set it to FALSE for a slight performance boost (default value is TRUE).

is_grad — logical; if TRUE then function returns gradients with respect to x and pc (default is FALSE).

Details

Functions dhpa0 and phpa0 are similar to dhpa and phpa, respectively. However, there are two key differences. First, dhpa0 and phpa0 deal only with univariate PGN distributions. Second, this distribution is standardized to zero mean and unit variance. Moreover pc is similar to pol_coefficients argument of dhpa but without the first component, i.e., pc = pol_coefficients[-1]. Also mean and sd are not the arguments of the normal density but actual mean and standard deviation of the resulting distribution. So if these arguments are different from 0 and 1 correspondingly then standardized PGN distribution will be linearly transformed to have mean mean and standard deviation sd.

Value

Both functions return a list. Function dhpa0 returns a list with element named "den" that is a numeric vector of density values. Function phpa0 returns a list with element named "prob" that is a numeric vector of probabilities.

If is_grad = TRUE, then elements "grad_x" and "grad_pc" will be added to the list containing gradients with respect to input argument x and parameters pc, respectively. If log = TRUE, then additional elements will be added to the list containing density, probability and gradient values for logarithms of corresponding functions. These elements will be named as "grad_x_log", "grad_pc_log", "prob_log" and "den_log".

Examples

```
# Calculate density and probability of standardized PGN distribution
  # distribution parameters
pc <- c(0.5, -0.2)
  # function arguments
x <- c(-0.3, 0.8, 1.5)
  # probability density function
dhpa0(x, pc)
  # cumulative distribution function
phpa0(x, pc)

# Additionally calculate gradients with respect to arguments
# and parameters of the PGN distribution
dhpa0(x, pc, is_grad = TRUE)
phpa0(x, pc, is_grad = TRUE)

# Let X be a standardized PGN random variable and repeat
# the calculations for 2 * X + 1
dhpa0(x, pc, is_grad = TRUE, mean = 1, sd = 2)
phpa0(x, pc, is_grad = TRUE, mean = 1, sd = 2)
```

hpaML *Semi-nonparametric maximum likelihood estimation*

Description

This function performs semi-nonparametric (SNP) maximum likelihood estimation of unknown (possibly truncated) multivariate density using Hermite polynomial based approximating function proposed by Gallant and Nychka in 1987. Please, see dhpa 'Details' section to get more information concerning this approximating function.

Usage

```
hpaML(
  data,
  pol_degrees = numeric(0),
  tr_left = numeric(0),
  tr_right = numeric(0),
  given_ind = numeric(0),
  omit_ind = numeric(0),
  x0 = numeric(0),
  cov_type = "sandwich",
  boot_iter = 100L,
  is_parallel = FALSE,
  opt_type = "optim",
  opt_control = NULL,
  is_validation = TRUE
)
```

Arguments

data	numeric matrix whose rows are realizations of independent and identically distributed random vectors while columns correspond to variables.
pol_degrees	a non-negative integer vector of polynomial degrees (orders).
tr_left	numeric vector of left (lower) truncation limits.
tr_right	numeric vector of right (upper) truncation limits.
given_ind	logical or numeric vector indicating whether corresponding random vector component is conditioned. By default it is a logical vector of FALSE values. If given_ind[i] equals TRUE or i then the i-th column of x matrix will contain conditional values.
omit_ind	logical or numeric vector indicating whether the corresponding random vector component is omitted. By default it is a logical vector of FALSE values. If omit_ind[i] equals TRUE or i then the values in the i-th column of the x matrix will be ignored.
x0	numeric vector of the optimization routine initial values. Note that x0=c(pol_coefficients[-1], mean, sd). For pol_coefficients, mean, and sd documentation, see dhpa function.

cov_type	character determining the type of covariance matrix to be returned and used for summary. If cov_type = "hessian" then negative inverse of the Hessian matrix will be applied. If cov_type = "gop" then inverse of the Jacobian outer product will be used. If cov_type = "sandwich" (default) then the sandwich covariance matrix estimator will be applied. If cov_type = "bootstrap" then bootstrap with boot_iter iterations will be used. If cov_type = "hessianFD" or cov_type = "sandwichFD" then (probably) more accurate but computationally demanding central difference Hessian approximation will be calculated for the inverse Hessian and sandwich estimators, respectively. Central differences are computed via analytically provided gradient. This Hessian matrix estimation approach seems to be less accurate than the BFGS approximation if polynomial order is high (usually greater than 5).
boot_iter	the number of bootstrap iterations for the covariance matrix estimator when cov_type = "bootstrap".
is_parallel	if TRUE then multiple cores will be used for some calculations. It usually provides speed advantage for large enough samples (more than 1000 observations).
opt_type	string value determining the type of the optimization routine to be applied. The default is "optim" meaning that the BFGS method from the optim function will be applied. If opt_type = "GA" then the ga function will be additionally applied.
opt_control	a list containing arguments to be passed to the optimization routine depending on the opt_type argument value. Please see details to get additional information.
is_validation	logical value indicating whether function input arguments should be validated. Set it to FALSE for a slight performance boost (default value is TRUE).

Details

The Hermite polynomial approximation approach for densities has been proposed by A. Gallant and D. W. Nychka in 1987. The main idea is to approximate unknown distribution density with a scaled Hermite polynomial. For more information please refer to the literature listed below.

Let's use notations introduced in dhpa 'Details' section. Function hpaML maximizes the following quasi log-likelihood function:

$$\ln L(\alpha, \mu, \sigma; x) = \sum_{i=1}^{n} \ln \left(f_\xi(x_i; \alpha, \mu, \sigma) \right),$$

where (in addition to previously defined notations):

x_i - are observations, i.e., the data matrix rows.

n - is sample size, i.e., the number of data matrix rows.

Arguments pol_degrees, tr_left, tr_right, given_ind and omit_ind affect the form of $f_\xi\left(x_i; \alpha, \mu, \sigma\right)$ in a way described in dhpa 'Details' section. Note that change of given_ind and omit_ind values may result in estimator whose statistical properties have not been rigorously investigated yet.

The first polynomial coefficient (zero powers) is set to 1 for identification purposes, i.e., $\alpha_{(0,\ldots,0)} = 1$.

All NA and NaN values will be removed from data matrix.

The function calculates standard errors via the sandwich estimator and significance levels are reported taking into account quasi maximum likelihood estimator (QMLE) asymptotic normality. If one wants to switch from QMLE to semi-nonparametric estimator (SNPE) during hypothesis testing then covariance matrix should be estimated again using the bootstrap.

