Package: smoothemplik (via r-universe)

June 10, 2026

Type Package

Title Smoothed Empirical Likelihood

Version 0.0.17

Maintainer Andreï Victorovitch Kostyrka <`andrei.kostyrka@gmail.com`>

Description Empirical likelihood methods for asymptotically efficient estimation of models based on conditional or unconditional moment restrictions; see Kitamura, Tripathi & Ahn (2004) <doi:10.1111/j.1468-0262.2004.00550.x> and Owen (2013) <doi:10.1002/cjs.11183>. Kernel-based non-parametric methods for density/regression estimation and numerical routines for empirical likelihood maximisation are implemented in 'Rcpp' for speed.

License EUPL

Encoding UTF-8

URL `https://github.com/Fifis/smoothemplik`

BugReports `https://github.com/Fifis/smoothemplik/issues`

Depends R (>= 3.0.0)

Imports parallel, Rcpp, RcppParallel, Rdpack, Matrix, data.table

Suggests knitr, rmarkdown, testthat (>= 3.0.0), xml2

RdMacros Rdpack

RoxygenNote 7.3.2

NeedsCompilation yes

LinkingTo Rcpp, RcppArmadillo, RcppParallel

SystemRequirements GNU make

VignetteBuilder knitr

Config/testthat/edition 3

Config/pak/sysreqs make

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

Date/Publication 2025-12-12 09:58:21 UTC

RemoteUrl https://github.com/fifis/smoothemplik

RemoteRef HEAD

RemoteSha 38abae6d526b09c751c05edffc7b322dbc1f78ed

Contents

`bartlettFactor` *Bartlett correction factor for empirical likelihood with estimating equations*

Description

Compute the Bartlett correction factor b for empirical likelihood based on the moment conditions $E\{g(X;\theta_0)\} = 0$. The function implements the rotation in (Liu and Chen 2010) and evaluates b either from raw moments (unadjusted) or from the bias-reduced moment estimators recommended in their paper.

Usage

```
bartlettFactor(x, centre = TRUE, bias.adj = TRUE)
```

Arguments

`x`	Numeric vector or matrix of estimating functions. If a matrix, rows are observations and columns are the components of g.
`centre`	Logical. If ‘TRUE‘ (default), centre each column of ‘x‘ by its sample mean before computing the correction (this corresponds to plugging in a consistent $\hat\theta$ so that $n^{-1}\sum g_i(\hat\theta) \approx 0$).
`bias.adj`	Logical. If ‘TRUE‘ (default), use the bias-reduced moment estimators. When $n \le 4$, the adjustment is disabled automatically.

Details

Let $V(\theta) = \mathrm{Var}\{g(X,\theta)\}$, and let P be the orthogonal matrix of eigenvectors of $V(\hat\theta)$. Define the rotated variables $Y_i = g_i(\hat\theta)P$ (observations in rows), and write $\alpha^{rs\cdots t} = E(Y^r Y^s \cdots Y^t)$ with $\alpha^{rr} = E(Y_r^2)$.

The Bartlett factor (Theorem 1 of (Liu and Chen 2010)) can be written compactly as

$$b = \frac{1}{q}\Big\{\frac{1}{2}\sum_{r,s}\frac{\alpha^{rrss}}{\alpha^{rr}\alpha^{ss}} - \frac{1}{3}\sum_{r,s,t}\frac{(\alpha^{rst})^2}{\alpha^{rr}\alpha^{ss}\alpha^{tt}}\Big\},$$

where q is the dimension of g. The first double sum is over all pairs `(r, s)`, and the triple sum is over all triples `(r, s, t)`.

For adjusted-EL applications, the implementation also uses the equivalent decomposition `b = b_1 - b_2`.

When `bias.adj = TRUE`, all moments are replaced by the bias-reduced estimators given in Eq. (10) and the table beneath it in (Liu and Chen 2010).

Value

Numeric scalar: the estimated Bartlett correction factor b. For multivariate inputs, the value has an attribute `"components"` equal to `c(b1, b2)` where `b = b_1 - b_2`. If `bias.adj = TRUE`, attributes `"unadjusted"` and `"unadjusted.components"` store the corresponding unadjusted estimates.

References

Liu Y, Chen J (2010). "Adjusted empirical likelihood with high-order precision." *The Annals of Statistics*, **38**(3). ISSN 0090-5364. doi:10.1214/09aos750.

Examples

```
set.seed(1)

# One-dimensional: Bartlett factor for the mean
x <- rchisq(50, df = 4)
bartlettFactor(x)  # Bias-adjusted
bartlettFactor(x, bias.adj=FALSE)

# Multi-variate g(X; theta): columns are components of g
n <- 100
g <- cbind(rchisq(n, 4)-4, rchisq(n, 3)-3, rchisq(n, 6)-6, rnorm(n))
bartlettFactor(g)  # Bias-adjusted, centred
bartlettFactor(g, centre = FALSE)  # The true average was used in g
```

`brentMin` *Brent's local minimisation*

Description

Brent's local minimisation

Usage

```
brentMin(
  f,
  interval,
  lower = NA_real_,
  upper = NA_real_,
  tol = 1e-08,
  maxiter = 200L,
  trace = 0L
)
```

Arguments

`f`	A function to be minimised on an interval.
`interval`	A length-2 vector containing the end-points of the search interval.
`lower`	Scalar: the lower end point of the search interval. Not necessary if `interval` is provided.
`upper`	Scalar: the upper end point of the search interval. Not necessary if `interval` is provided.

`tol`	Small positive scalar: stopping criterion. The search stops when the distance between the current candidate and the midpoint of the bracket is smaller than the dynamic threshold `2 * (sqrt(DBL_EPSILON) * abs(x) + tol)`
`maxiter`	Positive integer: the maximum number of iterations.
`trace`	Integer: 0, 1, or 2. Amount of tracing information on the optimisation progress printed. `trace = 0` produces no output, `trace = 1` reports the starting and final results, and `trace = 2` provides detailed iteration-level output.

Details

This is an adaptation of the implementation by John Burkardt (currently available at [https://people.math.sc.edu/Burkardt/m_s

This function is similar to `local_min` or `R_zeroin2`-style logic, but with the following additions: the number of iterations is tracked, and the algorithm stops when the standard Brent criterion is met or if the maximum iteration count is reached. The code stores the approximate final bracket width in `estim.prec`, like in [uniroot()]. If the minimiser is pinned to an end point, `estim.prec = NA`.

There are no preliminary iterations, unlike [brentZero()].

TODO: add preliminary iterations.

Value

A list with the following elements:

root Location of the minimum.

f.root Function value at the minimuim location.

iter Total iteration count used.

estim.prec Estimate of the final bracket size.

Examples

```
f <- function (x) (x - 1/3)^2
brentMin(f, c(0, 1), tol = 0.0001)
brentMin(function(x) x^2*(x-1), lower = 0, upper = 10, trace = 1)
```

`brentZero` *Brent's local root search with extended capabilities*

Description

Brent's local root search with extended capabilities

Usage

```
brentZero(
  f,
  interval,
  lower = NA_real_,
  upper = NA_real_,
  f_lower = NULL,
  f_upper = NULL,
  extendInt = "no",
  tol = 1e-08,
  maxiter = 500L,
  trace = 0L
)
```

Arguments

`f`	The function for which the root is sought.
`interval`	A length-2 vector containing the end-points of the search interval
`lower`	Scalar: the lower end point of the search interval. Not necessary if `interval` is provided.
`upper`	Scalar: the upper end point of the search interval. Not necessary if `interval` is provided.
`f_lower`	Scalar: same as f(upper). Passing this value saves time if f(lower) is slow to compute and is known.
`f_upper`	Scalar: same as f(lower).
`extendInt`	Character: `"no"` Do not extend the interval (default). `"yes"` Attempt to extend both ends until a sign change is found. `"upX"` Assumes the function is increasing around the root and extends upward if needed. `"downX"` Assumes the function is decreasing around the root and extends downward if needed. `"right"` Attempt to extend the upper (right) end until a sign change is found. `"left"` Attempt to extend the lower (left) end until a sign change is found. This behavior mirrors that of [uniroot()].
`tol`	Small positive scalar: convergence tolerance. The search stops when the bracket size is smaller than `2 * .Machine$double.eps * abs(x) + tol`, or if the function evaluates to zero at the candidate root.
`maxiter`	Positive integer: the maximum number of iterations before stopping.
`trace`	Integer: 0, 1, or 2. Controls the verbosity of the output. `trace = 0` produces no output, `trace = 1` reports the starting and final results, and `trace = 2` provides detailed iteration-level output.

Value

A list with the following elements:

root Location of the root.

f.root Function value at the root.

iter Total iteration count used.

init.it Number of initial `extendInt` iterations if there were any; NA otherwise.

estim.prec Estimate of the final bracket size.

exitcode 0 for success, 1 for maximum initial iteration limit, 2 for maximum main iteration limit.

Examples

```
f <- function (x, a) x - a
str(uniroot(f, c(0, 1), tol = 0.0001, a = 1/3))
uniroot(function(x) cos(x) - x, lower = -pi, upper = pi, tol = 1e-9)$root

# New capabilities: extending only one end of the interval
f <- function(x) x^2 - 1  # The roots are -1 and 1
brentZero(f, c(2, 3), extendInt = "left")
brentZero(f, c(2, 3), extendInt = "yes")
brentZero(f, c(2, 3), extendInt = "upX")
brentZero(f, c(0, 0.5), extendInt = "downX")  # This one finds the left crossing

# This function is faster than the base R uniroot, and this is the primary
# reason why it was written in C++
system.time(replicate(1000, { shift <- runif(1, 0, 2*pi)
  uniroot(function(x) cos(x+shift) - x, lower = -pi, upper = pi)
}))
system.time(replicate(1000, { shift <- runif(1, 0, 2*pi)
  brentZero(function(x) cos(x+shift) - x, lower = -pi, upper = pi)
}))
# Roughly twice as fast
```

`bw.CV` *Bandwidth Selectors for Kernel Density Estimation*

Description

Finds the optimal bandwidth by minimising the density cross-valication or least-squares criteria. Remember that since usually, the CV function is highly non-linear, the return value should be taken with a grain of salt. With non-smooth kernels (such as uniform), it will oftern return the local minimum after starting from a reasonable value. The user might want to standardise the input matrix `x` by column (divide by some estimator of scale, like `sd` or `IQR`) and examine the behaviour of the CV criterion as a function of unique bandwidth (`same` argument). If it seems that the optimum is unique, then they may proceed by multiplying the bandwidth by the scale measure, and start the search for the optimal bandwidth in multiple dimensions.

Usage

```
bw.CV(
  x,
  y = NULL,
  weights = NULL,
  kernel = "gaussian",
  order = 2,
  PIT = FALSE,
  chunks = 0,
  robust.iterations = 0,
  degree = 0,
  start.bw = NULL,
  same = FALSE,
  tol = 1e-04,
  try.grid = TRUE,
  ndeps = 1e-05,
  verbose = FALSE,
  attach.attributes = FALSE,
  control = list()
)
```

Arguments

`x`	A numeric vector, matrix, or data frame containing observations. For density, the points used to compute the density. For kernel regression, the points corresponding to explanatory variables.
`y`	A numeric vector of responses (dependent variable) if the user wants least-squares cross-validation.
`weights`	A numeric vector of observation weights (typically counts) to perform weighted operations. If null, `rep(1, NROW(x))` is used. In all calculations, the total number of observations is assumed to be the sum of `weights`.
`kernel`	Character describing the desired kernel type. NB: due to limited machine precision, even Gaussian has finite support.
`order`	An integer: 2, 4, or 6. Order-2 kernels are the standard kernels that are positive everywhere. Orders 4 and 6 produce some negative values, which reduces bias but may hamper density estimation.
`PIT`	If TRUE, the Probability Integral Transform (PIT) is applied to all columns of `x` via `ecdf` in order to map all values into the [0, 1] range. May be an integer vector of indices of columns to which the PIT should be applied.
`chunks`	Integer: the number of chunks to split the task into (limits RAM usage but increases overhead). `0` = auto-select (making sure that no matrix has more than 2^27 elements).
`robust.iterations`	Passed to `kernelSmooth` if `y` is not `NULL` (for least-squares CV).
`degree`	Passed to `kernelSmooth` if `y` is not `NULL` (for least-squares CV).
`start.bw`	Numeric vector: initial value for bandwidth search.

`same`	Logical: use the same bandwidth for all columns of `x`?
`tol`	Relative tolerance used by the optimiser as the stopping criterion.
`try.grid`	Logical: if true, 10 different bandwidths around the rule-of-thumb one are tried with multiplier `1.2^(-3:6)`
`ndeps`	Numerical-difference epsilon. Puts a lower bound on the result: the estimated optimal bw cannot be less than this value.
`verbose`	Logical: print out the optimiser return code for diagnostics?
`attach.attributes`	Logical: if TRUE, returns the output of ‘optim()‘ for diagnostics.
`control`	List: extra arguments to pass to the control-argument list of ‘optim‘.

Details

If `y` is NULL and only `x` is supplied, returns the density-cross-validated bandwidth (DCV). If `y` is supplied, then, returns the least-squares-cross-validated bandwidth (LSCV).

Value

Numeric vector or scalar of the optimal bandwidth.

