Package 'genscore'

December 16, 2023

Type Package

Title Generalized Score Matching Estimators

Version 1.0.2.2

Author Shiqing Yu, Lina Lin, Wally Gilks

Maintainer Shiqing Yu <syu.phd@gmail.com>

Description Implementation of the Generalized Score Matching estimator in Yu et al. (2019) <https://jmlr.org/papers/v20/18-278.html> for non-negative graphical models (truncated Gaussian, exponential square-root, gamma, a-b models) and univariate truncated Gaussian distributions. Also includes the original estimator for untruncated Gaussian graphical models from Lin et al. (2016) <doi:10.1214/16-EJS1126>, with the addition of a diagonal multiplier.

License GPL-3

Encoding UTF-8

Suggests Matrix, igraph, zoo, knitr, rmarkdown, cubature

Imports Rdpack, mvtnorm, tmvtnorm, stringr

URL https://github.com/sqyu/genscore

BugReports https://github.com/sqyu/genscore/issues

RdMacros Rdpack

RoxygenNote 7.1.1

VignetteBuilder knitr

NeedsCompilation yes

Repository CRAN

Date/Publication 2023-12-16 10:40:02 UTC

R topics documented:

AUC *Calculates the AUC of an ROC curve.*

Description

Calculates the area under an ROC curve (AUC).

Usage

```
AUC(tpfp)
```

Arguments

`tpfp`	A matrix with two columns, the true positive and the false positive rates.

Value

A number between 0 and 1, the area under the curve (AUC).

Examples

```
n <- 40
p <- 50
mu <- rep(0, p)
tol <- 1e-8
K <- cov_cons(mode="sub", p=p, seed=1, spars=0.2, eig=0.1, subgraphs=10)
true_edges <- which(abs(K) > tol & diag(p) == 0)
dm <- 1 + (1-1/(1+4*exp(1)*max(6*log(p)/n, sqrt(6*log(p)/n))))
set.seed(1)
domain <- make_domain("R+", p=p)
x <- tmvtnorm::rtmvnorm(n, mean = mu, sigma = solve(K),
       lower = rep(0, p), upper = rep(Inf, p), algorithm = "gibbs",
       burn.in.samples = 100, thinning = 10)
est <- estimate(x, setting="gaussian", elts=NULL, domain=domain, centered=TRUE,
         symmetric="symmetric", lambda_length=100, mode="min_pow",
         param1=1, param2=3, diagonal_multiplier=dm)
# Apply tp_fp to each estimated edges set for each lambda
TP_FP <- t(sapply(est$edgess, function(edges){tp_fp(edges, true_edges, p)}))
old.par <- par(mfrow=c(1,1), mar=c(5,5,5,5))
auc <- AUC(TP_FP)
plot(c(), c(),  ylim=c(0,1), xlim=c(0,1), cex.lab=1,
  main=paste("ROC curve, AUC",round(auc,4)), xlab="False Positives",
  ylab="True Positives")
points(TP_FP[,2], TP_FP[,1], type="l")
points(c(0,1), c(0,1), type = "l", lty = 2)
par(old.par)
```

`avgrocs` *Takes the vertical average of ROC curves.*

Description

Takes the vertical average of ROC curves using algorithm 3 from Fawcett (2006). The resulting ROC curve preserves the average AUC.

Usage

```
avgrocs(rocs, num_true_edges, p)
```

Arguments

`rocs`	A list of ROC curves, each of which is a matrix with two columns corresponding to the true positive and false positive rates, respectively.
`num_true_edges`	A positive integer, the number of true edges
`p`	A positive integer, the dimension

Value

The averaged ROC curve, a matrix with 2 columns and `(p^2-p-num_true_edges+1)` rows.

References

Fawcett T (2006). "An introduction to ROC analysis." *Pattern Recognition Letters*, **27**(8), 861–874.

Examples

```
n <- 40
p <- 50
mu <- rep(0, p)
tol <- 1e-8
domain <- make_domain("R+", p=p)
K <- cov_cons(mode="sub", p=p, seed=1, spars=0.2, eig=0.1, subgraphs=10)
true_edges <- which(abs(K) > tol & diag(p) == 0)
dm <- 1 + (1-1/(1+4*exp(1)*max(6*log(p)/n, sqrt(6*log(p)/n))))
ROCs <- list()
old.par <- par(mfrow=c(2,2), mar=c(5,5,5,5))
for (i in 1:3){
  set.seed(i)
  x <- tmvtnorm::rtmvnorm(n, mean = mu, sigma = solve(K),
         lower = rep(0, p), upper = rep(Inf, p), algorithm = "gibbs",
         burn.in.samples = 100, thinning = 10)
  est <- estimate(x, setting="gaussian", elts=NULL, domain=domain, centered=TRUE,
           symmetric="symmetric", lambda_length=100, mode="min_pow",
           param1=1, param2=3, diag=dm)
  # Apply tp_fp to each estimated edges set for each lambda
  TP_FP <- t(sapply(est$edgess, function(edges){tp_fp(edges, true_edges, p)}))
  ROCs[[i]] <- TP_FP
  plot(c(), c(),  ylim=c(0,1), xlim=c(0,1), cex.lab=1,
    main=paste("ROC, trial ",i,", AUC ",round(AUC(TP_FP),4),sep=""),
    xlab="False Positives", ylab="True Positives")
  points(TP_FP[,2], TP_FP[,1], type="l")
  points(c(0,1), c(0,1), type = "l", lty = 2)
}
average_ROC <- avgrocs(ROCs, length(true_edges), p)
plot(c(), c(),  ylim=c(0,1), xlim=c(0,1), cex.lab=1,
  main=paste("Average ROC, AUC",round(AUC(average_ROC),4)),
  xlab="False Positives", ylab="True Positives")
points(average_ROC[,2], average_ROC[,1], type="l")
points(c(0,1), c(0,1), type = "l", lty = 2)
par(old.par)
```

---

beautify_rule *Replaces consecutive "&"s and "|"s in a string to a single & and |.*

Description

Replaces consecutive "&"s and "|"s in a string to a single "&" and "|".

Usage

```
beautify_rule(rule)
```

Arguments

rule	A string containing positive integers, parentheses, and "&" and "|" only.

Details

Applied to domain$rule if domain$type == "polynomial".

Value

A string with extra "&"s and "|"s removed.

Examples

```
beautify_rule("(1 & 2 && 3 &&& 4) | 5 || 6 ||| 7")
```

---

binarySearch_bin *Finds the index of the bin a number belongs to using binary search.*

Description

Finds the index of the bin a number belongs to using binary search.

Usage

```
binarySearch_bin(arr, l, r, x)
```

Arguments

arr	A vector of size at least 2.
l	An integer between 1 and length(arr). Must be smaller than 1.
r	An integer between 1 and length(arr). Must be larger than l.
x	A number. Must be within the range of [arr[l], arr[r]].

Details

Finds the smallest index i such that arr[i] <= x <= arr[i+1].

Value

The index i such that arr[i] <= x <= arr[i+1].

Examples

```
binarySearch_bin(1:10, 1, 10, seq(1, 10, by=0.5))
binarySearch_bin(1:10, 5, 8, seq(5, 8, by=0.5))
```

`calc_crit` *Calculates penalized or unpenalized loss in K and eta given arbitrary data*

Description

Calculates penalized or unpenalized loss in K and eta given arbitrary data

Usage

```
calc_crit(elts, res, penalty)
```

Arguments

`elts`	An element list returned from `get_elts()`. Need not be the same as the elements used to estimate `res`, but they must be both centered or both non-centered, and their dimension `p` must match. `elts` cannot be profiled as this is supposed to be elements for a new data unseen by `res`, in which case the loss must be explicitly written in `K` and `eta` with `Gamma` and `g` from a new dataset `x`.
`res`	A result list returned from `get_results()`. Must be centered if `elts` is centered, and must be non-centered otherwise. Can be profiled. `res$p` must be equal to `elts$p`.
`penalty`	A boolean, indicates whether the loss should be penalized (using `elts$diagonals_with_multiplier`, `res$lambda1` and `res$lambda2`).

Details

This function calculates the loss in some estimated `K` and `eta` given an `elts` generated using `get_elts()` with a new dataset `x`. This is helpful for cross-validation.

Value

A number, the loss.

Examples

```
# In the following examples, all printed numbers should be close to 0.
# In practice, \code{res} need not be estimates fit to \code{elts},
# but in the examples we use \code{res <- get_results(elts)} just to
# demonstrate that the loss this function returns matches that returned
# by the C code during estimation using \code{get_results}.

n <- 6
p <- 3
eta <- rep(0, p)
K <- diag(p)
dm <- 1 + (1-1/(1+4*exp(1)*max(6*log(p)/n, sqrt(6*log(p)/n))))
```

```
domains <- list(make_domain("R", p=p),
                make_domain("R+", p=p),
                make_domain("uniform", p=p, lefts=c(0,2), rights=c(1,3)),
                make_domain("polynomial", p=p,
              ineqs=list(list("expression"="sum(x^2)<=1", nonnegative=FALSE, abs=FALSE))))

domains <- c(domains,
             list(make_domain("polynomial", p=p,
              ineqs=list(list("expression"="sum(x^2)<=1", nonnegative=TRUE, abs=FALSE))),
                  make_domain("polynomial", p=p,
                    ineqs=list(list("expression"=paste(paste(sapply(1:p,
                      function(j){paste(j, "x", j, sep="")}), collapse="+"), "<1"),
                      abs=FALSE, nonnegative=TRUE))),
                  make_domain("simplex", p=p)))

for (domain in domains) {
  if (domain$type == "R" ||
       (domain$type == "uniform" && any(domain$lefts < 0)) ||
       (domain$type == "polynomial" && !domain$ineqs[[1]]$nonnegative))
    settings <- c("gaussian")
  else if (domain$type == "simplex")
    settings <- c("log_log", "log_log_sum0")
  else
    settings <- c("gaussian", "exp", "gamma", "log_log", "ab_3/4_2/3")

  if (domain$type == "simplex")
    symms <- c("symmetric")
  else
    symms <- c("symmetric", "and", "or")

  for (setting in settings) {
    x <- gen(n, setting=setting, abs=FALSE, eta=eta, K=K, domain=domain,
         finite_infinity=100, xinit=NULL, burn_in=1000, thinning=100, verbose=FALSE)
    h_hp <- get_h_hp("min_pow", 1, 3)

    for (symm in symms) {

       # Centered, penalized loss
       elts <- get_elts(h_hp, x, setting, domain, centered=TRUE, scale="", diag=dm)
       res <- get_results(elts, symm, 0.1)
       print(calc_crit(elts, res, penalty=TRUE) - res$crit) # Close to 0

       # Non-centered, unpenalized loss
       elts_nopen <- get_elts(h_hp, x, setting, domain, centered=TRUE, scale="", diag=1)
       res_nopen <- get_results(elts_nopen, symm, 0)
      print(calc_crit(elts_nopen, res_nopen, penalty=FALSE) - res_nopen$crit) # Close to 0

       # Non-centered, non-profiled, penalized loss
       elts_nc_np <- get_elts(h_hp, x, setting, domain, centered=FALSE,
         profiled_if_noncenter=FALSE, scale="", diag=dm)
       res_nc_np <- get_results(elts_nc_np, symm, lambda1=0.1, lambda2=0.05)
      print(calc_crit(elts_nc_np, res_nc_np, penalty=TRUE) - res_nc_np$crit) # Close to 0
```

```
      # Non-centered, non-profiled, unpenalized loss
      elts_nc_np_nopen <- get_elts(h_hp, x, setting, domain, centered=FALSE,
        profiled_if_noncenter=FALSE, scale="", diag=1)
      res_nc_np_nopen <- get_results(elts_nc_np_nopen, symm, lambda1=0, lambda2=0)
      print(calc_crit(elts_nc_np_nopen, res_nc_np_nopen, penalty=FALSE) -
        res_nc_np_nopen$crit) # Close to 0

      if (domain$type != "simplex") {
        # Non-centered, profiled, penalized loss
        elts_nc_p <- get_elts(h_hp, x, setting, domain, centered=FALSE,
          profiled_if_noncenter=TRUE, scale="", diag=dm)
        res_nc_p <- get_results(elts_nc_p, symm, lambda1=0.1)
        if (elts_nc_np$setting != setting || elts_nc_np$domain_type != "R")
       res_nc_p$crit <- res_nc_p$crit - sum(elts_nc_np$g_eta ^ 2 / elts_nc_np$Gamma_eta) / 2
       print(calc_crit(elts_nc_np, res_nc_p, penalty=TRUE) - res_nc_p$crit)  # Close to 0
        # Note that the elts argument cannot be profiled, so
        # calc_crit(elts_nc_p, res_nc_p, penalty=TRUE) is not allowed

        # Non-centered, profiled, unpenalized loss
        elts_nc_p_nopen <- get_elts(h_hp, x, setting, domain, centered=FALSE,
          profiled_if_noncenter=TRUE, scale="", diag=1)
        res_nc_p_nopen <- get_results(elts_nc_p_nopen, symm, lambda1=0)
        if (elts_nc_np_nopen$setting != setting || elts_nc_np_nopen$domain_type != "R")
          res_nc_p_nopen$crit <- (res_nc_p_nopen$crit -
              sum(elts_nc_np_nopen$g_eta ^ 2 / elts_nc_np_nopen$Gamma_eta) / 2)
        print(calc_crit(elts_nc_np_nopen, res_nc_p_nopen, penalty=TRUE) -
          res_nc_p_nopen$crit) # Close to 0
         # Again, calc_crit(elts_nc_p_nopen, res_nc_p, penalty=TRUE) is not allowed
      } # if domain$type != "simplex"

    } # for symm in symms
  } # for setting in settings
} # for domain in domains
```

---

check_endpoints *Checks if two equally sized numeric vectors satisfy the requirements for being left and right endpoints of a domain defined as a union of intervals.*

Description

Checks if two equally sized numeric vectors satisfy the requirements for being left and right endpoints of a domain defined as a union of intervals.

Usage

```
check_endpoints(lefts, rights)
```

Arguments

lefts	A non-empty vector of numbers (may contain -Inf), the left endpoints of a domain defined as a union of intervals.
rights	A non-empty vector of numbers (may contain Inf), the right endpoints of a domain defined as a union of intervals. Must have the same size as lefts.

Details

Both lefts and rights must be non-empty and should have the same length. Suppose lefts and rights both have length l, [lefts[1], rights[1]], ..., [lefts[l], rights[l]] must be an increasing and non-overlapping set of valid intervals, meaning lefts[i] <= rights[i] <= lefts[j] for any i < j (singletons and overlapping at the boundary points are allowed). Inf is not allowed in lefts and -Inf is not allowed in rights.

Value

NULL. Program stops if lefts and rights do not define valid left and right endpoints.

Examples

```
## [-4,-3], [-2,-1], [0,1], [2,3], [4,5]
check_endpoints(lefts=c(-4,-2,0,2,4), rights=c(-3,-1,1,3,5))
## Not run:
check_endpoints(lefts=c(), rights=c()) # Cannot be empty
check_endpoints(lefts=c(-4,-2,0,2,4), rights=c(-3,-1,1,3)) # Unequal length
check_endpoints(lefts=c(Inf), rights=c(Inf)) # No Inf in lefts, otherwise invalid interval
check_endpoints(lefts=c(-Inf), rights=c(-Inf)) # No -Inf in rights, otherwise invalid interval
check_endpoints(lefts=c(0, 1), rights=c(2, 3)) # [0,2] and [1,3] overlap, not allowed
check_endpoints(lefts=c(2, 0), rights=c(3, 1)) # [2,3], [0,1] not increasing, not allowed

## End(Not run)
## Singletons and overlapping at the boundary points allowed
check_endpoints(lefts=c(0, 1, 2), rights=c(0, 2, 3))
```

---

compare_two_results *Compares two lists returned from estimate().*

Description

Compares two lists returned from estimate().

Usage

```
compare_two_results(res, res2)
```

Arguments

res	A res list returned from estimate().
res2	A res list returned from estimate().

Value

A list of numbers all of which should be close to 0 if `res` and `res2` are expected to be the same.

`compare_two_sub_results` *Compares two lists returned from get_results().*

Description

Compares two lists returned from `get_results()`.

Usage

```
compare_two_sub_results(res, res2)
```

Arguments

`res`	A res list returned from `get_results()`.
`res2`	A res list returned from `get_results()`.

Value

A list of numbers all of which should be close to 0 if `res` and `res2` are expected to be the same.

`cov_cons` *Random generator of inverse covariance matrices.*

Description

Random generator of inverse covariance matrices.

Usage

```
cov_cons(mode, p, seed = NULL, spars = 1, eig = 0.1, subgraphs = 1)
```

Arguments

`mode`	A string, see details.
`p`	A positive integer >= 2, the dimension.
`seed`	A number, the seed for the generator. Ignored if `NULL` or `mode == "band"` or `mode == "chain"`.
`spars`	A number, see details. Ignored if `mode == "chain"`. Default to 1.
`eig`	A positive number, the minimum eigenvalue of the returned matrix. Default to 0.1.
`subgraphs`	A positive integer, the number of subgraphs for the `"sub"` mode. Note that `p` must be divisible by `subgraphs`.

Details

The function generates an inverse covariance matrix according to the mode argument as follows. The diagonal entries of the matrix are set to the same value such that the minimum eigenvalue of the returned matrix is equal to eig.

Takes the Q matrix from the QR decomposition of a p by p random matrix with independent $Normal(0, 1)$ entries, and calculates $Q'diag(ev)Q$. Randomly zeros out its upper triangular entries using independent uniform Bernoulli(spars) variables, and then symmetrizes the matrix using the upper triangular part.

"randomsub" Constructs a block diagonal matrix with subgraphs disconnected subgraphs with equal number of nodes. In each subgraph, takes each entry independently from $Uniform(0.5, 1)$, and randomly zeros out its upper triangular entries using independent uniform Bernoulli(spars) variables, and finally symmetrizes the matrix using the upper triangular part. The construction from Section 4.2 of Lin et al. (2016).

"er" Constructs an Erd\H{o}s-R\'enyi game with probability spars, and sets the edges to independent $Uniform(0.5, 1)$ variables, and finally symmetrizes the matrix using the lower triangular entries.

"band" Constructs a banded matrix so that the (i,j)-th matrix is nonzero if and only if $|i-j| <= spars$, and is equal to $1 - |i - j|/(spars + 1)$ if $i! = j$.

"chain" A chain graph, where the (i,j)-th matrix is nonzero if and only if $|i - j| <= 1$, and is equal to 0.5 if $|i - j| == 1$. A special case of the "band" construction with spars equal to 1.

Value

A p by p inverse covariance matrix. See details.

References

Lin L, Drton M, Shojaie A (2016). "Estimation of high-dimensional graphical models using regularized score matching." *Electron. J. Stat.*, **10**(1), 806–854.

Examples

```
p <- 100
K1 <- cov_cons("random", p, seed = 1, spars = 0.05, eig = 0.1)
K2 <- cov_cons("sub", p, seed = 2, spars = 0.5, eig = 0.1, subgraphs=10)
K3 <- cov_cons("er", p, seed = 3, spars = 0.05, eig = 0.1)
K4 <- cov_cons("band", p, spars = 2, eig = 0.1)
K5 <- cov_cons("chain", p, eig = 0.1)
```

---

crbound_mu *The Cram\'er-Rao lower bound (times n) for estimating the mean parameter from a univariate truncated normal sample with known variance parameter.*

Description

The Cram\'er-Rao lower bound (times n) on the variance for estimating the mean parameter mu from a univariate truncated normal sample, assuming the true variance parameter sigmasq is known.

Usage

```
crbound_mu(mu, sigmasq)
```

Arguments

mu	The mean parameter.
sigmasq	The variance parameter.

Details

The Cram\'er-Rao lower bound in this case is defined as $\sigma^4/var(X-\mu)$.

Value

A number, the Cram\'er-Rao lower bound.

`crbound_sigma` *The Cram\'er-Rao lower bound (times* n*) for estimating the variance parameter from a univariate truncated normal sample with known mean parameter.*

Description

The Cram\'er-Rao lower bound (times n) on the variance for estimating the variance parameter sigmasq from a univariate truncated normal sample, assuming the true mean parameter mu is known.

Usage

```
crbound_sigma(mu, sigmasq)
```

Arguments

mu	The mean parameter.
sigmasq	The variance parameter.

Details

The Cram\'er-Rao lower bound in this case is defined as $4\sigma^8/var((X-\mu)^2)$.

Value

A number, the Cram\'er-Rao lower bound .

diff_lists *Computes the sum of absolute differences between two lists.*

Description

Computes the sum of absolute differences between two lists using diff_vecs().

Usage

```
diff_lists(l1, l2, name = NULL)
```

Arguments

l1	A list.
l2	A list.
name	A string, default to NULL. If not NULL, computes the differences in the l1[[name]] and l2[[name]].

Value

Returns the sum of absolute differences between l1 and l2 if name is NULL, or that between l1[[name]] and l2[[name]] otherwise. If name is not NULL and if name is in exactly one of l1 and l2, returns Inf; if name is in neither, returns NA. Exception: Returns a positive integer if the two elements compared hold NA, NULL or Inf values in different places.

diff_vecs *Computes the sum of absolute differences in the finite non-NA/NULL elements between two vectors.*

Description

Computes the sum of absolute differences in the finite non-NA/NULL elements between two vectors.

Usage

```
diff_vecs(l1, l2, relative = FALSE)
```

Arguments

l1	A vector.
l2	A vector.
relative	A boolean, default to FALSE. If TRUE, returns the relative difference (sum of absolute differences divided by the elementwise minimum between l1 and l2).

Value

The sum of (relative) absolute differences in `l1` and `l2`, or a positive integer if two vectors differ in length or hold `NA`, `NULL` or `Inf` values in different places.

`domain_for_C` *Returns a list to be passed to C that represents the domain.*

Description

Returns a list to be passed to C that represents the domain.

Usage

```
domain_for_C(domain)
```

Arguments

`domain`	A list returned from `make_domain()` that represents the domain.

Details

Construct a list to be read by C code that represents the domain.

Value

A list of the following elements.

`num_char_params`	An integer, length of `char_params`.
`char_params`	A vector of string (`char *` or `char **`) parameters.
`num_int_params`	An integer, length of `int_params`.
`int_params`	A vector of integer (`int`) parameters.
`num_double_params`	An integer, length of `double_params`.
`double_params`	A vector of double (`double`) parameters.

Examples

```
p <- 30
# The 30-dimensional real space R^30
domain <- make_domain("R", p=p)
domain_for_C(domain)

# The non-negative orthant of the 30-dimensional real space, R+^30
domain <- make_domain("R+", p=p)
domain_for_C(domain)
```

```
# x such that sum(x^2) > 10 && sum(x^(1/3)) > 10 with x allowed to be negative
domain <- make_domain("polynomial", p=p, rule="1 && 2",
       ineqs=list(list("expression"="sum(x^2)>10", abs=FALSE, nonnegative=FALSE),
                  list("expression"="sum(x^(1/3))>10", abs=FALSE, nonnegative=FALSE)))
domain_for_C(domain)

# ([0, 1] v [2,3]) ^ p
domain <- make_domain("uniform", p=p, lefts=c(0,2), rights=c(1,3))
domain_for_C(domain)

# x such that {x1 > 1 && log(1.3) < x2 < 1 && x3 > log(1.3) && ... && xp > log(1.3)}
domain <- make_domain("polynomial", p=p, rule="1 && 2 && 3",
       ineqs=list(list("expression"="x1>1", abs=FALSE, nonnegative=TRUE),
                  list("expression"="x2<1", abs=FALSE, nonnegative=TRUE),
                  list("expression"="exp(x)>1.3", abs=FALSE, nonnegative=FALSE)))
domain_for_C(domain)

# x in R_+^p such that {sum(log(x))<2 || (x1^(2/3)-1.3x2^(-3)<1 && exp(x1)+2.3*x2>2)}
domain <- make_domain("polynomial", p=p, rule="1 || (2 && 3)",
       ineqs=list(list("expression"="sum(log(x))<2", abs=FALSE, nonnegative=TRUE),
                 list("expression"="x1^(2/3)-1.3x2^(-3)<1", abs=FALSE, nonnegative=TRUE),
                   list("expression"="exp(x1)+2.3*x2^2>2", abs=FALSE, nonnegative=TRUE)))
domain_for_C(domain)

# x in R_+^p such that {x in R_+^p: sum_j j * xj <= 1}
domain <- make_domain("polynomial", p=p,
       ineqs=list(list("expression"=paste(paste(sapply(1:p,
                            function(j){paste(j, "x", j, sep="")}), collapse="+"), "<1"),
                    abs=FALSE, nonnegative=TRUE)))
domain_for_C(domain)

# The (p-1)-simplex
domain <- make_domain("simplex", p=p)
domain_for_C(domain)

# The l-1 ball {sum(|x|) < 1}
domain <- make_domain("polynomial", p=p,
       ineqs=list(list("expression"="sum(x)<1", abs=TRUE, nonnegative=FALSE)))
domain_for_C(domain)
```

---

eBIC *eBIC score with or without refitting.*

Description

Calculates the eBIC score both with and without refitting an unpenalized model restricted to the estimated support.

Usage

```
eBIC(res, elts, BIC_refit = TRUE, gammas = c(0, 0.5, 1))
```

Arguments

res	A list of results returned by get_results().
elts	A list, elements necessary for calculations returned by get_elts().
BIC_refit	A boolean, whether to get the BIC scores by refitting an unpenalized model restricted to the estimated edges, with lambda1=0, lambda2=0 and diagonal_multiplier=1. Default to TRUE.
gammas	Optional. A number of a vector of numbers. The γ parameter in eBIC. Default to c(0,0.5,1).

Value

A vector of length 2*length(gammas). The first length(gammas) numbers are the eBIC scores without refitting for each gamma value, and the rest are those with refitting if BIC_refit == TRUE, or Inf if BIC_refit == FALSE.

Examples

```
# Examples are shown for Gaussian truncated to R+^p only. For other distributions
#   on other types of domains, please refer to \code{gen()} or \code{get_elts()},
#   as the way to call this function (\code{eBIC()}) is exactly the same in those cases.
n <- 50
p <- 30
domain <- make_domain("R+", p=p)
h_hp <- get_h_hp("min_pow", 1, 3)
mu <- rep(0, p)
K <- diag(p)
dm <- 1 + (1-1/(1+4*exp(1)*max(6*log(p)/n, sqrt(6*log(p)/n))))
x <- tmvtnorm::rtmvnorm(n, mean = mu, sigma = solve(K),
       lower = rep(0, p), upper = rep(Inf, p), algorithm = "gibbs",
       burn.in.samples = 100, thinning = 10)

elts_gauss_np <- get_elts(h_hp, x, setting="gaussian", domain=domain,
                 centered=FALSE, profiled=FALSE, diag=dm)
res_nc_np <- get_results(elts_gauss_np, symmetric="symmetric",
                lambda1=0.35, lambda2=2, previous_res=NULL,
                is_refit=FALSE)
eBIC(res_nc_np, elts_gauss_np, BIC_refit=TRUE, gammas=c(0,0.5,1))
```

---

estimate *The main function for the generalized score-matching estimator for graphical models.*

Description

The main function for the generalized score-matching estimator for graphical models.

Usage

```
estimate(
  x,
  setting,
  domain,
  elts = NULL,
  centered = TRUE,
  symmetric = "symmetric",
  scale = "",
  lambda1s = NULL,
  lambda_length = NULL,
  lambda_ratio = Inf,
  mode = NULL,
  param1 = NULL,
  param2 = NULL,
  h_hp = NULL,
  unif_dist = NULL,
  verbose = TRUE,
  verbosetext = "",
  tol = 1e-06,
  maxit = 1000,
  BIC_refit = TRUE,
  warmstart = TRUE,
  diagonal_multiplier = NULL,
  eBIC_gammas = c(0, 0.5, 1),
  cv_fold = NULL,
  cv_fold_seed = NULL,
  return_raw = FALSE,
  return_elts = FALSE
)
```

Arguments

x	An n by p matrix, the data matrix, where n is the sample size and p the dimension.
setting	A string that indicates the distribution type, must be one of "exp", "gamma", "gaussian", "log_log", "log_log_sum0", or of the form "ab_NUM1_NUM2", where NUM1 is the a value and NUM2 is the b value, and NUM1 and NUM2 must be integers or two integers separated by "/", e.g. "ab_2_2", "ab_2_5/4" or "ab_2/3_1/2".
domain	A list returned from make_domain() that represents the domain.
elts	A list (optional), elements necessary for calculations returned by get_elts().
centered	A boolean, whether in the centered setting (assume $\boldsymbol{\mu} = \boldsymbol{\eta} = 0$) or not. Default to TRUE.
symmetric	A string. If equals "symmetric", estimates the minimizer $\mathbf{K}$ over all symmetric matrices; if "and" or "or", use the "and"/"or" rule to get the support. Default to "symmetric".