This function maximizes (quasi) log-likelihood function via the optim function setting its method argument to "BFGS". If opt_type = "GA" then the genetic algorithm from the ga function will be additionally (after optim putting its solution (par) into the suggestions matrix) applied in order to perform global optimization. Note that global optimization takes much more time (usually minutes but sometimes hours or even days). The number of iterations and population size of the genetic algorithm will grow linearly along with the number of estimated parameters. If it seems that global maximum has not been found then it is possible to continue the search restarting the function setting its input argument x0 to x1 output value. Note that if cov_type = "bootstrap" then the ga function will not be used for bootstrap iterations since it may be extremely time consuming.

If opt_type = "GA" then opt_control should be a list containing the values to be passed to the ga function. It is possible to pass arguments lower, upper, popSize, pcrossover, pmutation, elitism, maxiter, suggestions, optim, optimArgs, seed and monitor. Note that it is possible to set population, selection, crossover and mutation arguments changing ga default parameters via gaControl function. These arguments information is reported in ga. In order to provide manual values for lower and upper bounds please follow the parameters ordering mentioned above for the x0 argument. If these bounds are not provided manually then they (except those related to the polynomial coefficients) will depend on the estimates obtained by local optimization via optim function (these estimates will be in the middle between lower and upper). Specifically for each sd parameter lower (upper) bound is 5 times lower (higher) than this parameter's optim estimate. For each mean and regression coefficient parameter its lower and upper bounds deviate from corresponding optim estimate by two absolute values of this estimate. Finally, lower and upper bounds for each polynomial coefficient are -10 and 10, respectively (do not depend on their optim estimates).

The following arguments differ from their defaults in ga:

- pmutation = 0.2.
- optim = TRUE.
- optimArgs = list("method" = "Nelder-Mead", "poptim" = 0.2, "pressel" = 0.5).
- seed = 8.
- elitism = 2 + round(popSize * 0.1).

The arguments popSize and maxiter of the ga function have been set proportional to the number of estimated polynomial coefficients:

- popSize = 10 + (prod(pol_degrees + 1) - 1) * 2.
- maxiter = 50 * (prod(pol_degrees + 1))

Value

This function returns an object of class "hpaML".

An object of class "hpaML" is a list containing the following components:

- optim - optim function output. If opt_type = "GA", then it is the list containing optim and ga functions outputs.
- x1 - numeric vector of distribution parameters estimates.
- mean - density function mean vector estimate.
- sd - density function sd vector estimate.
- pol_coefficients - polynomial coefficients estimates.
- tr_left - the same as tr_left input parameter.
- tr_right - the same as tr_right input parameter.
- omit_ind - the same as omit_ind input parameter.
- given_ind - the same as given_ind input parameter.
- cov_mat - covariance matrix estimate.
- results - numeric matrix representing estimation results.
- log-likelihood - value of Log-Likelihood function.
- AIC - AIC value.
- data - the same as data input parameter but without NA observations.
- n_obs - number of observations.
- bootstrap - list where bootstrap estimation results are stored.

References

A. Gallant and D. W. Nychka (1987) <doi:10.2307/1913241>

See Also

summary.hpaML, predict.hpaML, logLik.hpaML, plot.hpaML

Examples

```
## Approximate Student (t) distribution

# Set seed for reproducibility
set.seed(123)

# Simulate 5000 realizations of Student distribution
# with 5 degrees of freedom
n <- 5000
df <- 5
x <- matrix(rt(n, df), ncol = 1)
pol_degrees <- c(4)

# Apply pseudo maximum likelihood routine
ml_result <- hpa::hpaML(data = x, pol_degrees = pol_degrees)
summary(ml_result)

# Get predicted probabilities (density values) approximations
predict(ml_result)
```

```
# Plot density approximation
plot(ml_result)

## Approximate chi-squared distribution

# Set seed for reproducibility
set.seed(123)

# Simulate 5000 realizations of chi-squared distribution
# with 5 degrees of freedom

n <- 5000
df <- 5
x <- matrix(rchisq(n, df), ncol = 1)
pol_degrees <- c(5)

# Apply pseudo maximum likelihood routine
ml_result <- hpaML(data = x, pol_degrees = as.vector(pol_degrees),
tr_left = 0)
summary(ml_result)

# Get predicted probabilities (density values) approximations
predict(ml_result)

# Plot density approximation
plot(ml_result)

## Approximate multivariate Student (t) distribution
## Note that calculations may take up to a minute

# Set seed for reproducibility
set.seed(123)

# Simulate 5000 realizations of three dimensional Student distribution
# with 5 degrees of freedom
library("mvtnorm")
cov_mat <- matrix(c(1, 0.5, -0.5, 0.5, 1, 0.5, -0.5, 0.5, 1), ncol = 3)
x <- rmvt(n = 5000, sigma = cov_mat, df = 5)

# Estimate approximating joint distribution parameters
ml_result <- hpaML(data = x, pol_degrees = c(1, 1, 1))

# Get summary
summary(ml_result)

# Get predicted values for joint density function
predict(ml_result)

# Plot density approximation for the
# second random variable
plot(ml_result, ind = 2)
```

```
# Plot density approximation for the
# second random variable conditioning
# on x1 = 1
plot(ml_result, ind = 2, given = c(1, NA, NA))

## Approximate Student (t) distribution and plot densities approximated
## under different Hermite polynomial degrees against
## the true density (of Student distribution)

# Simulate 5000 realizations of t-distribution with 5 degrees of freedom
n <- 5000
df <- 5
x <- matrix(rt(n, df), ncol=1)

# Apply pseudo maximum likelihood routine
# Create matrix of lists where i-th element contains hpaML results for K=i
ml_result <- matrix(list(), 4, 1)
for(i in 1:4)
{
 ml_result[[i]] <- hpa::hpaML(data = x, pol_degrees = i)
}

# Generate test values
test_values <- seq(qt(0.001, df), qt(0.999, df), 0.001)
n0 <- length(test_values)

# t-distribution density function at the test values points
true_pred <- dt(test_values, df)

# Create matrix of lists where i-th element contains
# density predictions for K=i
PGN_pred <- matrix(list(), 4, 1)
for(i in 1:4)
{
  PGN_pred[[i]] <- predict(object = ml_result[[i]],
                           newdata = matrix(test_values, ncol=1))
}
# Plot the result
library("ggplot2")

# prepare the data
h <- data.frame("values" = rep(test_values,5),
                "predictions" = c(PGN_pred[[1]],PGN_pred[[2]],
                                  PGN_pred[[3]],PGN_pred[[4]],
                                  true_pred),
                "Density" = c(
                  rep("K=1",n0), rep("K=2",n0),
                  rep("K=3",n0), rep("K=4",n0),
                  rep("t-distribution",n0))
                  )

# build the plot
ggplot(h, aes(values, predictions)) + geom_point(aes(color = Density)) +
```

```
  theme_minimal() + theme(legend.position = "top",
                          text = element_text(size=26),
                          legend.title=element_text(size=20),
                          legend.text=element_text(size=28)) +
  guides(colour = guide_legend(override.aes = list(size=10)))

# Get informative estimates summary for K=4
summary(ml_result[[4]])
```

---

hpaSelection *Perform semi-nonparametric selection model estimation*

Description

This function performs semi-nonparametric (SNP) maximum likelihood estimation of sample selection model using Hermite polynomial based approximating function proposed by Gallant and Nychka in 1987. Please, see dhpa 'Details' section to get more information concerning this approximating function.