Examples

```
set.seed(1)  # Creating a data set with many duplicates
n.uniq <- 200
n <- 500
inds <- sort(ceiling(runif(n, 0, n.uniq)))
x.uniq <- sort(rnorm(n.uniq))
y.uniq <- 1 + 0.1*x.uniq + sin(x.uniq) + rnorm(n.uniq)
x <- x.uniq[inds]
y <- y.uniq[inds]
w <- 1 + runif(n, 0, 2) # Relative importance
data.table::setDTthreads(1) # For measuring the pure gains and overhead
RcppParallel::setThreadOptions(numThreads = 1)
bw.grid <- seq(0.1, 1.3, 0.2)
CV <- LSCV(x, y, bw.grid, weights = w)
bw.init <- bw.grid[which.min(CV)]
bw.opt <- bw.CV(x, y, w) # 0.49, very close
g <- seq(-3.5, 3.5, 0.05)
yhat <- kernelSmooth(x, y, g, w, bw.opt, deduplicate.xout = FALSE)
oldpar <- par(mfrow = c(2, 1), mar = c(2, 2, 2, 0)+.1)
plot(bw.grid, CV, bty = "n", xlab = "", ylab = "", main = "Cross-validation")
points(bw.opt, LSCV(x, y, bw.opt, w), col = 2, pch = 15)
plot(x.uniq, y.uniq, bty = "n", xlab = "", ylab = "", main = "Optimal fit")
points(g, yhat, pch = 16, col = 2, cex = 0.5)
par(oldpar)
```

`bw.rot` *Silverman's rule-of-thumb bandwidth*

Description

A fail-safe function that would return a nice Silverman-like bandwidth suggestion for data for which the standard deviation might be NA or 0.

Usage

```
bw.rot(
  x,
  kernel = c("gaussian", "uniform", "triangular", "epanechnikov", "quartic"),
  na.rm = FALSE,
  robust = TRUE,
  discontinuous = FALSE
)
```

Arguments

`x`	A numeric vector without non-finite values.
`kernel`	A string character: `"gaussian"`, `"uniform"`, `"triangular"`, `"epanechnikov"`, or `"quartic"`.
`na.rm`	Logical: should missing values be removed? Setting it to TRUE may cause issues because variable-wise removal of NAs may return a bandwidth that is inappropriate for the final data set for which it is suggested.
`robust`	Logical: safeguard against extreme observations? If TRUE, uses `min(sd(x), IQR(x)/1.34)` to estimate the spread.
`discontinuous`	Logical: if the true density is discontinuous (i.e. has jumps), then, the formula for the optimal bandwidth for density estimation changes.

Details

$\Sigma = \mathrm{diag}(\sigma_k^2)$ with $\det \Sigma = \prod_k \sigma_k^2$ and $\Sigma^{-1} = \mathrm{diag}(1/\sigma_k^2)$). Then, the formula 4.12 in Silverman (1986) depends only on α, β. $\alpha = \mathrm{diag}(\sigma_k^2)$ (which depend only on the kernel and are fixed for a multivariate normal), and on the L2-norm of the second derivative of the density. The (i, i)th element of the Hessian of multi-variate normal ($\phi(x_1, \ldots, x_d) = \phi(X)$) is $\phi(X)(x_i^2 - \sigma_i^2)/\sigma_i^4$.

The rule-of-thumb bandwidth is obtained under the assumption that the true density is multivariate normal with zero covariances (i.e. a diagonal variance-covariance matrix). For details, see (Silverman 1986).

Value

A numeric vector of bandwidths that are a reasonable start optimal non-parametric density estimation of `x`.

References

Silverman BW (1986). *Density estimation for statistics and data analysis*. New York: Chapman and Hall.

Examples

```
set.seed(1); bw.rot(stats::rnorm(100)) # Should be 0.3787568 in R version 4.0.4
set.seed(1); bw.rot(matrix(stats::rnorm(500), ncol = 10)) # 0.4737872 ... 0.7089850
```

`ctracelr` *Compute empirical likelihood on a trajectory*

Description

Compute empirical likelihood on a trajectory

Usage

```
ctracelr(z, ct = NULL, mu0, mu1, N = 5, order = 4, verbose = FALSE, ...)
```

Arguments

`z`	Passed to [EL1()].
`ct`	Passed to [EL1()].
`mu0`	Starting point of trajectory
`mu1`	End point of trajectory
`N`	Number of segments into which the path is split (i. e. `N+1` steps are used).
`order`	Passed to [EL1()]. It is highly advised to avoid using `NA` (no extrapolation) because the lambda search may fail with unmodified logarithm.
`verbose`	Logical: report iteration progress?
`...`	Passed to [EL1()]. This function does not accept the starting lambda because it is much faster (3–5 times) to reuse the lambda from the previous iteration.

Value

A matrix with one row at each mean from mu0 to mu1 and a column for each EL return value (except EL weights).

Examples

```
# Plot 2.5 from Owen (2001)
earth <- c(
  5.5, 5.61, 4.88, 5.07, 5.26, 5.55, 5.36, 5.29, 5.58, 5.65, 5.57, 5.53, 5.62, 5.29,
  5.44, 5.34, 5.79, 5.1, 5.27, 5.39, 5.42, 5.47, 5.63, 5.34, 5.46, 5.3, 5.75, 5.68, 5.85
)
EL1(earth, mu = 5.1,  verbose = TRUE)
logELR <- ctracelr(earth, mu0 = 5.1, mu1 = 5.65, N = 55, verbose = TRUE)
hist(earth, breaks = seq(4.75, 6, 1/8))
plot(logELR[, 1], exp(logELR[, 2]), bty = "n", type = "l",
     xlab = "Earth density", ylab = "ELR")

# Two-dimensional trajectory
set.seed(1)
xy <- matrix(rexp(200), ncol = 2)
logELR2 <- ctracelr(xy, mu0 = c(0.5, 0.5), mu1 = c(1.5, 1.5), N = 100)
```

`dampedNewton` *Damped Newton optimiser*

Description

Damped Newton optimiser

Usage

```
dampedNewton(
  fn,
  par,
  thresh = 1e-30,
  itermax = 100,
  verbose = FALSE,
  alpha = 0.3,
  beta = 0.8,
  backeps = 0
)
```

Arguments

`fn`	A function that returns a list: f, f', f''. If the function takes vector arguments, the dimensions of the list components must be 1, `dim X`, `(dim X) x (dim X)`. The function must be (must be twice continuously differentiable at x)
`par`	Numeric vector: starting point.
`thresh`	A small scalar: stop when Newton decrement squared falls belowe `thresh`.
`itermax`	Maximum iterations. Consider optimisation failed if the maximum is reached.

`verbose`	Logical: if true, prints the tracing infornation (iteration log). This is a translation of Algorithm 9.5 from (Boyd and Vandenberghe 2004) into C++.
`alpha`	Back-tracking parameter strictly between 0 and 0.5: acceptance of a decrease in function value by alpha*f of the prediction.
`beta`	Back-tracking parameter strictly between 0 and 1: reduction of the step size until the stopping criterion is met. 0.1 corresponds to a very crude search, 0.8 corresponds to a less crude search.
`backeps`	Back-tracking threshold: the search can miss by this much. Consider setting it to 1e-10 if backtracking seems to be failing due to round-off.

Value

A list:

References

Boyd S, Vandenberghe L (2004). *Convex Optimization*. Cambridge University Press.

Examples

```
f1 <- function(x)
  list(fn = x - log(x), gradient = 1 - 1/x, Hessian = matrix(1/x^2, 1, 1))
optim(2, function(x) f1(x)[["fn"]], gr = function(x) f1(x)[["gradient"]], method = "BFGS")
dampedNewton(f1, 2, verbose = TRUE)

# The minimum of f3 should be roughly at -0.57
f3 <- function(x)
  list(fn = sum(exp(x) + 0.5 * x^2), gradient = exp(x) + x, Hessian =  diag(exp(x) + 1))
dampedNewton(f3, seq(0.1, 5, length.out = 11), verbose = TRUE)
```

`DCV` *Density cross-validation*

Description

Density cross-validation

Usage

```
DCV(
  x,
  bw,
  weights = NULL,
  same = FALSE,
  kernel = "gaussian",
  order = 2,
```

```
  PIT = FALSE,
  chunks = 0,
  no.dedup = FALSE
)
```

Arguments

`x`	A numeric vector, matrix, or data frame containing observations. For density, the points used to compute the density. For kernel regression, the points corresponding to explanatory variables.
`bw`	Candidate bandwidth values: scalar, vector, or a matrix (with columns corresponding to columns of `x`).
`weights`	A numeric vector of observation weights (typically counts) to perform weighted operations. If null, `rep(1, NROW(x))` is used. In all calculations, the total number of observations is assumed to be the sum of `weights`.
`same`	Logical: use the same bandwidth for all columns of `x`? Note: since DCV requires computing the leave-one-out estimator, repeated observations are combined first; the de-duplication is therefore forced in cross-validation. The only situation where de-duplication can be skipped is passing de-duplicated data sets from outside (e.g. inside optimisers).
`kernel`	Character describing the desired kernel type. NB: due to limited machine precision, even Gaussian has finite support.
`order`	An integer: 2, 4, or 6. Order-2 kernels are the standard kernels that are positive everywhere. Orders 4 and 6 produce some negative values, which reduces bias but may hamper density estimation.
`PIT`	If TRUE, the Probability Integral Transform (PIT) is applied to all columns of `x` via `ecdf` in order to map all values into the [0, 1] range. May be an integer vector of indices of columns to which the PIT should be applied.
`chunks`	Integer: the number of chunks to split the task into (limits RAM usage but increases overhead). `0` = auto-select (making sure that no matrix has more than 2^27 elements).
`no.dedup`	Logical: if TRUE, sets `deduplicate.x` and `deduplicate.xout` to FALSE (shorthand).

Value

A numeric vector of the same length as `bw` or `nrow(bw)`.

Examples

```
set.seed(1)
x <- rlnorm(100); x <- c(x[1], x)  # x with 1 duplicate
bws <- exp(seq(-3, 0.5, 0.1))
plot(bws, DCV(x, bws), log = "x", bty = "n", main = "Density CV")
```

`EL` *Unified empirical likelihood wrapper*

Description

Call `EL0()`, `EL1()`, or `EuL()` through a single interface. If extrapolation is requested, switch to dedicated functions. Anything method-specific goes into `EL.args`.

Usage

```
EL(
  z,
  ct = NULL,
  mu = NULL,
  shift = NULL,
  type = c("auto", "EL1", "EL0", "EuL"),
  chull.fail = c("none", "taylor", "wald", "adjusted", "adjusted2", "balanced"),
  renormalise = FALSE,
  return.weights = FALSE,
  weight.tolerance = NULL,
  verbose = FALSE,
  ...
)
```

Arguments

`z`	A numeric vector or a matrix with one data vector per column.
`ct`	Numeric count variable with non-negative values that indicates the multiplicity of observations.
`mu`	Hypothesised mean, default `(0 ... 0)` in $R^{\mathrm{ncol}(z)}$.
`shift`	The value to add in the denominator (useful in case there are extra Lagrange multipliers): $1 + \lambda' Z + shift$.
`type`	Character: one of `c("auto", "EL1", "EL0", "EuL")`. If `"auto"`, uses `"EL1"` for multi-variate data and `"EL0"` for uni-variate.
`chull.fail`	Character: `"none"` calls the original EL (which may return `-Inf` in case of a convex-hull violation), `"taylor"` calls [ExEL1()], `"wald"` calls [ExEL2()], `"adjusted"` adds one pseudo-observation as in (Chen et al. 2008), `"adjusted2"` adds one (in 1D) or two (2D+) pseudo-observations with improved coverage rate according to (Liu and Chen 2010), and `"balanced"` adds two pseudo-observations according to (Emerson and Owen 2009).
`renormalise`	If `FALSE`, then uses the total sum of counts as the number of observations, like in vanilla empirical likelihood, due to formula (2.9) in (Owen 2001), otherwise re-normalises the counts to 1 according to (Cosma et al. 2019) (p. 170, the topmost formula).

`return.weights`	Logical: if `TRUE`, returns the empirical probabilities. Default is memory-saving (`FALSE`).
`weight.tolerance`	Weight tolerance for counts to improve numerical stability (defaults to `sqrt(.Machine$double.eps)` times the maximum weight).
`verbose`	Logical: print output diagnostics?
`...`	Named extra arguments passed to the selected back-end (e.g. `order`, `itermax`, `lambda.init`, `vt`, `trunc.to`, `boundary.tolerance`, ...).

Value

A list with either the return value of the selected back-end or (for extrapolation methods) at least the `logelr` list value and extrapolation attributes.

References

Chen J, Variyath AM, Abraham B (2008). “Adjusted empirical likelihood and its properties.” *Journal of Computational and Graphical Statistics*, **17**(2), 426–443. doi:10.1198/106186008x321068.

Cosma A, Kostyrka AV, Tripathi G (2019). “Inference in conditional moment restriction models when there is selection due to stratification.” In Huynh KP, Jacho-Chavez DT, Tripathi G (eds.), *The Econometrics of Complex Survey Data: Theory and Applications*, 137–171. Emerald Publishing Limited. ISBN 978-1-78756-726-9.

Emerson SC, Owen AB (2009). “Calibration of the empirical likelihood method for a vector mean.” *Electronic Journal of Statistics*, **3**, 1161–1192. ISSN 1935-7524. doi:10.1214/09ejs518.

Liu Y, Chen J (2010). “Adjusted empirical likelihood with high-order precision.” *The Annals of Statistics*, **38**(3). ISSN 0090-5364. doi:10.1214/09aos750.

Owen AB (2001). *Empirical Likelihood*. Chapman and Hall/CRC, New York, USA.