`scale` A string indicating the scaling method. If contains `"sd"`, columns are scaled by standard deviation; if contains `"norm"`, columns are scaled by l2 norm; if contains `"center"` and `setting == "gaussian" && domain$type == "R"`, columns are centered to have mean zero. Default to `"norm"`.

`lambda1s` A vector of lambdas, the penalty parameter for K.

`lambda_length` An integer >= 2, the number of lambda1s. Ignored if `lambda1s` is provided, otherwise a grid of lambdas is automatically chosen so that the results range from an empty graph to a complete graph. Default to `10` if neither `lambda1s` nor `lambda_length` is provided.

`lambda_ratio` A positive number, the fixed ratio between $\lambda_{\mathbf{K}}$ and $\lambda_{\boldsymbol{\eta}}$, if $\lambda_{\boldsymbol{\eta}} \neq 0$ (non-profiled) in the non-centered setting.

`mode` A string, the class of the `h` function. Ignored if `elts`, or `h` and `hp` are provided, or if `setting == "gaussian" && domain$type == "R"`.

`param1` A number, the first parameter to the `h` function. Ignored if `elts`, or `h` and `hp` are provided, or if `setting == "gaussian" && domain$type == "R"`.

`param2` A number, the second parameter (may be optional depending on `mode`) to the `h` function. Ignored if `elts`, or `h` and `hp` are provided, or if `setting == "gaussian" && domain$type == "R"`.

`h_hp` A function that returns a list containing `hx=h(x)` (element-wise) and `hpx=hp(x)` (element-wise derivative of h) when applied to a vector or a matrix `x`, both of which has the same shape as `x`.

`unif_dist` Optional, defaults to `NULL`. If not `NULL`, `h_hp` must be `NULL` and `unif_dist(x)` must return a list containing `"g0"` of length `nrow(x)` and `"g0d"` of dimension `dim(x)`, representing the l2 distance and the gradient of the l2 distance to the boundary: the true l2 distance function to the boundary is used for all coordinates in place of h_of_dist; see "Estimating Density Models with Complex Truncation Boundaries" by Liu et al, 2019. That is, $(h_j \circ \phi_j)(x_i)$ in the score-matching loss is replaced by $g_0(x_i)$, the l2 distance of xi to the boundary of the domain.

`verbose` Optional. A boolean, whether to output intermediate results.

`verbosetext` Optional. A string, text to be added to the end of each printout if `verbose == TRUE`.

`tol` Optional. A number, the tolerance parameter. Default to `1e-6`.

`maxit` Optional. A positive integer, the maximum number of iterations for each fit. Default to `1000`.

`BIC_refit` A boolean, whether to get the BIC scores by refitting an unpenalized model restricted to the estimated edges, with `lambda1=lambda2=0` and `diagonal_multiplier=1`. Default to `TRUE`.

`warmstart` Optional. A boolean, whether to use the results from a previous (larger) lambda as a warm start for each new lambda. Default to `TRUE`.

`diagonal_multiplier`
A number >= 1, the diagonal multiplier. Optional and ignored if elts is provided. If `ncol(x) > ncol(n)`, a value strictly larger than 1 is recommended. Default to $1 + \left(1 - \left(1 + 4e \max\left(6 \log p/n, \sqrt{6 \log p/n}\right)\right)^{-1}\right)$.

eBIC_gammas Optional. A number of a vector of numbers. The γ parameter in eBIC. Default to `c(0,0.5,1)`.

cv_fold Optional. An integer larger than 1 if provided. The number of folds used for cross validation. If provided, losses will be calculated on each fold with model fitted on the other folds, and a `lambda_length` x `cv_fold` matrix `cv_losses` will be returned.

cv_fold_seed Optional. Seed for generating folds for cross validation.

return_raw A boolean, whether to return the raw estimates of `K`. Default to `FALSE`.

return_elts A boolean, whether to return the `elts` used for estimation. Default to `FALSE`.

Value

edgess A list of vectors of integers: indices of the non-zero edges.

BICs A `lambda_length` by `length(eBIC_gammas)` matrix of raw eBIC scores (without refitting). If `return_raw == FALSE`, may contain `Inf`s for rows after the first lambda that gives the complete graph.

lambda1s A vector of numbers of length `lambda_length`: the grid of `lambda1`s over which the estimates are obtained.

converged A vector of booleans of length `lambda_length`: indicators of convergence for each fit. If `return_raw == FALSE`, may contain `0`s for all lambdas after the first lambda that gives the complete graph.

iters A vector of integers of length `lambda_length`: the number of iterations run for each fit. If `return_raw == FALSE`, may contain `0`s for all lambdas after the first lambda that gives the complete graph.

In addition, if `centered == FALSE`,

etas A `lambda_length*p` matrix of `eta` estimates with the i-th row corresponding to the i-th `lambda1`. If `return_raw == FALSE`, may contain `NA`s after the first lambda that gives the complete graph.

if `centered == FALSE` and non-profiled,

lambda2s A vector of numbers of length `lambda_length`: the grid of `lambda2`s over which the estimates are obtained.

if `return_raw == TRUE`,

raw_estimate A list that contains `lambda_length` estimates for `K` of size `ncol(x)*ncol(x)`.

if `BIC_refit == TRUE`,

BIC_refits A `lambda_length` by `length(eBIC_gammas)` matrix of refitted eBIC scores, obtained by refitting unpenalized models restricted to the estimated edges. May contain `Inf`s for rows after the first lambda that gives the graph restricted to which an unpenalized model does not have a solution (loss unbounded from below).

if `cv_fold` is not `NULL`,

cv_losses A lambda_length x cv_fold matrix of cross validation losses. If return_raw == FALSE, may contain Infs for all lambdas after the first lambda that gives the complete graph.

if return_elts == TRUE,

elts A list of elements returned from get_elts().

Examples

```
# Examples are shown for Gaussian truncated to R+^p only. For other distributions
#   on other types of domains, please refer to \code{gen()} or \code{get_elts()},
#  as the way to call this function (\code{estimate()}) is exactly the same in those cases.
n <- 30
p <- 20
domain <- make_domain("R+", p=p)
mu <- rep(0, p)
K <- diag(p)
lambda1s <- c(0.01,0.1,0.2,0.3,0.4,0.5)
dm <- 1 + (1-1/(1+4*exp(1)*max(6*log(p)/n, sqrt(6*log(p)/n))))
x <- tmvtnorm::rtmvnorm(n, mean = mu, sigma = solve(K),
       lower = rep(0, p), upper = rep(Inf, p), algorithm = "gibbs",
       burn.in.samples = 100, thinning = 10)

## Centered estimates, no elts or h provided, mode and params provided
est1 <- estimate(x, "gaussian", domain=domain, elts=NULL, centered=TRUE,
          symmetric="symmetric", lambda1s=lambda1s, mode="min_pow",
          param1=1, param2=3, diag=dm, return_raw=TRUE, verbose=FALSE)

h_hp <- get_h_hp("min_pow", 1, 3)
## Centered estimates, no elts provided, h provided; equivalent to est1
est2 <- estimate(x, "gaussian", domain=domain, elts=NULL, centered=TRUE,
          symmetric="symmetric", lambda1s=lambda1s, h_hp=h_hp, diag=dm,
          return_raw=TRUE, verbose=FALSE)
compare_two_results(est1, est2) ## Should be almost all 0

elts_gauss_c <- get_elts(h_hp, x, setting="gaussian", domain=domain,
            centered=TRUE, diag=dm)
## Centered estimates, elts provided; equivalent to est1 and est2
## Here diagonal_multiplier will be set to the default value, equal to dm above
est3 <- estimate(x, "gaussian", domain=domain, elts=elts_gauss_c,
          symmetric="symmetric", lambda1s=lambda1s, diag=NULL,
          return_raw=TRUE, verbose=FALSE)
compare_two_results(est1, est3) ## Should be almost all 0

## Non-centered estimates with Inf penalty on eta; equivalent to est1~3
est4 <- estimate(x, "gaussian", domain=domain, elts=NULL, centered=FALSE,
          lambda_ratio=0, symmetric="symmetric", lambda1s=lambda1s,
          h=h_hp, diag=dm, return_raw=TRUE, verbose=FALSE)
sum(abs(est4$etas)) ## Should be 0 since non-centered with lambda ratio 0 is equivalent to centered
est4$etas <- NULL ## But different from est1 in that the zero etas are returned in est4
compare_two_results(est1, est4) ## Should be almost all 0
```

```
## Profiled estimates, no elts or h provided, mode and params provided
est5 <- estimate(x, "gaussian", domain=domain, elts=NULL, centered=FALSE,
          lambda_ratio=Inf, symmetric="or", lambda1s=lambda1s, mode="min_pow",
          param1=1, param2=3, diag=dm, return_raw=TRUE, verbose=FALSE)

## Profiled estimates, no elts provided, h provided; equivalent to est5
est6 <- estimate(x, "gaussian", domain=domain, elts=NULL, centered=FALSE,
          lambda_ratio=Inf, symmetric="or", lambda1s=lambda1s,
          h_hp=h_hp, diag=dm, return_raw=TRUE, verbose=FALSE)
compare_two_results(est5, est6) ## Should be almost all 0

elts_gauss_p <- get_elts(h_hp, x, setting="gaussian", domain=domain,
                centered=FALSE, profiled=TRUE, diag=dm)
## Profiled estimates, elts provided; equivalent to est5~6
est7 <- estimate(x, "gaussian", domain=domain, elts=elts_gauss_p, centered=FALSE,
          lambda_ratio=Inf, symmetric="or", lambda1s=lambda1s,
          diagonal_multiplier=NULL, return_raw=TRUE, verbose=FALSE)
compare_two_results(est5, est7) ## Should be almost all 0


## Non-centered estimates, no elts or h provided, mode and params provided
## Using 5-fold cross validation and no BIC refit
est8 <- estimate(x, "gaussian", domain=domain, elts=NULL, centered=FALSE,
          lambda_ratio=2, symmetric="and", lambda_length=100,
          mode="min_pow", param1=1, param2=3, diag=dm, return_raw=TRUE,
          BIC_refit=FALSE, cv_fold=5, cv_fold_seed=2, verbose=FALSE)

## Non-centered estimates, no elts provided, h provided; equivalent to est5
## Using 5-fold cross validation and no BIC refit
est9 <- estimate(x, "gaussian", domain=domain, elts=NULL, centered=FALSE,
          lambda_ratio=2, symmetric="and", lambda_length=100, h_hp=h_hp, diag=dm,
          return_raw=TRUE, BIC_refit=FALSE, cv_fold=5, cv_fold_seed=2, verbose=FALSE)
compare_two_results(est8, est9) ## Should be almost all 0

elts_gauss_np <- get_elts(h_hp, x, setting="gaussian", domain=domain, centered=FALSE,
                profiled=FALSE, diag=dm)
## Non-centered estimates, elts provided; equivalent to est8~9
## Using 5-fold cross validation and no BIC refit
est10 <- estimate(x, "gaussian", domain, elts=elts_gauss_np, centered=FALSE,
           lambda_ratio=2, symmetric="and", lambda_length=100, diag=NULL,
           return_raw=TRUE, BIC_refit=FALSE, cv_fold=5, cv_fold_seed=2, verbose=FALSE)
compare_two_results(est8, est10) ## Should be almost all 0
```

`find_max_ind` *Finds the max index in a vector that does not exceed a target number.*

Description

Finds the max index in a vector that does not exceed a target number.

Usage

```
find_max_ind(vals, target, start = 1)
```

Arguments

vals	A vector of numbers.
target	A number. Must not be smaller than vals[start].
start	A number, the starting index; default to 1. Must be such that vals[start] <= target.

Value

The max index i such that vals[i] <= target and i >= start.

Examples

```
for (i in 1:100) {
   vals <- 1:i
   for (start in 1:i)
      for (target in seq(start, i+0.5, by=0.5))
         if (find_max_ind(vals, target, start) != floor(target))
            stop()
}
```

---

frac_pow — *Evaluate x^(a/b) and |x|^(a/b) with integer a and b with extension to conventional operations.*

Description

Evaluate x^(a/b) and |x|^(a/b) with integer a and b with extension to conventional operations (listed under details) that would otherwise result in NaN.

Usage

```
frac_pow(x, a, b, abs)
```

Arguments

x	A number or a vector of numbers.
a	An integer.
b	An integer.
abs	TRUE or FALSE.

Details

Replace x by abs(x) below if abs == TRUE. If a == 0 && b == 0, returns log(x). If a != 0 && b == 0, returns exp(a*x). Otherwise, for b != 0, evaluates x^(a/b) with the following extensions. 0^0 evaluates to 1. If x < 0, returns (-1)^a * |x|^(a/b) if b is odd, or NaN otherwise. If x == 0 && a < 0, returns NaN.

Value

A vector of numbers of the same size as x. See details.

Examples

```
frac_pow(-5:5, 3, 2, TRUE) - abs(-5:5)^(3/2)
frac_pow(-5:5, 5, 3, FALSE) - sign(-5:5)^5*abs(-5:5)^(5/3)
frac_pow(-5:5, 2, 3, FALSE) - ((-5:5)^2)^(1/3)
frac_pow(c(-5:-1,1:5), 0, 0, TRUE) - log(abs(c(-5:-1,1:5)))
frac_pow(-5:5, 0, 1, FALSE) - 1
frac_pow(-5:5, 3, 0, FALSE) - exp(3*-5:5)
```

---

gcd *Finds the greatest (positive) common divisor of two integers.*

Description

Finds the greatest (positive) common divisor of two integers; if one of them is 0, returns the absolute value of the other number.

Usage

```
gcd(a, b)
```

Arguments

a	An integer.
b	An integer.

Value

The greatest (positive) common divisor of two integers; if one of them is 0, returns the absolute value of the other number.

Examples

```
gcd(1, 2)
gcd(1, -2)
gcd(12, -18)
gcd(-12, 18)
gcd(15, 0)
gcd(0, -15)
gcd(0, 0)
```

`gen` *Random data generator from general* `a-b` *distributions with general domain types, assuming a and b are rational numbers.*

Description

Random data generator from general a-b graphs with general domain types using adaptive rejection metropolis sampling (ARMS). x^(0/0) treated as log(x) and x^(n/0) as exp(x) for n non-zero. Density only guaranteed to be a proper density when 2*a > b >= 0 or when a = b = 0.

Usage

```
gen(
  n,
  setting,
  abs,
  eta,
  K,
  domain,
  finite_infinity = NULL,
  xinit = NULL,
  seed = NULL,
  burn_in = 1000,
  thinning = 100,
  verbose = TRUE,
  remove_outofbound = TRUE
)
```

Arguments

`n`	An integer, number of observations.
`setting`	A string that indicates the distribution type, must be one of `"exp"`, `"gamma"`, `"gaussian"`, `"log_log"`, `"log_log_sum0"`, or of the form `"ab_NUM1_NUM2"`, where `NUM1` is the `a` value and `NUM2` is the `b` value, and `NUM1` and `NUM2` must be integers or two integers separated by "/", e.g. "ab_2_2", "ab_2_5/4" or "ab_2/3_1/2".
`abs`	A boolean. If TRUE, density is rewritten as f(|x|), i.e. with |x|^(a_numer/a_denom) and |x|^(b_numer/b_denom)

`eta` A vector, the linear part in the distribution.

`K` A square matrix, the interaction matrix. There should exist some C > 0 such that

$$\boldsymbol{x}^{a\top}\mathbf{K}\boldsymbol{x}^{a}/(\boldsymbol{x}^{a\top}\boldsymbol{x}^{a}) >= C$$

for all x in the domain (i.e. `K` is positive definite if `domain$type == "R"` and `K` is co-positive if `domain$type == "R+"`.). If `a_numer == a_denom == b_numer == b_denom == 0 && domain$type == "simplex"`, K can also have all row and column sums equal to 0 but have all but one eigenvalues (0) positive.

`domain` A list returned from `make_domain()` that represents the domain.

`finite_infinity`
A finite positive number. `Inf` in actual generation will be truncated to `finite_infinity` if applicable. Although the code will adaptively increase `finite_infinity`, the user should set it to a large number initially so that `abs(x) > finite_infinity` with very small probability.

`xinit` Optional. A p-vector, an initial point in the domain. If the domain is defined by more than one ineq or by one ineq containing negative coefficients, this must be provided. In the unlikely case where the function fails to automatically generate an initial point this should also be provided.

`seed` Optional. A number, the seed for the random generator.

`burn_in` Optional. A positive integer, the number of burn-in samples in ARMS to be discarded, meaning that samples from the first `burn_in` x `thinning` iterations will be discarded.

`thinning` Optional. A positive integer, thinning factor in ARMS. Samples are taken at iteration steps $(\text{burn\_in}+1)\times\text{thinning}$, ..., $(\text{burn\_in}+n)\times\text{thinning}$. Default to `100`.

`verbose` Optional. A boolean. If `TRUE`, prints a progress bar showing the progress. Defaults to `TRUE`.

`remove_outofbound`
Optional. A logical, defaults to `TRUE`. If `TRUE`, a test whether each sample lies inside the domain will be done, which may take a while for larger sample sizes, and rows that do not lie in the domain will be removed (may happen for `domain$type == "polynomial"` with more than 1 ineq and an OR ("|") in `domain$rule`.).

Details

NOTE: For polynomial domains with many ineqs and a rule containing "OR" ("|"), not all samples generated are guaranteed to be inside the domain. It is thus recommended to set `remove_outofbound` to `TRUE` and rerun the function with new initial points until the desired number of in-bound samples have been generated.

Randomly generates `n` samples from the `p`-variate `a`-`b` distributions with parameters $\boldsymbol{\eta}$ and $\mathbf{K}$, where `p` is the length of $\boldsymbol{\eta}$ or the dimension of the square matrix $\mathbf{K}$.

Letting `a=a_numer/a_denom` and `b=b_numer/b_denom`, the `a`-`b` distribution is proportional to

$$\exp\left(-\frac{1}{2a}\boldsymbol{x}^{a\top}\mathbf{K}\boldsymbol{x}^{a}+\boldsymbol{\eta}^{\top}\frac{\boldsymbol{x}^{b}-\mathbf{1}_{p}}{b}\right)$$

. Note that `x^(0/0)` is understood as `log(x)`, and `x^(n/0)` with nonzero `n` is `exp(n*x)`, and in both cases the `a` and `b` in the denominators in the density are treated as 1.

Value

An $n \times p$ matrix of samples, where p is the length of `eta`.

Examples

```
n <- 20
p <- 10
eta <- rep(0, p)
K <- diag(p)
dm <- 1 + (1-1/(1+4*exp(1)*max(6*log(p)/n, sqrt(6*log(p)/n))))

# Gaussian on sum(x^2) > 10 && sum(x^(1/3)) > 10 with x allowed to be negative
domain <- make_domain("polynomial", p=p, rule="1 && 2",
       ineqs=list(list("expression"="sum(x^2)>10", abs=FALSE, nonnegative=FALSE),
                  list("expression"="sum(x^(1/3))>10", abs=FALSE, nonnegative=FALSE)))
xinit <- rep(sqrt(20/p), p)
x <- gen(n, setting="gaussian", abs=FALSE, eta=eta, K=K, domain=domain, finite_infinity=100,
       xinit=xinit, seed=2, burn_in=500, thinning=100, verbose=FALSE)

# exp on ([0, 1] v [2,3])^p
domain <- make_domain("uniform", p=p, lefts=c(0,2), rights=c(1,3))
x <- gen(n, setting="exp", abs=FALSE, eta=eta, K=K, domain=domain, xinit=NULL,
       seed=2, burn_in=500, thinning=100, verbose=TRUE)

# gamma on {x1 > 1 && log(1.3) < x2 < 1 && x3 > log(1.3) && ... && xp > log(1.3)}
domain <- make_domain("polynomial", p=p, rule="1 && 2 && 3",
       ineqs=list(list("expression"="x1>1", abs=FALSE, nonnegative=TRUE),
                  list("expression"="x2<1", abs=FALSE, nonnegative=TRUE),
                  list("expression"="exp(x)>1.3", abs=FALSE, nonnegative=FALSE)))
set.seed(1)
xinit <- c(1.5, 0.5, abs(stats::rnorm(p-2))+log(1.3))
x <- gen(n, setting="gamma", abs=FALSE, eta=eta, K=K, domain=domain, finite_infinity=100,
       xinit=xinit, seed=2, burn_in=500, thinning=100, verbose=FALSE)

# a0.6_b0.7 on {x in R_+^p: sum(log(x))<2 || (x1^(2/3)-1.3x2^(-3)<1 && exp(x1)+2.3*x2>2)}
domain <- make_domain("polynomial", p=p, rule="1 || (2 && 3)",
       ineqs=list(list("expression"="sum(log(x))<2", abs=FALSE, nonnegative=TRUE),
                 list("expression"="x1^(2/3)-1.3x2^(-3)<1", abs=FALSE, nonnegative=TRUE),
                   list("expression"="exp(x1)+2.3*x2^2>2", abs=FALSE, nonnegative=TRUE)))
xinit <- rep(1, p)
x <- gen(n, setting="ab_3/5_7/10", abs=FALSE, eta=eta, K=K, domain=domain, finite_infinity=1e4,
       xinit=xinit, seed=2, burn_in=500, thinning=100, verbose=FALSE)

# log_log model exp(-log(x) %*% K %*% log(x)/2 + eta %*% log(x)) on {x in R_+^p: sum_j j * xj <= 1}
domain <- make_domain("polynomial", p=p,
       ineqs=list(list("expression"=paste(paste(sapply(1:p,
                           function(j){paste(j, "x", j, sep="")}), collapse="+"), "<1"),
                     abs=FALSE, nonnegative=TRUE)))
x <- gen(n, setting="log_log", abs=FALSE, eta=eta, K=K, domain=domain, finite_infinity=100,
```

```
       xinit=NULL, seed=2, burn_in=500, thinning=100, verbose=FALSE)

# log_log model on the simplex with K having row and column sums 0 (Aitchison model)
domain <- make_domain("simplex", p=p)
K <- -cov_cons("band", p=p, spars=3, eig=1)
diag(K) <- diag(K) - rowSums(K) # So that rowSums(K) == colSums(K) == 0
eigen(K)$val[(p-1):p] # Make sure K has one 0 and p-1 positive eigenvalues
x <- gen(n, setting="log_log_sum0", abs=FALSE, eta=eta, K=K, domain=domain, xinit=NULL,
       seed=2, burn_in=500, thinning=100, verbose=FALSE)

# Gumbel_Gumbel model exp(-exp(2x) %*% K %*% exp(2x)/2 + eta %*% exp(-3x)) on {sum(|x|) < 1}
domain <- make_domain("polynomial", p=p,
       ineqs=list(list("expression"="sum(x)<1", abs=TRUE, nonnegative=FALSE)))
K <- diag(p)
x <- gen(n, setting="ab_2/0_-3/0", abs=FALSE, eta=eta, K=K, domain=domain, finite_infinity=100,
       xinit=NULL, seed=2, burn_in=500, thinning=100, verbose=FALSE)
```

`get_crit_nopenalty` *Minimized loss for unpenalized restricted asymmetric models.*

Description

Analytic solution of the minimized loss for an unpenalized asymmetric model restricted to a given support. Does not work if `symmetric == "symmetric"`.

Usage

```
get_crit_nopenalty(
  elts,
  exclude = NULL,
  exclude_eta = NULL,
  previous_res = NULL
)
```

Arguments

`elts`	A list, elements necessary for calculations returned by get_elts().
`exclude`	Optional. A p*p binary matrix or a p^2 binary vector, where 1 indicates the entry in K was estimated to 0 in the previous estimate. Default to `NULL`.
`exclude_eta`	Optional. A p-binary vector, similar to `exclude`. Default to `NULL`.
`previous_res`	Optional. A list, the returned list by `get_results()` run previously with another lambda value. Default to `NULL`.

Details

If `previous_res` is provided, `exclude` and `exclude_eta` must be `NULL` or be consistent with the estimated support in `previous_res`. If `previous_res` and `exclude` are both `NULL`, assume all edges are present. The same applies to the non-profiled non-centered case when `previous_res` and `exclude_eta` are both `NULL`.

Value

A number, the refitted loss.

Examples

```
# Examples are shown for Gaussian truncated to R+^p only. For other distributions
#   on other types of domains, please refer to \code{gen()} or \code{get_elts()}, as the
#  way to call this function (\code{get_crit_nopenalty()}) is exactly the same in those cases.
n <- 50
p <- 30
domain <- make_domain("R+", p=p)
h_hp <- get_h_hp("min_pow", 1, 3)
mu <- rep(0, p)
K <- diag(p)
dm <- 1 + (1-1/(1+4*exp(1)*max(6*log(p)/n, sqrt(6*log(p)/n))))
x <- tmvtnorm::rtmvnorm(n, mean = mu, sigma = solve(K),
       lower = rep(0, p), upper = rep(Inf, p), algorithm = "gibbs",
       burn.in.samples = 100, thinning = 10)

elts_gauss_np <- get_elts(h_hp, x, setting="gaussian", domain=domain,
                 centered=FALSE, profiled=FALSE, diag=dm)
res_nc_np <- get_results(elts_gauss_np, symmetric="symmetric", lambda1=0.35,
               lambda2=2, previous_res=NULL, is_refit=FALSE)
get_crit_nopenalty(elts_gauss_np, previous_res=res_nc_np)
```

`get_dist` *Finds the distance of each element in a matrix x to the its boundary of the domain while fixing the others in the same row.*

Description

Finds the distance of each element in a matrix `x` to its boundary of the `domain` while fixing the others in the same row.

Usage

```
get_dist(x, domain)
```

Arguments

`x`	An `n` by `p` matrix, the data matrix, where `n` is the sample size and `p` the dimension.
`domain`	A list returned from `make_domain()` that represents the domain.

Details

Returned matrix dx has its i, j-th component the distance of $x_{i,j}$ to the boundary of domain, assuming $x_{i,-j}$ are fixed. The matrix has the same size of x (n by p), or if domain$type == "simplex" and x has full dimension p, it has p-1 columns. Returned matrix dpx contains the component-wise derivatives of dx in its components. That is, its i, j-th component is 0 if $x_{i,j}$ is unbounded or is bounded from both below and above or is at the boundary, or -1 if $x_{i,j}$ is closer to its lower boundary (or if its bounded from below but unbounded from above), or 1 otherwise.

Value

A list that contains h(dist(x, domain)) and h\'(dist(x, domain)).

dx	Coordinate-wise distance to the boundary.
dpx	Coordinate-wise derivative of dx.