Usage

```
hpaSelection(
  selection,
  outcome,
  data,
  selection_K = 1L,
  outcome_K = 1L,
  pol_elements = 3L,
  is_Newey = FALSE,
  x0 = numeric(0),
  is_Newey_loocv = FALSE,
  cov_type = "sandwich",
  boot_iter = 100L,
  is_parallel = FALSE,
  opt_type = "optim",
  opt_control = NULL,
  is_validation = TRUE
)
```

Arguments

selection	an object of class "formula" (or one that can be coerced to that class): a symbolic description of the selection equation form. All variables in selection should be numeric vectors of the same length.

outcome	an object of class "formula" (or one that can be coerced to that class): a symbolic description of the outcome equation form. All variables in outcome should be numeric vectors of the same length.
data	data frame containing the variables in the model.
selection_K	a non-negative integer representing the polynomial degree related to the selection equation.
outcome_K	non-negative integer representing the polynomial degree related to the outcome equation.
pol_elements	number of conditional expectation approximating terms for Newey's method. If is_Newey_loocv is TRUE then determines maximum number of these terms during leave-one-out cross-validation.
is_Newey	logical; if TRUE then returns only Newey's method estimation results (default value is FALSE).
x0	numeric vector of the optimization routine initial values. Note that x0 = c(pol_coefficients[-1], mean, sd, z_coef, y_coef).
is_Newey_loocv	logical; if TRUE then number of conditional expectation approximating terms for Newey's method will be selected based on leave-one-out cross-validation criteria iterating through 0 to pol_elements number of these terms.
cov_type	character determining the type of covariance matrix to be returned and used for summary. If cov_type = "hessian" then negative inverse of the Hessian matrix will be applied. If cov_type = "gop" then inverse of the Jacobian outer product will be used. If cov_type = "sandwich" (default) then the sandwich covariance matrix estimator will be applied. If cov_type = "bootstrap" then bootstrap with boot_iter iterations will be used. If cov_type = "hessianFD" or cov_type = "sandwichFD" then (probably) more accurate but computationally demanding central difference Hessian approximation will be calculated for the inverse Hessian and sandwich estimators, respectively. Central differences are computed via analytically provided gradient. This Hessian matrix estimation approach seems to be less accurate than the BFGS approximation if polynomial order is high (usually greater than 5).
boot_iter	the number of bootstrap iterations for the covariance matrix estimator when cov_type = "bootstrap".
is_parallel	if TRUE then multiple cores will be used for some calculations. It usually provides speed advantage for large enough samples (more than 1000 observations).
opt_type	string value determining the type of the optimization routine to be applied. The default is "optim" meaning that the BFGS method from the optim function will be applied. If opt_type = "GA" then the ga function will be additionally applied.
opt_control	a list containing arguments to be passed to the optimization routine depending on the opt_type argument value. Please see details to get additional information.
is_validation	logical value indicating whether function input arguments should be validated. Set it to FALSE for a slight performance boost (default value is TRUE).

Details

The Hermite polynomial approximation approach for densities has been proposed by A. Gallant and D. W. Nychka in 1987. The main idea is to approximate unknown distribution density with a scaled Hermite polynomial. For more information please refer to the literature listed below.

Let's use the notations introduced in the dhpa 'Details' section. Function hpaSelection maximizes the following quasi log-likelihood function:

$$\ln L(\gamma,\beta,\alpha,\mu,\sigma;x) = \sum_{i:z_i=1} \ln\left(\overline{F}_{(\xi_1|\xi_2=y_i-x_i^o\beta)}\left(-\gamma x_i^s,\infty;\alpha,\mu,\sigma\right)\right) f_{\xi_2}\left(y_i - x_i^o\beta\right) + \\ + \sum_{i:z_i=0} \ln\left(\overline{F}_{\xi}(-\infty,-x_i^s\gamma;\alpha,\mu,\sigma)\right),$$

where (in addition to previously defined notations):

x_i^s - is row vector of selection equation regressors derived from data according to selection formula.

x_i^o - is row vector of outcome equation regressors derived from data according to outcome formula.

γ - is column vector of selection equation regression coefficients (constant will not be added by default).

β - is column vector of outcome equation regression coefficients (constant will not be added by default).

z_i - binary (0 or 1) dependent variable defined in selection formula.

y_i - continuous dependent variable defined in outcome formula.

Note that ξ is two-dimensional and selection_K corresponds to K_1 while outcome_K determines K_2.

The first polynomial coefficient (zero powers) is set to 1 for identification purposes, i.e., $\alpha_0 = 1$.

Rows in data corresponding to variables mentioned in selection and outcome formulas which have at least one NA value will be ignored. The exception is the continuous dependent variable y which may have NA values for observations where $z_i = 0$.

Note that coefficient for the first independent variable in selection will be fixed to 1 i.e. $\gamma_1 = 1$.

All variables mentioned in selection and outcome should be numeric vectors.

The function calculates standard errors via the sandwich estimator and significance levels are reported taking into account quasi maximum likelihood estimator (QMLE) asymptotic normality. If one wants to switch from QMLE to semi-nonparametric estimator (SNPE) during hypothesis testing then covariance matrix should be estimated again using the bootstrap.

Initial values for optimization routine are obtained by Newey's method (see the reference below). In order to obtain initial values via least squares, set pol_elements = 0. Initial values for the outcome equation are obtained via the hpaBinary function setting K to selection_K.

Note that selection equation dependent variables should have exactly two levels (0 and 1) where "0" states for the selection results which leads to unobservable values of dependent variable in outcome equation.

This function maximizes (quasi) log-likelihood function via the optim function setting its method argument to "BFGS". If opt_type = "GA" then the genetic algorithm from the ga function will be

additionally (after optim putting its solution (par) into the suggestions matrix) applied in order to perform global optimization. Note that global optimization takes much more time (usually minutes but sometimes hours or even days). The number of iterations and population size of the genetic algorithm will grow linearly along with the number of estimated parameters. If it seems that global maximum has not been found then it is possible to continue the search restarting the function setting its input argument x0 to x1 output value. Note that if cov_type = "bootstrap" then the ga function will not be used for bootstrap iterations since it may be extremely time consuming.