Examples

```
# EL0 with extras:
EL(type = "EL0", z = 1:9, mu = 4, boundary.tolerance = 1e-8)
# EL1 with a custom order and iteration cap:
set.seed(1)
x <- cbind(rnorm(30), runif(30)-0.5)
EL(type = "EL1", z = x, mu = c(0, 0), order = 4, itermax = 50, return.weights = TRUE)
# EuL with vt and truncation:
set.seed(1)
EL(type = "EuL", z = x, vt = runif(NROW(x)), weight.tolerance = 0.1, trunc.to = 0.1)

# Extrapolated variants
set.seed(1)
EL(type = "EL0", z = 1:9, mu = 12, chull.fail = "taylor", exel.control = list(xlim = c(2, 8)))
EL(type = "EL1", z = 1:9, mu = 12, chull.fail = "wald", exel.control = list(fmax = 10))
x <- matrix(runif(20), ncol = 2)
EL(x, mu = c(0, 0), chull.fail = "adjusted")
```

```
EL(x, mu = c(0, 0), chull.fail = "adjusted2")
EL(x, mu = c(0, 0), chull.fail = "balanced")
```

`EL0` *Uni-variate empirical likelihood via direct lambda search*

Description

Empirical likelihood with counts to solve one-dimensional problems efficiently with Brent's root search algorithm. Conducts an empirical likelihood ratio test of the hypothesis that the mean of `z` is `mu`. The names of the elements in the returned list are consistent with the original R code in (Owen 2017).

Usage

```
EL0(
  z,
  mu = NULL,
  ct = NULL,
  shift = NULL,
  renormalise = FALSE,
  return.weights = FALSE,
  weight.tolerance = NULL,
  boundary.tolerance = 1e-09,
  trunc.to = 0,
  deriv = FALSE,
  log.control = list(order = NULL, lower = NULL, upper = NULL),
  verbose = FALSE
)
```

Arguments

`z`	A numeric vector containing the observations.
`mu`	Hypothesised mean of `z` in the moment condition.
`ct`	Numeric count variable with non-negative values that indicates the multiplicity of observations. Can be fractional. Very small counts below the threshold `weight.tolerance` are zeroed.
`shift`	The value to add in the denominator (useful in case there are extra Lagrange multipliers): $1 + \lambda' Z + shift$.
`renormalise`	If `FALSE`, then uses the total sum of counts as the number of observations, like in vanilla empirical likelihood, due to formula (2.9) in (Owen 2001), otherwise re-normalises the counts to 1 according to (Cosma et al. 2019) (see p. 170, the topmost formula).
`return.weights`	Logical: if `TRUE`, returns the empirical probabilities. Default is memory-saving (`FALSE`).

`weight.tolerance`	Weight tolerance for counts to improve numerical stability (defaults to `sqrt(.Machine$double.eps)` times the maximum weight).
`boundary.tolerance`	Relative tolerance for determining when lambda is not an interior solution because it is too close to the boundary. Unit: fraction of the feasble bracket length.
`trunc.to`	Counts under `weight.tolerance` will be set to this value. In most cases, setting this to `0` (default) or `weight.tolerance` is a viable solution for the zero-denominator problem.
`deriv`	Logical: if `TRUE`, computes and returns the first two derivatives of log-ELR w.r.t. `mu`.
`log.control`	List of arguments passed to [logTaylor()].
`verbose`	Logical: if `TRUE`, prints warnings.

Details

This function provides the core functionality for univariate empirical likelihood. The technical details is given in (Cosma et al. 2019), although the algorithm used in that paper is slower than the one provided by this function.

Since we know that the EL probabilities belong to (0, 1), the interval (bracket) for λ search can be determined in the spirit of formula (2.9) from (Owen 2001). Let $z^*_i := z_i - \mu$ be the recentred observations.

$$p_i = c_i/N \cdot (1 + \lambda z^*_i + s)^{-1}$$

The probabilities are bounded from above: $p_i < 1$ for all *i*, therefore,

$$c_i/N \cdot (1 + \lambda z^*_i + s)^{-1} < 1$$

$$c_i/N - 1 - s < \lambda z^*_i$$

Two cases: either $z^*_i < 0$, or $z^*_i > 0$ (cases with $z^*_i = 0$ are trivially excluded because they do not affect the EL). Then,

$$(c_i/N - 1 - s)/z^*_i > \lambda,\ \forall i : z^*_i < 0$$

$$(c_i/N - 1 - s)/z^*_i < \lambda,\ \forall i : z^*_i > 0$$

which defines the search bracket:

$$\lambda_{\min} := \max_{i:z^*_i>0} (c_i/N - 1 - s)/z^*_i$$

$$\lambda_{\max} := \min_{i:z^*_i<0} (c_i/N - 1 - s)/z^*_i$$

$$\lambda_{\min} < \lambda < \lambda_{\max}$$

(This derivation contains *s*, which is the extra shift that extends the function to allow mixed conditional and unconditional estimation; Owen's textbook formula corresponds to $s = 0$.)

The actual tolerance of the lambda search in `brentZero` is $2|\lambda_{\max}|\epsilon_m + \mathrm{tol}/2$, where `tol` is `.Machine$double.eps` and ϵ_m is `.Machine$double.eps`.

The sum of log-weights is maximised without Taylor expansion, forcing `mu` to be inside the convex hull of `z`. If a violation is happening, consider using `log.control(order = 4)` or switching to Euclidean likelihood via [EuL()].

Value

A list with the following elements:

logelr Logarithm of the empirical likelihood ratio.

lam The Lagrange multiplier.

wts Observation weights/probabilities (of the same length as `z`).

converged `TRUE` if the algorithm converged, `FALSE` otherwise (usually means that `mu` is not within the range of `z`, i.e. the one-dimensional convex hull of `z`).

iter The number of iterations used (from `brentZero`).

bracket The admissible interval for lambda (that is, yielding weights between 0 and 1).

estim.prec The approximate estimated precision of lambda (from `brentZero`).

f.root The value of the derivative of the objective function w.r.t. lambda at the root (from `brentZero`). Values > `sqrt(.Machine$double.eps)` indicate convergence problems.

deriv If requested, the first two derivatives of log-ELR w.r.t. `mu`

exitcode An integer indicating the reason of termination.

message Character string describing the optimisation termination status.

References

Cosma A, Kostyrka AV, Tripathi G (2019). "Inference in conditional moment restriction models when there is selection due to stratification." In Huynh KP, Jacho-Chavez DT, Tripathi G (eds.), *The Econometrics of Complex Survey Data: Theory and Applications*, 137–171. Emerald Publishing Limited. ISBN 978-1-78756-726-9.

Owen AB (2001). *Empirical Likelihood*. Chapman and Hall/CRC, New York, USA.

Owen AB (2017). *A weighted self-concordant optimization for empirical likelihood*. `https://artowen.su.domains/empirical/countnotes.pdf`.

See Also

[EL1()] for multi-variate EL based on minimisation w.r.t. lambda.

Examples

```
# Figure 2.4 from Owen (2001) -- with a slightly different data point
earth <- c(
  5.5, 5.61, 4.88, 5.07, 5.26, 5.55, 5.36, 5.29, 5.58, 5.65, 5.57, 5.53, 5.62, 5.29,
  5.44, 5.34, 5.79, 5.1, 5.27, 5.39, 5.42, 5.47, 5.63, 5.34, 5.46, 5.3, 5.75, 5.68, 5.85
)
# Root searching (EL0) is faster than minimisation w.r.t. lambda (EL1)
set.seed(1)
system.time(r0 <- replicate(40, EL0(sample(earth, replace = TRUE), mu = 5.517)))
set.seed(1)
system.time(r1 <- replicate(40, EL1(sample(earth, replace = TRUE), mu = 5.517)))
plot(apply(r0, 2, "[[", "logelr"), apply(r1, 2, "[[", "logelr") - apply(r0, 2, "[[", "logelr"),
     bty = "n", xlab = "log(ELR) computed via dampened Newthon method",
```

```
     main = "Discrepancy between EL1 and EL0", ylab = "")
abline(h = 0, lty = 2)

# Handling the convex hull violation differently
EL0(1:9)
EL0(1:9, log.control = list(order = 2))  # Warning + huge lambda
EL0(1:9, log.control = list(order = 4))  # Warning + huge lambda

# Warning: depending on the compiler, the discrepancy between EL and EL0
# can be one million (1) times larger than the machine epsilon despite both of them
# being written in pure R
# The results from Apple clang-1400.0.29.202 and Fortran GCC 12.2.0 are different from
# those obtained under Ubuntu 22.04.4 + GCC 11.4.0-1ubuntu1~22.04,
# Arch Linux 6.6.21 + GCC 14.1.1, and Windows Server 2022 + GCC 13.2.0
out0 <- EL0(earth, mu = 5.517, return.weights = TRUE)[1:4]
out1 <- EL1(earth, mu = 5.517, return.weights = TRUE)[1:4]
print(c(out0$lam, out1$lam), 16)

# Value of lambda                                EL0                EL1
# aarch64-apple-darwin20          -1.5631313957?????  -1.5631313955?????
# Windows, Ubuntu, Arch           -1.563131395492627  -1.563131395492627
```

EL1 *Self-concordant multi-variate empirical likelihood with counts*

Description

Implements the empirical-likelihood-ratio test for the mean of the coordinates of `z` (with the hypothesised value `mu`). The counts need not be integer; in the context of local likelihoods, they can be kernel observation weights.

Usage

```
EL1(
  z,
  mu = NULL,
  ct = NULL,
  shift = NULL,
  lambda.init = NULL,
  renormalise = FALSE,
  return.weights = FALSE,
  lower = NULL,
  upper = NULL,
  order = NA,
  weight.tolerance = NULL,
  deriv = FALSE,
  thresh = 1e-30,
  itermax = 100L,
  verbose = FALSE,
```

```
  alpha = 0.3,
  beta = 0.8,
  backeps = 0,
  gradtol = 1e-12,
  steptol = 1e-12,
  ftol = 1e-14,
  stallmax = 5
)
```

Arguments

`z`	A numeric vector or a matrix with one data vector per column.
`mu`	Hypothesised mean, default `(0 ... 0)` in $R^{\mathrm{ncol}(z)}$.
`ct`	Numeric count variable with non-negative values that indicates the multiplicity of observations.
`shift`	The value to add in the denominator (useful in case there are extra Lagrange multipliers): $1 + \lambda' Z + shift$.
`lambda.init`	Starting lambda, default `(0 ... 0)`. Improves speed and accuracy in sequential problems if supplied from the previous iteration.
`renormalise`	If `FALSE`, then uses the total sum of counts as the number of observations, like in vanilla empirical likelihood, due to formula (2.9) in (Owen 2001), otherwise re-normalises the counts to 1 according to (Cosma et al. 2019) (p. 170, the topmost formula).
`return.weights`	Logical: if `TRUE`, returns the empirical probabilities. Default is memory-saving (`FALSE`).
`lower`	Lower cut-off for [logTaylor()], default `1/NROW(z)`.
`upper`	Upper cut-off for [logTaylor()], default `Inf`.
`order`	Positive even integer such that the Taylor approximation of this order to $\log x$ is self-concordant; usually 4 or 2. Passed to [logTaylor()].
`weight.tolerance`	Weight tolerance for counts to improve numerical stability (defaults to `sqrt(.Machine$double.eps)` times the maximum weight).
`deriv`	Logical: if `TRUE`, computes and returns the first two directional derivatives of log-ELR w.r.t. `mu` in the direction of the hypothesised value.
`thresh`	Target tolerance on the squared Newton decrement: loop stops when `decr^2 <= thresh`. (If `verbose` is `TRUE`, decrement itself is printed.)
`itermax`	Maximum number of outer iterations of the damped Newton method (seems ample).
`verbose`	Logical: print output diagnostics?
`alpha`	Backtracking line search Armijo parameter: acceptance of a decrease in function value by αf of the prediction based on the linear extrapolation. Smaller makes acceptance easier.
`beta`	Backtracking step shrinkage factor in `[0, 1]`. 0.1 corresponds to a very crude search, 0.8 corresponds to a less crude search.

`backeps`	Backtrack threshold, a small slack added to Armijo RHS: the search can miss by this much. Accept if $f(x+tp) \le f(x) + \alpha t g' p + \text{backeps}$. Consider setting it to `1e-10` if backtracking seems to be failing due to round-off.
`gradtol`	Gradient tolerance: stop if `||g|| <= gradtol`.
`steptol`	Step tolerance: stop if the relative size is tiny: `||x2-x1||/max(1, ||x2||) < ftol`.
`ftol`	Function change tolerance: stop if the relative function-value change is less than `ftol`.
`stallmax`	Stop if both `rel_step <= steptol` and `rel_f <= ftol` hold for this many consecutive iterations.

Details

Negative weights are not allowed. They could be useful in some applications, but they can destroy convexity or even boundedness. They also make the Newton step fail to be of least squares type.

This function relies on the improved computational strategy for the empirical likelihood. The search of the lambda multipliers is carried out via a dampened Newton method with guaranteed convergence owing to the fact that the log-likelihood is replaced by its Taylor approximation of any desired order (default: 4, the minimum value that ensures self-concordance).

Implementation note: the EL solver also guarantees a descent direction; if the Newton step is non-descent or non-finite, it falls back to steepest descent (negative gradient), which keeps the line search well-behaved.

Tweak `alpha` and `beta` with extreme caution. See (Boyd and Vandenberghe 2004), pp. 464–466 for details. It is necessary that `0 < alpha < 1/2` and `0 < beta < 1`. `alpha = 0.3` seems better than 0.01 on some 2-dimensional test data (sometimes fewer iterations).