Examples

```
n <- 20
p <- 10
eta <- rep(0, p)
K <- diag(p)
dm <- 1 + (1-1/(1+4*exp(1)*max(6*log(p)/n, sqrt(6*log(p)/n))))

# Gaussian on R^p:
domain <- make_domain("R", p=p)
x <- mvtnorm::rmvnorm(n, mean=solve(K, eta), sigma=solve(K))
# Equivalently:

x2 <- gen(n, setting="gaussian", abs=FALSE, eta=eta, K=K, domain=domain, finite_infinity=100,
        xinit=NULL, burn_in=1000, thinning=100, verbose=FALSE)

dist <- get_dist(x, domain)
# dx is all Inf and dpx is all 0 since each coordinate is unbounded with domain R
c(all(is.infinite(dist$dx)), all(dist$dpx==0))

# exp on R_+^p:
domain <- make_domain("R+", p=p)
x <- gen(n, setting="exp", abs=FALSE, eta=eta, K=K, domain=domain, finite_infinity=100,
       xinit=NULL, seed=2, burn_in=1000, thinning=100, verbose=FALSE)
dist <- get_dist(x, domain)
# dx is x and dpx is 1; with domain R+, the distance of x to the boundary is just x itself
c(max(abs(dist$dx - x))<.Machine$double.eps^0.5, all(dist$dpx == 1))

# Gaussian on sum(x^2) > p with x allowed to be negative
domain <- make_domain("polynomial", p=p,
     ineqs=list(list("expression"=paste("sum(x^2)>", p), abs=FALSE, nonnegative=FALSE)))
x <- gen(n, setting="gaussian", abs=FALSE, eta=eta, K=K, domain=domain, finite_infinity=100,
       xinit=NULL, seed=2, burn_in=1000, thinning=100, verbose=FALSE)
dist <- get_dist(x, domain)
quota <- p - (rowSums(x^2) - x^2) # How much should xij^2 at least be so that sum(xi^2) > p?
# How far is xij from +/-sqrt(quota), if quota >= 0?
```

```
dist_to_bound <- abs(x[quota >= 0]) - abs(sqrt(quota[quota >= 0]))
max(abs(dist$dx[is.finite(dist$dx)] - dist_to_bound)) # Should be equal to our own calculations
# dist'(x) should be the same as the sign of x
all(dist$dpx[is.finite(dist$dx)] == sign(x[quota >= 0]))
# quota is negative <-> sum of x_{i,-j}^2 already > p <-> xij unbounded
#   given others <-> distance to boundary is Inf
all(quota[is.infinite(dist$dx)] < 0)

# gamma on ([0, 1] v [2,3])^p
domain <- make_domain("uniform", p=p, lefts=c(0,2), rights=c(1,3))
x <- gen(n, setting="gamma", abs=FALSE, eta=eta, K=K, domain=domain,
       xinit=NULL, seed=2, burn_in=1000, thinning=100, verbose=FALSE)
dist <- get_dist(x, domain)
# If 0 <= xij <= 1, distance to boundary is min(x-0, 1-x)
max(abs(dist$dx - pmin(x, 1-x))[x >= 0 & x <= 1])
# If 0 <= xij <= 1, dist'(xij) is 1 if it is closer to 0, or -1 if it is closer 1,
#   assuming xij %in% c(0, 0.5, 1) with probability 0
all((dist$dpx == 2 * (1-x > x) - 1)[x >= 0 & x <= 1])
# If 2 <= xij <= 3, distance to boundary is min(x-2, 3-x)
max(abs(dist$dx - pmin(x-2, 3-x))[x >= 2 & x <= 3])
# If 2 <= xij <= 3, dist'(xij) is 1 if it is closer to 2, or -1 if it is closer 3,
#   assuming xij %in% c(2, 2.5, 3) with probability 0
all((dist$dpx == 2 * (3-x > x-2) - 1)[x >= 2 & x <= 3])

# a0.6_b0.7 on {x1 > 1 && 0 < x2 < 1 && x3 > 0 && ... && xp > 0}
domain <- make_domain("polynomial", p=p, rule="1 && 2 && 3",
       ineqs=list(list("expression"="x1>1", abs=FALSE, nonnegative=TRUE),
                       list("expression"="x2<1", abs=FALSE, nonnegative=TRUE),
                       list("expression"="exp(x)>1.3", abs=FALSE, nonnegative=FALSE)))
set.seed(1)
xinit <- c(1.5, 0.5, abs(stats::rnorm(p-2)) + log(1.3))
x <- gen(n, setting="ab_3/5_7/10", abs=FALSE, eta=eta, K=K, domain=domain,
       finite_infinity=100, xinit=xinit, seed=2, burn_in=1000, thinning=100,
       verbose=FALSE)
dist <- get_dist(x, domain)
# x_{i1} has uniform bound [1, +Inf), so its distance to its boundary is x_{i1} - 1
max(abs(dist$dx[,1] - (x[,1] - 1)))
# x_{i2} has uniform bound [log(1.3), 1], so its distance to its boundary
#   is min(x_{i2} - log(1.3), 1 - x_{i2})
max(abs(dist$dx[,2] - pmin(x[,2] - log(1.3), 1 - x[,2])))
# x_{ij} for j >= 3 has uniform bound [log(1.3), +Inf), so its distance to its boundary
#   is simply x_{ij} - log(1.3)
max(abs(dist$dx[,3:p] - (x[,3:p] - log(1.3))))
# dist\'(xi2) is 1 if it is closer to log(1.3), or -1 if it is closer 1,
#   assuming x_{i2} %in% c(log(1.3), (1+log(1.3))/2, 1) with probability 0
all((dist$dpx[,2] == 2 * (1 - x[,2] > x[,2] - log(1.3)) - 1))
# x_{ij} for j != 2 is bounded from below but unbounded from above, so dist\'(xij) is always 1
all(dist$dpx[,-2] == 1)

# log_log model on {x in R_+^p: sum_j j * xj <= 1}
domain <- make_domain("polynomial", p=p,
       ineqs=list(list("expression"=paste(paste(sapply(1:p,
                           function(j){paste(j, "x", j, sep="")}), collapse="+"), "<1"),
```

```
                    abs=FALSE, nonnegative=TRUE)))
x <- gen(n, setting="log_log", abs=FALSE, eta=eta, K=K, domain=domain, finite_infinity=100,
       xinit=NULL, seed=2, burn_in=1000, thinning=100, verbose=FALSE)
dist <- get_dist(x, domain)
# Upper bound for j * xij so that sum_j j * xij <= 1
quota <- 1 - (rowSums(t(t(x) * 1:p)) - t(t(x) * 1:p))
# Distance of xij to its boundary is min(xij - 0, quota_{i,j} / j - xij)
max(abs(dist$dx - pmin((t(t(quota) / 1:p) - x), x)))

# log_log model on the simplex with K having row and column sums 0 (Aitchison model)
domain <- make_domain("simplex", p=p)
K <- -cov_cons("band", p=p, spars=3, eig=1)
diag(K) <- diag(K) - rowSums(K) # So that rowSums(K) == colSums(K) == 0
eigen(K)$val[(p-1):p] # Make sure K has one 0 and p-1 positive eigenvalues
x <- gen(n, setting="log_log_sum0", abs=FALSE, eta=eta, K=K, domain=domain,
        xinit=NULL, seed=2, burn_in=1000, thinning=100, verbose=FALSE)
# Note that dist$dx and dist$dpx only has p-1 columns -- excluding the last coordinate in x
dist <- get_dist(x, domain)
# Upper bound for x_{i,j} so that x_{i,1} + ... + x_{i,p-1} <= 1
quota <- 1 - (rowSums(x[,-p]) - x[,-p])
# Distance of x_{i,j} to its boundary is min(xij - 0, quota_{i,j} - xij)
max(abs(dist$dx - pmin(quota - x[,-p], x[,-p])))
```

`get_elts` *The function wrapper to get the elements necessary for calculations for all settings.*

Description

The function wrapper to get the elements necessary for calculations for all settings.

Usage

```
get_elts(
  h_hp,
  x,
  setting,
  domain,
  centered = TRUE,
  profiled_if_noncenter = TRUE,
  scale = "",
  diagonal_multiplier = 1,
  use_C = TRUE,
  tol = .Machine$double.eps^0.5,
  unif_dist = NULL
)
```

Arguments

h_hp	A function that returns a list containing hx=h(x) (element-wise) and hpx=hp(x) (element-wise derivative of h) when applied to a vector or a matrix x, both of which has the same shape as x.
x	An n by p matrix, the data matrix, where n is the sample size and p the dimension.
setting	A string that indicates the distribution type, must be one of "exp", "gamma", "gaussian", "log_log", "log_log_sum0", or of the form "ab_NUM1_NUM2", where NUM1 is the a value and NUM2 is the b value, and NUM1 and NUM2 must be integers or two integers separated by "/", e.g. "ab_2_2", "ab_2_5/4" or "ab_2/3_1/2". If domain$type == "simplex", only "log_log" and "log_log_sum0" are supported, and on the other hand "log_log_sum0" is supported for domain$type == "simplex" only.
domain	A list returned from make_domain() that represents the domain.
centered	A boolean, whether in the centered setting(assume $\boldsymbol{\mu} = \boldsymbol{\eta} = 0$) or not. Default to TRUE.
profiled_if_noncenter	A boolean, whether in the profiled setting ($\lambda_{\boldsymbol{\eta}} = 0$) if non-centered. Parameter ignored if centered=TRUE. Default to TRUE. Can only be FALSE if setting == "log_log_sum0" && centered == FALSE.
scale	A string indicating the scaling method. If contains "sd", columns are scaled by standard deviation; if contains "norm", columns are scaled by l2 norm; if contains "center" and setting == "gaussian" && domain$type == "R", columns are centered to have mean zero. Default to "norm".
diagonal_multiplier	A number >= 1, the diagonal multiplier.
use_C	Optional. A boolean, use C (TRUE) or R (FALSE) functions for computation. Default to TRUE. Ignored if setting == "gaussian" && domain$type == "R".
tol	Optional. A positive number. If setting != "gaussian" || domain$type != "R", function stops if any entry if smaller than -tol, and all entries between -tol and 0 are set to tol, for numerical stability and to avoid violating the assumption that $h(\mathbf{x}) > 0$ almost surely.
unif_dist	Optional, defaults to NULL. If not NULL, h_hp must be NULL and unif_dist(x) must return a list containing "g0" of length nrow(x) and "g0d" of dimension dim(x), representing the l2 distance and the gradient of the l2 distance to the boundary: the true l2 distance function to the boundary is used for all coordinates in place of h_of_dist; see "Estimating Density Models with Complex Truncation Boundaries" by Liu et al, 2019. That is, $(h_j \circ \phi_j)(x_i)$ in the score-matching loss is replaced by $g_0(x_i)$, the l2 distance of xi to the boundary of the domain.

Details

Computes the $\boldsymbol{\Gamma}$ matrix and the $\boldsymbol{g}$ vector for generalized score matching.

Here, $\mathbf{\Gamma}$ is block-diagonal, and in the non-profiled non-centered setting, the j-th block is composed of $\mathbf{\Gamma}_{\mathbf{KK},j}$, $\mathbf{\Gamma}_{\mathbf{K}\boldsymbol{\eta},j}$ and its transpose, and finally $\mathbf{\Gamma}_{\boldsymbol{\eta}\boldsymbol{\eta},j}$. In the centered case, only $\mathbf{\Gamma}_{\mathbf{KK},j}$ is computed. In the profiled non-centered case,

$$\mathbf{\Gamma}_j \equiv \mathbf{\Gamma}_{\mathbf{KK},j} - \mathbf{\Gamma}_{\mathbf{K}\boldsymbol{\eta},j}\mathbf{\Gamma}_{\boldsymbol{\eta}\boldsymbol{\eta},j}^{-1}\mathbf{\Gamma}_{\mathbf{K}\boldsymbol{\eta}}^{\top}.$$

Similarly, in the non-profiled non-centered setting, $\boldsymbol{g}$ can be partitioned p parts, each with a p-vector $\boldsymbol{g}_{\mathbf{K},j}$ and a scalar $g_{\boldsymbol{\eta},j}$. In the centered setting, only $\boldsymbol{g}_{\mathbf{K},j}$ is needed. In the profiled non-centered case,

$$\boldsymbol{g}_j \equiv \boldsymbol{g}_{\mathbf{K},j} - \mathbf{\Gamma}_{\mathbf{K}\boldsymbol{\eta},j}\mathbf{\Gamma}_{\boldsymbol{\eta}\boldsymbol{\eta},j}^{-1}g_{\boldsymbol{\eta},j}.$$

The formulae for the pieces above are

$$\mathbf{\Gamma}_{\mathbf{KK},j} \equiv \frac{1}{n}\sum_{i=1}^{n} h\left(X_j^{(i)}\right) {X_j^{(i)}}^{2a-2} {\boldsymbol{X}^{(i)}}^{a} {\boldsymbol{X}^{(i)}}^{a\top},$$

$$\mathbf{\Gamma}_{\mathbf{K}\boldsymbol{\eta},j} \equiv -\frac{1}{n}\sum_{i=1}^{n} h\left(X_j^{(i)}\right) {X_j^{(i)}}^{a+b-2} {\boldsymbol{X}^{(i)}}^{a},$$

$$\mathbf{\Gamma}_{\boldsymbol{\eta}\boldsymbol{\eta},j} \equiv \frac{1}{n}\sum_{i=1}^{n} h\left(X_j^{(i)}\right) {X_j^{(i)}}^{2b-2},$$

$$\boldsymbol{g}_{\mathbf{K},j} \equiv \frac{1}{n}\sum_{i=1}^{n}\left(h'\left(X_j^{(i)}\right){X_j^{(i)}}^{a-1} + (a-1)h\left(X_j^{(i)}\right){X_j^{(i)}}^{a-2}\right){\boldsymbol{X}^{(i)}}^{a} + ah\left(X_j^{(i)}\right){X_j^{(i)}}^{2a-2}\boldsymbol{e}_{j,p},$$

$$\boldsymbol{g}_{\boldsymbol{\eta},j} \equiv \frac{1}{n}\sum_{i=1}^{n} -h'\left(X_j^{(i)}\right){X_j^{(i)}}^{b-1} - (b-1)h\left(X_j^{(i)}\right){X_j^{(i)}}^{b-2},$$

where $\boldsymbol{e}_{j,p}$ is the p-vector with 1 at the j-th position and 0 elsewhere.

In the profiled non-centered setting, the function also returns t_1 and t_2 defined as

$$\boldsymbol{t}_1 \equiv \mathbf{\Gamma}_{\boldsymbol{\eta}\boldsymbol{\eta}}^{-1}\boldsymbol{g}_{\boldsymbol{\eta}}, \quad \boldsymbol{t}_2 \equiv \mathbf{\Gamma}_{\boldsymbol{\eta}\boldsymbol{\eta}}^{-1}\mathbf{\Gamma}_{\mathbf{K}\boldsymbol{\eta}}^{\top},$$

so that $\hat{\boldsymbol{\eta}} = \boldsymbol{t}_1 - \boldsymbol{t}_2\mathrm{vec}(\hat{\mathbf{K}})$.

Value

A list that contains the elements necessary for estimation.

`n`	The sample size.
`p`	The dimension.
`centered`	The centered setting or not. Same as input.
`scale`	The scaling method. Same as input.
`diagonal_multiplier`	The diagonal multiplier. Same as input.
`diagonals_with_multiplier`	A vector that contains the diagonal entries of $\mathbf{\Gamma}$ after applying the multiplier.
`domain_type`	The domain type. Same as domain$type in the input.

`setting` The setting. Same as input.

`g_K` The $\boldsymbol{g}$ vector. In the non-profiled non-centered setting, this is the $\boldsymbol{g}$ sub-vector corresponding to $\mathbf{K}$. A p^2-vector. Not returned if `setting == "gaussian" && domain$type == "R"` since it is just $diag(p)$.

`Gamma_K` The $\boldsymbol{\Gamma}$ matrix with no diagonal multiplier. In the non-profiled non-centered setting, this is the $\boldsymbol{\Gamma}$ sub-matrix corresponding to $\mathbf{K}$. A vector of length p^2 if `setting == "gaussian" && domain$type == "R"` or p^3 otherwise.

`g_eta` Returned in the non-profiled non-centered setting. The $\boldsymbol{g}$ sub-vector corresponding to $\boldsymbol{\eta}$. A p-vector. Not returned if `setting == "gaussian" && domain$type == "R"` since it is just $numeric(p)$.

`Gamma_K_eta` Returned in the non-profiled non-centered setting. The $\boldsymbol{\Gamma}$ sub-matrix corresponding to interaction between $\mathbf{K}$ and $\boldsymbol{\eta}$. If `setting == "gaussian" && domain$type == "R"`, returns a vector of length p, or p^2 otherwise.

`Gamma_eta` Returned in the non-profiled non-centered setting. The $\boldsymbol{\Gamma}$ sub-matrix corresponding to $\boldsymbol{\eta}$. A p-vector. Not returned if `setting == "gaussian" && domain$type == "R"` since it is just `rep(1,p)`.

`t1,t2` Returned in the profiled non-centered setting, where the $\boldsymbol{\eta}$ estimate can be retrieved from $\boldsymbol{t_1} - \boldsymbol{t_2}\hat{\mathbf{K}}$ after appropriate resizing.

If `domain$type == "simplex"`, the following are also returned.

`Gamma_K_jp` A matrix of size `p` by `p(p-1)`. The `(j-1)*p+1` through `j*p` columns represent the interaction matrix between the `j`-th column and the `m`-th column of `K`.

`Gamma_Kj_etap` Non-centered only. A matrix of size `p` by `p(p-1)`. The `j`-th column represents the interaction between the `j`-th column of `K` and `eta[p]`.

`Gamma_Kp_etaj` Non-centered only. A matrix of size `p` by `p(p-1)`. The `j`-th column represents the interaction between the `p`-th column of `K` and `eta[j]`. Note that it is equal to `Gamma_Kj_etap` if `setting` does not contain substring `"sum0"`.

`Gamma_eta_jp` Non-centered only. A vector of size `p-1`. The `j`-th component represents the interaction between `eta[j]` and `eta[p]`.

Examples

```
n <- 30
p <- 10
eta <- rep(0, p)
K <- diag(p)
dm <- 1 + (1-1/(1+4*exp(1)*max(6*log(p)/n, sqrt(6*log(p)/n))))

# Gaussian on R^p:
domain <- make_domain("R", p=p)
x <- mvtnorm::rmvnorm(n, mean=solve(K, eta), sigma=solve(K))
# Equivalently:

x2 <- gen(n, setting="gaussian", abs=FALSE, eta=eta, K=K, domain=domain, finite_infinity=100,
        xinit=NULL, burn_in=1000, thinning=100, verbose=FALSE)

elts <- get_elts(NULL, x, "gaussian", domain, centered=TRUE, scale="norm", diag=dm)
```

```
elts <- get_elts(NULL, x, "gaussian", domain, FALSE, profiled=FALSE, scale="sd", diag=dm)

# Gaussian on R_+^p:
domain <- make_domain("R+", p=p)
x <- tmvtnorm::rtmvnorm(n, mean = solve(K, eta), sigma = solve(K),
       lower = rep(0, p), upper = rep(Inf, p), algorithm = "gibbs",
       burn.in.samples = 100, thinning = 10)
# Equivalently:

x2 <- gen(n, setting="gaussian", abs=FALSE, eta=eta, K=K, domain=domain,
       finite_infinity=100, xinit=NULL, burn_in=1000, thinning=100, verbose=FALSE)

h_hp <- get_h_hp("min_pow", 1, 3)
elts <- get_elts(h_hp, x, "gaussian", domain, centered=TRUE, scale="norm", diag=dm)

# Gaussian on sum(x^2) > 1 && sum(x^(1/3)) > 1 with x allowed to be negative
domain <- make_domain("polynomial", p=p, rule="1 && 2",
       ineqs=list(list("expression"="sum(x^2)>1", abs=FALSE, nonnegative=FALSE),
                  list("expression"="sum(x^(1/3))>1", abs=FALSE, nonnegative=FALSE)))
xinit <- rep(sqrt(2/p), p)
x <- gen(n, setting="gaussian", abs=FALSE, eta=eta, K=K, domain=domain, finite_infinity=100,
       xinit=xinit, seed=2, burn_in=1000, thinning=100, verbose=FALSE)
h_hp <- get_h_hp("min_pow", 1, 3)
elts <- get_elts(h_hp, x, "gaussian", domain, centered=FALSE,
       profiled_if_noncenter=TRUE, scale="", diag=dm)

# exp on ([0, 1] v [2,3])^p
domain <- make_domain("uniform", p=p, lefts=c(0,2), rights=c(1,3))
x <- gen(n, setting="exp", abs=FALSE, eta=eta, K=K, domain=domain,
       xinit=NULL, seed=2, burn_in=1000, thinning=100, verbose=FALSE)
h_hp <- get_h_hp("min_pow", 1.5, 3)
elts <- get_elts(h_hp, x, "exp", domain, centered=TRUE, scale="", diag=dm)
elts <- get_elts(h_hp, x, "exp", domain, centered=FALSE,
       profiled_if_noncenter=FALSE, scale="", diag=dm)

# gamma on {x1 > 1 && log(1.3) < x2 < 1 && x3 > log(1.3) && ... && xp > log(1.3)}
domain <- make_domain("polynomial", p=p, rule="1 && 2 && 3",
       ineqs=list(list("expression"="x1>1", abs=FALSE, nonnegative=TRUE),
                  list("expression"="x2<1", abs=FALSE, nonnegative=TRUE),
                  list("expression"="exp(x)>1.3", abs=FALSE, nonnegative=TRUE)))
set.seed(1)
xinit <- c(1.5, 0.5, abs(stats::rnorm(p-2))+log(1.3))
x <- gen(n, setting="gamma", abs=FALSE, eta=eta, K=K, domain=domain, finite_infinity=100,
       xinit=xinit, seed=2, burn_in=1000, thinning=100, verbose=FALSE)
h_hp <- get_h_hp("min_pow", 1.5, 3)
elts <- get_elts(h_hp, x, "gamma", domain, centered=TRUE, scale="", diag=dm)
elts <- get_elts(h_hp, x, "gamma", domain, centered=FALSE,
       profiled_if_noncenter=FALSE, scale="", diag=dm)

# a0.6_b0.7 on {x in R_+^p: sum(log(x))<2 || (x1^(2/3)-1.3x2^(-3)<1 && exp(x1)+2.3*x2>2)}
domain <- make_domain("polynomial", p=p, rule="1 || (2 && 3)",
       ineqs=list(list("expression"="sum(log(x))<2", abs=FALSE, nonnegative=TRUE),
                  list("expression"="x1^(2/3)-1.3x2^(-3)<1", abs=FALSE, nonnegative=TRUE),
```

```
                  list("expression"="exp(x1)+2.3*x2^2>2", abs=FALSE, nonnegative=TRUE)))
xinit <- rep(1, p)
x <- gen(n, setting="ab_3/5_7/10", abs=FALSE, eta=eta, K=K, domain=domain, finite_infinity=100,
       xinit=xinit, seed=2, burn_in=1000, thinning=100, verbose=FALSE)
h_hp <- get_h_hp("min_pow", 1.4, 3)
elts <- get_elts(h_hp, x, "ab_3/5_7/10", domain, centered=TRUE, scale="", diag=dm)
elts <- get_elts(h_hp, x, "ab_3/5_7/10", domain, centered=FALSE,
       profiled_if_noncenter=TRUE, scale="", diag=dm)

# log_log model on {x in R_+^p: sum_j j * xj <= 1}
domain <- make_domain("polynomial", p=p,
       ineqs=list(list("expression"=paste(paste(sapply(1:p,
                           function(j){paste(j, "x", j, sep="")}), collapse="+"), "<1"),
                     abs=FALSE, nonnegative=TRUE)))
x <- gen(n, setting="log_log", abs=FALSE, eta=eta, K=K, domain=domain,
       finite_infinity=100, xinit=NULL, seed=2, burn_in=1000, thinning=100,
       verbose=FALSE)
h_hp <- get_h_hp("min_pow", 2, 3)
elts <- get_elts(h_hp, x, "log_log", domain, centered=TRUE, scale="", diag=dm)
elts <- get_elts(h_hp, x, "log_log", domain, centered=FALSE,
       profiled_if_noncenter=FALSE, scale="", diag=dm)
# Example of using the uniform distance function to boundary as in Liu (2019)
g0 <- function(x) {
       row_min <- apply(x, 1, min)
       row_which_min <- apply(x, 1, which.min)
       dist_to_sum_boundary <- apply(x, 1, function(xx){
                   (1 - sum(1:p * xx)) / sqrt(p*(p+1)*(2*p+1)/6)})
       grad_sum_boundary <- -(1:p) / sqrt(p*(p+1)*(2*p+1)/6)
       g0 <- pmin(row_min, dist_to_sum_boundary)
       g0d <- t(sapply(1:nrow(x), function(i){
          if (row_min[i] < dist_to_sum_boundary[i]){
             tmp <- numeric(ncol(x)); tmp[row_which_min[i]] <- 1
          } else {tmp <- grad_sum_boundary}
          tmp
       }))
       list("g0"=g0, "g0d"=g0d)
}
elts <- get_elts(NULL, x, "exp", domain, centered=TRUE, profiled_if_noncenter=FALSE,
       scale="", diag=dm, unif_dist=g0)

# log_log_sum0 model on the simplex with K having row and column sums 0 (Aitchison model)
domain <- make_domain("simplex", p=p)
K <- -cov_cons("band", p=p, spars=3, eig=1)
diag(K) <- diag(K) - rowSums(K) # So that rowSums(K) == colSums(K) == 0
eigen(K)$val[(p-1):p] # Make sure K has one 0 and p-1 positive eigenvalues
x <- gen(n, setting="log_log_sum0", abs=FALSE, eta=eta, K=K, domain=domain,
       xinit=NULL, seed=2, burn_in=1000, thinning=100, verbose=FALSE)
h_hp <- get_h_hp("min_pow", 2, 3)
h_hp_dx <- h_of_dist(h_hp, x, domain) # h and h' applied to distance from x to boundary

# Does not assume K has 0 row and column sums
elts_simplex_0 <- get_elts(h_hp, x, "log_log", domain, centered=TRUE, profiled=FALSE,
       scale="", diag=1.5)
```

```
# If want K to have row sums and column sums equal to 0 (Aitchison); estimate off-diagonals only
elts_simplex_1 <- get_elts(h_hp, x, "log_log_sum0", domain, centered=FALSE,
       profiled=FALSE, scale="", diag=1.5)
# All entries corresponding to the diagonals of K should be 0:
max(abs(sapply(1:p, function(j){c(elts_simplex_1$Gamma_K[j, (j-1)*p+1:p],
       elts_simplex_1$Gamma_K[, (j-1)*p+j])})))
max(abs(diag(elts_simplex_1$Gamma_K_eta)))
max(abs(diag(matrix(elts_simplex_1$g_K, nrow=p))))
```

`get_elts_ab` *The R implementation to get the elements necessary for calculations for general a and b.*

Description

The R implementation to get the elements necessary for calculations for general a and b.

Usage

```
get_elts_ab(
  hdx,
  hpdx,
  x,
  a,
  b,
  setting,
  centered = TRUE,
  profiled_if_noncenter = TRUE,
  scale = "",
  diagonal_multiplier = 1
)
```

Arguments

hdx	A matrix, $h(\mathbf{x})$ applied to the distance of x from the boundary of the domain, should be of the same dimension as x.
hpdx	A matrix, $h'(\mathbf{x})$ applied to the distance of x from the boundary of the domain, should be of the same dimension as x.
x	An n by p matrix, the data matrix, where n is the sample size and p the dimension.
a	A number, must be strictly larger than $b/2$.
b	A number, must be >= 0.
setting	A string that indicates the distribution type. Returned without being checked or used in the function body.

`centered` A boolean, whether in the centered setting (assume $\boldsymbol{\mu} = \boldsymbol{\eta} = 0$) or not. Default to `TRUE`.

`profiled_if_noncenter`
A boolean, whether in the profiled setting ($\lambda_{\boldsymbol{\eta}} = 0$) if non-centered. Parameter ignored if `centered=TRUE`. Default to `TRUE`.

`scale` A string indicating the scaling method. Returned without being checked or used in the function body. Default to `"norm"`.

`diagonal_multiplier`
A number >= 1, the diagonal multiplier.

Details

Computes the $\boldsymbol{\Gamma}$ matrix and the $\boldsymbol{g}$ vector for generalized score matching.