If opt_type = "GA" then opt_control should be a list containing the values to be passed to the ga function. It is possible to pass arguments lower, upper, popSize, pcrossover, pmutation, elitism, maxiter, suggestions, optim, optimArgs, seed and monitor. Note that it is possible to set population, selection, crossover and mutation arguments changing ga default parameters via gaControl function. These arguments information is reported in ga. In order to provide manual values for lower and upper bounds please follow the parameters ordering mentioned above for the x0 argument. If these bounds are not provided manually then they (except those related to the polynomial coefficients) will depend on the estimates obtained by local optimization via optim function (these estimates will be in the middle between lower and upper). Specifically for each sd parameter lower (upper) bound is 5 times lower (higher) than this parameter's optim estimate. For each mean and regression coefficient parameter its lower and upper bounds deviate from corresponding optim estimate by two absolute values of this estimate. Finally, lower and upper bounds for each polynomial coefficient are -10 and 10, respectively (do not depend on their optim estimates).

The following arguments differ from their defaults in ga:

- pmutation = 0.2.
- optim = TRUE.
- optimArgs = list("method" = "Nelder-Mead", "poptim" = 0.2, "pressel" = 0.5).
- seed = 8.
- elitism = 2 + round(popSize * 0.1).

Let's denote by n_reg the number of regressors included in the selection and outcome formulas. The arguments popSize and maxiter of the ga function have been set proportional to the number of estimated polynomial coefficients and independent variables:

- popSize = 10 + 5 * (z_K + 1) * (y_K + 1) + 2 * n_reg
- maxiter = 50 * (z_K + 1) * (y_K + 1) + 10 * n_reg

Value

This function returns an object of class "hpaSelection".

An object of class "hpaSelection" is a list containing the following components:

- optim - optim function output. If opt_type = "GA" then it is the list containing optim and ga functions outputs.
- x1 - numeric vector of distribution parameters estimates.
- Newey - list containing information concerning Newey's method estimation results.
- selection_mean - estimate of the hermite polynomial mean parameter related to selection equation random error marginal distribution.

- outcome_mean - estimate of the hermite polynomial mean parameter related to outcome equation random error marginal distribution.
- selection_sd - estimate of sd parameter related to selection equation random error marginal distribution.
- outcome_sd - estimate of the hermite polynomial sd parameter related to outcome equation random error marginal distribution.
- pol_coefficients - polynomial coefficients estimates.
- pol_degrees - numeric vector which first element is selection_K and the second is outcome_K.
- selection_coef - selection equation regression coefficients estimates.
- outcome_coef - outcome equation regression coefficients estimates.
- cov_mat - covariance matrix estimate.
- results - numeric matrix representing estimation results.
- log-likelihood - value of Log-Likelihood function.
- re_moments - list which contains information about random errors expectations, variances and correlation.
- data_List - list containing model variables and their partition according to outcome and selection equations.
- n_obs - number of observations.
- ind_List - list which contains information about parameters indexes in x1.
- selection_formula - the same as selection input parameter.
- outcome_formula - the same as outcome input parameter.

Abovementioned list Newey has class "hpaNewey" and contains the following components:

- outcome_coef - regression coefficients estimates (except constant term which is part of conditional expectation approximating polynomial).
- selection_coef - regression coefficients estimates related to selection equation.
- constant_biased - biased estimate of constant term.
- inv_mills - inverse mills ratios estimates and their powers (including constant).
- inv_mills_coef - coefficients related to inv_mills.
- pol_elements - the same as pol_elements input parameter. However if is_Newey_loocv is TRUE then it will equal to the number of conditional expectation approximating terms for Newey's method which minimize leave-one-out cross-validation criteria.
- outcome_exp_cond - dependent variable conditional expectation estimates.
- selection_exp - selection equation random error expectation estimate.
- selection_var - selection equation random error variance estimate.
- hpaBinaryModel - object of class "hpaBinary" which contains selection equation estimation results.

Abovementioned list re_moments contains the following components:

- selection_exp - selection equation random errors expectation estimate.

- `selection_var` - selection equation random errors variance estimate.
- `outcome_exp` - outcome equation random errors expectation estimate.
- `outcome_var` - outcome equation random errors variance estimate.
- `errors_covariance` - outcome and selection equation random errors covariance estimate.
- `rho` - outcome and selection equation random errors correlation estimate.
- `rho_std` - outcome and selection equation random errors correlation estimator standard error estimate.

References

A. Gallant and D. W. Nychka (1987) <doi:10.2307/1913241>

W. K. Newey (2009) <https://doi.org/10.1111/j.1368-423X.2008.00263.x>

Mroz T. A. (1987) <doi:10.2307/1911029>

See Also

summary.hpaSelection, predict.hpaSelection, plot.hpaSelection, logLik.hpaSelection

Examples

```
## Let's estimate wage equation accounting for non-random selection.
## See the reference to Mroz TA (1987) to get additional details about
## the data this examples uses

# Prepare data
library("sampleSelection")
data("Mroz87")
h = data.frame("kids" = as.numeric(Mroz87$kids5 + Mroz87$kids618 > 0),
"age" = as.numeric(Mroz87$age),
"faminc" = as.numeric(Mroz87$faminc),
"educ" = as.numeric(Mroz87$educ),
"exper" = as.numeric(Mroz87$exper),
"city" = as.numeric(Mroz87$city),
"wage" = as.numeric(Mroz87$wage),
"lfp" = as.numeric(Mroz87$lfp))

# Estimate model parameters
model <- hpaSelection(selection = lfp ~ educ + age + I(age ^ 2) +
                                        kids + log(faminc),
                      outcome = log(wage) ~ exper + I(exper ^ 2) +
                                            educ + city,
                                  selection_K = 2, outcome_K = 3,
                                  data = h,
                                  pol_elements = 3, is_Newey_loocv = TRUE)
summary(model)

# Plot outcome equation random errors density
plot(model, type = "outcome")
# Plot selection equation random errors density
```

```
plot(model, type = "selection")


## Estimate semi-nonparametric sample selection model
## parameters on simulated data given chi-squared random errors


set.seed(100)
library("mvtnorm")

# Sample size

n <- 1000

# Simulate independent variables
X_rho <- 0.5
X_sigma <- matrix(c(1, X_rho, X_rho,
                    X_rho, 1, X_rho,
                    X_rho,X_rho,1),
                  ncol=3)
X <- rmvnorm(n=n, mean = c(0,0,0),
             sigma = X_sigma)

# Simulate random errors
epsilon <- matrix(0, n, 2)
epsilon_z_y <- rchisq(n, 5)
epsilon[, 1] <- (rchisq(n, 5) + epsilon_z_y) * (sqrt(3/20)) - 3.8736
epsilon[, 2] <- (rchisq(n, 5) + epsilon_z_y) * (sqrt(3/20)) - 3.8736
# Simulate selection equation
z_star <- 1 + 1 * X[,1] + 1 * X[,2] + epsilon[,1]
z <- as.numeric((z_star > 0))

# Simulate outcome equation
y_star <- 1 + 1 * X[,1] + 1 * X[,3] + epsilon[,2]
z <- as.numeric((z_star > 0))
y <- y_star
y[z==0] <- NA
h <- as.data.frame(cbind(z, y, X))
names(h) <- c("z", "y", "x1", "x2", "x3")

# Estimate parameters
model <- hpaSelection(selection = z ~ x1 + x2,
                      outcome = y ~ x1 + x3,
                      data = h,
                      selection_K = 1, outcome_K = 3)
summary(model)

# Get conditional predictions for outcome equation
model_pred_c <- predict(model, is_cond = TRUE)
# Conditional predictions y|z=1
model_pred_c$y_1
# Conditional predictions y|z=0
model_pred_c$y_0
```

```
# Get unconditional predictions for outcome equation
model_pred_u <- predict(model, is_cond = FALSE)
model_pred_u$y

# Get conditional predictions for selection equation
# Note that for z=0 these predictions are NA
predict(model, is_cond = TRUE, type = "selection")
# Get unconditional predictions for selection equation
predict(model, is_cond = FALSE, type = "selection")
```