The argument names, except for `lambda.init`, are matching the original names in Art B. Owen's implementation. The highly optimised one-dimensional counterpart, [EL0()], is designed to return a faster and a more accurate solution in the one-dimensional case.

Value

A list with the following values:

logelr Log of empirical likelihood ratio (equal to 0 if the hypothesised mean is equal to the sample mean)

lam Vector of Lagrange multipliers

wts Observation weights/probabilities (vector of length n)

deriv Length-2 vector: directional first and second derivatives along the ray toward mu (if `deriv = TRUE`)

converged `TRUE` if algorithm converged. `FALSE` usually means that mu is not in the convex hull of the data. Then, a very small likelihood is returned (instead of zero).

iter Number of iterations taken.

ndec Newton decrement (see Boyd & Vandenberghe).

gradnorm Norm of the gradient of log empirical likelihood.

Source

This original code was written for (Owen 2013) and [published online](https://artowen.su.domains/empirical/) by Art B. Owen (March 2015, February 2017). The present version was rewritten in `Rcpp` and slightly reworked to contain fewer inner functions and loops.

References

Boyd S, Vandenberghe L (2004). *Convex Optimization*. Cambridge University Press.

Cosma A, Kostyrka AV, Tripathi G (2019). "Inference in conditional moment restriction models when there is selection due to stratification." In Huynh KP, Jacho-Chavez DT, Tripathi G (eds.), *The Econometrics of Complex Survey Data: Theory and Applications*, 137–171. Emerald Publishing Limited. ISBN 978-1-78756-726-9.

Owen AB (2001). *Empirical Likelihood*. Chapman and Hall/CRC, New York, USA.

Owen AB (2013). "Self-concordance for empirical likelihood." *Canadian Journal of Statistics*, **41**(3), 387–397.

See Also

[logTaylor()], [EL0()]

Examples

```
earth <- c(
  5.5, 5.61, 4.88, 5.07, 5.26, 5.55, 5.36, 5.29, 5.58, 5.65, 5.57, 5.53, 5.62, 5.29,
  5.44, 5.34, 5.79, 5.1, 5.27, 5.39, 5.42, 5.47, 5.63, 5.34, 5.46, 5.3, 5.75, 5.68, 5.85
)
EL1(earth, mu = 5.517, verbose = TRUE) # 5.517 is the modern accepted value

# Linear regression through empirical likelihood
coef.lm <- coef(lm(mpg ~ hp + am, data = mtcars))
xmat <- cbind(1, as.matrix(mtcars[, c("hp", "am")]))
yvec <- mtcars$mpg
foc.lm <- function(par, x, y) {  # The sample average of this
  resid <- y - drop(x %*% par)   # must be 0
  resid * x
}
minusEL <- function(par) -EL1(foc.lm(par, xmat, yvec), itermax = 10)$logelr
coef.el <- optim(c(26, -0.06, 5.3), minusEL, control = list(maxit = 100))$par
abs(coef.el - coef.lm) / coef.lm  # Relative difference

# Likelihood ratio testing without any variance estimation
# Define the profile empirical likelihood for the coefficient on am
minusPEL <- function(par.free, par.am)
  -EL1(foc.lm(c(par.free, par.am), xmat, yvec), itermax = 20)$logelr
# Constrained maximisation assuming that the coef on par.am is 3.14
coef.el.constr <- optim(coef.el[1:2], minusPEL, par.am = 3.14)$par
print(-2 * EL1(foc.lm(c(coef.el.constr, 3.14), xmat, yvec))$logelr)
# Exceeds the critical value qchisq(0.95, df = 1)
```

`EuL` *Multi-variate Euclidean likelihood with analytical solution*

Description

Multi-variate Euclidean likelihood with analytical solution

Usage

```
EuL(
  z,
  mu = NULL,
  ct = NULL,
  vt = NULL,
  shift = NULL,
  weight.tolerance = NULL,
  trunc.to = 0,
  renormalise = TRUE,
  return.weights = FALSE,
  verbose = FALSE
)
```

Arguments

`z`	Numeric data vector.
`mu`	Hypothesised mean of `z` in the moment condition.
`ct`	Numeric count variable with non-negative values that indicates the multiplicity of observations. Can be fractional. Very small counts below the threshold `weight.tolerance` are zeroed.
`vt`	Numeric vector: non-negative variance weights for estimating the conditional variance of `z`. Probabilities are returned only for the observations where `vt > 0`.
`shift`	The value to add in the denominator (useful in case there are extra Lagrange multipliers): $1 + \lambda' Z + shift$.
`weight.tolerance`	Weight tolerance for counts to improve numerical stability (defaults to `sqrt(.Machine$double.eps)` times the maximum weight).
`trunc.to`	Counts under `weight.tolerance` will be set to this value. In most cases, setting this to `0` (default) or `weight.tolerance` is a viable solution for the zero-denominator problem.
`renormalise`	If `FALSE`, then uses the total sum of counts as the number of observations, like in vanilla empirical likelihood, due to formula (2.9) in (Owen 2001), otherwise re-normalises the counts to 1 according to (Cosma et al. 2019) (see p. 170, the topmost formula).
`return.weights`	Logical: if `TRUE`, returns the empirical probabilities. Default is memory-saving (`FALSE`).
`verbose`	Logical: if `TRUE`, prints warnings.

Details

The arguments `ct` and `vt` are responsible for smoothing of the moment function and conditional variance, respectively. The objective function is

$$\min_{p_{ij}} \frac{1}{n} \sum_{i=1}^{n} \sum_{j=1}^{n} \mathbb{I}_{ij} \frac{(p_{ij} - c_{ij})^2}{2v_{ij}}$$

, where $\mathbb{I}_{ij}$ is 1 if $v_{ij} \neq 0$.

This estimator is numerically equivalent to the Sieve Minimum Distance estimator of (Ai and Chen 2003) with kernel sieves, but this interface provides more flexibility through the two sets of weights. If `ct` and `vt` are not provided, their default value is set to 1, and the resulting estimator is the CUE-GMM estimator: a quadratic form in which the unconditional mean vector is weighted by the inverse of the unconditional variance.

Value

A list with the same structure as that in [EL1()].

References

Ai C, Chen X (2003). “Efficient Estimation of Models with Conditional Moment Restrictions Containing Unknown Functions.” *Econometrica*, **71**(6), 1795–1843. ISSN 1468-0262. doi:10.1111/14680262.00470.

Cosma A, Kostyrka AV, Tripathi G (2019). “Inference in conditional moment restriction models when there is selection due to stratification.” In Huynh KP, Jacho-Chavez DT, Tripathi G (eds.), *The Econometrics of Complex Survey Data: Theory and Applications*, 137–171. Emerald Publishing Limited. ISBN 978-1-78756-726-9.

Owen AB (2001). *Empirical Likelihood*. Chapman and Hall/CRC, New York, USA.

See Also

[EL1()]

Examples

```
set.seed(1)
z <- cbind(rnorm(10), runif(10))
colMeans(z)
a <- EuL(z, return.weights = TRUE)
a$wts
sum(a$wts)  # Unity
colSums(a$wts * z)  # Zero
```

`ExEL1` *Extrapolated EL of the first kind (Taylor expansion)*

Description

Extrapolated EL of the first kind (Taylor expansion)

Usage

```
ExEL1(
  z,
  mu,
  type = c("auto", "EL0", "EL1"),
  exel.control = list(xlim = "auto", fmax = NA, p = 0.999, df = NA),
  ...
)

ExEL2(
  z,
  mu,
  type = c("auto", "EL0", "EL1"),
  exel.control = list(xlim = "auto", fmax = NA, p = 0.999, df = NA),
  ...
)
```

Arguments

`z`	Passed to `EL0`/`EL1`.
`mu`	Passed to `EL0`/`EL1`.
`type`	If `"EL0"`, uses uni-variate [EL0()] for calculations; same for `"EL1"`.
`exel.control`	A list with the following elements: `xlim` – if `"auto"`, uses a quick boundary detection, otherwise should be a length-two numeric vector; `fmax` – maximum allowed chi-squared statistic value for a thorough root search with probability `p` and degrees of freedom `df`.
`...`	Also passed to `EL0`/`EL1`.

Value

A numeric vector of log-ELR statistic of the same length as `mu`.

Examples

```
z <- c(1, 4, 5, 5, 6, 6)
ExEL1(z, 0.5, ct = 1:6)

xseq <- seq(0, 7, 0.2)
```

```
plot(xseq, -2*ExEL1(z, mu = xseq, ct = 1:6))
abline(v = c(1.2, 5.8), h = qchisq(0.99, 1), lty = 3)

# User-defined 'good' interval
ctrl0 <- list(xlim = c(-1, 8)); ctrl1 <- list(xlim = c(2.5, 5.5))
plot(xseq, -2*ExEL1(z, xseq, ct = 1:6, exel.control = ctrl0), bty = "n")
lines(xseq, -2*ExEL1(z, xseq, ct = 1:6, exel.control = ctrl1), col = 3)
abline(v = ctrl1$xlim, lty = 3)

# Root searching
ctrl2 <- list(fmax = qchisq(0.99, 1))
plot(xseq, -2*ExEL1(z, xseq, ct = 1:6, exel.control = ctrl0), bty = "n")
lines(xseq, -2*ExEL1(z, xseq, ct = 1:6, exel.control = ctrl2), col = 3)
abline(h = qchisq(0.99, 1), lty = 3)

# With EL1 vs. EL0 -- very little discrepancy
xseq <- seq(0.8, 1.4, length.out = 101)
plot(xseq, -2*ExEL1(z, xseq, ct = 1:6, exel.control = ctrl0), bty = "n")
lines(xseq, -2*ExEL1(z, xseq, ct = 1:6, type = "EL0"), col = 3)
lines(xseq, -2*ExEL1(z, xseq, ct = 1:6, type = "EL1"), col = 2, lty = 2, lwd = 2)

# Comparing ExEL2 vs ExEL1 with bridges containing exp(x)
z <- -4:4
ct <- 9:1
xseq <- seq(-7, 10.5, 0.1)
xl <- range(xseq)
a0 <- ExEL1(z, mu = xseq, ct = ct, exel.control = list(xlim = c(-11, 11)))
a1 <- ExEL1(z, mu = xseq, ct = ct)
a2 <- ExEL2(z, mu = xseq, ct = ct)
v1 <- attr(a1, "xlim")
v2 <- c(attr(a2, "bridge.left")[c("x1", "x2")], attr(a2, "bridge.right")[c("x1", "x2")])

plot(xseq, a0, ylim = c(-300, 0), xlim = xl, main = "ExEL splices",
  bty = "n", xlab = "mu", ylab = "logELR(mu)")
lines(xseq, a1, col = 2, lwd = 2)
lines(xseq, a2, col = 4, lwd = 2)
abline(v = v2, lty = 3)
lines(xseq, attr(a2, "parabola.coef") * (xseq - attr(a2, "parabola.centre"))^2, lty = 2)
legend("topright", c("Taylor", "Wald", "ax^2"),
       col = c(2, 4, 1), lwd = c(2, 2, 1), lty = c(1, 1, 2))

dx <- diff(xseq[1:2])
plot(xseq[-1], diff(a1)/dx, col = 2, type = "l", lwd = 2,
  main = "Derivatives of ExEL splice", bty = "n", ylim = c(-100, 100),
  xlab = "mu", ylab = "d/dmu logELR(mu)")
lines(xseq[-1], diff(a2)/dx, col = 4, lwd = 2)
abline(v = c(v1, v2), lty = 3, col = "#00000055")
legend("topright", c("Taylor", "Wald"), col = c(2, 4), lwd = 2)

# Multivariate extension
set.seed(1)
X <- cbind(rchisq(30, 3), rchisq(30, 3))
ct <- runif(30)
```

```
-2*ExEL1(X, mu = c(-1, -1),  ct = ct)  # Outside the hull
-2*ExEL2(X, mu = c(-1, -1),  ct = ct)
```

`getSELWeights` *Construct memory-efficient weights for estimation*

Description

This function constructs SEL weights with appropriate trimming for numerical stability and optional renormalisation so that the sum of the weights be unity

Usage

```
getSELWeights(x, bw = NULL, ..., trim = NULL, renormalise = TRUE)
```

Arguments

`x`	A numeric vector (with many close-to-zero elements).
`bw`	A numeric scalar or a vector passed to ‘kernelWeights‘.
`...`	Other arguments pased to `kernelWeights`.
`trim`	A trimming function that returns a threshold value below which the weights are ignored. In common applications, this function should tend to 0 as the length of `x` increases.
`renormalise`	Logical; passed to ‘sparseVectorToList‘.

Value

A list with indices of large enough elements.