Here, $\boldsymbol{\Gamma}$ is block-diagonal, and in the non-profiled non-centered setting, the j-th block is composed of $\boldsymbol{\Gamma}_{\mathbf{KK},j}$, $\boldsymbol{\Gamma}_{\mathbf{K}\boldsymbol{\eta},j}$ and its transpose, and finally $\boldsymbol{\Gamma}_{\boldsymbol{\eta}\boldsymbol{\eta},j}$. In the centered case, only $\boldsymbol{\Gamma}_{\mathbf{KK},j}$ is computed. In the profiled non-centered case,

$$\boldsymbol{\Gamma}_j \equiv \boldsymbol{\Gamma}_{\mathbf{KK},j} - \boldsymbol{\Gamma}_{\mathbf{K}\boldsymbol{\eta},j}\boldsymbol{\Gamma}_{\boldsymbol{\eta}\boldsymbol{\eta},j}^{-1}\boldsymbol{\Gamma}_{\mathbf{K}\boldsymbol{\eta}}^{\top}.$$

Similarly, in the non-profiled non-centered setting, $\boldsymbol{g}$ can be partitioned p parts, each with a p-vector $\boldsymbol{g}_{\mathbf{K},j}$ and a scalar $g_{\boldsymbol{\eta},j}$. In the centered setting, only $\boldsymbol{g}_{\mathbf{K},j}$ is needed. In the profiled non-centered case,

$$\boldsymbol{g}_j \equiv \boldsymbol{g}_{\mathbf{K},j} - \boldsymbol{\Gamma}_{\mathbf{K}\boldsymbol{\eta},j}\boldsymbol{\Gamma}_{\boldsymbol{\eta}\boldsymbol{\eta},j}^{-1} g_{\boldsymbol{\eta},j}.$$

The formulae for the pieces above are

$$\boldsymbol{\Gamma}_{\mathbf{KK},j} \equiv \frac{1}{n}\sum_{i=1}^{n} h\left(X_j^{(i)}\right) {X_j^{(i)}}^{2a-2} {\boldsymbol{X}^{(i)}}^{a} {\boldsymbol{X}^{(i)}}^{a\top},$$

$$\boldsymbol{\Gamma}_{\mathbf{K}\boldsymbol{\eta},j} \equiv -\frac{1}{n}\sum_{i=1}^{n} h\left(X_j^{(i)}\right) {X_j^{(i)}}^{a+b-2} {\boldsymbol{X}^{(i)}}^{a},$$

$$\boldsymbol{\Gamma}_{\boldsymbol{\eta}\boldsymbol{\eta},j} \equiv \frac{1}{n}\sum_{i=1}^{n} h\left(X_j^{(i)}\right) {X_j^{(i)}}^{2b-2},$$

$$\boldsymbol{g}_{\mathbf{K},j} \equiv \frac{1}{n}\sum_{i=1}^{n} \left(h'\left(X_j^{(i)}\right) {X_j^{(i)}}^{a-1} + (a-1)h\left(X_j^{(i)}\right) {X_j^{(i)}}^{a-2}\right) {\boldsymbol{X}^{(i)}}^{a} + ah\left(X_j^{(i)}\right) {X_j^{(i)}}^{2a-2} \boldsymbol{e}_{j,p},$$

$$\boldsymbol{g}_{\boldsymbol{\eta},j} \equiv \frac{1}{n}\sum_{i=1}^{n} -h'\left(X_j^{(i)}\right) {X_j^{(i)}}^{b-1} - (b-1)h\left(X_j^{(i)}\right) {X_j^{(i)}}^{b-2},$$

where $\boldsymbol{e}_{j,p}$ is the p-vector with 1 at the j-th position and 0 elsewhere.

In the profiled non-centered setting, the function also returns t_1 and t_2 defined as

$$\boldsymbol{t}_1 \equiv \boldsymbol{\Gamma}_{\boldsymbol{\eta}\boldsymbol{\eta}}^{-1}\boldsymbol{g}_{\boldsymbol{\eta}}, \quad \boldsymbol{t}_2 \equiv \boldsymbol{\Gamma}_{\boldsymbol{\eta}\boldsymbol{\eta}}^{-1}\boldsymbol{\Gamma}_{\mathbf{K}\boldsymbol{\eta}}^{\top},$$

so that $\hat{\boldsymbol{\eta}} = \boldsymbol{t}_1 - \boldsymbol{t}_2 \mathrm{vec}(\hat{\mathbf{K}})$.

Value

A list that contains the elements necessary for estimation.

n	The sample size.
p	The dimension.
centered	The centered setting or not. Same as input.
scale	The scaling method. Same as input.
diagonal_multiplier	The diagonal multiplier. Same as input.
diagonals_with_multiplier	A vector that contains the diagonal entries of $\mathbf{\Gamma}$ after applying the multiplier.
setting	The setting. Same as input.
g_K	The $\boldsymbol{g}$ vector. In the non-profiled non-centered setting, this is the $\boldsymbol{g}$ sub-vector corresponding to $\mathbf{K}$.
Gamma_K	The $\mathbf{\Gamma}$ matrix with no diagonal multiplier. In the non-profiled non-centered setting, this is the $\mathbf{\Gamma}$ sub-matrix corresponding to $\mathbf{K}$.
g_eta	Returned in the non-profiled non-centered setting. The $\boldsymbol{g}$ sub-vector corresponding to $\boldsymbol{\eta}$.
Gamma_K_eta	Returned in the non-profiled non-centered setting. The $\mathbf{\Gamma}$ sub-matrix corresponding to interaction between $\mathbf{K}$ and $\boldsymbol{\eta}$.
Gamma_eta	Returned in the non-profiled non-centered setting. The $\mathbf{\Gamma}$ sub-matrix corresponding to $\boldsymbol{\eta}$.
t1,t2	Returned in the profiled non-centered setting, where the $\boldsymbol{\eta}$ estimate can be retrieved from $\boldsymbol{t_1} - \boldsymbol{t_2}\hat{\mathbf{K}}$ after appropriate resizing.

Examples

```
n <- 50
p <- 30
eta <- rep(0, p)
K <- diag(p)
domain <- make_domain("R+", p=p)
x <- gen(n, setting="ab_1/2_7/10", abs=FALSE, eta=eta, K=K, domain=domain, finite_infinity=100,
       xinit=NULL, seed=2, burn_in=1000, thinning=100, verbose=FALSE)
h_hp <- get_h_hp("min_pow", 1.5, 3)
h_hp_dx <- h_of_dist(h_hp, x, domain) # h and h' applied to distance from x to boundary
elts <- get_elts_ab(h_hp_dx$hdx, h_hp_dx$hpdx, x, a=0.5, b=0.7, setting="ab_1/2_7/10",
            centered=TRUE, scale="norm", diag=1.5)
elts <- get_elts_ab(h_hp_dx$hdx, h_hp_dx$hpdx, x, a=0.5, b=0.7, setting="ab_1/2_7/10",
            centered=FALSE, profiled_if_noncenter=TRUE, scale="norm", diag=1.7)
elts <- get_elts_ab(h_hp_dx$hdx, h_hp_dx$hpdx, x, a=0.5, b=0.7, setting="ab_1/2_7/10",
            centered=FALSE, profiled_if_noncenter=FALSE, scale="norm", diag=1.9)
```

`get_elts_exp` *The R implementation to get the elements necessary for calculations for the exponential square-root setting (a=0.5, b=0.5).*

Description

The R implementation to get the elements necessary for calculations for the exponential square-root setting (a=0.5, b=0.5).

Usage

```
get_elts_exp(
  hdx,
  hpdx,
  x,
  centered = TRUE,
  profiled_if_noncenter = TRUE,
  scale = "",
  diagonal_multiplier = 1
)
```

Arguments

`hdx`	A matrix, $h(\mathbf{x})$ applied to the distance of x from the boundary of the domain, should be of the same dimension as `x`.
`hpdx`	A matrix, $h'(\mathbf{x})$ applied to the distance of x from the boundary of the domain, should be of the same dimension as `x`.
`x`	An `n` by `p` matrix, the data matrix, where `n` is the sample size and `p` the dimension.
`centered`	A boolean, whether in the centered setting (assume $\boldsymbol{\mu} = \boldsymbol{\eta} = 0$) or not. Default to `TRUE`.
`profiled_if_noncenter`	A boolean, whether in the profiled setting ($\lambda_{\boldsymbol{\eta}} = 0$) if non-centered. Parameter ignored if centered=TRUE. Default to `TRUE`.
`scale`	A string indicating the scaling method. Returned without being checked or used in the function body. Default to `"norm"`.
`diagonal_multiplier`	A number >= 1, the diagonal multiplier.

Details

For details on the returned values, please refer to `get_elts_ab` or `get_elts`.

Value

A list that contains the elements necessary for estimation.

n	The sample size.
p	The dimension.
centered	The centered setting or not. Same as input.
scale	The scaling method. Same as input.
diagonal_multiplier	The diagonal multiplier. Same as input.
diagonals_with_multiplier	A vector that contains the diagonal entries of $\mathbf{\Gamma}$ after applying the multiplier.
setting	The setting "exp".
g_K	The $\boldsymbol{g}$ vector. In the non-profiled non-centered setting, this is the $\boldsymbol{g}$ sub-vector corresponding to $\mathbf{K}$.
Gamma_K	The $\mathbf{\Gamma}$ matrix with no diagonal multiplier. In the non-profiled non-centered setting, this is the $\mathbf{\Gamma}$ sub-matrix corresponding to $\mathbf{K}$.
g_eta	Returned in the non-profiled non-centered setting. The $\boldsymbol{g}$ sub-vector corresponding to $\boldsymbol{\eta}$.
Gamma_K_eta	Returned in the non-profiled non-centered setting. The $\mathbf{\Gamma}$ sub-matrix corresponding to interaction between $\mathbf{K}$ and $\boldsymbol{\eta}$.
Gamma_eta	Returned in the non-profiled non-centered setting. The $\mathbf{\Gamma}$ sub-matrix corresponding to $\boldsymbol{\eta}$.
t1,t2	Returned in the profiled non-centered setting, where the $\boldsymbol{\eta}$ estimate can be retrieved from $\boldsymbol{t_1} - \boldsymbol{t_2}\hat{\mathbf{K}}$ after appropriate resizing.

Examples

```
n <- 50
p <- 30
eta <- rep(0, p)
K <- diag(p)
domain <- make_domain("R+", p=p)
x <- gen(n, setting="exp", abs=FALSE, eta=eta, K=K, domain=domain, finite_infinity=100,
       xinit=NULL, seed=2, burn_in=1000, thinning=100, verbose=FALSE)
h_hp <- get_h_hp("min_pow", 1, 3)
h_hp_dx <- h_of_dist(h_hp, x, domain) # h and h' applied to distance from x to boundary
elts <- get_elts_exp(h_hp_dx$hdx, h_hp_dx$hpdx, x, centered=TRUE, scale="norm", diag=1.5)
elts <- get_elts_exp(h_hp_dx$hdx, h_hp_dx$hpdx, x, centered=FALSE, profiled_if_noncenter=TRUE,
      scale="norm", diag=1.7)
elts <- get_elts_exp(h_hp_dx$hdx, h_hp_dx$hpdx, x, centered=FALSE, profiled_if_noncenter=FALSE,
      scale="norm", diag=1.7)
```

`get_elts_gamma` *The R implementation to get the elements necessary for calculations for the gamma setting (a=0.5, b=0).*

Description

The R implementation to get the elements necessary for calculations for the gamma setting (a=0.5, b=0).

Usage

```
get_elts_gamma(
  hdx,
  hpdx,
  x,
  centered = TRUE,
  profiled_if_noncenter = TRUE,
  scale = "",
  diagonal_multiplier = 1
)
```

Arguments

`hdx`	A matrix, $h(\mathbf{x})$ applied to the distance of x from the boundary of the domain, should be of the same dimension as `x`.
`hpdx`	A matrix, $h'(\mathbf{x})$ applied to the distance of x from the boundary of the domain, should be of the same dimension as `x`.
`x`	An `n` by `p` matrix, the data matrix, where `n` is the sample size and `p` the dimension.
`centered`	A boolean, whether in the centered setting (assume $\boldsymbol{\mu} = \boldsymbol{\eta} = 0$) or not. Default to `TRUE`.
`profiled_if_noncenter`	A boolean, whether in the profiled setting ($\lambda_{\boldsymbol{\eta}} = 0$) if non-centered. Parameter ignored if centered=TRUE. Default to `TRUE`.
`scale`	A string indicating the scaling method. Returned without being checked or used in the function body. Default to `"norm"`.
`diagonal_multiplier`	A number >= 1, the diagonal multiplier.

Details

For details on the returned values, please refer to `get_elts_ab` or `get_elts`.

Value

A list that contains the elements necessary for estimation.

`n`	The sample size.
`p`	The dimension.
`centered`	The centered setting or not. Same as input.
`scale`	The scaling method. Same as input.
`diagonal_multiplier`	The diagonal multiplier. Same as input.
`diagonals_with_multiplier`	A vector that contains the diagonal entries of $\mathbf{\Gamma}$ after applying the multiplier.
`setting`	The setting `"gamma"`.
`g_K`	The $\boldsymbol{g}$ vector. In the non-profiled non-centered setting, this is the $\boldsymbol{g}$ sub-vector corresponding to $\mathbf{K}$.
`Gamma_K`	The $\mathbf{\Gamma}$ matrix with no diagonal multiplier. In the non-profiled non-centered setting, this is the $\mathbf{\Gamma}$ sub-matrix corresponding to $\mathbf{K}$.
`g_eta`	Returned in the non-profiled non-centered setting. The $\boldsymbol{g}$ sub-vector corresponding to $\boldsymbol{\eta}$.
`Gamma_K_eta`	Returned in the non-profiled non-centered setting. The $\mathbf{\Gamma}$ sub-matrix corresponding to interaction between $\mathbf{K}$ and $\boldsymbol{\eta}$.
`Gamma_eta`	Returned in the non-profiled non-centered setting. The $\mathbf{\Gamma}$ sub-matrix corresponding to $\boldsymbol{\eta}$.
`t1,t2`	Returned in the profiled non-centered setting, where the $\boldsymbol{\eta}$ estimate can be retrieved from $\boldsymbol{t_1} - \boldsymbol{t_2}\hat{\mathbf{K}}$ after appropriate resizing.

Examples

```
n <- 50
p <- 30
eta <- rep(0, p)
K <- diag(p)
domain <- make_domain("R+", p=p)
x <- gen(n, setting="gamma", abs=FALSE, eta=eta, K=K, domain=domain, finite_infinity=100,
       xinit=NULL, seed=2, burn_in=1000, thinning=100, verbose=FALSE)
h_hp <- get_h_hp("min_pow", 1.5, 3)
h_hp_dx <- h_of_dist(h_hp, x, domain) # h and h' applied to distance from x to boundary
elts <- get_elts_gamma(h_hp_dx$hdx, h_hp_dx$hpdx, x, centered=TRUE, scale="norm", diag=1.5)
elts <- get_elts_gamma(h_hp_dx$hdx, h_hp_dx$hpdx, x, centered=FALSE, profiled_if_noncenter=TRUE,
       scale="norm", diag=1.7)
elts <- get_elts_gamma(h_hp_dx$hdx, h_hp_dx$hpdx, x, centered=FALSE, profiled_if_noncenter=FALSE,
       scale="norm", diag=1.9)
```

`get_elts_gauss` *The R implementation to get the elements necessary for calculations for the gaussian setting on R^p.*

Description

The R implementation to get the elements necessary for calculations for the gaussian setting on R^p.

Usage

```
get_elts_gauss(
  x,
  centered = TRUE,
  profiled_if_noncenter = TRUE,
  scale = "",
  diagonal_multiplier = 1
)
```

Arguments

`x`	An `n` by `p` matrix, the data matrix, where `n` is the sample size and `p` the dimension.
`centered`	A boolean, whether in the centered setting (assume $\boldsymbol{\mu} = \boldsymbol{\eta} = 0$) or not. Default to `TRUE`.
`profiled_if_noncenter`	A boolean, whether in the profiled setting ($\lambda_{\boldsymbol{\eta}} = 0$) if non-centered. Parameter ignored if `centered==TRUE`. Default to `TRUE`.
`scale`	A string indicating the scaling method. Returned without being checked or used in the function body. Default to `"norm"`.
`diagonal_multiplier`	A number >= 1, the diagonal multiplier.

Details

For details on the returned values, please refer to `get_elts_ab` or `get_elts`.

Value

A list that contains the elements necessary for estimation.

`n`	The sample size.
`p`	The dimension.
`centered`	The centered setting or not. Same as input.
`scale`	The scaling method. Same as input.

`diagonal_multiplier`
: The diagonal multiplier. Same as input.

`diagonals_with_multiplier`
: A vector that contains the diagonal entries of $\mathbf{\Gamma}$ after applying the multiplier.

`setting`
: The setting `"gaussian"`.

`Gamma_K`
: The $\mathbf{\Gamma}$ matrix with no diagonal multiplier. In the non-profiled non-centered setting, this is the $\mathbf{\Gamma}$ sub-matrix corresponding to $\mathbf{K}$. Except for the *profiled* setting, this is $\mathbf{x}\mathbf{x}^\top/n$.

`Gamma_K_eta`
: Returned in the non-profiled non-centered setting. The $\mathbf{\Gamma}$ sub-matrix corresponding to interaction between $\mathbf{K}$ and $\boldsymbol{\eta}$. The minus column means of `x`.

`t1,t2`
: Returned in the profiled non-centered setting, where the$\boldsymbol{\eta}$ estimate can be retrieved from $\boldsymbol{t_1} - \boldsymbol{t_2}\hat{\mathbf{K}}$ after appropriate resizing.

Examples

```
n <- 50
p <- 30
mu <- rep(0, p)
K <- diag(p)
x <- mvtnorm::rmvnorm(n, mean=mu, sigma=solve(K))
# Equivalently:

x2 <- gen(n, setting="gaussian", abs=FALSE, eta=c(K%*%mu), K=K, domain=make_domain("R",p),
       finite_infinity=100, xinit=NULL, burn_in=1000, thinning=100, verbose=FALSE)

elts <- get_elts_gauss(x, centered=TRUE, scale="norm", diag=1.5)
elts <- get_elts_gauss(x, centered=FALSE, profiled=FALSE, scale="sd", diag=1.9)
```

`get_elts_loglog` *The R implementation to get the elements necessary for calculations for the log-log setting (a=0, b=0).*

Description

The R implementation to get the elements necessary for calculations for the log-log setting (a=0, b=0).

Usage

```
get_elts_loglog(
  hdx,
  hpdx,
  x,
  setting,
  centered = TRUE,
  profiled_if_noncenter = TRUE,
  scale = "",
  diagonal_multiplier = 1
)
```

Arguments

hdx	A matrix, $h(\mathbf{x})$ applied to the distance of x from the boundary of the domain, should be of the same dimension as x.
hpdx	A matrix, $h'(\mathbf{x})$ applied to the distance of x from the boundary of the domain, should be of the same dimension as x.
x	An n by p matrix, the data matrix, where n is the sample size and p the dimension.
setting	A string, log_log.
centered	A boolean, whether in the centered setting (assume $\boldsymbol{\mu} = \boldsymbol{\eta} = 0$) or not. Default to TRUE.
profiled_if_noncenter	A boolean, whether in the profiled setting ($\lambda_{\boldsymbol{\eta}} = 0$) if non-centered. Parameter ignored if centered=TRUE. Default to TRUE.
scale	A string indicating the scaling method. Returned without being checked or used in the function body. Default to "norm".
diagonal_multiplier	A number >= 1, the diagonal multiplier.

Details

For details on the returned values, please refer to get_elts_ab or get_elts.

Value

A list that contains the elements necessary for estimation.

n	The sample size.
p	The dimension.
centered	The centered setting or not. Same as input.
scale	The scaling method. Same as input.
diagonal_multiplier	The diagonal multiplier. Same as input.
diagonals_with_multiplier	A vector that contains the diagonal entries of $\boldsymbol{\Gamma}$ after applying the multiplier.
setting	The same setting as in the function argument.
g_K	The $\boldsymbol{g}$ vector. In the non-profiled non-centered setting, this is the $\boldsymbol{g}$ sub-vector corresponding to $\mathbf{K}$.
Gamma_K	The $\boldsymbol{\Gamma}$ matrix with no diagonal multiplier. In the non-profiled non-centered setting, this is the $\boldsymbol{\Gamma}$ sub-matrix corresponding to $\mathbf{K}$.
g_eta	Returned in the non-profiled non-centered setting. The $\boldsymbol{g}$ sub-vector corresponding to $\boldsymbol{\eta}$.
Gamma_K_eta	Returned in the non-profiled non-centered setting. The $\boldsymbol{\Gamma}$ sub-matrix corresponding to interaction between $\mathbf{K}$ and $\boldsymbol{\eta}$.

`Gamma_eta` Returned in the non-profiled non-centered setting. The $\mathbf{\Gamma}$ sub-matrix corresponding to $\boldsymbol{\eta}$.

`t1,t2` Returned in the profiled non-centered setting, where the $\boldsymbol{\eta}$ estimate can be retrieved from $\boldsymbol{t_1} - \boldsymbol{t_2}\hat{\mathbf{K}}$ after appropriate resizing.

Examples

```
n <- 50
p <- 30
eta <- rep(0, p)
K <- diag(p)
domain <- make_domain("uniform", p=p, lefts=c(0), rights=c(1))
x <- gen(n, setting="log_log", abs=FALSE, eta=eta, K=K, domain=domain,
       xinit=NULL, seed=2, burn_in=1000, thinning=100, verbose=FALSE)
h_hp <- get_h_hp("min_pow", 1.5, 3)
h_hp_dx <- h_of_dist(h_hp, x, domain) # h and h' applied to distance from x to boundary
elts <- get_elts_loglog(h_hp_dx$hdx, h_hp_dx$hpdx, x, setting="log_log", centered=TRUE,
       scale="", diag=1.5)
elts <- get_elts_loglog(h_hp_dx$hdx, h_hp_dx$hpdx, x, setting="log_log", centered=FALSE,
       profiled_if_noncenter=TRUE, scale="", diag=1.7)
elts <- get_elts_loglog(h_hp_dx$hdx, h_hp_dx$hpdx, x, setting="log_log", centered=FALSE,
       profiled_if_noncenter=FALSE, scale="", diag=1.9)
```

---

get_elts_loglog_simplex

The R implementation to get the elements necessary for calculations for the log-log setting (a=0, b=0) on the p-simplex.

Description

The R implementation to get the elements necessary for calculations for the log-log setting (a=0, b=0) on the p-simplex.

Usage

```
get_elts_loglog_simplex(
  hdx,
  hpdx,
  x,
  setting,
  centered = TRUE,
  profiled_if_noncenter = TRUE,
  scale = "",
  diagonal_multiplier = 1
)
```

Arguments

hdx	A matrix, $h(\mathbf{x})$ applied to the distance of x from the boundary of the domain, should be of the same dimension as x.
hpdx	A matrix, $h'(\mathbf{x})$ applied to the distance of x from the boundary of the domain, should be of the same dimension as x.
x	An n by p matrix, the data matrix, where n is the sample size and p the dimension.
setting	A string, log_log or log_log_sum0. If log_log_sum0, assumes that the true K has row and column sums 0 (see the A^d model), so only the off-diagonal entries will be estimated; the diagonal entries will be profiled out in the loss), so elements corresponding to the diagonals of K will be set to 0, and the loss will be rewritten in the off-diagonal entries only.
centered	A boolean, whether in the centered setting (assume $\boldsymbol{\mu} = \boldsymbol{\eta} = 0$) or not. Default to TRUE.
profiled_if_noncenter	A boolean, whether in the profiled setting ($\lambda_{\boldsymbol{\eta}} = 0$) if non-centered. Parameter ignored if centered=TRUE. Default to TRUE.
scale	A string indicating the scaling method. Returned without being checked or used in the function body. Default to "norm".
diagonal_multiplier	A number >= 1, the diagonal multiplier.

Details

For details on the returned values, please refer to get_elts_ab or get_elts.

Value

A list that contains the elements necessary for estimation.

n	The sample size.
p	The dimension.
centered	The centered setting or not. Same as input.
scale	The scaling method. Same as input.
diagonal_multiplier	The diagonal multiplier. Same as input.
diagonals_with_multiplier	A vector that contains the diagonal entries of $\boldsymbol{\Gamma}$ after applying the multiplier.
setting	The same setting as in the function argument.
g_K	The $\boldsymbol{g}$ vector. In the non-profiled non-centered setting, this is the $\boldsymbol{g}$ sub-vector corresponding to $\mathbf{K}$.
Gamma_K	The $\boldsymbol{\Gamma}$ matrix with no diagonal multiplier. In the non-profiled non-centered setting, this is the $\boldsymbol{\Gamma}$ sub-matrix corresponding to $\mathbf{K}$.
g_eta	Returned in the non-profiled non-centered setting. The $\boldsymbol{g}$ sub-vector corresponding to $\boldsymbol{\eta}$.

Gamma_K_eta	Returned in the non-profiled non-centered setting. The $\mathbf{\Gamma}$ sub-matrix corresponding to interaction between $\mathbf{K}$ and $\boldsymbol{\eta}$.
Gamma_eta	Returned in the non-profiled non-centered setting. The $\mathbf{\Gamma}$ sub-matrix corresponding to $\boldsymbol{\eta}$.
t1,t2	Returned in the profiled non-centered setting, where the $\boldsymbol{\eta}$ estimate can be retrieved from $\boldsymbol{t_1} - \boldsymbol{t_2}\hat{\mathbf{K}}$ after appropriate resizing.

Examples

```
n <- 50
p <- 30
eta <- rep(0, p)
K <- -cov_cons("band", p=p, spars=3, eig=1)
diag(K) <- diag(K) - rowSums(K) # So that rowSums(K) == colSums(K) == 0
eigen(K)$val[(p-1):p] # Make sure K has one 0 and p-1 positive eigenvalues
domain <- make_domain("simplex", p=p)
x <- gen(n, setting="log_log_sum0", abs=FALSE, eta=eta, K=K, domain=domain,
       xinit=NULL, seed=2, burn_in=1000, thinning=100, verbose=FALSE)
h_hp <- get_h_hp("min_pow", 2, 3)
h_hp_dx <- h_of_dist(h_hp, x, domain) # h and h' applied to distance from x to boundary

elts_simplex_0 <- get_elts_loglog_simplex(h_hp_dx$hdx, h_hp_dx$hpdx, x,
       setting="log_log", centered=FALSE, profiled=FALSE, scale="", diag=1.5)

# If want K to have row sums and column sums equal to 0; estimate off-diagonals only
elts_simplex_1 <- get_elts_loglog_simplex(h_hp_dx$hdx, h_hp_dx$hpdx, x,
       setting="log_log_sum0", centered=FALSE, profiled=FALSE, scale="", diag=1.5)
# All entries corresponding to the diagonals of K should be 0:
max(abs(sapply(1:p, function(j){c(elts_simplex_1$Gamma_K[j, (j-1)*p+1:p],
       elts_simplex_1$Gamma_K[, (j-1)*p+j])})))
max(abs(diag(elts_simplex_1$Gamma_K_eta)))
max(abs(diag(matrix(elts_simplex_1$g_K, nrow=p))))
```

---

get_elts_trun_gauss *The R implementation to get the elements necessary for calculations for the gaussian setting (a=1, b=1) on domains other than R^p.*

Description

The R implementation to get the elements necessary for calculations for the gaussian setting (a=1, b=1) on domains other than R^p.

Usage

```
get_elts_trun_gauss(
  hdx,
  hpdx,
  x,
```

```
  centered = TRUE,
  profiled_if_noncenter = TRUE,
  scale = "",
  diagonal_multiplier = 1
)
```

Arguments

hdx	A matrix, $h(\mathbf{x})$ applied to the distance of x from the boundary of the domain, should be of the same dimension as x.
hpdx	A matrix, $h'(\mathbf{x})$ applied to the distance of x from the boundary of the domain, should be of the same dimension as x.
x	An n by p matrix, the data matrix, where n is the sample size and p the dimension.
centered	A boolean, whether in the centered setting (assume $\boldsymbol{\mu} = \boldsymbol{\eta} = 0$) or not. Default to TRUE.
profiled_if_noncenter	A boolean, whether in the profiled setting ($\lambda_{\boldsymbol{\eta}} = 0$) if non-centered. Parameter ignored if centered=TRUE. Default to TRUE.
scale	A string indicating the scaling method. Returned without being checked or used in the function body. Default to "norm".
diagonal_multiplier	A number >= 1, the diagonal multiplier.