`hsaDist` *Probabilities and Moments Hermite Spline Approximation*

Description

The set of functions similar to dhpa-like functions. The difference is that instead of a polynomial these functions utilize a spline.

Usage

```
dhsa(x, m, knots, mean = 0, sd = 1, log = FALSE)

ehsa(m, knots, mean = 0, sd = 1, power = 1)
```

Arguments

`x`	numeric vector of values for which the function should be estimated.
`m`	numeric matrix whose rows correspond to the spline intervals while columns represent the variables powers. Therefore the element in i-th row and j-th column represents the coefficient associated with the variable that 1) belongs to the i-th interval i.e. between the i-th and (i + 1)-th knots 2) raised to the power of (j - 1).
`knots`	a numeric vector sorted in ascending order representing the knots of the spline.
`mean`	expected value of a normal distribution.
`sd`	standard deviation of a normal distribution.
`log`	logical; if `TRUE` then probabilities p are given as log(p) or derivatives will be given with respect to log(p).
`power`	non-negative integer representing the moment order, i.e., E(X ^ power) will be estimated.

Details

In contrast to dhpa-like functions these functions may deal with univariate distributions only. In future this functions will be generalized to work with multivariate distributions. The main idea of these functions is to use squared spline instead of squared polynomial in order to provide greater numeric stability and approximation accuracy. To provide spline parameters please use `m` and `knots` arguments (i.e. instead of `pol_degrees` and `pol_coefficients` arguments that were used to specify the polynomial for dhpa-like functions).

Value

Function dhsa returns vector of probability densities of the same length as `x`. Function ehsa returns moment value.

See Also

dhpa, bsplineGenerate

Examples

```
## Examples demonstrating dhsa and ehsa functions' application.

# Generate b-splines
b <- bsplineGenerate(knots = c(-2.1, 1.5, 1.5, 2.2, 3.7, 4.2, 5),
                     degree = 3)

# Combine b-splines into a spline
spline <- bsplineComb(splines = b, weights = c(1.6, -1.2, 3.2))

# Assign parameters using the spline created above
knots <- spline$knots
m <- spline$m
mean <- 1
sd <- 2

# Estimate the density at particular points
x <- c(2, 3.7, 8)
dhsa(x,
     m = m, knots = knots,
     mean = mean, sd = sd)

# Calculate expected value
ehsa(m = m, knots = knots,
     mean = mean, sd = sd,
     power = 1)

# Evaluate the third moment
ehsa(m = m, knots = knots,
     mean = mean, sd = sd,
     power = 3)
```

logLik.hpaBinary *Calculates log-likelihood for "hpaBinary" object*

Description

This function calculates log-likelihood for a "hpaBinary" object

Usage

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

Arguments

object	Object of class "hpaBinary"
...	further arguments (currently ignored)

logLik.hpaML *Calculates log-likelihood for "hpaML" object*

Description

This function calculates log-likelihood for a "hpaML" object

Usage

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

Arguments

object	Object of class "hpaML"
...	further arguments (currently ignored)

logLik.hpaSelection *Calculates log-likelihood for "hpaSelection" object*

Description

This function calculates log-likelihood for a "hpaSelection" object

Usage

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

Arguments

object	Object of class "hpaSelection"
...	further arguments (currently ignored)

logLik_hpaBinary *Calculates log-likelihood for "hpaBinary" object*

Description

This function calculates log-likelihood for "hpaBinary" object

Usage

```
logLik_hpaBinary(object)
```

Arguments

object	Object of class "hpaBinary"

logLik_hpaML *Calculates log-likelihood for "hpaML" object*

Description

This function calculates log-likelihood for "hpaML" object

Usage

```
logLik_hpaML(object)
```

Arguments

object	Object of class "hpaML"

logLik_hpaSelection *Calculates log-likelihood for "hpaSelection" object*

Description

This function calculates log-likelihood for "hpaSelection" object

Usage

```
logLik_hpaSelection(object)
```

Arguments

object	Object of class "hpaSelection"

mecdf *Calculates multivariate empirical cumulative distribution function*

Description

This function calculates multivariate empirical cumulative distribution function at each point of the sample

Usage

```
mecdf(x)
```

Arguments

x	numeric matrix which rows are observations

normalMoment *Calculate k-th order moment of normal distribution*

Description

This function recursively calculates k-th order moment of normal distribution.

Usage

```
normalMoment(
  k = 0L,
  mean = 0,
  sd = 1,
  return_all_moments = FALSE,
  is_validation = TRUE,
  is_central = FALSE,
  diff_type = "NO"
)
```

Arguments

k	non-negative integer moment order.
mean	numeric expected value.
sd	positive numeric standard deviation.
return_all_moments	logical; if TRUE, function returns (k+1)-dimensional numeric vector of moments of normally distributed random variable with mean = mean and standard deviation = sd. Note that i-th vector's component value corresponds to the (i-1)-th moment.
is_validation	logical value indicating whether function input arguments should be validated. Set it to FALSE for a slight performance boost (default value is TRUE).
is_central	logical; if TRUE, then central moments will be calculated.
diff_type	string value indicating the type of the argument the moment should be differentiated with respect to. Default value is "NO" so the moments themselves will be returned. Alternative values are "mean" and "sd". Also "x_lower" and "x_upper" values are available for truncatedNormalMoment.

Details

This function estimates k-th order moment of normal distribution which mean equals to mean and standard deviation equals to sd.