Examples

```
getSELWeights(1:5, bw = 2, kernel = "triangular")
```

`kernelDensity` *Kernel density estimation*

Description

Kernel density estimation

Usage

```
kernelDensity(
  x,
  xout = NULL,
  weights = NULL,
  bw = NULL,
  kernel = c("gaussian", "uniform", "triangular", "epanechnikov", "quartic"),
  order = 2,
  convolution = FALSE,
  chunks = 0,
  PIT = FALSE,
  deduplicate.x = TRUE,
  deduplicate.xout = TRUE,
  no.dedup = FALSE,
  return.grid = FALSE
)
```

Arguments

`x`	A numeric vector, matrix, or data frame containing observations. For density, the points used to compute the density. For kernel regression, the points corresponding to explanatory variables.
`xout`	A vector or a matrix of data points with `ncol(xout) = ncol(x)` at which the user desires to compute the weights, density, or predictions. In other words, this is the requested evaluation grid. If `NULL`, then `x` itself is used as the grid.
`weights`	A numeric vector of observation weights (typically counts) to perform weighted operations. If null, `rep(1, NROW(x))` is used. In all calculations, the total number of observations is assumed to be the sum of `weights`.
`bw`	Bandwidth for the kernel: a scalar or a vector of the same length as `ncol(x)`. Since it is the crucial parameter in many applications, a warning is thrown if the bandwidth is not supplied, and then, Silverman's rule of thumb (via `bw.row()`) is applied to *every dimension* of `x`.
`kernel`	Character describing the desired kernel type. NB: due to limited machine precision, even Gaussian has finite support.
`order`	An integer: 2, 4, or 6. Order-2 kernels are the standard kernels that are positive everywhere. Orders 4 and 6 produce some negative values, which reduces bias but may hamper density estimation.
`convolution`	Logical: if FALSE, returns the usual kernel. If TRUE, returns the convolution kernel that is used in density cross-validation.
`chunks`	Integer: the number of chunks to split the task into (limits RAM usage but increases overhead). `0` = auto-select (making sure that no matrix has more than 2^27 elements).
`PIT`	If TRUE, the Probability Integral Transform (PIT) is applied to all columns of `x` via `ecdf` in order to map all values into the [0, 1] range. May be an integer vector of indices of columns to which the PIT should be applied.

`deduplicate.x`	Logical: if TRUE, full duplicates in the input `x` and `y` are counted and transformed into weights; subsetting indices to reconstruct the duplicated data set from the unique one are also returned.
`deduplicate.xout`	Logical: if TRUE, full duplicates in the input `xout` are counted; subsetting indices to reconstruct the duplicated data set from the unique one are returned.
`no.dedup`	Logical: if TRUE, sets `deduplicate.x` and `deduplicate.xout` to FALSE (shorthand).
`return.grid`	Logical: if `TRUE`, returns `xout` and appends the estimated density as the last column. The number of chunks for kernel density and regression estimation is chosen in such a manner that the number of elements in the internal weight matrix should not exceed $2^{27} = 1.3 \cdot 10^8$, which caps RAM use (64 bits = 8 bytes per element) at 1 GB. Larger matrices are processed in parallel in chunks of size at most $2^{26} = 6.7 \cdot 10^7$ elements. The number of threads is 4 by default, which can be changed by `RcppParallel::setThreadOptions(numThreads = 8)` or something similar.

Value

A vector of density estimates evaluated at the grid points or, if `return.grid`, a matrix with the density in the last column.

Examples

```
set.seed(1)
x <- sort(rt(10000, df = 5)) # Observed values
g <- seq(-6, 6, 0.05) # Grid for evaluation
d2 <- kernelDensity(x, g, bw = 0.3, kernel = "epanechnikov", no.dedup = TRUE)
d4 <- kernelDensity(x, g, bw = 0.4, kernel = "quartic", order = 4, no.dedup = TRUE)
plot(g, d2, ylim = range(0, d2, d4), type = "l"); lines(g, d4, col = 2)

# De-duplication facilities for faster operations
set.seed(1)  # Creating a data set with many duplicates
n.uniq <- 1000
n <- 4000
inds <- ceiling(runif(n, 0, n.uniq))
x.uniq <- matrix(rnorm(n.uniq*10), ncol = 10)
x <- x.uniq[inds, ]
xout <- x.uniq[ceiling(runif(n.uniq*3, 0, n.uniq)), ]
w <- runif(n)
data.table::setDTthreads(1) # For measuring the pure gains and overhead
RcppParallel::setThreadOptions(numThreads = 1)
kd1 <- kernelDensity(x, xout, w, bw = 0.5)
kd2 <- kernelDensity(x, xout, w, bw = 0.5, no.dedup = TRUE)
stat1 <- attr(kd1, "duplicate.stats")
stat2 <- attr(kd2, "duplicate.stats")
print(stat1[3:5]) # De-duplication time -- worth it
print(stat2[3:5]) # Without de-duplication, slower
unname(prod((1 - stat1[1:2])) / (stat1[5] / stat2[5])) # > 1 = better time
```

```
# savings than expected, < 1 = worse time savings than expected
all.equal(as.numeric(kd1), as.numeric(kd2))
max(abs(kd1 - kd2)) # Should be around machine epsilon or less
```

`kernelDiscreteDensitySmooth` *Density and/or kernel regression estimator with conditioning on discrete variables*

Description

Density and/or kernel regression estimator with conditioning on discrete variables

Usage

```
kernelDiscreteDensitySmooth(x, y = NULL, compact = FALSE, fun = mean)
```

Arguments

`x`	A vector or a matrix/data frame of discrete explanatory variables (exogenous). Non-integer values are fine because the data are split into bins defined by interactions of these variables.
`y`	Optional: a vector of dependent variable values.
`compact`	Logical: return unique values instead of full data with repeated observations?
`fun`	A function that computes a statistic of `y` inside every category defined by `x`.

Value

A list with `x`, density estimator (`fhat`) and, if `y` was provided, regression estimate.

Examples

```
set.seed(1)
x <- sort(rnorm(1000))
p <- 0.5*pnorm(x) + 0.25 # Propensity score
d <- as.numeric(runif(1000) < p)
# g = discrete version of x for binning
g <- as.numeric(as.character(cut(x, -4:4, labels = -4:3+0.5)))
dhat.x <- kernelSmooth(x = x, y = d, bw = 0.4, no.dedup = TRUE)
dhat.g <- kernelDiscreteDensitySmooth(x = g, y = d)
dhat.comp <- kernelDiscreteDensitySmooth(g, d, compact = TRUE)
plot(x, p, ylim = c(0, 1), bty = "n", type = "l", lty = 2)
points(x, dhat.x, col = "#00000044")
points(dhat.comp, col = 2, pch = 16, cex = 2)
lines(dhat.comp$x, dhat.comp$fhat, col = 4, pch = 16, lty = 3)
```

`kernelFun` *Basic univatiate kernel functions*

Description

Computes 5 most popular kernel functions of orders 2 and 4 with the potential of returning an analytical convolution kernel for density cross-validation. These kernels appear in (Silverman 1986).

Usage

```
kernelFun(
  x,
  kernel = c("gaussian", "uniform", "triangular", "epanechnikov", "quartic"),
  order = c(2, 4),
  convolution = FALSE
)
```

Arguments

`x`	A numeric vector of values at which to compute the kernel function.
`kernel`	Kernel type: uniform, Epanechnikov, triangular, quartic, or Gaussian.
`order`	Kernel order. 2nd-order kernels are always non-negative. *k*-th-order kernels have all moments from 1 to (k-1) equal to zero, which is achieved by having some negative values. $\int_{-\infty}^{+\infty} x^2 k(x) = \sigma^2_k = 1$. This is useful because in this case, the constant `k_2` in formulæ 3.12 and 3.21 from Silverman (1986) is equal to 1.
`convolution`	Logical: return the convolution kernel? (Useful for density cross-validation.)

Details

The kernel functions take non-zero values on $[-1, 1]$, except for the Gaussian one, which is supposed to have full support, but due to the rapid decay, is indistinguishable from machine epsilon outside $[-8.2924, 8.2924]$.

Value

A numeric vector of the same length as input.

References

Silverman BW (1986). *Density estimation for statistics and data analysis*. New York: Chapman and Hall.

Examples

```
ks <- c("uniform", "triangular", "epanechnikov", "quartic", "gaussian"); names(ks) <- ks
os <- c(2, 4); names(os) <- paste0("o", os)
cols <- c("#000000CC", "#0000CCCC", "#CC0000CC", "#00AA00CC", "#BB8800CC")
put.legend <- function() legend("topright", legend = ks, lty = 1, col = cols, bty = "n")
xout <- seq(-4, 4, length.out = 301)
plot(NULL, NULL, xlim = range(xout), ylim = c(0, 1.1),
  xlab = "", ylab = "", main = "Unscaled kernels", bty = "n"); put.legend()
for (i in 1:5) lines(xout, kernelFun(xout, kernel = ks[i]), col = cols[i])
oldpar <- par(mfrow = c(1, 2))
plot(NULL, NULL, xlim = range(xout), ylim = c(-0.1, 0.8), xlab = "", ylab = "",
  main = "4th-order kernels", bty = "n"); put.legend()
for (i in 1:5) lines(xout, kernelFun(xout, kernel = ks[i], order = 4), col = cols[i])
par(mfrow = c(1, 1))
plot(NULL, NULL, xlim = range(xout), ylim = c(-0.25, 1.4), xlab = "", ylab = "",
  main = "Convolution kernels", bty = "n"); put.legend()
for (i in 1:5) {
  for (j in 1:2) lines(xout, kernelFun(xout, kernel = ks[i], order = os[j],
  convolution = TRUE), col = cols[i], lty = j)
}; legend("topleft", c("2nd order", "4th order"), lty = 1:2, bty = "n")
par(oldpar)

# All kernels integrate to correct values; we compute the moments
mom <- Vectorize(function(k, o, m, c) integrate(function(x) x^m * kernelFun(x, k, o,
  convolution = c), lower = -Inf, upper = Inf)$value)
for (m in 0:4) {
  cat("\nComputing integrals of x^", m, " * f(x). \nSimple unscaled kernel:\n", sep = "")
  print(round(outer(os, ks, function(o, k) mom(k, o, m = m, c = FALSE)), 4))
  cat("Convolution kernel:\n")
  print(round(outer(os, ks, function(o, k) mom(k, o, m = m, c = TRUE)), 4))
}
```

`kernelMixedDensity` *Density with conditioning on discrete and continuous variables*

Description

Density with conditioning on discrete and continuous variables

Usage

```
kernelMixedDensity(
  x,
  by,
  xout = NULL,
  byout = NULL,
  weights = NULL,
  parallel = FALSE,
```

```
  cores = 1,
  preschedule = TRUE,
  ...
)
```

Arguments

`x`	A numeric vector, matrix, or data frame containing observations. For density, the points used to compute the density. For kernel regression, the points corresponding to explanatory variables.
`by`	A variable containing unique identifiers of discrete categories.
`xout`	A vector or a matrix of data points with `ncol(xout) = ncol(x)` at which the user desires to compute the weights, density, or predictions. In other words, this is the requested evaluation grid. If `NULL`, then `x` itself is used as the grid.
`byout`	A variable containing unique identifiers of discrete categories for the output grid (same points as `xout`)
`weights`	A numeric vector of observation weights (typically counts) to perform weighted operations. If null, `rep(1, NROW(x))` is used. In all calculations, the total number of observations is assumed to be the sum of `weights`.
`parallel`	Logical: if `TRUE`, parallelises the calculation over the unique values of `by`. At this moment, supports only `parallel::mclapply` (therefore, will not work on Windows).
`cores`	Integer: the number of CPU cores to use. High core count = high RAM usage. If the number of unique values of 'by' is less than the number of cores requested, then, only `length(unique(by))` cores are used.
`preschedule`	Logical: passed as `mc.preschedule` to `mclapply`.
`...`	Passed to `kernelDensity`.

Value

A numeric vector of the density estimate of the same length as `nrow(xout)`.

Examples

```
# Estimating 3 densities on something like a panel
set.seed(1)
n <- 200
x <- c(rnorm(n), rchisq(n, 4)/4, rexp(n, 1))
by <- rep(1:3, each = n)
xgrid <- seq(-3, 6, 0.1)
out <- expand.grid(x = xgrid, by = 1:3)
fhat <- kernelMixedDensity(x = x, xout = out$x, by = by, byout = out$by)
plot(xgrid, dnorm(xgrid)/3, type = "l", bty = "n", lty = 2, ylim = c(0, 0.35),
     xlab = "", ylab = "Density")
lines(xgrid, dchisq(xgrid*4, 4)*4/3, lty = 2, col = 2)
lines(xgrid, dexp(xgrid, 1)/3, lty = 2, col = 3)
for (i in 1:3) {
  lines(xgrid, fhat[out$by == i], col = i, lwd = 2)
```

```
  rug(x[by == i], col = i)
}
legend("top", c("00", "10", "01", "11"), col = 2:5, lwd  = 2)
```

`kernelMixedSmooth` *Smoothing with conditioning on discrete and continuous variables*

Description

Smoothing with conditioning on discrete and continuous variables

Usage

```
kernelMixedSmooth(
  x,
  y,
  by,
  xout = NULL,
  byout = NULL,
  weights = NULL,
  parallel = FALSE,
  cores = 1,
  preschedule = TRUE,
  ...
)
```

Arguments

`x`	A numeric vector, matrix, or data frame containing observations. For density, the points used to compute the density. For kernel regression, the points corresponding to explanatory variables.
`y`	A numeric vector of dependent variable values.
`by`	A variable containing unique identifiers of discrete categories.
`xout`	A vector or a matrix of data points with `ncol(xout) = ncol(x)` at which the user desires to compute the weights, density, or predictions. In other words, this is the requested evaluation grid. If `NULL`, then `x` itself is used as the grid.
`byout`	A variable containing unique identifiers of discrete categories for the output grid (same points as `xout`)
`weights`	A numeric vector of observation weights (typically counts) to perform weighted operations. If null, `rep(1, NROW(x))` is used. In all calculations, the total number of observations is assumed to be the sum of `weights`.
`parallel`	Logical: if `TRUE`, parallelises the calculation over the unique values of `by`. At this moment, supports only `parallel::mclapply` (therefore, will not work on Windows).

`cores`	Integer: the number of CPU cores to use. High core count = high RAM usage. If the number of unique values of 'by' is less than the number of cores requested, then, only `length(unique(by))` cores are used.
`preschedule`	Logical: passed as `mc.preschedule` to `mclapply`.
`...`	Passed to `kernelSmooth` (usually `bw`, `gaussian` for both; `degree` and `robust.iterations` for "smooth"),

Value

A numeric vector of the kernel estimate of the same length as `nrow(xout)`.