Details

For details on the returned values, please refer to get_elts_ab or get_elts.

Value

A list that contains the elements necessary for estimation.

n	The sample size.
p	The dimension.
centered	The centered setting or not. Same as input.
scale	The scaling method. Same as input.
diagonal_multiplier	The diagonal multiplier. Same as input.
diagonals_with_multiplier	A vector that contains the diagonal entries of $\boldsymbol{\Gamma}$ after applying the multiplier.
setting	The setting "gaussian".
g_K	The $\boldsymbol{g}$ vector. In the non-profiled non-centered setting, this is the $\boldsymbol{g}$ sub-vector corresponding to $\mathbf{K}$.
Gamma_K	The $\boldsymbol{\Gamma}$ matrix with no diagonal multiplier. In the non-profiled non-centered setting, this is the $\boldsymbol{\Gamma}$ sub-matrix corresponding to $\mathbf{K}$.

g_eta	Returned in the non-profiled non-centered setting. The $\boldsymbol{g}$ sub-vector corresponding to $\boldsymbol{\eta}$.
Gamma_K_eta	Returned in the non-profiled non-centered setting. The $\boldsymbol{\Gamma}$ sub-matrix corresponding to interaction between $\mathbf{K}$ and $\boldsymbol{\eta}$.
Gamma_eta	Returned in the non-profiled non-centered setting. The $\boldsymbol{\Gamma}$ sub-matrix corresponding to $\boldsymbol{\eta}$.
t1,t2	Returned in the profiled non-centered setting, where the $\boldsymbol{\eta}$ estimate can be retrieved from $\boldsymbol{t_1} - \boldsymbol{t_2}\hat{\mathbf{K}}$ after appropriate resizing.

Examples

```
n <- 50
p <- 30
mu <- rep(0, p)
K <- diag(p)
eta <- K %*% mu
domain <- make_domain("R+", p=p)
x <- gen(n, setting="gaussian", abs=FALSE, eta=eta, K=K, domain=domain, finite_infinity=100,
       xinit=NULL, seed=2, burn_in=1000, thinning=100, verbose=FALSE)
h_hp <- get_h_hp("min_pow", 1, 3)
h_hp_dx <- h_of_dist(h_hp, x, domain) # h and h' applied to distance from x to boundary
elts <- get_elts_trun_gauss(h_hp_dx$hdx, h_hp_dx$hpdx, x, centered=TRUE, scale="norm", diag=1.5)
elts <- get_elts_trun_gauss(h_hp_dx$hdx, h_hp_dx$hpdx, x, centered=FALSE,
       profiled_if_noncenter=TRUE, scale="norm", diag=1.7)
elts <- get_elts_trun_gauss(h_hp_dx$hdx, h_hp_dx$hpdx, x, centered=FALSE,
       profiled_if_noncenter=FALSE, scale="norm", diag=1.9)
```

---

get_g0 *Calculates the l2 distance to the boundary of the domain and its gradient for some domains.*

Description

Calculates the l2 distance to the boundary of the domain and its gradient for some domains.

Usage

```
get_g0(domain, C)
```

Arguments

domain	A list returned from `make_domain()` that represents the domain.
C	A positive number, cannot be `Inf` if `domain$type == "R"`. If not `Inf`, the l2 distance will be truncated to `C`, i.e. the function returns `pmin(g0(x), C)` and its gradient.

Details

Calculates the l2 distance to the boundary of the domain, with the distance truncated above by a constant `C`. Matches the `g0` function and its gradient from Liu (2019) if `C == Inf` and domain is bounded. Currently only R, R+, simplex, uniform and polynomial-type domains of the form sum(x^2) <= d or sum(x^2) >= d or sum(abs(x)) <= d are implemented.

Value

A function that takes `x` and returns a list of a vector `g0` and a matrix `g0d`.

Examples

```
n <- 15
p <- 5
K <- diag(p)
eta <- numeric(p)

domain <- make_domain("R", p=p)
x <- gen(n, "gaussian", FALSE, eta, K, domain, 100)
get_g0(domain, 1)(x)

domain <- make_domain("R+", p=p)
x <- gen(n, "gaussian", FALSE, eta, K, domain, 100)
get_g0(domain, 1)(x)

domain <- make_domain("uniform", p=p, lefts=c(-Inf,-3,3), rights=c(-5,1,Inf))
x <- gen(n, "gaussian", FALSE, eta, K, domain, 100)
get_g0(domain, 1)(x)

domain <- make_domain("simplex", p=p)
x <- gen(n, "gaussian", FALSE, eta, K, domain, 100)
max(abs(get_g0(domain, 1)(x)$g0 - get_g0(domain, 1)(x[,-p])$g0))
max(abs(get_g0(domain, 1)(x)$g0d - get_g0(domain, 1)(x[,-p])$g0d))

domain <- make_domain("polynomial", p=p, ineqs=
     list(list("expression"="sum(x^2)>1.3", "nonnegative"=FALSE, "abs"=FALSE)))
x <- gen(n, "gaussian", FALSE, eta, K, domain, 100)
get_g0(domain, 1)(x)

domain <- make_domain("polynomial", p=p, ineqs=
     list(list("expression"="sum(x^2)>1.3", "nonnegative"=TRUE, "abs"=FALSE)))
x <- gen(n, "gaussian", FALSE, eta, K, domain, 100)
get_g0(domain, 1)(x)

domain <- make_domain("polynomial", p=p, ineqs=
     list(list("expression"="sum(x^2)<1.3", "nonnegative"=FALSE, "abs"=FALSE)))
x <- gen(n, "gaussian", FALSE, eta, K, domain, 100)
get_g0(domain, 1)(x)

domain <- make_domain("polynomial", p=p, ineqs=
     list(list("expression"="sum(x^2)<1.3", "nonnegative"=TRUE, "abs"=FALSE)))
x <- gen(n, "gaussian", FALSE, eta, K, domain, 100)
```

```
get_g0(domain, 1)(x)

domain <- make_domain("polynomial", p=p, ineqs=
     list(list("expression"="sum(x)<1.3", "nonnegative"=FALSE, "abs"=TRUE)))
x <- gen(n, "gaussian", FALSE, eta, K, domain, 100)
get_g0(domain, 1)(x)

domain <- make_domain("polynomial", p=p, ineqs=
     list(list("expression"="sum(x)<1.3", "nonnegative"=TRUE, "abs"=TRUE)))
x <- gen(n, "gaussian", FALSE, eta, K, domain, 100)
get_g0(domain, 1)(x)
```

`get_g0_ada` *Adaptively truncates the l2 distance to the boundary of the domain and its gradient for some domains.*

Description

Adaptively truncates the l2 distance to the boundary of the domain and its gradient for some domains.

Usage

```
get_g0_ada(domain, percentile)
```

Arguments

`domain`	A list returned from `make_domain()` that represents the domain.
`percentile`	A number between 0 and 1, the percentile. The returned l2 distance will be truncated to its `percentile`-th quantile, i.e. the function returns `pmin(g0(x), stats::quantile(g0(x), percentile))` and its gradient. The quantile is calculated using finite values only, and if no finite values exist the quantile is set to 1.

Details

Calculates the l2 distance to the boundary of the domain, with the distance truncated above at a specified quantile. Matches the `g0` function and its gradient from Liu (2019) if `percentile == 1` and domain is bounded. Currently only R, R+, simplex, uniform and polynomial-type domains of the form sum(x^2) <= d or sum(x^2) >= d or sum(abs(x)) <= d are implemented.

Value

A function that takes `x` and returns a list of a vector `g0` and a matrix `g0d`.

Examples

```
n <- 15
p <- 5
K <- diag(p)
eta <- numeric(p)

domain <- make_domain("R", p=p)
x <- gen(n, "gaussian", FALSE, eta, K, domain, 100)
get_g0_ada(domain, 0.3)(x)

domain <- make_domain("R+", p=p)
x <- gen(n, "gaussian", FALSE, eta, K, domain, 100)
get_g0_ada(domain, 0.3)(x)

domain <- make_domain("uniform", p=p, lefts=c(-Inf,-3,3), rights=c(-5,1,Inf))
x <- gen(n, "gaussian", FALSE, eta, K, domain, 100)
get_g0_ada(domain, 0.6)(x)

domain <- make_domain("simplex", p=p)
x <- gen(n, "gaussian", FALSE, eta, K, domain, 100)
max(abs(get_g0_ada(domain, 0.4)(x)$g0 - get_g0_ada(domain, 0.4)(x[,-p])$g0))
max(abs(get_g0_ada(domain, 0.4)(x)$g0d - get_g0_ada(domain, 0.4)(x[,-p])$g0d))

domain <- make_domain("polynomial", p=p, ineqs=
     list(list("expression"="sum(x^2)>1.3", "nonnegative"=FALSE, "abs"=FALSE)))
x <- gen(n, "gaussian", FALSE, eta, K, domain, 100)
get_g0_ada(domain, 0.5)(x)

domain <- make_domain("polynomial", p=p, ineqs=
     list(list("expression"="sum(x^2)>1.3", "nonnegative"=TRUE, "abs"=FALSE)))
x <- gen(n, "gaussian", FALSE, eta, K, domain, 100)
get_g0_ada(domain, 0.7)(x)

domain <- make_domain("polynomial", p=p, ineqs=
     list(list("expression"="sum(x^2)<1.3", "nonnegative"=FALSE, "abs"=FALSE)))
x <- gen(n, "gaussian", FALSE, eta, K, domain, 100)
get_g0_ada(domain, 0.6)(x)

domain <- make_domain("polynomial", p=p, ineqs=
     list(list("expression"="sum(x^2)<1.3", "nonnegative"=TRUE, "abs"=FALSE)))
x <- gen(n, "gaussian", FALSE, eta, K, domain, 100)
get_g0_ada(domain, 0.25)(x)

domain <- make_domain("polynomial", p=p, ineqs=
     list(list("expression"="sum(x)<1.3", "nonnegative"=FALSE, "abs"=TRUE)))
x <- gen(n, "gaussian", FALSE, eta, K, domain, 100)
get_g0_ada(domain, 0.37)(x)

domain <- make_domain("polynomial", p=p, ineqs=
     list(list("expression"="sum(x)<1.3", "nonnegative"=TRUE, "abs"=TRUE)))
x <- gen(n, "gaussian", FALSE, eta, K, domain, 100)
get_g0_ada(domain, 0.45)(x)
```

get_h_hp *Generator of h and hp (derivative of h) functions.*

Description

Generator of h and hp (derivative of h) functions.

Usage

```
get_h_hp(mode, para = NULL, para2 = NULL)
```

Arguments

mode	A string, see details.
para	May be optional. A number, the first parameter. Default to NULL.
para2	May be optional. A number, the second parameter. If mode is one of the adaptive mode below, this specifies the percentile (see details). Default to NULL.

Details

The mode parameter can be chosen from the options listed below along with the corresponding definitions of h under appropriate choices of para and para2 parameters. Unless otherwise noted, para and para2, must both be strictly positive if provided, and are set to 1 if not provided. Functions h and hp should only be applied to non-negative values x and this is not enforced or checked by the functions. Internally calls get_h_hp_vector.

asinh An asinh function $\boldsymbol{h}(\boldsymbol{x}) = \mathrm{asinh}(\mathrm{para} \cdot \boldsymbol{x}) = \log\left(\mathrm{para} \cdot \boldsymbol{x} + \sqrt{(\mathrm{para} \cdot \boldsymbol{x})^2 + 1}\right)$. Unbounded and takes one parameter. Equivalent to min_asinh(x, para, Inf).

cosh A shifted cosh function $\boldsymbol{h}(\boldsymbol{x}) = \cosh(\mathrm{para} \cdot \boldsymbol{x}) - 1$. Unbounded and takes one parameter. Equivalent to min_cosh(x, para, Inf).

exp A shifted exponential function $\boldsymbol{h}(\boldsymbol{x}) = \exp(\mathrm{para} \cdot \boldsymbol{x}) - 1$. Unbounded and takes one parameter. Equivalent to min_exp(x, para, Inf).

identity The identity function $\boldsymbol{h}(\boldsymbol{x}) = \boldsymbol{x}$. Unbounded and does not take any parameter. Equivalent to pow(x, 1) or min_pow(x, 1, Inf).

log_pow A power function on a log scale $\boldsymbol{h}(\boldsymbol{x}) = \log(1 + \boldsymbol{x})^{\mathrm{para}}$. Unbounded and takes one parameter. Equivalent to min_log_pow(x, para, Inf).

mcp Treating λ=para, γ=para2, the step-wise MCP function applied element-wise: $\lambda x - x^2/(2\gamma)$ if $x \leq \lambda\gamma$, or $\gamma\lambda^2/2$ otherwise. Bounded and takes two parameters.

min_asinh A truncated asinh function applied element-wise: $\min(\mathrm{asinh}(\mathrm{para} \cdot \boldsymbol{x}), \mathrm{para}_2)$. Bounded and takes two parameters.

min_asinh_ada Adaptive version of min_asinh.

`min_cosh` A truncated shifted cosh function applied element-wise: $\min(\cosh(\text{para}\cdot\boldsymbol{x})-1, \text{para}_2)$. Bounded and takes two parameters.

`min_cosh_ada` Adaptive version of `min_cosh`.

`min_exp` A truncated shifted exponential function applied element-wise: $\boldsymbol{h}(\boldsymbol{x}) = \min(\exp(\text{para}\cdot\boldsymbol{x})-1, \text{para}_2)$. Bounded and takes two parameters.

`min_exp_ada` Adaptive version of `min_exp`.

`min_log_pow` A truncated power on a log scale applied element-wise: $\boldsymbol{h}(\boldsymbol{x}) = \min(\log(1+\boldsymbol{x}), \text{para}_2)^{\text{para}}$. Bounded and takes two parameters.

`min_log_pow_ada` Adaptive version of `min_log_pow`.

`min_pow` A truncated power function applied element-wise: $\boldsymbol{h}(\boldsymbol{x}) = \min(\boldsymbol{x}, \text{para}_2)^{\text{para}}$. Bounded and takes two parameters.

`min_pow_ada` Adaptive version of `min_pow`.

`min_sinh` A truncated sinh function applied element-wise: $\min(\sinh(\text{para}\cdot\boldsymbol{x}), \text{para}_2)$. Bounded and takes two parameters.

`min_sinh_ada` Adaptive version of `min_sinh`.

`min_softplus` A truncated shifted softplus function applied element-wise: $\min(\log(1+\exp(\text{para}\cdot\boldsymbol{x}))-\log(2), \text{para}_2)$. Bounded and takes two parameters.

`min_softplus_ada` Adaptive version of `min_softplus`.

`pow` A power function $\boldsymbol{h}(\boldsymbol{x}) = \boldsymbol{x}^{\text{para}}$. Unbounded and takes two parameter. Equivalent to `min_pow(x, para, Inf)`.

`scad` Treating λ=para, γ=para2, the step-wise SCAD function applied element-wise: λx if $x \le \lambda$, or $(2\gamma\lambda x - x^2 - \lambda^2)/(2(\gamma-1))$ if $\lambda < x < \gamma\lambda$, or $\lambda^2(\gamma+1)/2$ otherwise. Bounded and takes two parameters, where `para2` must be larger than 1, and will be set to 2 by default if not provided.

`sinh` A sinh function $\boldsymbol{h}(\boldsymbol{x}) = \sinh(\text{para}\cdot\boldsymbol{x})$. Unbounded and takes one parameter. Equivalent to `min_sinh(x, para, Inf)`.

`softplus` A shifted softplus function $\boldsymbol{h}(\boldsymbol{x}) = \log(1+\exp(\text{para}\cdot\boldsymbol{x}))-\log(2)$. Unbounded and takes one parameter. Equivalent to `min_softplus(x, para, Inf)`.

`tanh` A tanh function $\boldsymbol{h}(\boldsymbol{x}) = \tanh(\text{para}\cdot\boldsymbol{x})$. Bounded and takes one parameter.

`truncated_sin` A truncated sin function applied element-wise: $\sin(\text{para}\cdot x)$ if $\text{para}\cdot x \le \pi/2$, or 1 otherwise. Bounded and takes one parameter.

`truncated_tan` A truncated tan function applied element-wise: $\tan(\text{para}\cdot x)$ if $\text{para}\cdot x \le \pi/4$, or 1 otherwise. Bounded and takes one parameter.

For the adaptive modes (names ending with `"_ada"`), `h` and `hp` are first applied to `x` without truncation. Then inside each column, values that are larger than the `para2`-th quantile will be truncated. The quantile is calculated using finite values only, and if no finite values exist the quantile is set to 1. For example, if `mode == "min_pow_ada"`, `para == 2`, `para2 == 0.4`, the `j`-th column of the returned `hx` will be `pmin(x[,j]^2, stats::quantile(x[,j]^2, 0.4))`, and the `j`-th column of `hpx` will be `2*x[,j]*(x[,j] <= stats::quantile(x[,j]^2, 0.4))`.

Value

A function that returns a list containing `hx=h(x)` (element-wise) and `hpx=hp(x)` (element-wise derivative of h) when applied to a vector (for mode names not ending with `"_ada"` only) or a matrix `x`, with both of the results having the same shape as `x`.

Examples

```
get_h_hp("mcp", 2, 4)(0:10)
get_h_hp("min_log_pow", 1, log(1+3))(matrix(0:11, nrow=3))
get_h_hp("min_pow", 1.5, 3)(seq(0, 5, by=0.5))
get_h_hp("min_softplus")(matrix(seq(0, 2, by=0.1), nrow=7))

get_h_hp("min_log_pow_ada", 1, 0.4)(matrix(0:49, nrow=10))
get_h_hp("min_pow_ada", 2, 0.3)(matrix(0:49, nrow=10))
get_h_hp("min_softplus_ada", 2, 0.6)(matrix(seq(0, 0.49, by=0.01), nrow=10))
```

---

get_h_hp_adaptive *Generator of adaptive h and hp (derivative of h) functions.*

Description

Generator of adaptive h and hp (derivative of h) functions.

Usage

```
get_h_hp_adaptive(mode, para, percentile)
```

Arguments

mode	A string, the corresponding mode (with the suffix "_ada" removed from the input to get_h_hp()). Must be one of the modes starting with "min_" supported by get_h_hp_vector().
para	Must be provided, but can be NULL. A number, the first parameter; see get_h_hp() or get_h_hp_vector().
percentile	A number, the percentile for column-wise truncation on hx and hpx.

Details

Helper function of get_h_hp(). Please refer to get_hs_hp().

Value

A function that returns a list containing hx=h(x) (element-wise) and hpx=hp(x) (element-wise derivative of h) when applied to a matrix x, with both of the results having the same shape as x.

Examples

```
get_h_hp_adaptive("min_log_pow", 1, 0.4)(matrix(0:49, nrow=10))
get_h_hp_adaptive("min_pow", 2, 0.3)(matrix(0:49, nrow=10))
get_h_hp_adaptive("min_softplus", 2, 0.6)(matrix(seq(0, 0.49, by=0.01), nrow=10))

hx_hpx <- get_h_hp_adaptive("min_log_pow", 1, 0.4)(matrix(0:49, nrow=10))
hx_hpx2 <- get_h_hp("min_log_pow_ada", 1, 0.4)(matrix(0:49, nrow=10))
c(max(abs(hx_hpx$hx - hx_hpx2$hx)), max(abs(hx_hpx$hpx - hx_hpx2$hpx)))
```

get_h_hp_vector *Generator of h and hp (derivative of h) functions.*

Description

Generator of h and hp (derivative of h) functions.

Usage

```
get_h_hp_vector(mode, para = NULL, para2 = NULL)
```

Arguments

mode	A string, see details.
para	May be optional. A number, the first parameter. Default to NULL.
para2	May be optional. A number, the second parameter. Default to NULL.

Details

Helper function of get_h_hp(). Please refer to get_hs_hp().

Value

A function that returns a matrix with hx=h(x) (element-wise) and hpx=hp(x) (element-wise derivative of h) cbinded when applied to a vector or a matrix x, where if x is a vector, the returned value will have two columns and number of rows equal to length(x), otherwise it will have the same number of rows as x and number of columns doubled.

Examples

```
get_h_hp_vector("mcp", 2, 4)
get_h_hp_vector("min_log_pow", 1, log(1+3))
get_h_hp_vector("min_pow", 1, 3)
get_h_hp_vector("min_softplus")
```

get_postfix_rule *Changes a logical expression in infix notation to postfix notation using the shunting-yard algorithm.*

Description

Changes a logical expression in infix notation to postfix notation using the shunting-yard algorithm.

Usage

```
get_postfix_rule(rule, num_eqs)
```

Arguments

rule	A string containing positive integers, parentheses, and "&" and "|" only. "&&" and "&" are not differentiated, and similarly for "||" and "|". Chained operations are only allowed for the same operation ("&" or "|"), so instead of "1 && 2 || 3" one should write either "(1 && 2) || 3" or "1 && (2 || 3)" to avoid ambiguity.
num_eqs	An integer, must be larger than or equal to the largest integer appearing in rule.

Details

Applied to domain$rule if domain$type == "polynomial", and internally calls beautify_rule().

Value

rule in postfix notation.

Examples

```
get_postfix_rule("1 & 2 && 3", 3)
get_postfix_rule("1 & (2 || 3)", 3)
get_postfix_rule("(1 & 2) || 3 | (4 & (5 || 6) && 7) | 8 | (9 && (10 || 11 | 12) & 13)", 13)
## Not run:
get_postfix_rule("1 && 2 & 3 && 4", 3) # Error, ineq number 4 appearing in \code{rule}.

## End(Not run)
## Not run:
# Error, ambigious rule. Change to either \code{"1 & (2 | 3)"} or \code{"(1 & 2) | 3"}.
get_postfix_rule("1 & 2 | 3", 3)

## End(Not run)
```

---

get_results *Estimate $\mathbf{K}$ and $\boldsymbol{\eta}$ using elts from* get_elts() *given one* λ_$\mathbf{K}$ *(and* λ_$\boldsymbol{\eta}$ *if non-profiled non-centered) and applying warm-start with strong screening rules.*

Description

Estimate $\mathbf{K}$ and $\boldsymbol{\eta}$ using elts from get_elts() given one $\lambda_{\mathbf{K}}$ (and $\lambda_{\boldsymbol{\eta}}$ if non-profiled non-centered) and applying warm-start with strong screening rules.

Usage

```
get_results(
  elts,
  symmetric,
  lambda1,
  lambda2 = 0,
```

```
  tol = 1e-06,
  maxit = 10000,
  previous_res = NULL,
  is_refit = FALSE
)
```

Arguments

elts	A list, elements necessary for calculations returned by get_elts().
symmetric	A string. If equals "symmetric", estimates the minimizer $\mathbf{K}$ over all symmetric matrices; if "and" or "or", use the "and"/"or" rule to get the support.
lambda1	A number, the penalty parameter for $\mathbf{K}$.
lambda2	A number, the penalty parameter for $\boldsymbol{\eta}$. Default to 0. Cannot be Inf if non-profiled non-centered.
tol	Optional. A number, the tolerance parameter.
maxit	Optional. A positive integer, the maximum number of iterations.
previous_res	Optional. A list or NULL, the returned list by this function run previously with another lambda value.
is_refit	A boolean, in the refit mode for BIC estimation if TRUE. If TRUE, lambda1, previous_lambda1 and lambda2 are all set to 0, and estimation is restricted to entries in exclude that are 0.

Details

If elts$domain_type == "simplex", symmetric != "symmetric" or elts$centered == FALSE && elts$profiled_if_noncenter are currently not supported. If elts$domain_type == "simplex" and elts$setting contains substring "sum0", it is assumed that the column and row sums of K are all 0 and estimation will be done by profiling out the diagonal entries.

Value

converged	A boolean indicating convergence.
crit	A number, the final penalized loss.
edges	A vector of the indices of entries in the K estimate that are non-zero.
eta	A p-vector, the eta estimate. Returned only if elts$centered == FALSE.
eta_support	A vector of the indices of entries in the eta estimate that are non-zero. Returned only if elts$centered == FALSE && elts$profiled_if_noncenter == TRUE.
iters	An integer, number of iterations run.
K	A p*p matrix, the K estimate.
n	An integer, the number of samples.
p	An integer, the dimension.
is_refit,lambda1,maxit,previous_lambda1,symmetric,tol	Same as in the input.
lambda2	Same as in the input, and returned only if elts$centered == FALSE and elts$profiled_if_noncenter == FALSE.

Examples

```
# Examples are shown for Gaussian truncated to R+^p only. For other distributions
#   on other types of domains, please refer to \code{gen()} or \code{get_elts()}, as the
#   way to call this function (\code{get_results()}) is exactly the same in those cases.
n <- 50
p <- 30
domain <- make_domain("R+", p=p)
mu <- rep(0, p)
K <- diag(p)
x <- tmvtnorm::rtmvnorm(n, mean = mu, sigma = solve(K),
       lower = rep(0, p), upper = rep(Inf, p), algorithm = "gibbs",
       burn.in.samples = 100, thinning = 10)

h_hp <- get_h_hp("min_pow", 1, 3)
dm <- 1 + (1-1/(1+4*exp(1)*max(6*log(p)/n, sqrt(6*log(p)/n))))
elts_gauss_np <- get_elts(h_hp, x, setting="gaussian", domain=domain,
                 centered=FALSE, profiled=FALSE, scale="norm", diag=dm)
test_nc_np <- get_results(elts_gauss_np, symmetric="symmetric", lambda1=0.35,
                 lambda2=2, previous_res=NULL, is_refit=FALSE)
test_nc_np2 <- get_results(elts_gauss_np, symmetric="and", lambda1=0.25,
                  lambda2=2, previous_res=test_nc_np, is_refit=FALSE)

elts_gauss_p <- get_elts(h_hp, x, setting="gaussian", domain=domain,
                centered=FALSE, profiled=TRUE, scale="norm", diag=dm)
test_nc_p <- get_results(elts_gauss_p, symmetric="symmetric",
                lambda1=0.35, lambda2=NULL, previous_res=NULL, is_refit=FALSE)

elts_gauss_c <- get_elts(h_hp, x, setting="gaussian", domain=domain,
                centered=TRUE, scale="norm", diag=dm)
test_c <- get_results(elts_gauss_c, symmetric="or", lambda1=0.35,
                lambda2=NULL, previous_res=NULL, is_refit=FALSE)
```

`get_safe_log_h_hp` *Asymptotic log of* `h` *and* `hp` *functions for large* `x` *for modes with an unbounded* `h`.

Description

Asymptotic log of `h` and `hp` functions for large `x` for modes with an unbounded `h`.

Usage

```
get_safe_log_h_hp(mode, para)
```

Arguments

`mode`	A string, the class of the `h` function. Must be one of `"asinh"`, `"cosh"`, `"exp"`, `"identity"`, `"log_pow"`, `"pow"`, `"sinh"`, `"softplus"`, and `"tanh"`.
`para`	A number, the first parameter to the `h` function.

Value

A list of two vectorized functions, logh and loghp.

Examples

```
para <- 2.3
x <- seq(from=0.1, to=150, by=0.1)
for (mode in c("asinh", "cosh", "exp", "identity", "log_pow", "pow", "sinh", "softplus", "tanh")) {
  print(mode)
  hx_hpx <- get_h_hp(mode, para)(x)
  print(c(max(abs(get_safe_log_h_hp(mode, para)$logh(x) - log(hx_hpx$hx))),
          max(abs(get_safe_log_h_hp(mode, para)$loghp(x) - log(hx_hpx$hpx)))))
}
```

---

get_trun *The truncation point for* h *for* h *that is truncated (bounded but not naturally bounded).*

Description

The truncation point for h for h that is truncated (bounded but not naturally bounded).

Usage

```
get_trun(mode, param1, param2)
```

Arguments

mode	A string, the class of the h function. Must be one of "mcp", "scad", "min_asinh", "min_cosh", "min_exp", "min_log_pow", "min_pow", "min_sinh", "min_softplus", "truncated_sin", and "truncated_tan".
param1	A number, the first parameter to the h function.
param2	A number, the second parameter (may be optional depending on mode) to the h function.