Note that the value of parameter k automatically converts to an integer. So passing a non-integer k value will not cause any errors but the calculations will be performed for the rounded k value only.

Value

This function returns k-th order moment of normal distribution whose mean equals mean and whose standard deviation equals sd. If return_all_moments is TRUE then see this argument description above for output details.

Examples

```
## Calculate 5-th order moment of normal random variable which
## mean equals to 3 and standard deviation is 5.
```

```
# 5-th moment
normalMoment(k = 5, mean = 3, sd = 5)

# (0-5)-th moments
normalMoment(k = 5, mean = 3, sd = 5, return_all_moments = TRUE)

# 5-th moment derivative respect to mean
normalMoment(k = 5, mean = 3, sd = 5, diff_type = "mean")

# 5-th moment derivative respect to sd
normalMoment(k = 5, mean = 3, sd = 5, diff_type = "sd")
```

`plot.hpaBinary` *Plot the approximated density of hpaBinary random errors*

Description

Plot the approximated density of hpaBinary random errors

Usage

```
## S3 method for class 'hpaBinary'
plot(x, y = NULL, ...)
```

Arguments

`x`	Object of class "hpaBinary"
`y`	this parameter currently ignored
`...`	further arguments to be passed to the `plot` function.

`plot.hpaML` *Plot approximated marginal density using hpaML output*

Description

Plot approximated marginal density using hpaML output

Usage

```
## S3 method for class 'hpaML'
plot(x, y = NULL, ..., ind = 1, given = NULL)
```

Arguments

x	Object of class "hpaML"
y	this parameter currently ignored
...	further arguments to be passed to the plot function.
ind	index of random variable for which approximation to marginal density should be plotted
given	numeric vector of the same length as given_ind from x. Determines conditional values for the corresponding components. NA values in given vector indicate that corresponding random variable is not conditioned. By default all given components are NA so unconditional marginal density will be plotted for the ind-th random variable.

---

plot.hpaSelection *Plot hpaSelection random errors approximated density*

Description

Plot hpaSelection random errors approximated density

Usage

```
## S3 method for class 'hpaSelection'
plot(x, y = NULL, ..., type = "outcome")
```

Arguments

x	Object of class "hpaSelection"
y	this parameter currently ignored
...	further arguments to be passed to the plot function.
type	character; if "outcome" then function plots the graph for outcome equation random errors, if "selection" then plot for selection equation random errors will be generated.

pnorm_parallel *Calculate normal cdf in parallel*

Description

Calculate in parallel for each value from vector `x` distribution function of normal distribution with mean equal to `mean` and standard deviation equal to `sd`.

Usage

```
pnorm_parallel(x, mean = 0, sd = 1, is_parallel = FALSE)
```

Arguments

`x`	vector of quantiles: should be a numeric vector, not just a double value.
`mean`	double value.
`sd`	double positive value.
`is_parallel`	if `TRUE` then multiple cores will be used for some calculations. It usually provides speed advantage for large enough samples (more than 1000 observations).

polynomialIndex *Multivariate Polynomial Representation*

Description

Function `polynomialIndex` provides a matrix which allows to iterate through the elements of multivariate polynomial being aware of these elements' powers. So (i, j)-th element of the matrix is power of j-th variable in i-th multivariate polynomial element.

Function `printPolynomial` prints multivariate polynomial given its degrees (`pol_degrees`) and coefficients (`pol_coefficients`) vectors.

Usage

```
polynomialIndex(pol_degrees = numeric(0), is_validation = TRUE)

printPolynomial(pol_degrees, pol_coefficients, is_validation = TRUE)
```

Arguments

`pol_degrees`	a non-negative integer vector of polynomial degrees (orders).
`is_validation`	logical value indicating whether function input arguments should be validated. Set it to `FALSE` for a slight performance boost (default value is `TRUE`).
`pol_coefficients`	numeric vector of polynomial coefficients.

Details

Multivariate polynomial of degrees $(K_1, ..., K_m)$ (`pol_degrees`) has the form:

$$a_{(0,...,0)}x_1^0 * ... * x_m^0 + ... + a_{(K_1,...,K_m)}x_1^{K_1} * ... * x_m^{K_m},$$

where $a_{(i_1,...,i_m)}$ are polynomial coefficients, while polynomial elements are:

$$a_{(i_1,...,i_m)}x_1^{i_1} * ... * x_m^{i_m},$$

where $(i_1, ..., i_m)$ are polynomial element's powers corresponding to variables $(x_1, ..., x_m)$ respectively. Note that $i_j \in \{0, ..., K_j\}$.

Function `printPolynomial` removes polynomial elements whose coefficients are zero and variables whose powers are zero. Output may contain long coefficient representations as they are not rounded.

Value

Function `polynomialIndex` returns a matrix whose rows correspond to variables while columns correspond to powers. So (i, j)-th element of this matrix corresponds to the power i_j of the x_j variable in i-th polynomial element. Therefore i-th column of this matrix contains vector of powers $(i_1, ..., i_m)$ for the i-th polynomial element. So the function transforms m-dimensional elements indexing to one-dimensional.

Function `printPolynomial` returns the string which contains polynomial symbolic representation.

Examples

```
## Get polynomial indexes matrix for the polynomial
## whose degrees are (1, 3, 5)

polynomialIndex(c(1, 3, 5))

## Consider multivariate polynomial of degrees (2, 1) such that coefficients
## for elements whose powers sum is even are 2 and for those whose powers sum
## is odd are 5. So the polynomial is 2+5x2+5x1+2x1x2+2x1^2+5x1^2x2 where
## x1 and x2 are polynomial variables.

# Create variable to store polynomial degrees
pol_degrees <- c(2, 1)

# Let's represent its powers (not coefficients) in a matrix form
pol_matrix <- polynomialIndex(pol_degrees)

# Calculate polynomial elements' powers sums
pol_powers_sum <- pol_matrix[1, ] + pol_matrix[2, ]

# Let's create a polynomial coefficients vector filling it
# with NA values
pol_coefficients <- rep(NA, (pol_degrees[1] + 1) * (pol_degrees[2] + 1))

# Now let's fill the coefficients vector with correct values
pol_coefficients[pol_powers_sum %% 2 == 0] <- 2
```

```
pol_coefficients[pol_powers_sum %% 2 != 0] <- 5

# Finally, let's check that the correspondence is correct
printPolynomial(pol_degrees, pol_coefficients)

## Let's represent a polynomial 0.3+0.5x2-x2^2+2x1+1.5x1x2+x1x2^2

pol_degrees <- c(1, 2)
pol_coefficients <- c(0.3, 0.5, -1, 2, 1.5, 1)

printPolynomial(pol_degrees, pol_coefficients)
```

`predict.hpaBinary` *Predict method for hpaBinary*

Description

Predict method for hpaBinary

Usage

```
## S3 method for class 'hpaBinary'
predict(object, ..., newdata = NULL, is_prob = TRUE)
```

Arguments

`object`	Object of class "hpaBinary"
`...`	further arguments (currently ignored)
`newdata`	An optional data frame (for hpaBinary and hpaSelection) or numeric matrix (for hpaML) in which to look for variables with which to predict. If omitted, the original data frame (matrix) is used.
`is_prob`	logical; if TRUE (default) then function returns predicted probabilities. Otherwise latent variable (single index) estimates will be returned.