Examples

```
set.seed(1)
n <- 1000
z1 <- rbinom(n, 1, 0.5)
z2 <- rbinom(n, 1, 0.5)
x <- rnorm(n)
u <- rnorm(n)
y <- 1 + x^2 + z1 + 2*z2 + z1*z2 + u
by <- as.integer(interaction(list(z1, z2)))
out <- expand.grid(x = seq(-4, 4, 0.25), by = 1:4)
yhat <- kernelMixedSmooth(x = x, y = y, by = by, bw = 1, degree = 1,
                          xout = out$x, byout = out$by)
plot(x, y)
for (i in 1:4) lines(out$x[out$by == i], yhat[out$by == i], col = i+1, lwd = 2)
legend("top", c("00", "10", "01", "11"), col = 2:5, lwd  = 2)

# The function works faster if there are duplicated values of the
# conditioning variables in the prediction grid and there are many
# observations; this is illustrated by the following example
# without a custom grid
# In this example, ignore the fact that the conditioning variable is rounded
# and therefore contains measurement error (ruining consistency)
x  <- rnorm(10000)
xout <- rnorm(5000)
xr <- round(x)
xrout <- round(xout)
w <- runif(10000, 1, 3)
y  <- 1 + x^2 + rnorm(10000)
by <- rep(1:4, each = 2500)
byout <- rep(1:4, each = 1250)
system.time(kernelMixedSmooth(x = x, y = y, by = by, weights = w,
                              xout = xout, byout = byout, bw = 1))
#  user  system elapsed
# 0.144   0.000   0.144
system.time(km1 <- kernelMixedSmooth(x = xr, y = y, by = by, weights = w,
                                     xout = xrout, byout = byout, bw = 1))
#  user  system elapsed
# 0.021   0.000   0.022
system.time(km2 <- kernelMixedSmooth(x = xr, y = y, by = by, weights = w,
                     xout = xrout, byout = byout, bw = 1, no.dedup = TRUE))
```

```
#  user  system elapsed
# 0.138   0.001   0.137
all.equal(km1, km2)

# Parallel capabilities shine in large data sets
if (.Platform$OS.type != "windows") {
# A function to carry out the same estimation in multiple cores
pFun <- function(n) kernelMixedSmooth(x = rep(x, 2), y = rep(y, 2),
         weights = rep(w, 2), by = rep(by, 2),
         bw = 1, degree = 0, parallel = TRUE, cores = n)
system.time(pFun(1))  # 0.6--0.7 s
system.time(pFun(2))  # 0.4--0.5 s
}
```

`kernelSmooth` *Local kernel smoother*

Description

Local kernel smoother

Usage

```
kernelSmooth(
  x,
  y,
  xout = NULL,
  weights = NULL,
  bw = NULL,
  kernel = c("gaussian", "uniform", "triangular", "epanechnikov", "quartic"),
  order = 2,
  convolution = FALSE,
  chunks = 0,
  PIT = FALSE,
  LOO = FALSE,
  degree = 0,
  trim = function(x) 0.01/length(x),
  robust.iterations = 0,
  robust = c("bisquare", "huber"),
  deduplicate.x = TRUE,
  deduplicate.xout = TRUE,
  no.dedup = FALSE,
  return.grid = FALSE
)
```

Arguments

`x`	A numeric vector, matrix, or data frame containing observations. For density, the points used to compute the density. For kernel regression, the points corresponding to explanatory variables.
`y`	A numeric vector of dependent variable values.
`xout`	A vector or a matrix of data points with `ncol(xout) = ncol(x)` at which the user desires to compute the weights, density, or predictions. In other words, this is the requested evaluation grid. If `NULL`, then `x` itself is used as the grid.
`weights`	A numeric vector of observation weights (typically counts) to perform weighted operations. If null, `rep(1, NROW(x))` is used. In all calculations, the total number of observations is assumed to be the sum of `weights`.
`bw`	Bandwidth for the kernel: a scalar or a vector of the same length as `ncol(x)`. Since it is the crucial parameter in many applications, a warning is thrown if the bandwidth is not supplied, and then, Silverman's rule of thumb (via `bw.row()`) is applied to *every dimension* of `x`.
`kernel`	Character describing the desired kernel type. NB: due to limited machine precision, even Gaussian has finite support.
`order`	An integer: 2, 4, or 6. Order-2 kernels are the standard kernels that are positive everywhere. Orders 4 and 6 produce some negative values, which reduces bias but may hamper density estimation.
`convolution`	Logical: if FALSE, returns the usual kernel. If TRUE, returns the convolution kernel that is used in density cross-validation.
`chunks`	Integer: the number of chunks to split the task into (limits RAM usage but increases overhead). `0` = auto-select (making sure that no matrix has more than 2^27 elements).
`PIT`	If TRUE, the Probability Integral Transform (PIT) is applied to all columns of `x` via `ecdf` in order to map all values into the [0, 1] range. May be an integer vector of indices of columns to which the PIT should be applied.
`LOO`	Logical: If `TRUE`, the leave-one-out estimator is returned.
`degree`	Integer: 0 for locally constant estimator (Nadaraya–Watson), 1 for locally linear (Cleveland's LOESS), 2 for locally quadratic (use with care, less stable, requires larger bandwidths)
`trim`	Trimming function for small weights to speed up locally weighted regression (if `degree` is 1 or 2).
`robust.iterations`	The number of robustifying iterations (due to Cleveland, 1979). If greater than 0, `xout` is ignored.
`robust`	Character: "huber" for Huber's local regression weights, "bisquare" for more robust bi-square ones
`deduplicate.x`	Logical: if TRUE, full duplicates in the input `x` and `y` are counted and transformed into weights; subsetting indices to reconstruct the duplicated data set from the unique one are also returned.

`deduplicate.xout`	Logical: if TRUE, full duplicates in the input `xout` are counted; subsetting indices to reconstruct the duplicated data set from the unique one are returned.
`no.dedup`	Logical: if TRUE, sets `deduplicate.x` and `deduplicate.xout` to FALSE (shorthand).
`return.grid`	If `TRUE`, prepends `xout` to the return results.

Standardisation is recommended for the purposes of numerical stability (sometimes `lm()` might choke when the dependent variable takes very large absolute values and its square is used).

The robust iterations are carried out, if requested, according to @cleveland1979robust. Huber weights are never zero; bisquare weights create a more robust re-descending estimator.

Note: if `x` and `xout` are different but robust iterations were requested, the robustification can take longer. TODO: do not estimate on (x, grid), do the calculation with K.full straight away.

Note: if `LOO` is used, it makes sense to de-duplicate observations first. By default, this behaviour is not enforced in this function, but when it is called in cross-validation routines, the de-duplication is forced. It makes no sense to zero out once observation out of many repeated.

Value

A vector of predicted values or, if `return.grid` is `TRUE`, a matrix with the predicted values in the last column.

Examples

```
set.seed(1)
n <- 300
x <- sort(rt(n, df = 6)) # Observed values
g <- seq(-4, 5, 0.1) # Grid for evaluation
f <- function(x) 1 + x + sin(x) # True E(Y | X) = f(X)
y <- f(x) + rt(n, df = 4)
# 3 estimators: locally constant + 2nd-order kernel,
# locally constant + 4th-order kernel, locally linear robust
b2lc <- suppressWarnings(bw.CV(x, y = y, kernel = "quartic")
                         + 0.8)
b4lc <- suppressWarnings(bw.CV(x, y = y, kernel = "quartic", order = 4,
              try.grid = FALSE, start.bw = 3) + 1)
b2ll <- bw.CV(x, y = y, kernel = "quartic", degree = 1, robust.iterations = 1,
              try.grid = FALSE, start.bw = 3, verbose = TRUE)
m2lc <- kernelSmooth(x, y, g, bw = b2lc, kernel = "quartic", no.dedup = TRUE)
m4lc <- kernelSmooth(x, y, g, bw = b4lc, kernel = "quartic", order = 4, no.dedup = TRUE)
m2ll <- kernelSmooth(x, y, g, bw = b2ll, kernel = "quartic",
                     degree = 1, robust.iterations = 1, no.dedup = TRUE)
plot(x, y, xlim = c(-6, 7), col = "#00000088", bty = "n")
lines(g, f(g), col = "white", lwd = 5); lines(g, f(g))
lines(g, m2lc, col = 2); lines(g, m4lc, col = 3); lines(g, m2ll, col = 4)
# De-duplication facilities for faster operations
set.seed(1)  # Creating a data set with many duplicates
```

```
n.uniq <- 1000
n <- 4000
inds <- sort(ceiling(runif(n, 0, n.uniq)))
x.uniq <- sort(rnorm(n.uniq))
y.uniq <- 1 + x.uniq + sin(x.uniq*2) + rnorm(n.uniq)
x <- x.uniq[inds]
y <- y.uniq[inds]
xout <- x.uniq[sort(ceiling(runif(n.uniq*3, 0, n.uniq)))]
w <- runif(n)
data.table::setDTthreads(1) # For measuring the pure gains and overhead
RcppParallel::setThreadOptions(numThreads = 1)
kr1 <- kernelSmooth(x, y, xout, w, bw = 0.2)
kr2 <- kernelSmooth(x, y, xout, w, bw = 0.5, no.dedup = TRUE)
stat1 <- attr(kr1, "duplicate.stats")
stat2 <- attr(kr2, "duplicate.stats")
print(stat1[3:5]) # De-duplication time -- worth it
print(stat2[3:5]) # Without de-duplication, slower
unname(prod((1 - stat1[1:2])) / (stat1[5] / stat2[5])) # > 1 = better time
# savings than expected, < 1 = worse time savings than expected
all.equal(as.numeric(kr1), as.numeric(kr2))
max(abs(kr1 - kr2)) # Should be around machine epsilon or less

# Example in 2 dimensions
# TODO
```

`kernelWeights` *Kernel-based weights*

Description

Kernel-based weights

Usage

```
kernelWeights(
  x,
  xout = NULL,
  bw = NULL,
  kernel = c("gaussian", "uniform", "triangular", "epanechnikov", "quartic"),
  order = 2,
  convolution = FALSE,
  sparse = FALSE,
  PIT = FALSE,
  deduplicate.x = FALSE,
  deduplicate.xout = FALSE,
  no.dedup = FALSE
)
```

Arguments

`x`	A numeric vector, matrix, or data frame containing observations. For density, the points used to compute the density. For kernel regression, the points corresponding to explanatory variables.
`xout`	A vector or a matrix of data points with `ncol(xout) = ncol(x)` at which the user desires to compute the weights, density, or predictions. In other words, this is the requested evaluation grid. If `NULL`, then `x` itself is used as the grid.
`bw`	Bandwidth for the kernel: a scalar or a vector of the same length as `ncol(x)`. Since it is the crucial parameter in many applications, a warning is thrown if the bandwidth is not supplied, and then, Silverman's rule of thumb (via `bw.row()`) is applied to *every dimension* of `x`.
`kernel`	Character describing the desired kernel type. NB: due to limited machine precision, even Gaussian has finite support.
`order`	An integer: 2, 4, or 6. Order-2 kernels are the standard kernels that are positive everywhere. Orders 4 and 6 produce some negative values, which reduces bias but may hamper density estimation.
`convolution`	Logical: if FALSE, returns the usual kernel. If TRUE, returns the convolution kernel that is used in density cross-validation.
`sparse`	Logical: TODO (should be ignored?) Note that if `pit = TRUE`, then the kernel-based weights become nearest-neighbour weights (i.e. not much different from the ones used internally in the built-in `loess` function) since the distances now depend on the ordering of data, not the values per se. Technical remark: if the kernel is Gaussian, then, the ratio of the tail density to the maximum value (at 0) is less than mach.eps/2 when abs(x) > 2*sqrt(106*log(2)) ~ 8.572. This has implications the relative error of the calculation: even the kernel with full support (theoretically) may fail to produce numerically distinct values if the argument values are more than ~8.5 standard deviations away from the mean.
`PIT`	If TRUE, the Probability Integral Transform (PIT) is applied to all columns of `x` via `ecdf` in order to map all values into the [0, 1] range. May be an integer vector of indices of columns to which the PIT should be applied.
`deduplicate.x`	Logical: if TRUE, full duplicates in the input `x` and `y` are counted and transformed into weights; subsetting indices to reconstruct the duplicated data set from the unique one are also returned.
`deduplicate.xout`	Logical: if TRUE, full duplicates in the input `xout` are counted; subsetting indices to reconstruct the duplicated data set from the unique one are returned.
`no.dedup`	Logical: if TRUE, sets `deduplicate.x` and `deduplicate.xout` to FALSE (shorthand).

Value

A matrix of weights of dimensions nrow(xout) x nrow(x).