Value

Returns the truncation point (the point x0 such that h becomes constant and hp becomes 0 for x >= x0) for some selected modes.

Examples

```
param1 <- 1.3; param2 <- 2.3
for (mode in c("mcp", "scad", "min_asinh", "min_cosh", "min_exp", "min_log_pow",
    "min_pow", "min_sinh", "min_softplus", "truncated_tan")) {
  # Valgrind complains about "truncated_sin" for unknown reason; omitted
  print(mode)
  trun <- get_trun(mode, param1, param2)
```

```
  x <- trun + -3:3 / 1e5
  hx_hpx <- get_h_hp(mode, param1, param2)(x)
  print(round(x, 6))
  print(paste("hx:", paste(hx_hpx$hx, collapse=" ")))
  print(paste("hpx:", paste(hx_hpx$hpx, collapse=" ")))
}
```

`h_of_dist` *Finds the distance of each element in a matrix x to the its boundary of the domain while fixing the others in the same row (dist(x, domain)), and calculates element-wise h(dist(x, domain)) and h\'(dist(x, domain)) (w.r.t. each element in x).*

Description

Finds the distance of each element in a matrix `x` to its boundary of the `domain` while fixing the others in the same row (`dist(x, domain)`), and calculates element-wise `h(dist(x, domain))` and `h\'(dist(x, domain))` (w.r.t. each element in `x`).

Usage

```
h_of_dist(h_hp, x, domain, log = FALSE)
```

Arguments

`h_hp`	A function, the h and hp (the derivative of `h`) functions. `h_hp(x)` should return a list of elements `hx` (`h(x)`) and `hpx` (`hp(x)`), both of which have the same size as `x`.
`x`	An `n` by `p` matrix, the data matrix, where `n` is the sample size and `p` the dimension.
`domain`	A list returned from `make_domain()` that represents the domain.
`log`	A logical, defaults to `FALSE`. If `TRUE`, assumes that `h_hp` contains in fact the log of `h` and `hp`, and this function will return the log of `h(dist(x, domain))` and `abs(h\'(dist(x, domain)))` along with the sign of `h\'(dist(x, domain))`.

Details

Define `dist(x, domain)` as the matrix whose `i,j`-th component is the distance of $x_{i,j}$ to the boundary of `domain`, assuming $x_{i,-j}$ are fixed. The matrix has the same size of `x` (`n` by `p`), or if `domain$type == "simplex"` and `x` has full dimension `p`, it has `p-1` columns.
Define `dist\'(x, domain)` as the component-wise derivative of `dist(x, domain)` in its components. That is, its `i,j`-th component is 0 if $x_{i,j}$ is unbounded or is bounded from both below and above or is at the boundary, or -1 if $x_{i,j}$ is closer to its lower boundary (or if its bounded from below but unbounded from above), or 1 otherwise.
`h_of_dist(h_hp, x, domain)` simply returns `h_hp(dist(x, domain))$hx` and `h_hp(dist(x, domain))$hpx * dist\'(x, domain)` (element-wise derivative of `h_hp(dist(x, domain))$hx` w.r.t. `x`).

Value

If log == FALSE, a list that contains h(dist(x, domain)) and h\'(dist(x, domain)).

hdx	h(dist(x, domain)).
hpdx	hp(dist(x, domain)).

If log == TRUE, a list that contains the log of h(dist(x, domain)) and abs(h\'(dist(x, domain))) as well as the sign of h\'(dist(x, domain)).

log_hdx	log(h(dist(x, domain))).
log_hpdx	log(abs(hp(dist(x, domain)))).
sign_hpdx	sign(hp(dist(x, domain))).

Examples

```
n <- 20
p <- 10
eta <- rep(0, p)
K <- diag(p)
dm <- 1 + (1-1/(1+4*exp(1)*max(6*log(p)/n, sqrt(6*log(p)/n))))

# Gaussian on R^p:
domain <- make_domain("R", p=p)
x <- mvtnorm::rmvnorm(n, mean=solve(K, eta), sigma=solve(K))
# Equivalently:

x2 <- gen(n, setting="gaussian", abs=FALSE, eta=eta, K=K, domain=domain,
       finite_infinity=100, xinit=NULL, burn_in=1000, thinning=100, verbose=FALSE)

h_hp <- get_h_hp("pow", 2) # For demonstration only
hd <- h_of_dist(h_hp, x, domain)
# hdx is all Inf and hpdx is all 0 since each coordinate is unbounded with domain R
c(all(is.infinite(hd$hdx)), all(hd$hpdx==0))


# exp on R_+^p:
domain <- make_domain("R+", p=p)
x <- gen(n, setting="exp", abs=FALSE, eta=eta, K=K, domain=domain, finite_infinity=100,
       xinit=NULL, seed=2, burn_in=1000, thinning=100, verbose=FALSE)
h_hp <- get_h_hp("pow", 2) # For demonstration only
hd <- h_of_dist(h_hp, x, domain)
# hdx is x^2 and hpdx is 2*x; with domain R+, the distance of x to the boundary is just x itself
c(max(abs(hd$hdx - x^2)), max(abs(hd$hpdx - 2*x)))


# Gaussian on sum(x^2) > p with x allowed to be negative
domain <- make_domain("polynomial", p=p,
     ineqs=list(list("expression"=paste("sum(x^2)>", p), abs=FALSE, nonnegative=FALSE)))
x <- gen(n, setting="gaussian", abs=FALSE, eta=eta, K=K, domain=domain, finite_infinity=100,
       xinit=NULL, seed=2, burn_in=1000, thinning=100, verbose=FALSE)
dist <- get_dist(x, domain)
```

```
quota <- p - (rowSums(x^2) - x^2) # How much should xij^2 at least be so that sum(xi^2) > p?
# How far is xij from +/-sqrt(quota), if quota >= 0?
dist_to_bound <- abs(x[quota >= 0]) - abs(sqrt(quota[quota >= 0]))
# Should be equal to our own calculations
max(abs(dist$dx[is.finite(dist$dx)] - dist_to_bound))
# dist'(x) should be the same as the sign of x
all(dist$dpx[is.finite(dist$dx)] == sign(x[quota >= 0]))
# quota is negative <-> sum of x_{i,-j}^2 already > p <-> xij unbounded given others
#      <-> distance to boundary is Inf
all(quota[is.infinite(dist$dx)] < 0)

h_hp <- get_h_hp("pow", 2) # For demonstration only
# Now confirm that h_of_dist indeed applies h and hp to dists
hd <- h_of_dist(h_hp, x, domain)
# hdx = dist ^ 2
print(max(abs(hd$hdx[is.finite(dist$dx)] - dist$dx[is.finite(dist$dx)]^2)))
# hdx = Inf if dist = Inf
print(all(is.infinite(hd$hdx[is.infinite(dist$dx)])))
 # hpdx = 2 * dist' * dist
print(max(abs(hd$hpdx[is.finite(dist$dx)] - 2*(dist$dpx*dist$dx)[is.finite(dist$dx)])))
print(all(hd$hpdx[is.infinite(dist$dx)] == 0)) # hpdx = 0 if dist = Inf


# gamma on ([0, 1] v [2,3])^p
domain <- make_domain("uniform", p=p, lefts=c(0,2), rights=c(1,3))
x <- gen(n, setting="gamma", abs=FALSE, eta=eta, K=K, domain=domain,
       xinit=NULL, seed=2, burn_in=1000, thinning=100, verbose=FALSE)
dist <- get_dist(x, domain)
# If 0 <= xij <= 1, distance to boundary is min(x-0, 1-x)
max(abs(dist$dx - pmin(x, 1-x))[x >= 0 & x <= 1])
# If 0 <= xij <= 1, dist'(xij) is 1 if it is closer to 0, or -1 if it is closer 1,
#   assuming xij %in% c(0, 0.5, 1) with probability 0
all((dist$dpx == 2 * (1-x > x) - 1)[x >= 0 & x <= 1])
# If 2 <= xij <= 3, distance to boundary is min(x-2, 3-x)
max(abs(dist$dx - pmin(x-2, 3-x))[x >= 2 & x <= 3])
# If 2 <= xij <= 3, dist'(xij) is 1 if it is closer to 2, or -1 if it is closer 3,
#   assuming xij %in% c(2, 2.5, 3) with probability 0
all((dist$dpx == 2 * (3-x > x-2) - 1)[x >= 2 & x <= 3])
h_hp <- get_h_hp("pow", 2) # For demonstration only
# Now confirm that h_of_dist indeed applies h and hp to dists
hd <- h_of_dist(h_hp, x, domain)
# hdx = dist ^ 2
print(max(abs(hd$hdx - dist$dx^2)))
# hpdx = 2 * dist' * dist
print(max(abs(hd$hpdx - 2*dist$dpx*dist$dx)))


# a0.6_b0.7 on {x1 > 1 && log(1.3) < x2 < 1 && x3 > log(1.3) && ... && xp > log(1.3)}
domain <- make_domain("polynomial", p=p, rule="1 && 2 && 3",
       ineqs=list(list("expression"="x1>1", abs=FALSE, nonnegative=TRUE),
                       list("expression"="x2<1", abs=FALSE, nonnegative=TRUE),
                       list("expression"="exp(x)>1.3", abs=FALSE, nonnegative=FALSE)))
set.seed(1)
```

```
xinit <- c(1.5, 0.5, abs(stats::rnorm(p-2)) + log(1.3))
x <- gen(n, setting="ab_3/5_7/10", abs=FALSE, eta=eta, K=K, domain=domain, finite_infinity=100,
       xinit=xinit, seed=2, burn_in=1000, thinning=100, verbose=FALSE)
dist <- get_dist(x, domain)
# x_{i1} has uniform bound [1, +Inf), so its distance to its boundary is x_{i1} - 1
max(abs(dist$dx[,1] - (x[,1] - 1)))
# x_{i2} has uniform bound [log(1.3), 1], so its distance to its boundary
#    is min(x_{i2} - log(1.3), 1 - x_{i2})
max(abs(dist$dx[,2] - pmin(x[,2] - log(1.3), 1 - x[,2])))
# x_{ij} for j >= 3 has uniform bound [log(1.3), +Inf), so its distance to its boundary is
#    simply x_{ij} - log(1.3)
max(abs(dist$dx[,3:p] - (x[,3:p] - log(1.3))))
# dist'(xi2) is 1 if it is closer to log(1.3), or -1 if it is closer 1,
#    assuming x_{i2} %in% c(log(1.3), (1+log(1.3))/2, 1) with probability 0
all((dist$dpx[,2] == 2 * (1 - x[,2] > x[,2] - log(1.3)) - 1))
all(dist$dpx[,-2] == 1) # x_{ij} for j != 2 is bounded from below but unbounded from above,
#    so dist'(xij) is always 1
h_hp <- get_h_hp("pow", 2) # For demonstration only
# Now confirm that h_of_dist indeed applies h and hp to dists
hd <- h_of_dist(h_hp, x, domain)
# hdx = dist ^ 2
print(max(abs(hd$hdx - dist$dx^2)))
# hpdx = 2 * dist' * dist
print(max(abs(hd$hpdx - 2*dist$dpx*dist$dx)))


# log_log model on {x in R_+^p: sum_j j * xj <= 1}
domain <- make_domain("polynomial", p=p,
       ineqs=list(list("expression"=paste(paste(sapply(1:p,
                           function(j){paste(j, "x", j, sep="")}), collapse="+"), "<1"),
                  abs=FALSE, nonnegative=TRUE)))
x <- gen(n, setting="log_log", abs=FALSE, eta=eta, K=K, domain=domain, finite_infinity=100,
       xinit=NULL, seed=2, burn_in=1000, thinning=100, verbose=FALSE)
dist <- get_dist(x, domain)
# Upper bound for j * xij so that sum_j j * xij <= 1
quota <- 1 - (rowSums(t(t(x) * 1:p)) - t(t(x) * 1:p))
# Distance of xij to its boundary is min(xij - 0, quota_{i,j} / j - xij)
max(abs(dist$dx - pmin((t(t(quota) / 1:p) - x), x)))
h_hp <- get_h_hp("pow", 2) # For demonstration only
# Now confirm that h_of_dist indeed applies h and hp to dists
hd <- h_of_dist(h_hp, x, domain)
# hdx = dist ^ 2
print(max(abs(hd$hdx - dist$dx^2)))
# hpdx = 2 * dist' * dist
print(max(abs(hd$hpdx - 2*dist$dpx*dist$dx)))


# log_log_sum0 model on the simplex with K having row and column sums 0 (Aitchison model)
domain <- make_domain("simplex", p=p)
K <- -cov_cons("band", p=p, spars=3, eig=1)
diag(K) <- diag(K) - rowSums(K) # So that rowSums(K) == colSums(K) == 0
eigen(K)$val[(p-1):p] # Make sure K has one 0 and p-1 positive eigenvalues
x <- gen(n, setting="log_log_sum0", abs=FALSE, eta=eta, K=K, domain=domain,
```

```
       xinit=NULL, seed=2, burn_in=1000, thinning=100, verbose=FALSE)
# Note that dist$dx and dist$dpx only has p-1 columns -- excluding the last coordinate in x
dist <- get_dist(x, domain)
# Upper bound for x_{i,j} so that x_{i,1} + ... + x_{i,p-1} <= 1
quota <- 1 - (rowSums(x[,-p]) - x[,-p])
# Distance of x_{i,j} to its boundary is min(xij - 0, quota_{i,j} - xij)
max(abs(dist$dx - pmin(quota - x[,-p], x[,-p])))
h_hp <- get_h_hp("pow", 2) # For demonstration only
# Now confirm that h_of_dist indeed applies h and hp to dists
hd <- h_of_dist(h_hp, x, domain)
# hdx = dist ^ 2
print(max(abs(hd$hdx - dist$dx^2)))
# hpdx = 2 * dist' * dist
print(max(abs(hd$hpdx - 2*dist$dpx*dist$dx)))
```

`interval_intersection` *Finds the intersection between two unions of intervals.*

Description

Finds the intersection between two unions of intervals.

Usage

```
interval_intersection(A, B)
```

Arguments

`A`	A list of vectors of size 2, each representing an interval. It is required that `A[[i]][1]` <= `A[[i]][2]` <= `A[[j]][1]` for any `i` < `j`.
`B`	A list of vectors of size 2, each representing an interval. It is required that `A[[i]][1]` <= `A[[i]][2]` <= `A[[j]][1]` for any `i` < `j`.

Details

Finds the intersection between the union of all intervals in `A` and the union of all intervals in `B`.

Value

A list of vectors of size 2, whose union represents the intersection between `A` and `B`.

Examples

```
interval_intersection(list(c(1.2,1.5), c(2.3,2.7)),
       list(c(0.6,1.4), c(2.5,3.6), c(6.3,6.9)))
interval_intersection(list(c(-0.3,0.55), c(2.35,2.8)),
       list(c(0.54,0.62), c(2.5,2.9)))
interval_intersection(list(c(0,1)), list(c(1,2)))
interval_intersection(list(c(0,1+1e-8)), list(c(1,2)))
```

```
interval_intersection(list(c(0,1), c(2,3)),
       list(c(1,2)))
interval_intersection(list(c(0,1+1e-8), c(2-1e-8,3)),
       list(c(1,2)))
interval_intersection(list(c(0,1)), list())
```

`interval_union` *Finds the union between two unions of intervals.*

Description

Finds the union between two unions of intervals.

Usage

```
interval_union(A, B)
```

Arguments

`A`	A list of vectors of size 2, each representing an interval. It is required that `A[[i]][1]` <= `A[[i]][2]` <= `A[[j]][1]` for any `i` < `j`.
`B`	A list of vectors of size 2, each representing an interval. It is required that `A[[i]][1]` <= `A[[i]][2]` <= `A[[j]][1]` for any `i` < `j`.

Details

Finds the union between the union of all intervals in `A` and the union of all intervals in `B`.

Value

A list of vectors of size 2, whose union represents the union between `A` and `B`.

Examples

```
interval_union(list(c(1.2,1.5), c(2.3,2.7)),
       list(c(0.6,1.4), c(2.5,3.6), c(6.3,6.9)))
interval_union(list(c(-0.3,0.55), c(2.35,2.8)),
       list(c(0.54,0.62), c(2.5,2.9)))
interval_union(list(c(0,1)), list(c(1,2)))
interval_union(list(c(0,1-1e-8)), list(c(1,2)))
interval_union(list(c(0,1), c(2,3)),
       list(c(1,2)))
interval_union(list(c(0,1-1e-8), c(2+1e-8,3)),
       list(c(1,2)))
interval_union(list(c(0,1)), list())
```

in_bound *Returns whether a vector or each row of a matrix falls inside a domain.*

Description

Returns whether a vector or each row of a matrix falls inside a domain.

Usage

```
in_bound(x, domain)
```

Arguments

x	A vector of length or a matrix of number of columns equal to domain$p if domain$type != "simplex", or either domain$p or domain$p-1 otherwise.
domain	A list returned from make_domain() that represents the domain.

Details

Returns whether a vector or each row of a matrix falls inside a domain. If domain$type == "simplex", if the length/number of columns is domain$p, returns all(x > 0) && abs(sum(x) - 1) < domain$simplex_tol; if the dimension is domain$p-1, returns all(x > 0) && sum(x) < 1.

Value

A logical vector of length equal to the number of rows in x (1 if x is a vector).

Examples

```
p <- 30
n <- 10

# The 30-dimensional real space R^30, assuming probability of
domain <- make_domain("R", p=p)
in_bound(1:p, domain)
in_bound(matrix(1:(p*n), ncol=p), domain)

# The non-negative orthant of the 30-dimensional real space, R+^30
domain <- make_domain("R+", p=p)
in_bound(matrix(1:(p*n), ncol=p), domain)
in_bound(matrix(1:(p*n) * (2*rbinom(p*n, 1, 0.98)-1), ncol=p), domain)

# x such that sum(x^2) > 10 && sum(x^(1/3)) > 10 with x allowed to be negative
domain <- make_domain("polynomial", p=p, rule="1 && 2",
       ineqs=list(list("expression"="sum(x^2)>10", abs=FALSE, nonnegative=FALSE),
                  list("expression"="sum(x^(1/3))>10", abs=FALSE, nonnegative=FALSE)))
in_bound(rep((5/p)^3, p), domain)
in_bound(rep((10/p)^3, p), domain)
in_bound(rep((15/p)^3, p), domain)
```

```
in_bound(rep((5/p)^(1/2), p), domain)
in_bound(rep((10/p)^(1/2), p), domain)
in_bound(rep((15/p)^(1/2), p), domain)

# ([0, 1] v [2,3]) ^ p
domain <- make_domain("uniform", p=p, lefts=c(0,2), rights=c(1,3))
in_bound(c(0.5, 2.5)[rbinom(p, 1, 0.5)+1], domain)
in_bound(c(rep(0.5, p/2), rep(2.5, p/2)), domain)
in_bound(c(rep(0.5, p/2), rep(2.5, p/2-1), 4), domain)

# x such that {x1 > 1 && log(1.3) < x2 < 1 && x3 > log(1.3) && ... && xp > log(1.3)}
domain <- make_domain("polynomial", p=p, rule="1 && 2 && 3",
       ineqs=list(list("expression"="x1>1", abs=FALSE, nonnegative=TRUE),
                       list("expression"="x2<1", abs=FALSE, nonnegative=TRUE),
                       list("expression"="exp(x)>1.3", abs=FALSE, nonnegative=FALSE)))
in_bound(c(1.5, (log(1.3)+1)/2, rep(log(1.3)*2, p-2)), domain)
in_bound(c(0.5, (log(1.3)+1)/2, rep(log(1.3)*2, p-2)), domain)
in_bound(c(1.5, log(1.3)/2, rep(log(1.3)*2, p-2)), domain)
in_bound(c(1.5, (log(1.3)+1)/2, rep(log(1.3)/2, p-2)), domain)

# x in R_+^p such that {sum(log(x))<2 || (x1^(2/3)-1.3x2^(-3)<1 && exp(x1)+2.3*x2>2)}
domain <- make_domain("polynomial", p=p, rule="1 || (2 && 3)",
       ineqs=list(list("expression"="sum(log(x))<2", abs=FALSE, nonnegative=TRUE),
                 list("expression"="x1^(2/3)-1.3x2^(-3)<1", abs=FALSE, nonnegative=TRUE),
                   list("expression"="exp(x1)+2.3*x2^2>2", abs=FALSE, nonnegative=TRUE)))
in_bound(rep(exp(1/p), p), domain)
in_bound(c(1, 1, rep(1e5, p-2)), domain)

# x in R_+^p such that {x in R_+^p: sum_j j * xj <= 1}
domain <- make_domain("polynomial", p=p,
       ineqs=list(list("expression"=paste(paste(sapply(1:p,
                           function(j){paste(j, "x", j, sep="")}), collapse="+"), "<1"),
                     abs=FALSE, nonnegative=TRUE)))
in_bound(0.5/p/1:p, domain)
in_bound(2/p/1:p, domain)
in_bound(rep(1/p, p), domain)
in_bound(rep(1/p^2, p), domain)

# The (p-1)-simplex
domain <- make_domain("simplex", p=p)
x <- abs(matrix(rnorm(p*n), ncol=p))
x <- x / rowSums(x)
in_bound(x, domain) # TRUE
in_bound(x[,1:(p-1)], domain) # TRUE
x2 <- x
x2[,1] <- -x2[,1]
in_bound(x2, domain) # FALSE since the first component is now negative
in_bound(x2[,1:(p-1)], domain) # FALSE since the first component is now negative
x3 <- x
x3[,1] <- x3[,1] + domain$simplex_tol * 10
in_bound(x3, domain) # FALSE since the rows do not sum to 1
in_bound(x3[,1:(p-1)], domain) # TRUE since the first (p-1) elts in each row still sum to < 1
x3[,1] <- x3[,1] + x3[,p]
```

```
in_bound(x3[,1:(p-1)], domain) # FALSE since the first (p-1) elts in each row now sum to > 1

# The l-1 ball {sum(|x|) < 1}
domain <- make_domain("polynomial", p=p,
       ineqs=list(list("expression"="sum(x)<1", abs=TRUE, nonnegative=FALSE)))
in_bound(rep(0.5/p, p)*(2*rbinom(p, 1, 0.5)-1), domain)
in_bound(rep(1.5/p, p)*(2*rbinom(p, 1, 0.5)-1), domain)
```

`lambda_max` *Analytic solution for the minimum λ_$\mathbf{K}$ that gives the empty graph.*

Description

Analytic solution for the minimum $\lambda_{\mathbf{K}}$ that gives the empty graph. In the non-centered setting the bound is not tight, as it is such that both $\mathbf{K}$ and $\boldsymbol{\eta}$ are empty. The bound is also not tight if `symmetric == "and"`.

Usage

```
lambda_max(elts, symmetric, lambda_ratio = Inf)
```

Arguments

`elts`	A list, elements necessary for calculations returned by `get_elts()`.
`symmetric`	A string. If equals `"symmetric"`, estimates the minimizer $\mathbf{K}$ over all symmetric matrices; if `"and"` or `"or"`, use the "and"/"or" rule to get the support.
`lambda_ratio`	A positive number (or `Inf`), the fixed ratio $\lambda_{\mathbf{K}}$ and $\lambda_{\boldsymbol{\eta}}$, if $\lambda_{\boldsymbol{\eta}} \neq 0$ (non-profiled) in the non-centered setting.

Value

A number, the smallest lambda that produces the empty graph in the centered case, or that gives zero solutions for $\mathbf{K}$ and $\boldsymbol{\eta}$ in the non-centered case. If `symmetric == "and"`, it is not a tight bound for the empty graph.

Examples

```
# Examples are shown for Gaussian truncated to R+^p only. For other distributions
#   on other types of domains, please refer to \code{gen()} or \code{get_elts()},
#  as the way to call this function (\code{lambda_max()}) is exactly the same in those cases.
n <- 50
p <- 30
domain <- make_domain("R+", p=p)
mu <- rep(0, p)
K <- diag(p)
x <- tmvtnorm::rtmvnorm(n, mean = mu, sigma = solve(K),
       lower = rep(0, p), upper = rep(Inf, p), algorithm = "gibbs",
       burn.in.samples = 100, thinning = 10)
```

```
dm <- 1 + (1-1/(1+4*exp(1)*max(6*log(p)/n, sqrt(6*log(p)/n))))
h_hp <- get_h_hp("min_pow", 1, 3)
elts_gauss_np <- get_elts(h_hp, x, setting="gaussian", domain=domain,
                centered=FALSE, profiled=FALSE, diag=dm)

# Exact analytic solution for the smallest lambda such that K and eta are both zero,
#  but not a tight bound for K ONLY
lambda_max(elts_gauss_np, "symmetric", 2)
# Use the upper bound as a starting point for numerical search
test_lambda_bounds2(elts_gauss_np, "symmetric", lambda_ratio=2, lower = FALSE,
     lambda_start = lambda_max(elts_gauss_np, "symmetric", 2))

# Exact analytic solution for the smallest lambda such that K and eta are both zero,
#  but not a tight bound for K ONLY
lambda_max(elts_gauss_np, "or", 2)
# Use the upper bound as a starting point for numerical search
test_lambda_bounds2(elts_gauss_np, "or", lambda_ratio=2, lower = FALSE,
     lambda_start = lambda_max(elts_gauss_np, "or", 2))

# An upper bound, not tight.
lambda_max(elts_gauss_np, "and", 2)
# Use the upper bound as a starting point for numerical search
test_lambda_bounds2(elts_gauss_np, "and", lambda_ratio=2, lower = FALSE,
     lambda_start = lambda_max(elts_gauss_np, "and", 2))


elts_gauss_p <- get_elts(h_hp, x, setting="gaussian", domain=domain,
              centered=FALSE, profiled=TRUE, diag=dm)
# Exact analytic solution
lambda_max(elts_gauss_p, "symmetric")
# Numerical solution, should be close to the analytic solution
test_lambda_bounds2(elts_gauss_p, "symmetric", lambda_ratio=Inf, lower = FALSE,
     lambda_start = NULL)

# Exact analytic solution
lambda_max(elts_gauss_p, "or")
# Numerical solution, should be close to the analytic solution
test_lambda_bounds2(elts_gauss_p, "or", lambda_ratio=Inf, lower = FALSE,
     lambda_start = NULL)

# An upper bound, not tight
lambda_max(elts_gauss_p, "and")
# Use the upper bound as a starting point for numerical search
test_lambda_bounds2(elts_gauss_p, "and", lambda_ratio=Inf, lower = FALSE,
     lambda_start = lambda_max(elts_gauss_p, "and"))
```

---

makecoprime *Makes two integers coprime.*

Description

Divides both integers by their greatest common divisor, switching their signs if the second integer is negative. If either integer is 0, returns without modification.

Usage

```
makecoprime(a, b)
```

Arguments

a	An integer.
b	An integer.

Value

The greatest (positive) common divisor of two integers; if one of them is 0, returns the absolute value of the other number.

Examples

```
makecoprime(1, 2)
makecoprime(1, -2)
makecoprime(12, -18)
makecoprime(-12, 18)
makecoprime(15, 0)
makecoprime(0, -15)
makecoprime(0, 0)
```

---

make_domain *Creates a list of elements that defines the domain for a multivariate distribution.*

Description

Creates a list of elements that define the domain for a multivariate distribution.

Usage

```
make_domain(type, p, lefts = NULL, rights = NULL, ineqs = NULL, rule = NULL)
```

Arguments

type	A string, the domain type. Currently support "R", "R+", "uniform", "polynomial", "simplex". See details.
p	An integer, the dimension of the domain.

`lefts` Optional, required if `type == "uniform"` and must have the same length as `rights`. A non-empty vector of numbers (may contain `-Inf`), the left endpoints of a domain defined as a union of intervals. It is required that `lefts[i] <= rights[i] <= lefts[j]` for any `i < j`.

`rights` Optional, required if `type == "uniform"` and must have the same length as `lefts`. A non-empty vector of numbers (may contain `Inf`), the right endpoints of a domain defined as a union of intervals. It is required that `lefts[i] <= rights[i] <= lefts[j]` for any `i < j`.