Value

This function returns predicted probabilities based on `hpaBinary` estimation results.

predict.hpaML *Predict method for hpaML*

Description

Predict method for hpaML

Usage

```
## S3 method for class 'hpaML'
predict(object, ..., newdata = matrix(c(0)))
```

Arguments

object	Object of class "hpaML"
...	further arguments (currently ignored)
newdata	An optional data frame (for hpaBinary and hpaSelection) or numeric matrix (for hpaML) in which to look for variables with which to predict. If omitted, the original data frame (matrix) is used.

Value

This function returns predictions based on hpaML estimation results.

predict.hpaSelection *Predict outcome and selection equation values from hpaSelection model*

Description

This function predicts outcome and selection equation values from hpaSelection model.

Usage

```
## S3 method for class 'hpaSelection'
predict(
  object,
  ...,
  newdata = NULL,
  method = "HPA",
  is_cond = TRUE,
  type = "outcome"
)
```

Arguments

object	Object of class "hpaSelection"
...	further arguments (currently ignored)
newdata	An optional data frame (for hpaBinary and hpaSelection) or numeric matrix (for hpaML) in which to look for variables with which to predict. If omitted, the original data frame (matrix) is used.
method	string value indicating prediction method based on hermite polynomial approximation "HPA" or Newey method "Newey".
is_cond	logical; if TRUE (default) then conditional predictions will be estimated. Otherwise unconditional predictions will be returned.
type	character; if "outcome" (default) then predictions for outcome equation will be estimated according to method. If "selection" then selection equation predictions (probabilities) will be returned.

Details

Note that Newey method can't predict conditional outcomes for zero selection equation value. Conditional probabilities for selection equation could be estimated only when dependent variable from outcome equation is observable.

Value

This function returns the list which structure depends on method, is_probit and is_outcome values.

---

predict_hpaBinary *Predict method for hpaBinary*

Description

Predict method for hpaBinary

Usage

```
predict_hpaBinary(object, newdata = NULL, is_prob = TRUE)
```

Arguments

object	Object of class "hpaBinary"
newdata	An optional data frame (for hpaBinary and hpaSelection) or numeric matrix (for hpaML) in which to look for variables with which to predict. If omitted, the original data frame (matrix) is used.
is_prob	logical; if TRUE (default) then function returns predicted probabilities. Otherwise latent variable (single index) estimates will be returned.

Value

This function returns predicted probabilities based on hpaBinary estimation results.

`predict_hpaML` *Predict method for hpaML*

Description

Predict method for hpaML

Usage

```
predict_hpaML(object, newdata = matrix(1, 1))
```

Arguments

`object`	Object of class "hpaML"
`newdata`	An optional data frame (for hpaBinary and hpaSelection) or numeric matrix (for hpaML) in which to look for variables with which to predict. If omitted, the original data frame (matrix) is used.

Value

This function returns predictions based on hpaML estimation results.

`predict_hpaSelection` *Predict outcome and selection equation values from hpaSelection model*

Description

This function predicts outcome and selection equation values from hpaSelection model.

Usage

```
predict_hpaSelection(
  object,
  newdata = NULL,
  method = "HPA",
  is_cond = TRUE,
  is_outcome = TRUE
)
```

Arguments

object	Object of class "hpaSelection"
newdata	An optional data frame (for hpaBinary and hpaSelection) or numeric matrix (for hpaML) in which to look for variables with which to predict. If omitted, the original data frame (matrix) is used.
method	string value indicating prediction method based on hermite polynomial approximation "HPA" or Newey method "Newey".
is_cond	logical; if TRUE (default) then conditional predictions will be estimated. Otherwise unconditional predictions will be returned.
is_outcome	logical; if TRUE (default) then predictions for selection equation will be estimated using "HPA" method. Otherwise selection equation predictions (probabilities) will be returned.

Details

Note that Newey method can't predict conditional outcomes for zero selection equation value. Conditional probabilities for selection equation could be estimated only when dependent variable from outcome equation is observable.

Value

This function returns the list which structure depends on method, is_probit and is_outcome values.

---

print.hpaBinary *Print method for "hpaBinary" object*

Description

Print method for "hpaBinary" object

Usage

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

Arguments

x	Object of class "hpaBinary"
...	further arguments (currently ignored)

print.hpaML *Print method for "hpaML" object*

Description

Print method for "hpaML" object

Usage

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

Arguments

x	Object of class "hpaML"
...	further arguments (currently ignored)

print.hpaSelection *Print method for "hpaSelection" object*

Description

Print method for "hpaSelection" object

Usage

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

Arguments

x	Object of class "hpaSelection"
...	further arguments (currently ignored)

print.summary.hpaBinary *Summary for "hpaBinary" object*

Description

Summary for "hpaBinary" object

Usage

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

Arguments

x	Object of class "hpaBinary"
...	further arguments (currently ignored)

print.summary.hpaML *Summary for hpaML output*

Description

Summary for hpaML output

Usage

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

Arguments

x	Object of class "hpaML"
...	further arguments (currently ignored)

`print.summary.hpaSelection` *Summary for "hpaSelection" object*

Description

Summary for "hpaSelection" object

Usage

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

Arguments

`x`	Object of class "hpaSelection"
`...`	further arguments (currently ignored)

`print_summary_hpaBinary` *Summary for hpaBinary output*

Description

Summary for hpaBinary output

Usage

```
print_summary_hpaBinary(x)
```

Arguments

`x`	Object of class "hpaBinary"

print_summary_hpaML *Summary for hpaML output*

Description

Summary for hpaML output

Usage

```
print_summary_hpaML(x)
```

Arguments

x	Object of class "hpaML"

print_summary_hpaSelection *Summary for hpaSelection output*

Description

Summary for hpaSelection output

Usage

```
print_summary_hpaSelection(x)
```

Arguments

x	Object of class "hpaSelection"

summary.hpaBinary *Summarizing hpaBinary Fits*

Description

Summarizing hpaBinary Fits

Usage

```
## S3 method for class 'hpaBinary'
summary(object, ...)
```

Arguments

object	Object of class "hpaBinary"
...	further arguments (currently ignored)

Value

This function returns the same list as hpaBinary function changing its class to "summary.hpaBinary".