Examples

```
set.seed(1)
x   <- sort(rnorm(1000)) # Observed values
g   <- seq(-10, 10, 0.1) # Grid for evaluation
w   <- kernelWeights(x, g, bw = 2, kernel = "triangular")
wsp <- kernelWeights(x, g, bw = 2, kernel = "triangular", sparse = TRUE)
print(c(object.size(w), object.size(wsp)) / 1024) # Kilobytes used
image(g, x, w)
all.equal(w[, 1],  # Internal calculation for one column
            kernelFun((g - x[1])/2, "triangular", 2, FALSE))

# Bare-bones interface to the C++ functions
# Example: 4th-order convolution kernels
x <- seq(-3, 5, length.out = 301)
ks <- c("uniform", "triangular", "epanechnikov", "quartic", "gaussian")
kmat <- sapply(ks, function(k) kernelFun(x, k, 4, TRUE))
matplot(x, kmat, type = "l", lty = 1, bty = "n", lwd = 2)
legend("topright", ks, col = 1:5, lwd = 2)
```

`logTaylor` *Modified logarithm with derivatives*

Description

Modified logarithm with derivatives

Usage

```
logTaylor(x, lower = NULL, upper = NULL, der = 0, order = 4)
```

Arguments

`x`	Numeric vector for which approximated logarithm is to be computed.
`lower`	Lower threshold below which approximation starts; can be a scalar of a vector of the same length as `x`.
`upper`	Upper threshold above which approximation starts; can be a scalar of a vector of the same length as `x`.
`der`	Non-negative integer: 0 yields the function, 1 and higher yields derivatives
`order`	Positive integer: Taylor approximation order. If `NA`, returns `log(x)` or its derivative.

Details

Provides a family of alternatives to -log() and derivative thereof in order to attain self-concordance and computes the modified negative logarithm and its first derivatives. For lower <= x <= upper, returns just the logarithm. For x < lower and x > upper, returns the Taylor approximation of the given `order`. 4th order is the lowest that gives self concordance.

Value

A numeric matrix with `(order+1)` columns containing the values of the modified log and its derivatives.

Examples

```
x <- seq(0.01^0.25, 2^0.25, length.out = 51)^4 - 0.11 # Denser where |f'| is higher
plot(x,  log(x)); abline(v = 0, lty = 2) # Observe the warning
lines(x, logTaylor(x, lower = 0.2), col = 2)
lines(x, logTaylor(x, lower = 0.5), col = 3)
lines(x, logTaylor(x, lower = 1, upper = 1.2, order = 6), col = 4)

# Substitute log with its Taylor approx. around 1
x <- seq(0.1, 2, 0.05)
ae <- abs(sapply(2:6, function(o) log(x) - logTaylor(x, lower=1, upper=1, order=o)))
matplot(x[x!=1], ae[x!=1,], type = "l", log = "y", lwd = 2,
  main = "Abs. trunc. err. of Taylor expansion at 1", ylab = "")

# Vanilla logarithm
identical(logTaylor(2, order = NA), log(2))
```

`LSCV` *Least-squares cross-validation function for the Nadaraya-Watson estimator*

Description

Least-squares cross-validation function for the Nadaraya-Watson estimator

Usage

```
LSCV(
  x,
  y,
  bw,
  weights = NULL,
  same = FALSE,
  degree = 0,
  kernel = "gaussian",
  order = 2,
  PIT = FALSE,
  chunks = 0,
  robust.iterations = 0,
  cores = 1
)
```

Arguments

`x`	A numeric vector, matrix, or data frame containing observations. For density, the points used to compute the density. For kernel regression, the points corresponding to explanatory variables.
`y`	A numeric vector of dependent variable values.
`bw`	Candidate bandwidth values: scalar, vector, or a matrix (with columns corresponding to columns of `x`).
`weights`	A numeric vector of observation weights (typically counts) to perform weighted operations. If null, `rep(1, NROW(x))` is used. In all calculations, the total number of observations is assumed to be the sum of `weights`.
`same`	Logical: use the same bandwidth for all columns of `x`?
`degree`	Integer: 0 for locally constant estimator (Nadaraya–Watson), 1 for locally linear (Cleveland's LOESS), 2 for locally quadratic (use with care, less stable, requires larger bandwidths)
`kernel`	Character describing the desired kernel type. NB: due to limited machine precision, even Gaussian has finite support.
`order`	An integer: 2, 4, or 6. Order-2 kernels are the standard kernels that are positive everywhere. Orders 4 and 6 produce some negative values, which reduces bias but may hamper density estimation.
`PIT`	If TRUE, the Probability Integral Transform (PIT) is applied to all columns of `x` via `ecdf` in order to map all values into the [0, 1] range. May be an integer vector of indices of columns to which the PIT should be applied.
`chunks`	Integer: the number of chunks to split the task into (limits RAM usage but increases overhead). `0` = auto-select (making sure that no matrix has more than 2^27 elements).
`robust.iterations`	The number of robustifying iterations (due to Cleveland, 1979). If greater than 0, `xout` is ignored.
`cores`	Integer: the number of CPU cores to use. High core count = high RAM usage. Note: since LSCV requires zeroing out the diagonals of the weight matrix, repeated observations are combined first; the de-duplication is therefore forced in cross-validation. The only situation where de-duplication can be skipped is passing de-duplicated data sets from outside (e.g. inside optimisers).

Value

A numeric vector of the same length as `bw` or `nrow(bw)`.

Examples

```
set.seed(1)  # Creating a data set with many duplicates
n.uniq <- 1000
n <- 4000
inds <- sort(ceiling(runif(n, 0, n.uniq)))
x.uniq <- sort(rnorm(n.uniq))
```

```
y.uniq <- 1 + 0.2*x.uniq + 0.3*sin(x.uniq) + rnorm(n.uniq)
x <- x.uniq[inds]
y <- y.uniq[inds]
w <- 1 + runif(n, 0, 2) # Relative importance
data.table::setDTthreads(1) # For measuring pure gains and overhead
RcppParallel::setThreadOptions(numThreads = 1)
bw.grid <- seq(0.1, 1.2, 0.05)
ncores <- if (.Platform$OS.type == "windows") 1 else 2
CV <- LSCV(x, y, bw.grid, weights = w, cores = ncores)  # Parallel capabilities
bw.opt <- bw.grid[which.min(CV)]
g <- seq(-3.5, 3.5, 0.05)
yhat <- kernelSmooth(x, y, xout = g, weights = w,
                     bw = bw.opt, deduplicate.xout = FALSE)
oldpar <- par(mfrow = c(2, 1), mar = c(2, 2, 2, 0)+.1)
plot(bw.grid, CV, bty = "n", xlab = "", ylab = "", main = "Cross-validation")
plot(x.uniq, y.uniq, bty = "n", xlab = "", ylab = "", main = "Optimal fit")
points(g, yhat, pch = 16, col = 2, cex = 0.5)
par(oldpar)
```

`pit` *Probability integral transform*

Description

Probability integral transform

Usage

```
pit(x, xout = NULL)
```

Arguments

`x`	A numeric vector of data points.
`xout`	A numeric vector. If supplied, then the transformed function at the grid points different from `x` takes values equidistant between themselves and the ends of the interval to which they belong.

Value

A numeric vector of values strictly between 0 and 1 of the same length as `xout` (or `x`, if `xout` is `NULL`).

Examples

```
set.seed(2)
x1 <- c(4, 3, 7, 10, 2, 2, 7, 2, 5, 6)
x2 <- sample(c(0, 0.5, 1, 2, 2.5, 3, 3.5, 10, 100), 25, replace = TRUE)
l <- length(x1)
pit(x1)
```

```
plot(pit(x1), ecdf(x1)(x1), xlim = c(0, 1), ylim = c(0, 1), asp = 1)
abline(v = seq(0.5 / l, 1 - 0.5 / l, length.out = l), col = "#00000044", lty = 2)
abline(v = c(0, 1))
points(pit(x1, x2), ecdf(x1)(x2), pch = 16, col = "#CC000088", cex = 0.9)
abline(v = pit(x1, x2), col = "#CC000044", lty = 2)

x1 <- c(1, 1, 3, 4, 6)
x2 <- c(0, 2, 2, 5.9, 7, 8)
pit(x1)
pit(x1, x2)

set.seed(1)
l <- 10
x1 <- rlnorm(l)
x2 <- sample(c(x1, rlnorm(10)))
plot(pit(x1), ecdf(x1)(x1), xlim = c(0, 1), ylim = c(0, 1), asp = 1)
abline(v = seq(0.5 / l, 1 - 0.5 / l, length.out = l), col = "#00000044", lty = 2)
abline(v = c(0, 1))
points(pit(x1, x2), ecdf(x1)(x2), pch = 16, col = "#CC000088", cex = 0.9)
```

`prepareKernel` *Check the data for kernel estimation*

Description

Checks if the order is 2, 4, or 6, transforms the objects into matrices, checks the dimensions, provides the bandwidth, creates default arguments to pass to the C++ functions, carries out deduplication for speed-up etc.

Usage

```
prepareKernel(
  x,
  y = NULL,
  xout = NULL,
  weights = NULL,
  bw = NULL,
  kernel = c("gaussian", "uniform", "triangular", "epanechnikov", "quartic"),
  order = 2,
  convolution = FALSE,
  sparse = FALSE,
  deduplicate.x = TRUE,
  deduplicate.xout = TRUE,
  no.dedup = FALSE,
  PIT = FALSE
)
```

Arguments

`x`	A numeric vector, matrix, or data frame containing observations. For density, the points used to compute the density. For kernel regression, the points corresponding to explanatory variables.
`y`	Optional: a vector of dependent variable values.
`xout`	A vector or a matrix of data points with `ncol(xout) = ncol(x)` at which the user desires to compute the weights, density, or predictions. In other words, this is the requested evaluation grid. If `NULL`, then `x` itself is used as the grid.
`weights`	A numeric vector of observation weights (typically counts) to perform weighted operations. If null, `rep(1, NROW(x))` is used. In all calculations, the total number of observations is assumed to be the sum of `weights`.
`bw`	Bandwidth for the kernel: a scalar or a vector of the same length as `ncol(x)`. Since it is the crucial parameter in many applications, a warning is thrown if the bandwidth is not supplied, and then, Silverman's rule of thumb (via `bw.row()`) is applied to *every dimension* of `x`.
`kernel`	Character describing the desired kernel type. NB: due to limited machine precision, even Gaussian has finite support.
`order`	An integer: 2, 4, or 6. Order-2 kernels are the standard kernels that are positive everywhere. Orders 4 and 6 produce some negative values, which reduces bias but may hamper density estimation.
`convolution`	Logical: if FALSE, returns the usual kernel. If TRUE, returns the convolution kernel that is used in density cross-validation.
`sparse`	Logical: TODO (ignored)
`deduplicate.x`	Logical: if TRUE, full duplicates in the input `x` and `y` are counted and transformed into weights; subsetting indices to reconstruct the duplicated data set from the unique one are also returned.
`deduplicate.xout`	Logical: if TRUE, full duplicates in the input `xout` are counted; subsetting indices to reconstruct the duplicated data set from the unique one are returned.
`no.dedup`	Logical: if TRUE, sets `deduplicate.x` and `deduplicate.xout` to FALSE (shorthand).
`PIT`	If TRUE, the Probability Integral Transform (PIT) is applied to all columns of `x` via `ecdf` in order to map all values into the [0, 1] range. May be an integer vector of indices of columns to which the PIT should be applied.

Value

A list of arguments that are taken by [kernelDensity()] and [kernelSmooth()].

Examples

```
# De-duplication facilities
set.seed(1)  # Creating a data set with many duplicates
n.uniq <- 10000
n <- 60000
```

```
inds <- ceiling(runif(n, 0, n.uniq))
x.uniq <- matrix(rnorm(n.uniq*10), ncol = 10)
x <- x.uniq[inds, ]
y <- runif(n.uniq)[inds]
xout <- x.uniq[ceiling(runif(n.uniq*3, 0, n.uniq)), ]
w <- runif(n)
print(system.time(a1 <- prepareKernel(x, y, xout, w, bw = 0.5)))
print(system.time(a2 <- prepareKernel(x, y, xout, w, bw = 0.5,
                 deduplicate.x = FALSE, deduplicate.xout = FALSE)))
print(c(object.size(a1), object.size(a2)) / 1024) # Kilobytes used
# Speed-memory trade-off: 4 times smaller, takes 0.2 s, but reduces the
# number of matrix operations by a factor of
1 - prod(1 - a1$duplicate.stats[1:2])    # 95% fewer operations
sum(a1$weights) - sum(a2$weights)  # Should be 0 or near machine epsilon
```

`smoothEmplik` *Smoothed Empirical Likelihood function value*

Description

Evaluates SEL function for a given moment function at a certain parameter value.

Usage

```
smoothEmplik(
  rho,
  theta,
  data,
  sel.weights = NULL,
  weight.tolerance = 0,
  type = c("auto", "EL0", "EL1", "EuL"),
  chull.fail = c("none", "taylor", "wald", "adjusted", "adjusted2", "balanced"),
  kernel.args = list(bw = NULL, kernel = "epanechnikov", order = 2, PIT = TRUE, sparse =
    TRUE),
  minus = FALSE,
  cores = 1,
  chunks = NULL,
  sparse = FALSE,
  verbose = FALSE,
  bad.value = -Inf,
  attach.attributes = c("none", "all", "ELRs", "residuals", "lam", "nabla", "converged",
    "exitcode", "probabilities"),
  ...
)
```

Arguments

`rho`	The moment function depending on parameters and data (and potentially other parameters). Must return a numeric vector.