`ineqs` Optional, required if `type == "polynomial"`. A list of lists, each sublist representing an inequality that defines the domain. Each sublist must contain fields `abs` (logical) and `nonnegative` (logical), and in addition either a single `expression` (string), or all of the following: `uniform` (logical), `larger` (logical), `power_numers` (1 or p integers), `power_denoms` (1 or p integers), `const` (a number), `coeffs` (1 or p numbers).

`rule` Optional, required if `type == "polynomial" && length(ineqs) > 1`. A string containing inequality numbers, spaces, parentheses, '&' and 'I' only. Used to indicate the logic operation on how to combine the domains defined by each inequality, i.e. "(1 & 2 && 3) || 4 | 5". Chained operations not separated by parentheses are only allowed for the same type of operation ('&'/'I'), i.e. "1 & 2 | 3" is not allowed; it should be either "(1 & 2) | 3" or "1 & (2 | 3)".

Details

The following types of domains are supported:

`"R"` The entire p-dimensional real space. Equivalent to `"uniform"` type with `lefts=-Inf` and `rights=Inf`.

`"R+"` The non-negative orthant of the p-dimensional real space. Equivalent to `"uniform"` type with `lefts=0` and `rights=Inf`.

`"uniform"` A union of finitely many disjoint intervals as a uniform domain for all components. The left endpoints should be specified through `lefts` and the right endpoints through `rights`. The intervals must be disjoint and strictly increasing, i.e. `lefts[i] <= rights[i] <= lefts[j]` for any `i < j`. E.g. `lefts=c(0, 10)` and `rights=c(5, Inf)` represents the domain ([0,5]v[10,+Inf])^p.

`"simplex"` The standard `p-1`-simplex with all components positive and sum to 1, i.e. `sum(x) == 1` and `x > 0`.

`"polynomial"` A finite intersection/union of domains defined by comparing a constant to a polynomial with at most one term in each component and no interaction terms (e.g. `x1^3+x1^2>1` or `x1*x2>1` not supported). The following is supported: `{x1^2 + 2*x2^(3/2) > 1} && ({3.14*x1 - 0.7*x3^3 < 1} || {-exp(3*x2) + 3.7*log(x3) + 2.4*x4^(-3/2)})`. To specify a polynomial-type domain, one should define the `ineqs`, and in case of more than one inequality, the logical `rule` to combine the domains defined by each inequality.

`rule` A logical rule in infix notation, e.g. `"(1 && 2 && 3) || (4 && 5) || 6"`, where the numbers represent the inequality numbers starting from 1. `"&&"` and `"&"` are not differentiated, and similarly for `"||"` and `"|"`. Chained operations are only allowed for the same operation (`"&"` or `"|"`), so instead of `"1 && 2 || 3"` one should write either `"(1 && 2) || 3"` or `"1 && (2 || 3)"` to avoid ambiguity.

ineqs A list of lists, each sublist represents one inequality, and must contain the following fields:

abs A logical, indicates whether one should evaluate the polynomial in abs(x) instead of x (e.g. "sum(x) > 1" with abs == TRUE is interpreted as sum(abs(x)) > 1).

nonnegative A logical, indicates whether the domain of this inequality should be restricted to the non-negative orthant.

In addition, one must in addition specify either a single string "expression" (highly recommended, detailed below), or all of the following fields (discouraged usage):

uniform A logical, indicates whether the inequality should be uniformly applied to all components (e.g. "x>1" -> "x1>1 && ... && xp>1").

larger A logical, indicates whether the polynomial should be larger or smaller than the constant (e.g. TRUE for x1 + ... + xp > C, and FALSE for x1 + ... + xp < C).

const A number, the constant the polynomial should be compared to (e.g. 2.3 for x1 + ... + xp > 2.3).

power_numers A single integer or a vector of p integers. x[i] will be raised to the power of power_numers[i] / power_denoms[i] (or without subscript if a singer integer). Note that x^(0/0) is interpreted as log(x), and x^(n/0) as exp(n*x) for n non-zero. For a negative x, x^(a/b) is defined as (-1)^a*|x|^(a/b) if b is odd, or NaN otherwise. (Example: c(2,3,5,0,-2) for x1^2+2*x2^(3/2)+3*x3^(5/3)+4*log(x4)+5*exp(-2*x)>1).

power_denoms A single integer or a vector of p integers. (Example: c(1,2,3,0,0) for x1^2+2*x2^(3/2)+3*x3^(5/3)+4*log(x4)+5*exp(-2*x)>1).

coeffs Required if uniform == FALSE. A vector of p doubles, where coeffs[i] is the coefficient on x[i] in the inequality.

The user is recommended to use a single expression for ease and to avoid potential errors. The user may safely skip the explanations and directly look at the examples to get a better understanding.

The expression should have the form "POLYNOMIAL SIGN CONST", where "SIGN" is one of "<", "<=", ">", ">=", and "CONST" is a single number (scientific notation allowed).

"POLYNOMIAL" must be (1) a single term (see below) in "x" with no coefficient (e.g. "x^(2/3)", "exp(3x)"), or (2) such a term surrounded by "sum()" (e.g. "sum(x^(2/3))", "sum(exp(3x))"), or (3) a sum of such terms in "x1" through "xp" (one term max for each component) with or without coefficients, separated by the plus or the minus sign (e.g. "2.3x1^(2/3)-3.4exp(x2)+x3^(-3/5)").

For (1) and (2), the term must be in one of the following forms: "x", "x^2", "x^(-2)", "x^(2/3)", "x^(-2/3)", "log(x)", "exp(x)", "exp(2x)", "exp(2*x)", "exp(-3x)", where the 2 and 3 can be changed to any other non-zero integers.
For (3), each term should be as above but in "x1", ..., "xp" instead of "x", following an optional double number and optionally a "*" sign.

Examples: For p=10,
(1) "x^2 > 2" defines the domain abs(x1) > sqrt(2) && ... && abs(x10) > sqrt(2).
(2) "sum(x^2) > 2" defines the domain x1^2 + ... + x10^2 > 2.
(3) "2.3x3^(2/3)-3.4x4+x5^(-3/5)+3.7*x6^(-4)-1.9*log(x7)+1.3e5*exp(-3x8)}\cr
\code{-2*exp(x9)+0.5exp(2*x10) <= 2" defines a domain using a polynomial in x3 through

x10, and x1 and x2 are thus allowed to vary freely.

Note that ">" and ">=" are not differentiated, and so are "<" and "<=".

Value

A list containing the elements that define the domain. For all types of domains, the following are returned.

type	A string, same as the input.
p	An integer, same as the input.
p_deemed	An integer, equal to p-1 if type == "simplex" or p otherwise.
checked	A logical, TRUE. Used in other functions to test whether a list is returned by this function.

In addition,

- For type == "simplex", returns in addition

  simplex_tol Tolerance used for simplex domains. Defaults to 1e-10.

- For type == "uniform", returns in addition

  lefts A non-empty vector of numbers, same as the input.

  rights A non-empty vector of numbers, same as the input.

  left_inf A logical, indicates whether lefts[1] is -Inf.

  right_inf A logical, indicates whether rights[length(rights)] is Inf.

- For type == "polynomial", returns in addition

  rule A string, same as the input if provided and valid; if not provided and length(ineqs) == 1, set to "1" by default.

  postfix_rule A string, rule in postfix notation (reverse Polish notation) containing numbers, " ", "&" and "|" only.

  ineqs A list of lists, each sublist representing one inequality containing the following fields:

  - uniform A logical, indicates whether the inequality should be uniformly applied to all components (e.g. "x>1" -> "x1>1 && ... && xp>1").
  - larger A logical, indicates whether the polynomial should be larger or smaller than the constant (e.g. TRUE for x1 + ... + xp > C, and FALSE for x1 + ... + xp < C).
  - const A number, the constant the polynomial should be compared to (e.g. 2.3 for x1 + ... + xp > 2.3).
  - abs A logical, indicates whether one should evaluate the polynomial in abs(x) instead of x.
  - nonnegative A logical, indicates whether the domain of this inequality should be restricted to the non-negative orthant.
  - power_numers A single integer or a vector of p integers. x[i] will be raised to the power of power_numers[i] / power_denoms[i] (or without subscript if a singer integer). Note that x^(0/0) is interpreted as log(x), and x^(n/0) as exp(n*x) for n non-zero. For a negative x, x^(a/b) is defined as (-1)^a*|x|^(a/b) if b is odd, or NaN otherwise.

power_denoms A single integer or a vector of p integers.

coeffs NULL if uniform == TRUE. A vector of p doubles, where coeffs[i] is the coefficient on x[i] in the inequality

Examples

```
p <- 30
# The 30-dimensional real space R^30
domain <- make_domain("R", p=p)

# The non-negative orthant of the 30-dimensional real space, R+^30
domain <- make_domain("R+", p=p)

# x such that sum(x^2) > 10 && sum(x^(1/3)) > 10 with x allowed to be negative
domain <- make_domain("polynomial", p=p, rule="1 && 2",
       ineqs=list(list("expression"="sum(x^2)>10", abs=FALSE, nonnegative=FALSE),
                  list("expression"="sum(x^(1/3))>10", abs=FALSE, nonnegative=FALSE)))
# Alternatively
domain2 <- make_domain("polynomial", p=p, rule="1 && 2",
      ineqs=list(list(uniform=FALSE, power_numers=2, power_denoms=1, const=10, coeffs=1,
                 larger=1, abs=FALSE, nonnegative=FALSE),
                 list(uniform=FALSE, power_numers=1, power_denoms=3, const=10, coeffs=1,
                 larger=1, abs=FALSE, nonnegative=FALSE)))


# ([0, 1] v [2,3]) ^ p
domain <- make_domain("uniform", p=p, lefts=c(0,2), rights=c(1,3))

# x such that {x1 > 1 && log(1.3) < x2 < 1 && x3 > log(1.3) && ... && xp > log(1.3)}
domain <- make_domain("polynomial", p=p, rule="1 && 2 && 3",
       ineqs=list(list("expression"="x1>1", abs=FALSE, nonnegative=TRUE),
                     list("expression"="x2<1", abs=FALSE, nonnegative=TRUE),
                     list("expression"="exp(x)>1.3", abs=FALSE, nonnegative=FALSE)))
# Alternatively
domain2 <- make_domain("polynomial", p=p, rule="1 && 2",
       ineqs=list(list(uniform=FALSE, power_numers=1, power_denoms=1, const=1,
                 coeffs=c(1,rep(0,p-1)), larger=1, abs=FALSE, nonnegative=TRUE),
                 list(uniform=FALSE, power_numers=1, power_denoms=1, const=1,
                 coeffs=c(0,1,rep(0,p-2)), larger=0, abs=FALSE, nonnegative=TRUE),
                 list(uniform=TRUE, power_numers=1, power_denoms=0, const=1.3,
                 larger=1, abs=FALSE, nonnegative=FALSE)))


# x in R_+^p such that {sum(log(x))<2 || (x1^(2/3)-1.3x2^(-3)<1 && exp(x1)+2.3*x2>2)}
domain <- make_domain("polynomial", p=p, rule="1 || (2 && 3)",
       ineqs=list(list("expression"="sum(log(x))<2", abs=FALSE, nonnegative=TRUE),
                list("expression"="x1^(2/3)-1.3x2^(-3)<1", abs=FALSE, nonnegative=TRUE),
                  list("expression"="exp(x1)+2.3*x2^2>2", abs=FALSE, nonnegative=TRUE)))
# Alternatively
domain2 <- make_domain("polynomial", p=p, rule="1 && 2",
       ineqs=list(list(uniform=FALSE, power_numers=0, power_denoms=0, const=2,
                 coeffs=1, larger=0, abs=FALSE, nonnegative=TRUE),
           list(uniform=FALSE, power_numers=c(2,-3,rep(1,p-2)), power_denoms=c(3,rep(1,p-1)),
```

```
            const=1, coeffs=c(1.0,-1.3,rep(0,p-2)), larger=0, abs=FALSE, nonnegative=TRUE),
            list(uniform=FALSE, power_numers=c(1,2,rep(1,p-2)), power_denoms=c(0,rep(1,p-1)),
             const=2, coeffs=c(1,2.3,rep(0,p-2)), larger=1, abs=FALSE, nonnegative=TRUE)))


# x in R_+^p such that {x in R_+^p: sum_j j * xj <= 1}
domain <- make_domain("polynomial", p=p,
       ineqs=list(list("expression"=paste(paste(sapply(1:p,
                           function(j){paste(j, "x", j, sep="")}), collapse="+"), "<1"),
                     abs=FALSE, nonnegative=TRUE)))
# Alternatively
domain2 <- make_domain("polynomial", p=p,
       ineqs=list(list(uniform=FALSE, power_numers=1, power_denoms=1, const=1,
                 coeffs=1:p, larger=0, abs=FALSE, nonnegative=TRUE)))


# The (p-1)-simplex
domain <- make_domain("simplex", p=p)

# The l-1 ball {sum(|x|) < 1}
domain <- make_domain("polynomial", p=p,
       ineqs=list(list("expression"="sum(x)<1", abs=TRUE, nonnegative=FALSE)))
```

`make_folds` *Helper function for making fold IDs for cross validation.*

Description

Helper function for making fold IDs for cross validation.

Usage

```
make_folds(nsamp, nfold, cv_fold_seed)
```

Arguments

`nsamp`	Number of samples.
`nfold`	Number of cross validation folds.
`cv_fold_seed`	Seed for random shuffling.

Value

A list of `nsamp` vectors, numbers 1 to `nsamp` shuffled and grouped into vectors of length `floor(nsamp/nfold)` followed by vectors of length `floor(nsamp/nfold)+1`.

Examples

```
make_folds(37, 5, NULL)
make_folds(100, 5, 2)
make_folds(100, 10, 3)
```

`mu_sigmasqhat` *Estimates the mu and sigma squared parameters from a univariate truncated normal sample.*

Description

Estimates the mu and sigma squared parameters from a univariate truncated normal sample.

Usage

```
mu_sigmasqhat(x, mode, param1, param2, mu = NULL, sigmasq = NULL)
```

Arguments

`x`	A vector, the data.
`mode`	A string, the class of the `h` function.
`param1`	A number, the first parameter to the `h` function.
`param2`	A number, the second parameter (may be optional depending on `mode`) to the `h` function.
`mu`	A number, may be `NULL`. If `NULL`, an estimate will be given; otherwise, the value will be treated as the known true `mu` parameter and is used to calculate an estimate for `sigmasq`, if `sigmasq` is `NULL`.
`sigmasq`	A number, may be `NULL`. If `NULL`, an estimate will be given; otherwise, the value will be treated as the known true `sigmasq` parameter and is used to calculate an estimate for `mu`, if `mu` is `NULL`.

Details

If both `mu` and `sigmasq` are provided, they are returned immediately. If neither is provided, the estimates are given as

$$[1/\sigma^2, \mu/\sigma^2] = \left\{\sum_{i=1}^{n} h(X_i)[X_i, -1][X_i, -1]^\top\right\}^{-1} \left\{\sum_{i=1}^{n} [h(X_i) + h'(X_i)X_i, -h'(X_i)]\right\}.$$

If only `sigmasq` is provided, the estimate for `mu` is given as

$$\sum_{i=1}^{n}[h(X_i)X_i - \sigma^2 h'(X_i)] / \sum_{i=1}^{n} h(X_i).$$

If only `mu` is given, the estimate for `sigmasq` is given as

$$\sum_{i=1}^{n} h(X_i)(X_i - \mu)^2 / \sum_{i=1}^{n}[h(X_i) + h'(X_i)(X_i - \mu)].$$

Value

A vector that contains the `mu` and the `sigmasq` estimates.

naiveSearch_bin *Finds the index of the bin a number belongs to using naive search.*

Description

Finds the index of the bin a number belongs to using naive search.

Usage

```
naiveSearch_bin(arr, x)
```

Arguments

arr	A vector of size at least 2.
x	A number. Must be within the range of [arr[1], arr[length(arr)]].

Details

Finds the smallest index i such that arr[i] <= x <= arr[i+1].

Value

The index i such that arr[i] <= x <= arr[i+1].

Examples

```
naiveSearch_bin(1:10, seq(1, 10, by=0.5))
```

parse_ab *Parses an ab setting into rational numbers a and b.*

Description

Parses an ab setting into rational numbers a and b.

Usage

```
parse_ab(s)
```

Arguments

s	A string starting with "ab_", followed by rational numbers a and b separated by "_". a and b must be integers or rational numbers of the form "int/int". See examples.

Value

A list of the following elements:

a_numer	The numerator of a.
a_denom	The denominator of a.
b_numer	The numerator of b.
b_denom	The denominator of b.

Examples

```
parse_ab("ab_1_1") # gaussian: a = 1, b = 1
parse_ab("ab_2_5/4") # a = 2, b = 5/4
parse_ab("ab_5/4_3/2") # a = 5/4, b = 3/2
parse_ab("ab_3/2_0/0") # a = 3/2, b = 0/0 (log)
parse_ab("ab_1/2_0/0") # exp: a = 1/2, b = 0/0 (log)
```

---

parse_ineq *Parses an ineq expression into a list of elements that represents the ineq.*

Description

Parses an ineq expression into a list of elements that represents the ineq.

Usage

```
parse_ineq(s, p)
```

Arguments

s	A string, an ineq expression. Please refer make_domain().
p	An integer, the dimension.

Details

Please refer make_domain() for the syntax of the expression.

Value

A list containing the following elements:

uniform	A logical, indicates whether the ineq is a uniform expression that applies to each component independently (e.g. x^2>1, exp(3*|x|)<3.4).
const	A number, the constant side of the ineq that the variable side should compare to (e.g. 1.3 in x1^2+2*x2^3>1.3).
larger	A logical, indicates whether the variable side of the expression should be larger or smaller than const.

power_numers A single number or a vector of length p. The numerators of the powers in the ineq for each component (e.g. c(2,3,5,0,-2) for x1^2+2*x2^(3/2)+3*x3^(5/3)+4*log(x4)+5*exp(-2*x)>1).

power_denoms A single number or a vector of length p. The denominators of the powers in the ineq for each component (e.g. c(1,2,3,0,0) for x1^2+2*x2^(3/2)+3*x3^(5/3)+4*log(x4)+5*exp(-2*x)>1).

coeffs A vector of length p that represents the coefficients in the ineq associated with each component. Returned only if uniform == FALSE.

Examples

```
p <- 30
parse_ineq("sum(x^2)>10", p)
parse_ineq("sum(x^(1/3))>10", p)
parse_ineq("x1>1", p)
parse_ineq("x2<1", p)
parse_ineq("exp(x)>1.3", p)
parse_ineq("sum(log(x)) < 2", p)
parse_ineq("x1^(2/3)-1.3x2^(-3)<1", p)
parse_ineq("exp(x1)+2.3*x2^2 > 2", p)
parse_ineq(paste(paste(sapply(1:p,
                        function(j){paste(j, "x", j, sep="")}), collapse="+"), "<1"), p)

parse_ineq("0.5*x1^(-2/3)-x3^3 + 2log(x2)- 1.3e4exp(-25*x6)+x8-.3x5^(-3/-4) >= 2", 8)
parse_ineq("0.5*x1^(-2/3)-x2^(4/-6)+2e3x3^(-6/9) < 3.5e5", 3)
parse_ineq("x^(-2/3)<=3e3", 5)
parse_ineq("sum(x^(-2/3))<=3e3", 5)
parse_ineq("x<=3e3", 5)
parse_ineq("sum(x)<=3e3", 5)
parse_ineq("exp(-23x)<=3e3", 5)
parse_ineq("sum(exp(-23x))<=3e3", 5)
```

---

random_init_polynomial

Randomly generate an initial point in the domain defined by a single polynomial with no negative coefficient.

Description

Randomly generate an initial point in the domain defined by a single polynomial with no negative coefficient.

Usage

```
random_init_polynomial(domain)
```

Arguments

domain	A list returned from make_domain() that represents the domain. Currently only supports domain$type == "polynomial" && length(domain$ineqs) == 1. If domain$ineqs[[1]]$uniform == FALSE, domain$ineqs[[1]]$coeffs must not contain negative numbers.

Details

If inequality is uniform, find the uniform bound for each component and generate each coordinate using random_init_uniform(). Otherwise, first randomly generate centered laplace variables for components with coefficient 0 (free variables). Then assign a quota of eq$const / length(nonzero_coefficient) to each coordinate (so that each frac_pow(x[i], eq$power_numers[i], eq$power_denoms[i], eq$abs) * eq$coeffs[i] is compared to quota). Deal with components with exp() term first, and generate each coordinate while fulfilling quota if possible; if not, randomly generate from [-0.01,0.01]/abs(eq$power_numers[i]). Then recalculate the new quota which subtracts the exp() terms from eq$const, and this time divided by the number of remaining components. If quota becomes negative and eq$larger == FALSE, each component, after frac_pow() is assumed to give a negative number. This is not possible if the term has the form x^{even_number/even_number}, or if the term is not log() in the case where eq$nonnegative == TRUE || eq$abs == TRUE. Change quota to a positive smaller in absolute value for these bad terms and generate. Finally, recalculate quota as before and generate the rest of the "good" components.

In some extreme domains the function may fail to generate a point within the domain. Also, it is not guaranteed that the function returns a point in an area with a high probability density.

Value

A p vector inside the domain defined by domain.

Examples

```
p <- 30
poly_d <- function(ex, abs, nng){
   return (make_domain("polynomial", p=p,
                       ineqs=list(list(expression=ex, abs=abs, nonnegative=nng))))
}

random_init_polynomial(poly_d(paste("sum(exp(x))<=", p*1.01), abs=TRUE, nng=TRUE))
random_init_polynomial(poly_d(paste("sum(exp(x))<=", p*1.01), abs=FALSE, nng=FALSE))
random_init_polynomial(poly_d(paste("sum(exp(x))>", p*1.01), abs=TRUE, nng=TRUE))
random_init_polynomial(poly_d(paste("sum(exp(x))>", p*1.01), abs=TRUE, nng=FALSE))
random_init_polynomial(poly_d(paste("sum(log(x))<=", 0.01), abs=TRUE, nng=TRUE))
random_init_polynomial(poly_d(paste("sum(log(x))>", 0.01), abs=TRUE, nng=TRUE))
random_init_polynomial(poly_d(paste("sum(x^2)<=", 0.01), abs=TRUE, nng=TRUE))
random_init_polynomial(poly_d(paste("sum(x^2)>", 0.01), abs=TRUE, nng=TRUE))

random_init_polynomial(poly_d(paste("exp(x)<=", 1.01), abs=TRUE, nng=TRUE))
random_init_polynomial(poly_d(paste("exp(x)<=", 1.01), abs=FALSE, nng=FALSE))
random_init_polynomial(poly_d(paste("exp(x)>", 1.01), abs=TRUE, nng=TRUE))
random_init_polynomial(poly_d(paste("exp(x)>", 1.01), abs=TRUE, nng=FALSE))
```

```
random_init_polynomial(poly_d(paste("log(x)<=", 0.01), abs=TRUE, nng=TRUE))
random_init_polynomial(poly_d(paste("log(x)>", 0.01), abs=TRUE, nng=TRUE))
random_init_polynomial(poly_d(paste("x^2<=", 0.01), abs=TRUE, nng=TRUE))
random_init_polynomial(poly_d(paste("x^2>", 0.01), abs=TRUE, nng=TRUE))

random_init_polynomial(poly_d("x1^2+x2^2+log(x3)<-2", abs=TRUE, nng=TRUE))
random_init_polynomial(poly_d("x1^2+x2^2+log(x3)>-2", abs=FALSE, nng=FALSE))
random_init_polynomial(poly_d("x1^(3/5)+x2^2+x3^(1/3)<-2", abs=FALSE, nng=FALSE))
random_init_polynomial(poly_d("x1^(3/5)+x2^2+x3^(1/3)>-2", abs=FALSE, nng=FALSE))
random_init_polynomial(poly_d("x1^(3/5)+1.2*exp(2*x2)+2.3*exp(-3*x3)<-2", abs=FALSE, nng=FALSE))
random_init_polynomial(poly_d("x1^(3/5)+1.2*exp(2*x2)+2.3*exp(-3*x3)<2", abs=TRUE, nng=FALSE))
random_init_polynomial(poly_d("x1^(3/5)+1.2*exp(2*x2)+2.3*exp(-3*x3)>-2", abs=TRUE, nng=FALSE))
random_init_polynomial(poly_d("x1^(3/5)+2.3*log(x4)+1.3*exp(2*x2)+0.7*exp(-3*x3)<-2",
                       abs=TRUE, nng=FALSE))
random_init_polynomial(poly_d("x1^(3/5)+2.3*log(x4)+1.3*exp(2*x2)+0.7*exp(-3*x3)>-2",
                       abs=FALSE, nng=FALSE))
random_init_polynomial(poly_d(
  "x1^(3/5)+0.9*x2^(2/3)+2.7*x3^(-3/2)+1.1*x4^(-5)+1.1*exp(2x5)+1.3*exp(-3x6)+0.7*log(x7)<-2",
                       abs=TRUE, nng=FALSE))
random_init_polynomial(poly_d(
  "x1^(3/5)+0.9*x2^(2/3)+2.7*x3^(-3/2)+1.1*x4^(-5)+1.1*exp(2x5)+1.3*exp(-3x6)+0.7*log(x7)<-2",
                       abs=FALSE, nng=TRUE))
random_init_polynomial(poly_d(
  "x1^(3/5)+0.9*x2^(2/3)+2.7*x3^(-3/2)+1.1*x4^(-5)+1.1*exp(2x5)+1.3*exp(-3x6)+0.7*log(x7)>-2",
                       abs=TRUE, nng=FALSE))
random_init_polynomial(poly_d(
  "x1^(3/5)+0.9*x2^(2/3)+2.7*x3^(-3/2)+1.1*x4^(-5)+1.1*exp(2x5)+1.3*exp(-3x6)+0.7*log(x7)>2",
                       abs=TRUE, nng=TRUE))
random_init_polynomial(poly_d(
  "x1^(3/5)+0.9*x2^(2/3)+2.7*x3^(-3/2)+1.1*x4^(-5)+1.1*exp(2x5)+1.3*exp(-3x6)+0.7*log(x7)>2",
                       abs=FALSE, nng=FALSE))
```

`random_init_simplex` *Generates a random point in the (p-1)-simplex.*

Description

Generates a random point in the `(p-1)`-simplex.

Usage

```
random_init_simplex(p, rdist = stats::rnorm, tol = 1e-10, maxtimes = 100)
```

Arguments

`p`	An integer, the dimension.
`rdist`	A function that generates a random number when called using `rdist(1)`. Must return a non-zero number with a large enough probability.

tol	A small positive number. Samples are regenerated until each of the p components are at least of size tol.
maxtimes	An integer, maximum number of attempts.

Details

p numbers are generated from rdist and their absolute values are normalized to sum to 1. This will be repeated up to maxtimes times until all p components are larger than or equal to tol.

Value

A random point (p-vector) in the (p-1)-simplex, i.e. sum(x) == 1 && x > 0.

Examples

```
random_init_simplex(100, stats::rnorm, 1e-10, 100)
```

---

random_init_uniform *Generates random numbers from a finite union of intervals.*

Description

Generates random numbers from a finite union of intervals.

Usage

```
random_init_uniform(n, lefts, rights)
```

Arguments

n	An integer, the number of samples to return.
lefts	A vector of numbers, must have the same length as rights. A non-empty vector of numbers (may contain -Inf), the left endpoints of a domain defined as a union of intervals. It is required that lefts[i] <= rights[i] <= lefts[j] for any i < j.
rights	A vector of numbers, must have the same length as lefts. A non-empty vector of numbers (may contain Inf), the right endpoints of a domain defined as a union of intervals. It is required that lefts[i] <= rights[i] <= lefts[j] for any i < j.

Details

For each sample, a random bin i is uniformly chosen from 1 through length(lefts); if the lefts[i] and rights[i] define a finite interval, a random uniform variable is drawn from the interval; if the interval is infinite, a truncated laplace variable with location 0 and scale 1 is drawn. Used for randomly generating initial points for generators of truncated multivariate distributions.