---

summary.hpaML *Summarizing hpaML Fits*

Description

Summarizing hpaML Fits

Usage

```
## S3 method for class 'hpaML'
summary(object, ...)
```

Arguments

object	Object of class "hpaML"
...	further arguments (currently ignored)

Value

This function returns the same list as hpaML function changing its class to "summary.hpaML".

---

summary.hpaSelection *Summarizing hpaSelection Fits*

Description

This function summarizes hpaSelection Fits

Usage

```
## S3 method for class 'hpaSelection'
summary(object, ...)
```

Arguments

object	Object of class "hpaSelection"
...	further arguments (currently ignored)

Value

This function returns the same list as hpaSelection function changing its class to "summary.hpaSelection".

---

summary_hpaBinary *Summarizing hpaBinary Fits*

Description

Summarizing hpaBinary Fits

Usage

```
summary_hpaBinary(object)
```

Arguments

object	Object of class "hpaBinary"

Value

This function returns the same list as hpaBinary function changing its class to "summary.hpaBinary".

---

summary_hpaML *Summarizing hpaML Fits*

Description

Summarizing hpaML Fits

Usage

```
summary_hpaML(object)
```

Arguments

object	Object of class "hpaML"

Value

This function returns the same list as hpaML function changing its class to "summary.hpaML".

summary_hpaSelection *Summarizing hpaSelection Fits*

Description

This function summarizes hpaSelection Fits.

Usage

```
summary_hpaSelection(object)
```

Arguments

object	Object of class "hpaSelection".

Value

This function returns the same list as hpaSelection function changing its class to "summary.hpaSelection".

truncatedNormalMoment *Calculate k-th order moment of truncated normal distribution*

Description

This function recursively calculates k-th order moment of truncated normal distribution.

Usage

```
truncatedNormalMoment(
  k = 1L,
  x_lower = numeric(0),
  x_upper = numeric(0),
  mean = 0,
  sd = 1,
  pdf_lower = numeric(0),
  cdf_lower = numeric(0),
  pdf_upper = numeric(0),
  cdf_upper = numeric(0),
  cdf_difference = numeric(0),
  return_all_moments = FALSE,
  is_validation = TRUE,
  is_parallel = FALSE,
  diff_type = "NO"
)
```

Arguments

k	non-negative integer moment order.
x_lower	numeric vector of lower truncation points.
x_upper	numeric vector of upper truncation points.
mean	numeric expected value.
sd	positive numeric standard deviation.
pdf_lower	non-negative numeric matrix of precalculated normal density functions with mean mean and the standard deviation sd at points given by x_lower.
cdf_lower	non-negative numeric matrix of precalculated values of normal cumulative distribution functions with mean mean and standard deviation sd at points given by x_lower.
pdf_upper	non-negative numeric matrix of precalculated normal density functions with mean mean and the standard deviation sd at points given by x_upper.
cdf_upper	non-negative numeric matrix of precalculated values of normal cumulative distribution functions with mean mean and standard deviation sd at points given by x_upper.
cdf_difference	non-negative numeric matrix of precalculated cdf_upper-cdf_lower values.
return_all_moments	logical; if TRUE, function returns the matrix of moments of normally distributed random variable with mean = mean and standard deviation = sd under lower and upper truncation points x_lower and x_upper correspondingly. Note that element in i-th row and j-th column of this matrix corresponds to the i-th observation (j-1)-th order moment.
is_validation	logical value indicating whether function input arguments should be validated. Set it to FALSE for a slight performance boost (default value is TRUE).
is_parallel	if TRUE then multiple cores will be used for some calculations. It usually provides speed advantage for large enough samples (more than 1000 observations).
diff_type	string value indicating the type of the argument the moment should be differentiated with respect to. Default value is "NO" so the moments themselves will be returned. Alternative values are "mean" and "sd". Also "x_lower" and "x_upper" values are available for truncatedNormalMoment.

Details

This function estimates k-th order moment of normal distribution which mean equals to mean and standard deviation equals to sd truncated at points given by x_lower and x_upper. Note that the function is vectorized so you can provide x_lower and x_upper as vectors of equal size. If vectors values for x_lower and x_upper are not provided then their default values will be set to -(.Machine$double.xmin * 0.99) and (.Machine$double.xmax * 0.99) correspondingly.

Note that the value of parameter k automatically converts to an integer. So passing a non-integer k value will not cause any errors but the calculations will be performed for the rounded k value only.

If there are precalculated density functions or cumulative distribution functions at standardized truncation points (subtract mean and then divide by sd) then it is possible to provide them through pdf_lower, pdf_upper, cdf_lower and cdf_upper arguments in order to decrease the number of calculations.

Value

This function returns vector of k-th order moments for normally distributed random variable with mean = mean and standard deviation = sd under x_lower and x_upper truncation points x_lower and x_upper correspondingly. If return_all_moments is TRUE then see this argument description above for output details.

Examples

```
## Calculate 5-th order moment of three truncated normal random
## variables (x1, x2, x3) which mean is 5 and standard deviation is 3.
## These random variables truncation points are given
## as follows:-1<x1<1, 0<x2<2, 1<x3<3.
k <- 3
x_lower <- c(-1, 0, 1, -Inf, -Inf)
x_upper <- c(1, 2 , 3, 2, Inf)
mean <- 3
sd <- 5

# get the moments
truncatedNormalMoment(k, x_lower, x_upper, mean, sd)

# get matrix of (0-5)-th moments (columns) for each variable (rows)
truncatedNormalMoment(k, x_lower, x_upper,
                      mean, sd,
                      return_all_moments = TRUE)

# get the moments derivatives respect to mean
truncatedNormalMoment(k, x_lower, x_upper,
                      mean, sd,
                      diff_type = "mean")

# get the moments derivatives respect to standard deviation
truncatedNormalMoment(k, x_lower, x_upper,
                      mean, sd,
                      diff_type = "sd")
```

---

vcov.hpaBinary *Extract covariance matrix from a hpaBinary object*

Description

Extract covariance matrix from a hpaBinary object

Usage

```
## S3 method for class 'hpaBinary'
vcov(object, ...)
```

Arguments

object	Object of class "hpaBinary"
...	further arguments (currently ignored)

---

vcov.hpaML *Extract covariance matrix from a hpaML object*

Description

Extract covariance matrix from a hpaML object

Usage

```
## S3 method for class 'hpaML'
vcov(object, ...)
```

Arguments

object	Object of class "hpaML"
...	further arguments (currently ignored)

---

vcov.hpaSelection *Extract covariance matrix from a hpaSelection object*

Description

Extract covariance matrix from a hpaSelection object

Usage

```
## S3 method for class 'hpaSelection'
vcov(object, ...)
```

Arguments

object	Object of class "hpaSelection"
...	further arguments (currently ignored)

Index