`theta`	A parameter at which the moment function is evaluated.
`data`	A data object on which the moment function is computed.
`sel.weights`	Either a matrix with valid kernel smoothing weights with rows adding up to 1, or a function that computes the kernel weights based on the `data` argument passed to `...`.
`weight.tolerance`	Passed to [EL()].
`type`	Character: `"auto"` for empirical likelihood, `"EuL"` for Euclidean likelihood, `"EL0"` for one-dimensional empirical likelihood. `"EL0"` is *strongly* recommended for 1-dimensional moment functions because it is faster and more robust: it searches for the Lagrange multiplier directly and has nice fail-safe options for convex hull failure.
`chull.fail`	Passed to [EL()].
`kernel.args`	A list of arguments passed to `kernelWeights()` if `sel.weights` is a function.
`minus`	If TRUE, returns SEL times -1 (for optimisation via minimisation).
`cores`	The number of cores used by `parallel::mclapply` to speed up the computation.
`chunks`	The number of chunks into which the weight matrix is split for memory saving. One chunk is good for sample sizes 2000 and below. If equal to the number of observations, then, the smoothed likelihoods are computed in series, which saves memory but computes kernel weights at every step of a loop, increasing CPU time. If `cores` is greater than 1, parallelisation occurs within each chunk.
`sparse`	Logical: convert the weight matrix to a sparse one?
`verbose`	If `TRUE`, a progress bar is made to display the evaluation progress in case partial or full memory saving is in place.
`bad.value`	Replace non-finite individual SEL values with this value. May be useful if the optimiser does not allow specific non-finite values (like L-BFGS-B).
`attach.attributes`	If `"none"`, returns just the sum of expected likelihoods; otherwise, attaches certain attributes for diagnostics: `"ELRs"` for expected likelihoods, `"residuals"` for the residuals (moment function values), `"lam"` for the Lagrange multipliers lambda in the EL problems, `"nabla"` for d/d(lambda)EL (should be close to zero because this must be true for any `theta`), `"converged"` for the convergence of #' individual EL problems, `"exitcode"` for the `EL` exit codes (0 for success), `"probabilities"` for the matrix of weights (very large, not recommended for sample sizes larger than 2000).
`...`	Passed to `rho`.

Value

A scalar with the SEL value and, if requested, attributes containing the diagnostic information attached to it.

Examples

```
set.seed(1)
x <- sort(rlnorm(50))
# Heteroskedastic DGP
y <- abs(1 + 1*x + rnorm(50) * (1 + x + sin(x)))
mod.OLS <- lm(y ~ x)
rho <- function(theta, ...) y - theta[1] - theta[2]*x  # Moment fn
w <- kernelWeights(x, PIT = TRUE, bw = 0.25, kernel = "epanechnikov")
w <- w / rowSums(w)
image(x, x, w, log = "xy")
theta.vals <- list(c(1, 1), coef(mod.OLS))
SEL <- function(b, ...) smoothEmplik(rho = rho, theta = b, sel.weights = w, ...)
sapply(theta.vals, SEL) # Smoothed empirical likelihood
# SEL maximisation
ctl <- list(fnscale = -1, reltol = 1e-6, ndeps = rep(1e-5, 2),
            trace = 1, REPORT = 5)
b.init <- coef(mod.OLS)
b.init <- c(1.790207, 1.007491)  # Only to speed up estimation
b.SEL <- optim(b.init, SEL, method = "BFGS", control = ctl)
print(b.SEL$par) # Closer to the true value (1, 1) than OLS
plot(x, y)
abline(1, 1, lty = 2)
abline(mod.OLS, col = 2)
abline(b.SEL$par, col = 4)

# Euclidean likelihood
SEuL <- function(b, ...)  smoothEmplik(rho = rho, theta = b,
                                       type = "EuL", sel.weights = w, ...)
b.SEuL <- optim(coef(mod.OLS), SEuL, method = "BFGS", control = ctl)
abline(b.SEuL$par, col = 3)
cbind(SEL = b.SEL$par, SEuL = b.SEuL$par)

# Now we start from (0, 0), for which an extension is necessary
# because all residuals at this starting value are positive and the
# unmodified EL ratio for the test of equality to 0 is -Inf
if (FALSE) {
SEL(c(0, 0))
SEL(c(0, 0), chull.fail = "taylor")
SEL(c(0, 0), chull.fail = "wald")
SEL(c(0, 0), chull.fail = "adjusted")
SEL(c(0, 0), chull.fail = "adjusted2")
SEL(c(0, 0), chull.fail = "balanced")

# The next example is very slow; approx. 1 minute

# Experiment: a small bandwidth so that the spanning condition should fail often
# It yields an appalling estimator
w <- kernelWeights(x, PIT = TRUE, bw = 0.15, kernel = "epanechnikov")
w <- w / rowSums(w)
# The first option is faster but it may sometimes fails
b.SELt <- optim(c(0, 0), SEL, chull.fail = "taylor",
                method = "BFGS", control = ctl)
```

```
b.SELw <- optim(c(0, 0), SEL, chull.fail = "wald",
                method = "BFGS", control = ctl)

w <- kernelWeights(x, PIT = TRUE, bw = 0.15, kernel = "epanechnikov")
w <- w / rowSums(w)
# In this sense, Euclidean likelihood is robust to convex hull violations
b.SELu <- optim(c(0, 0), SEuL, method = "BFGS", control = ctl)
b0grid <- seq(-1.5, 7, length.out = 51)
b1grid <- seq(-1.5, 4.5, length.out = 51)
bgrid <- as.matrix(expand.grid(b0grid, b1grid))
fi <- function(i) smoothEmplik(rho, bgrid[i, ], sel.weights = w, type = "EL0",
                  EL.args = list(chull.fail = "taylor"))
ncores <- max(floor(parallel::detectCores()/2 - 1), 1)
chk <- Sys.getenv("_R_CHECK_LIMIT_CORES_", "")  # Limit to 2 cores for CRAN checks
if (nzchar(chk) && chk == "TRUE") ncores <- min(ncores, 2L)
selgrid <- unlist(parallel::mclapply(1:nrow(bgrid), fi, mc.cores = ncores))
selgrid <- matrix(selgrid, nrow = length(b0grid))
probs <- c(0.25, 0.5, 0.75, 0.8, 0.9, 0.95, 0.99, 1-10^seq(-4, -16, -2))
levs <- qchisq(probs, df = 2)
# levs <- c(1, 2, 5, 10, 20, 50, 100, 200, 500, 1000, 2000, 5000, 10000)
labs <- round(levs, 1)
cols <- rainbow(length(levs), end = 0.7, v = 0.7)
oldpar <- par(mar = c(4, 4, 2, 0) + .1)
selgrid2 <- -2*(selgrid - max(selgrid, na.rm = TRUE))
contour(b0grid, b1grid, selgrid2, levels = levs,
        labels = labs, col = cols, lwd = 1.5, bty = "n",
        main = "'Safe' likelihood contours", asp = 1)
image(b0grid, b1grid, log1p(selgrid2))
# The narrow lines are caused by the fact that if two observations are close together
# at the edge, the curvature at that point is extreme

# The same with Euclidean likelihood
seulgrid <- unlist(parallel::mclapply(1:nrow(bgrid), function(i)
  smoothEmplik(rho, bgrid[i, ], sel.weights = w, type = "EuL"),
    mc.cores = ncores))
seulgrid <- matrix(seulgrid, nrow = length(b0grid))
seulgrid2 <- -50*(seulgrid - max(seulgrid, na.rm = TRUE))
par(mar = c(4, 4, 2, 0) + .1)
contour(b0grid, b1grid, seulgrid2, levels = levs,
        labels = labs, col = cols, lwd = 1.5, bty = "n",
        main = "'Safe' likelihood contours", asp = 1)
image(b0grid, b1grid, log1p(seulgrid2))
par(oldpar)
}
```

`sparseVectorToList` *Convert a weight vector to list*

Description

This function saves memory (which is crucial in large samples) and allows one to speed up the code by minimising the number of time-consuming subsetting operations and memory-consuming matrix multiplications. We do not want to rely on extra packages for sparse matrix manipulation since the EL smoothing weights are usually fixed at the beginning, and need not be recomputed dynamically, so we recommend applying this function to the rows of a matrix. In order to avoid numerical instability, the weights are trimmed at `0.01 / length(x)`. Using too much trimming may cause the spanning condition to fail (the moment function values can have the same sign in some neighbourhoods).

Usage

```
sparseVectorToList(x, trim = NULL, renormalise = FALSE)

sparseMatrixToList(x, trim = NULL, renormalise = FALSE)
```

Arguments

`x`	A numeric vector or matrix (with many close-to-zero elements).
`trim`	A trimming function that returns a threshold value below which the weights are ignored. In common applications, this function should tend to 0 as the length of `x` increases.
`renormalise`	Logical: renormalise the sum of weights to one after trimming?

Value

A list with indices and values of non-zero elements.

Examples

```
set.seed(1)
m <- round(matrix(rnorm(100), 10, 10), 2)
m[as.logical(rbinom(100, 1, 0.7))] <- 0
sparseVectorToList(m[, 3])
sparseMatrixToList(m)
```

`svdlm` *Least-squares regression via SVD*

Description

Least-squares regression via SVD

Usage

```
svdlm(x, y, rel.tol = 1e-09, abs.tol = 1e-100)
```

Arguments

`x`	Model matrix.
`y`	Response vector.
`rel.tol`	Relative zero tolerance for generalised inverse via SVD.
`abs.tol`	Absolute zero tolerance for generalised inverse via SVD. Newton steps for many empirical likelihoods are of least-squares type. Denote x^+ to be the generalised inverse of `x`. If SVD algorithm failures are encountered, it sometimes helps to try `svd(t(x))` and translate back. First check to ensure that `x` does not contain `NaN`, or `Inf`, or `-Inf`. The tolerances are used to check the closeness of singular values to zero. The values of the singular-value vector `d` that are less than `max(rel.tol * max(d), abs.tol)` are set to zero.

Value

A vector of coefficients.

Examples

```
b.svd <- svdlm(x = cbind(1, as.matrix(mtcars[, -1])), y = mtcars[, 1])
b.lm  <- coef(lm(mpg ~ ., data = mtcars))
b.lm - b.svd  # Negligible differences
```

`tlog` *d-th derivative of the k-th-order Taylor expansion of log(x)*

Description

d-th derivative of the k-th-order Taylor expansion of log(x)

Usage

```
tlog(x, a = as.numeric(c(1)), k = 4L, d = 0L)
```

Arguments

`x`	Numeric: a vector of points for which the logarithm is to be evaluated
`a`	Scalar: the point at which the polynomial approximation is computed
`k`	Non-negative integer: maximum polynomial order in the Taylor expansion of the original function. `k = 0` returns a constant.
`d`	Non-negative integer: derivative order Note that this function returns the d-th derivative of the k-th-order Taylor expansion, not the k-th-order approximation of the d-th derivative. Therefore, the degree of the resulting polynomial is $d - k$.

Value

The approximating Taylor polynomial around `a` of the order `d-k` evaluated at `x`.

Examples

```
cl <- rainbow(9, end = 0.8, v = 0.8, alpha = 0.8)
a <- 1.5
x <- seq(a*2, a/2, length.out = 101)
f <- function(x, d = 0)  if (d == 0) log(x) else ((d%%2 == 1)*2-1) * 1/x^d * gamma(d)
oldpar <- par(mfrow = c(2, 3), mar = c(2, 2, 2.5, 0.2))
for (d in 0:5) {
y <- f(x, d = d)
plot(x, y, type = "l", lwd = 7, bty = "n", ylim = range(0, y),
       main = paste0("d^", d, "/dx^", d, " Taylor(Log(x))"))
  for (k in 0:8) lines(x, tlog(x, a = a, k = k, d = d), col = cl[k+1], lwd = 1.5)
  points(a, f(a, d = d), pch = 16, cex = 1.5, col = "white")
}
legend("topright", as.character(0:8), title = "Order", col = cl, lwd = 1)
par(oldpar)
```

`trimmed.weighted.mean` *Weighted trimmed mean*

Description

Compute a weighted trimmed mean, i.e. a mean that assigns non-negative weights to the observations and (2) discards an equal share of total weight from each tail of the distribution before averaging.

Usage

```
trimmed.weighted.mean(x, trim = 0, w = NULL, na.rm = FALSE, ...)
```

Arguments

`x`	Numeric vector of data values.
`trim`	Single number in $[0, 0.5]$. Fraction of the total weight to cut from each tail.
`w`	Numeric vector of non-negative weights of the same length as ‘x‘. If ‘NULL‘ (default), equal weights are used.
`na.rm`	Logical: should ‘NA‘ values in ‘x‘ or ‘w‘ be removed?
`...`	Further arguments passed to [‘weighted.mean()‘] (for compatibility).

Details

For example, ‘trim = 0.10‘ removes 10 from the right (20 Setting ‘trim = 0.5‘ returns the weighted median.

The algorithm follows these steps:

1. Sort the data by ‘x‘ and accumulate the corresponding weights.
2. Identify the lower and upper cut-points that mark the central share of the total weight.
3. Drop observations whose cumulative weight lies entirely outside the cut-points and proportionally down-weight the two (at most) remaining outermost observations.
4. Return the weighted mean of the retained mass. If ‘trim == 0.5‘, only the 50

Value

A single numeric value: the trimmed weighted mean of ‘x‘. Returns ‘NA_real_‘ if no non-‘NA‘ observations remain after optional ‘na.rm‘ handling.

See Also

[‘mean()‘] for the unweighted trimmed mean, [‘weighted.mean()‘] for the untrimmed weighted mean.

Examples

```
set.seed(1)
z <- rt(100, df = 3)
w <- pmin(1, 1 / abs(z)^2)  # Far-away observations tails get lower weight

mean(z, trim = 0.20)  # Ordinary trimmed mean
trimmed.weighted.mean(z, trim = 0.20)  # Same

weighted.mean(z, w)   # Ordinary weighted mean (no trimming)
trimmed.weighted.mean(z, w = w)  # Same

trimmed.weighted.mean(z, trim = 0.20, w = w)  # Weighted trimmed mean
trimmed.weighted.mean(z, trim = 0.5,  w = w)  # Weighted median
```

Index