Value

n random numbers from the union of intervals.

Examples

```
hist(random_init_uniform(1e4, -Inf, Inf), breaks=200)
hist(random_init_uniform(1e4, c(0, 5), c(2, Inf)), breaks=200)
hist(random_init_uniform(1e4, c(-Inf, 0, 3), c(-3, 1, 12)), breaks=200)
hist(random_init_uniform(1e4, c(-5, 0), c(-2, 2)), breaks=200)
hist(random_init_uniform(1e4, c(-10, 1), c(-7, 10)), breaks=200)
hist(random_init_uniform(1e4, c(-Inf, 100), c(-100, Inf)), breaks=200)
hist(random_init_uniform(1e4, c(-100, -90), c(-95, -85)), breaks=200)
```

---

ran_mat *Random generator of matrices with given eigenvalues.*

Description

Random generator of matrices with given eigenvalues.

Usage

```
ran_mat(p, ev = stats::runif(p, 0, 10), seed = NULL)
```

Arguments

p	A positive integer >= 2, the dimension.
ev	A vector of length p, the eigenvalues of the output matrix.
seed	A number, the seed for the generator. Ignored if NULL.

Details

The function randomly fills a p by p matrix with independent $Normal(0, 1)$ entries, takes the Q matrix from its QR decomposition, and returns $Q'diag(ev)Q$.

Value

A p by p matrix whose eigenvalues equal to ev.

Examples

```
p <- 30
eigen_values <- (0.1*p-1+1:p)/p
K <- ran_mat(p, ev=eigen_values, seed=1)
sort(eigen(K)$val)-eigen_values
```

`read_exponent` *Parses the exponent part into power_numer and power_denom.*

Description

Parses the exponent part into power_numer and power_denom.

Usage

```
read_exponent(s)
```

Arguments

`s`	A string. Must be of the form "" (empty string), "^2", "^(-5/3)" followed by other terms (starting with "+" or "-").

Details

Parses the exponential part of the first term into power_numer and power_denom and returns the rest of the terms. Please refer to the examples. `s` must be of the form "", "^2", "^(-5/3)" followed by other terms, e.g. "+x2^2", "^2+x2^2", "^(-5/3)+x2^2". Assuming these come from "x1+x2^2", "x1^2+x2^2" and "x1^(-5/3)+x2^2", respectively, these will parsed into `power_numer=1, power_denom=1`, `power_numer=2, power_denom=1`, and `power_numer=-5, power_denom=3`, respectively.

Value

A list containing the following elements:

`power_numer`	An integer, the numerator of the power.
`power_denom`	An integer, the denominator of the power.
`s`	A string, the rest of the unparsed string.

If parsing is unsuccessful, `NULL` is returned.

Examples

```
read_exponent("")
read_exponent("^(-2*4)") # Unsuccessful parsing, returns \code{NULL}.
read_exponent("+x2^(2/3)+x3^(-3/4)") # Comes from "x1+x2^(2/3)+x3^(-3/4)"
read_exponent("^2+x2^(2/3)+x3^(-3/4)") # Comes from "x1^2+x2^(2/3)+x3^(-3/4)"
read_exponent("^(2/3)+x2^(2/3)+x3^(-3/4)") # Comes from "x1^(2/3)+x2^(2/3)+x3^(-3/4)"
read_exponent("^(-2)+x2^(2/3)+x3^(-3/4)") # Comes from "x1^(-2)+x2^(2/3)+x3^(-3/4)"
read_exponent("^(-2/3)+x2^(2/3)+x3^(-3/4)") # Comes from "x1^(-2/3)+x2^(2/3)+x3^(-3/4)"
```

read_exponential *Parses the integer coefficient in an exponential term.*

Description

Parses the integer coefficient in an exponential term.

Usage

```
read_exponential(s, has_index)
```

Arguments

s	A string that starts with one of the following forms: `exp(x)`, `exp(-x)`, `exp(2x)`, `exp(-2x)`, `exp(12*x)`, `exp(-123*x)`, followed by other terms. If `has_index == TRUE`, the first term should be rewritten in `x` with an index (e.g. `exp(x1)`, `exp(-2*x2)`).
has_index	A logical, indicates whether the term is written in a component (e.g. `x1`, `x2`) as opposed to a uniform term (i.e. `x`).

Details

Parses the coefficient in the first exponential term and returns the rest of the terms.

Value

A list containing the following elements:

power_numer	An integer, the integer coefficient inside the first exponential term.
idx	An integer, the index of the term matched (e.g. `3` for `exp(2*x3)`). `NULL` if `has_index == FALSE`.
s	A string, the rest of the unparsed string.

If parsing is unsuccessful, `NULL` is returned.

Examples

```
# Unsuccessful parsing, not starting with exponential, returns \code{NULL}.
read_exponential("x", FALSE)
# Unsuccessful parsing, not starting with exponential, returns \code{NULL}.
read_exponential("x1^2+exp(2x2)", TRUE)
read_exponential("exp(x)", FALSE)
read_exponential("exp(x1)", TRUE)
read_exponential("exp(-x)", FALSE)
read_exponential("exp(-x1)+x2^2", TRUE)
read_exponential("exp(2x)", FALSE)
read_exponential("exp(2x1)+x2^(-2/3)", TRUE)
read_exponential("exp(-2x)", FALSE)
```

```
read_exponential("exp(-2x1)+exp(x3)", TRUE)
read_exponential("exp(12x)", FALSE)
read_exponential("exp(12x2)+x3^(-3)+x4^2", TRUE)
read_exponential("exp(-12x)", FALSE)
read_exponential("exp(-12x3)+x1^(2/5)+log(x2)", TRUE)
read_exponential("exp(123*x)", FALSE)
read_exponential("exp(123*x1)+x2^4", TRUE)
read_exponential("exp(-123*x)", FALSE)
read_exponential("exp(-123*x4)+exp(2*x3)", TRUE)
```

`read_one_term` *Parses the first term of a non-uniform expression.*

Description

Parses the first term of a non-uniform expression.

Usage

```
read_one_term(s)
```

Arguments

`s`	A string, the variable side of a non-uniform inequality expression (i.e. terms must be rewritten in e.g. `x1`, `x2` as opposed to `x`).

Details

Parses the first term in a non-uniform expression and returns the rest of the terms.

Value

A list containing the following elements:

`idx`	An integer, the index of the first term (e.g. `3` for `1.3*x3^(-2/5)`).
`power_numer`	An integer, the power_numer of the first term.
`power_denom`	An integer, the power_denom of the first term.
`coef`	A number, the coefficient on the first term (e.g. `1.3` for `1.3*x3^(-2/5)`).
`s`	A string, the rest of the unparsed string.

Examples

```
read_one_term("0.5*x1+x2^2")
read_one_term("2e3x1^(2/3)-1.3x2^(-3)")
read_one_term("2exp(3x1)+2.3*x2^2")
read_one_term(paste(sapply(1:10, function(j){paste(j, "x", j, "^", (11-j), sep="")}), collapse="+"))
read_one_term("0.5*x1^(-2/3)-x3^3 + 2log(x2)- 1.3e4exp(-25*x6)+x8-.3x5^(-3/-4)")
read_one_term("-1e-4x1^(-2/3)-x2^(4/-6)+2e3x3^(-6/9) < 3.5e5")
```

read_uniform_term *Attempts to parse a single term in x into power_numer and power_denom.*

Description

Attempts to parse a single term in x into power_numer and power_denom.

Usage

```
read_uniform_term(s)
```

Arguments

s	A string, the variable side of an inequality expression. Please refer to `make_domain()`.

Details

Returns `NULL` if `s` is not a single uniform term in `x` (e.g. `x^2` is uniform, while `x1^2+x2^2` is not uniform).

Value

power_numers	The uniform numerator in the power (e.g. `-2` for `x^(-2/3)`).
power_denoms	The uniform denominator in the power (e.g. `3` for `x^(-2/3)`).

Examples

```
p <- 30
read_uniform_term("x^2")
read_uniform_term("x^(1/3)")
read_uniform_term("exp(x)")
read_uniform_term("log(x)")
read_uniform_term("x^(-2/3)")
read_uniform_term("x")
read_uniform_term("exp(-23x)")
```

refit *Loss for a refitted (restricted) unpenalized model*

Description

Refits an unpenalized model restricted to the estimated edges, with `lambda1=0`, `lambda2=0` and `diagonal_multiplier=1`. Returns `Inf` if no unique solution exists (when the loss is unbounded from below or has infinitely many minimizers).

Usage

```
refit(res, elts)
```

Arguments

res	A list of results returned by get_results().
elts	A list, elements necessary for calculations returned by get_elts().

Details

Currently the function only returns Inf when the maximum node degree is >= the sample size, which is a sufficient and necessary condition for nonexistence of a unique solution with probability 1 if symmetric != "symmetric". In practice it is also a sufficient and necessary condition for most cases and symmetric == "symmetric".

Value

A number, the loss of the refitted model.

Examples

```
# Examples are shown for Gaussian truncated to R+^p only. For other distributions
#   on other types of domains, please refer to \code{gen()} or \code{get_elts()},
#   as the way to call this function (\code{refit()}) is exactly the same in those cases.
n <- 50
p <- 30
domain <- make_domain("R+", p=p)
h_hp <- get_h_hp("min_pow", 1, 3)
mu <- rep(0, p)
K <- diag(p)
dm <- 1 + (1-1/(1+4*exp(1)*max(6*log(p)/n, sqrt(6*log(p)/n))))
x <- tmvtnorm::rtmvnorm(n, mean = mu, sigma = solve(K),
       lower = rep(0, p), upper = rep(Inf, p), algorithm = "gibbs",
       burn.in.samples = 100, thinning = 10)

elts_gauss_np <- get_elts(h_hp, x, setting="gaussian", domain=domain,
                centered=FALSE, profiled=FALSE, diag=dm)
res_nc_np <- get_results(elts_gauss_np, symmetric="symmetric",
               lambda1=0.35, lambda2=2, previous_res=NULL, is_refit=FALSE)
refit(res_nc_np, elts_gauss_np)

elts_gauss_p <- get_elts(h_hp, x, setting="gaussian", domain=domain,
               centered=FALSE, profiled=TRUE, diag=dm)
res_nc_p <- get_results(elts_gauss_p, symmetric="symmetric",
              lambda1=0.35, lambda2=NULL, previous_res=NULL, is_refit=FALSE)
refit(res_nc_p, elts_gauss_p)

elts_gauss_c <- get_elts(h_hp, x, setting="gaussian", domain=domain,
            centered=TRUE, diag=dm)
res_c <- get_results(elts_gauss_c, symmetric="or", lambda1=0.35,
           lambda2=NULL, previous_res=NULL, is_refit=FALSE)
```

```
refit(res_c, elts_gauss_c)
```

`rexp_truncated` *Generates translated and truncated exponential variables.*

Description

Generates translated and truncated exponential variables.

Usage

```
rexp_truncated(n, lo, hi)
```

Arguments

`n`	An integer, the number of samples to return.
`lo`	A double, the lower limit of the distribution, cannot be `-Inf`.
`hi`	A double, the upper limit of the distribution.

Details

Returns `n` random variables from the translated truncated exponential distribution with density $\exp(-(x - lo))/(1 - \exp(lo - hi))$ on `[lo,hi]`.

Value

`n` random variables from the translated truncated exponential distribution.

Examples

```
hist(rexp_truncated(1e4, 0, Inf), breaks=200)
hist(rexp_truncated(1e4, 10, 12), breaks=200)
hist(rexp_truncated(1e4, -2, 2), breaks=200)
hist(rexp_truncated(1e4, -10, 0), breaks=200)
hist(rexp_truncated(1e4, -100, Inf), breaks=200)
hist(rexp_truncated(1e4, -100, -95), breaks=200)
```

`rlaplace_truncated` *Generates laplace variables truncated to a finite union of intervals.*

Description

Generates laplace variables truncated to a finite union of intervals.

Usage

```
rlaplace_truncated(n, lefts, rights, m = 0, s = 1)
```

Arguments

`n`	An integer, the number of samples to return.
`lefts`	A vector of numbers, must have the same length as `rights`. A non-empty vector of numbers (may contain `-Inf`), the left endpoints of a domain defined as a union of intervals. It is required that `lefts[i] <= rights[i] <= lefts[j]` for any `i < j`.
`rights`	A vector of numbers, must have the same length as `lefts`. A non-empty vector of numbers (may contain `Inf`), the right endpoints of a domain defined as a union of intervals. It is required that `lefts[i] <= rights[i] <= lefts[j]` for any `i < j`.
`m`	A number, the location parameter of the laplace distribution.
`s`	A number, the scale/dispersion parameter of the laplace distribution.

Details

Returns `n` random variables from the truncated laplace distribution with density proportional to $\exp(-|x-m|/s)$ truncated to the domain defined by the union of `[lefts[i], rights[i]]`.

Value

`n` random variables from the truncated laplace distribution.

Examples

```
hist(rlaplace_truncated(1e4, -Inf, Inf), breaks=200)
hist(rlaplace_truncated(1e4, c(0, 5), c(2, Inf), m=2, s=3), breaks=200)
hist(rlaplace_truncated(1e4, c(-Inf, 0, 3), c(-3, 1, 12), m=8, s=4), breaks=200)
hist(rlaplace_truncated(1e4, c(-5, 0), c(-2, 2), s=0.8), breaks=200)
hist(rlaplace_truncated(1e4, c(-10, 1), c(-7, 10), m=-4), breaks=200)
hist(rlaplace_truncated(1e4, c(-Inf, 100), c(-100, Inf), m=100), breaks=200)
hist(rlaplace_truncated(1e4, c(-Inf, 100), c(-100, Inf), m=-100), breaks=200)
hist(rlaplace_truncated(1e4, c(-100, -90), c(-95, -85), s=2), breaks=200)
```

`rlaplace_truncated_centered` *Generates centered laplace variables with scale 1.*

Description

Generates centered laplace variables with scale 1.

Usage

```
rlaplace_truncated_centered(n, lo, hi)
```

Arguments

`n`	An integer, the number of samples to return.
`lo`	A double, the lower limit of the distribution.
`hi`	A double, the upper limit of the distribution.

Details

Returns `n` random variables from the truncated laplace distribution with density proportional to $\exp(-|x|)$ on `[lo,hi]`.

Value

`n` random variables from the truncated laplace distribution.

Examples

```
hist(rlaplace_truncated_centered(1e4, -Inf, Inf), breaks=200)
hist(rlaplace_truncated_centered(1e4, 0, Inf), breaks=200)
hist(rlaplace_truncated_centered(1e4, 10, 12), breaks=200)
hist(rlaplace_truncated_centered(1e4, -2, 2), breaks=200)
hist(rlaplace_truncated_centered(1e4, -10, 0), breaks=200)
hist(rlaplace_truncated_centered(1e4, -100, Inf), breaks=200)
hist(rlaplace_truncated_centered(1e4, -100, -95), breaks=200)
```

search_bin *Finds the index of the bin a number belongs to.*

Description

Finds the index of the bin a number belongs to.

Usage

```
search_bin(arr, x)
```

Arguments

arr	A vector of size at least 2.
x	A number. Must be within the range of [arr[1], arr[length(arr)]].

Details

Finds the smallest index i such that arr[i] <= x <= arr[i+1]. Calls binarySearch_bin() if length(arr) > 8 and calls naiveSearch_bin() otherwise.

Value

The index i such that arr[i] <= x <= arr[i+1].

Examples

```
search_bin(1:10, seq(1, 10, by=0.5))
```

s_at *Returns the character at a position of a string.*

Description

Returns the character at a position of a string.

Usage

```
s_at(string, position)
```

Arguments

string	A string.
position	A positive number.

Details

Calls `substr(string, position, position)`.

Value

A character

Examples

```
s_at("123", 1)
s_at("123", 2)
s_at("123", 3)
s_at("123", 4)
s_at("123", 0)
```

`s_output` *Helper function for outputting if verbose.*

Description

Helper function for outputting if verbose.

Usage

```
s_output(out, verbose, verbosetext)
```

Arguments

`out`	Text string.
`verbose`	Boolean.
`verbosetext`	Text string.

Value

If `verbose == TRUE`, outputs a string that concatenates `verbosetext` and `out`.

`test_lambda_bounds` *Searches for a tight bound for $\lambda\_\boldsymbol{K}$ that gives the empty or complete graph starting from a given lambda with a given step size*

Description

Searches for the smallest lambda that gives the empty graph (if `lower == FALSE`) or the largest that gives the complete graph (if `lower == TRUE`) starting from the given lambda, each time updating by multiplying or dividing by `step` depending on the search direction.

Usage

```
test_lambda_bounds(
  elts,
  symmetric,
  lambda = 1,
  lambda_ratio = 1,
  step = 2,
  lower = TRUE,
  verbose = TRUE,
  tol = 1e-06,
  maxit = 10000,
  cur_res = NULL
)
```

Arguments

`elts`	A list, elements necessary for calculations returned by `get_elts()`.
`symmetric`	A string. If equals `"symmetric"`, estimates the minimizer $\mathbf{K}$ over all symmetric matrices; if `"and"` or `"or"`, use the "and"/"or" rule to get the support
`lambda`	A number, the initial searching point for $\lambda_{\mathbf{K}}$.
`lambda_ratio`	A positive number (or `Inf`), the fixed ratio $\lambda_{\mathbf{K}}$ and $\lambda_{\boldsymbol{\eta}}$, if $\lambda_{\boldsymbol{\eta}} \neq 0$ (non-profiled) in the non-centered setting.
`step`	A number, the multiplicative constant applied to lambda at each iteration. Must be strictly larger than 1.
`lower`	A boolean. If `TRUE`, finds the largest possible lambda that gives the complete graph (a $lower$ bound). If `FALSE`, finds the smallest possible lambda that gives the empty graph (an $upper$ bound).
`verbose`	Optional. A boolean. If `TRUE`, prints out the lambda value at each iteration.
`tol`	Optional. A number, the tolerance parameter.
`maxit`	Optional. A positive integer, the maximum number of iterations in model fitting for each lambda.
`cur_res`	Optional. A list, current results returned from a previous lambda. If provided, used as a warm start. Default to `NULL`.

Value

lambda	A number, the best lambda that produces the desired number of edges. 1e-10 or 1e15 is returned if out of bound.
cur_res	A list, results for this lambda. May be NULL if lambda is out of bound.

Examples

```
# Examples are shown for Gaussian truncated to R+^p only. For other distributions
#   on other types of domains, please refer to \code{gen()} or \code{get_elts()}, as the
#  way to call this function (\code{test_lambda_bounds()}) is exactly the same in those cases.
n <- 50
p <- 30
domain <- make_domain("R+", p=p)
mu <- rep(0, p)
K <- diag(p)
x <- tmvtnorm::rtmvnorm(n, mean = mu, sigma = solve(K),
       lower = rep(0, p), upper = rep(Inf, p), algorithm = "gibbs",
       burn.in.samples = 100, thinning = 10)

h_hp <- get_h_hp("min_pow", 1, 3)
dm <- 1 + (1-1/(1+4*exp(1)*max(6*log(p)/n, sqrt(6*log(p)/n))))
elts_gauss_np <- get_elts(h_hp, x, setting="gaussian", domain=domain,
                 centered=FALSE, profiled=FALSE, diag=dm)
lambda_cur_res <- test_lambda_bounds(elts_gauss_np, "symmetric", lambda=1,
                  lambda_ratio=1, step=1.5, lower=TRUE, cur_res=NULL)
lambda_cur_res2 <- test_lambda_bounds(elts_gauss_np, "symmetric", lambda=1,
                  lambda_ratio=1, step=1.5, lower=FALSE, cur_res=lambda_cur_res$cur_res)
```

---

test_lambda_bounds2 *Searches for a tight bound for $\lambda\_\boldsymbol{K}$ that gives the empty or complete graph starting from a given lambda*

Description

Searches for the smallest lambda that gives the empty graph (if lower == FALSE) or the largest that gives the complete graph (if lower == TRUE) starting from the given lambda.

Usage

```
test_lambda_bounds2(
  elts,
  symmetric,
  lambda_ratio = Inf,
  lower = TRUE,
  verbose = TRUE,
  tol = 1e-06,
  maxit = 10000,
  lambda_start = NULL
)
```

Arguments

elts	A list, elements necessary for calculations returned by get_elts().
symmetric	A string. If equals "symmetric", estimates the minimizer $\mathbf{K}$ over all symmetric matrices; if "and" or "or", use the "and"/"or" rule to get the support
lambda_ratio	A positive number (or Inf), the fixed ratio $\lambda_{\mathbf{K}}$ and $\lambda_{\boldsymbol{\eta}}$, if $\lambda_{\boldsymbol{\eta}} \neq 0$ (non-profiled) in the non-centered setting.
lower	A boolean. If TRUE, finds the largest possible lambda that gives the complete graph (a $lower$ bound). If FALSE, finds the smallest possible lambda that gives the empty graph (an $upper$ bound).
verbose	Optional. A boolean. If TRUE, prints out the lambda value at each iteration.
tol	Optional. A number, the tolerance parameter.
maxit	Optional. A positive integer, the maximum number of iterations in model fitting for each lambda.
lambda_start	Optional. A number, the starting point for searching. If NULL, set to 1e-4 if lower == TRUE, or 1 if lower == FALSE.

Details

This function calls test_lambda_bounds three times with step set to 10, 10^(1/4), 10^(1/16), respectively.

Value

A number, the best lambda that produces the desired number of edges. 1e-10 or 1e15 is returned if out of bound.

Examples

```
# Examples are shown for Gaussian truncated to R+^p only. For other distributions
#   on other types of domains, please refer to \code{gen()} or \code{get_elts()}, as the
#  way to call this function (\code{test_lambda_bounds2()}) is exactly the same in those cases.
n <- 50
p <- 30
domain <- make_domain("R+", p=p)
mu <- rep(0, p)
K <- diag(p)
x <- tmvtnorm::rtmvnorm(n, mean = mu, sigma = solve(K),
       lower = rep(0, p), upper = rep(Inf, p), algorithm = "gibbs",
       burn.in.samples = 100, thinning = 10)
h_hp <- get_h_hp("min_pow", 1, 3)
dm <- 1 + (1-1/(1+4*exp(1)*max(6*log(p)/n, sqrt(6*log(p)/n))))
elts_gauss_np <- get_elts(h_hp, x, setting="gaussian", domain=domain,
                centered=FALSE, profiled=FALSE, diag=dm)
test_lambda_bounds2(elts_gauss_np, "symmetric", lambda_ratio=2,
     lower=TRUE, verbose=TRUE, lambda_start=NULL)
test_lambda_bounds2(elts_gauss_np, "symmetric", lambda_ratio=2,
     lower=FALSE, verbose=TRUE, lambda_start=1.0)
```

tp_fp *Calculates the true and false positive rates given the estimated and true edges.*

Description

Calculates the true and false positive rates given the estimated and true edges.

Usage

```
tp_fp(edges, true_edges, p)
```

Arguments

edges	A vector of indices corresponding to the estimated edges. Should not contain the diagonals.
true_edges	A vector of indices corresponding to the true edges.
p	A positive integer, the dimension.

Value

A vector containing the true positive rate and the false positive rate.

Examples

```
n <- 40
p <- 50
mu <- rep(0, p)
tol <- 1e-8
K <- cov_cons(mode="sub", p=p, seed=1, spars=0.2, eig=0.1, subgraphs=10)
true_edges <- which(abs(K) > tol & diag(p) == 0)
dm <- 1 + (1-1/(1+4*exp(1)*max(6*log(p)/n, sqrt(6*log(p)/n))))
set.seed(1)
domain <- make_domain("R+", p=p)
x <- tmvtnorm::rtmvnorm(n, mean = mu, sigma = solve(K),
       lower = rep(0, p), upper = rep(Inf, p), algorithm = "gibbs",
       burn.in.samples = 100, thinning = 10)
est <- estimate(x, setting="gaussian", elts=NULL, domain=domain, centered=TRUE,
         symmetric="symmetric", lambda_length=100, mode="min_pow",
         param1=1, param2=3, diagonal_multiplier=dm, verbose=FALSE)
# Apply tp_fp to each estimated edges set for each lambda
TP_FP <- t(sapply(est$edgess, function(edges){tp_fp(edges, true_edges, p)}))
old.par <- par(mfrow=c(1,1), mar=c(5,5,5,5))
plot(c(), c(),  ylim=c(0,1), xlim=c(0,1), cex.lab=1, main = "ROC curve",
  xlab="False Positives", ylab="True Positives")
points(TP_FP[,2], TP_FP[,1], type="l")
points(c(0,1), c(0,1), type = "l", lty = 2)
par(old.par)
```

update_finite_infinity_for_uniform *Maximum between finite_infinity and 10 times the max abs value of finite elements in* lefts *and* rights.

Description

Maximum between finite_infinity and 10 times the max abs value of finite elements in lefts and rights.

Usage

```
update_finite_infinity_for_uniform(lefts, rights, finite_infinity)
```

Arguments

lefts	A non-empty vector of numbers (may contain -Inf), the left endpoints of a domain defined as a union of intervals. Must pass check_endpoints(lefts, rights).
rights	A non-empty vector of numbers (may contain Inf), the right endpoints of a domain defined as a union of intervals. Must pass check_endpoints(lefts, rights).
finite_infinity	A finite positive number. Inf will be truncated to finite_infinity if applicable. See details.

Details

Since we assume that lefts[i] <= rights[i] <= lefts[j] for any i < j, the function takes the maximum between finite_infinity and 10 times the absolute values of lefts[1], lefts[length(lefts)], rights[1], and rights[length(rights)], if they are finite.

Value

A double, larger than or equal to finite_infinity.

Examples

```
# Does not change since 1000 > 12 * 10
update_finite_infinity_for_uniform(c(-10,-5,0,5,9), c(-8,-3,2,7,12), 1000)
# Changed to 12 * 10
update_finite_infinity_for_uniform(c(-10,-5,0,5,9), c(-8,-3,2,7,12), 10)
# Changed to 12 * 10
update_finite_infinity_for_uniform(c(-Inf,-5,0,5,9), c(-8,-3,2,7,12), 10)
# Changed to 9 * 10
update_finite_infinity_for_uniform(c(-Inf,-5,0,5,9), c(-8,-3,2,7,Inf), 10)
```

varhat — *Asymptotic variance (times* n*) of the estimator for* mu *or* sigmasq *for the univariate normal on a general domain assuming the other parameter is known.*

Description

Asymptotic variance (times n) of the estimator for mu or sigmasq for the univariate normal on a general domain assuming the other parameter is known.

Usage

```
varhat(mu, sigmasq, mode, param1, param2, est_mu, domain, tol = 1e-10)
```

Arguments

mu	A number, the true mu parameter.
sigmasq	A number, the true sigmasq parameter.
mode	A string, the class of the h function.
param1	A number, the first parameter to the h function.
param2	A number, the second parameter (may be optional depending on mode) to the h function.
est_mu	A boolean. If TRUE, returns the asymptotic variance of muhat assuming sigmasq is known; if FALSE, returns the asymptotic variance of sigmasqhat assuming mu is known.
domain	A list returned from make_domain() that represents the domain.
tol	A positive number, tolerance for numerical integration. Defaults to 1e-10.

Details

The estimates may be off from the empirical variance, or may even be Inf or NaN if "mode" is one of "cosh", "exp", and "sinh") as the functions grow too fast. If est_mu == TRUE, the function numerically calculates

$$E\left[\sigma^2 h^2(X) + \sigma^4 h'^2(X)\right] / E^2[h(X)],$$

and if est_mu == FALSE, the function numerically calculates

$$E\left[\left(2\sigma^6 h^2(X) + \sigma^8 h'^2(X)\right)(X-\mu)^2\right] / E^2\left[h(X)(X-\mu)^2\right],$$

where E is the expectation over the true distribution $TN(\mu, \sigma)$ of X.

Value

A number, the asymptotic variance.

Examples

```
varhat(0, 1, "min_log_pow", 1, 1, TRUE, make_domain("R+", 1))
varhat(0.5, 4, "min_pow", 1, 1, TRUE, make_domain("R+", 1))
```

Index