Package: steinsampling (via r-universe)

July 21, 2026

Title Kernel Stein Discrepancy Goodness-of-Fit and Stein Sampling Tools

Version 0.1.0

Date 2026-07-11

Maintainer Junhao Gao <jug049@ucsd.edu>

Description Provides Stein-discrepancy goodness-of-fit tests and Stein-method-based sampling tools. The tests include kernel Stein discrepancy U- and V-statistics following Liu et al. (2016) <doi:10.48550/arXiv.1602.03253> and Chwialkowski et al. (2016) <doi:10.48550/arXiv.1602.02964>, plus the finite set Stein discrepancy test of Jitkrittum et al. (2017) <doi:10.48550/arXiv.1705.07673>. The sampling tools include Stein thinning, Stein Points, Stein Point Markov chain Monte Carlo, and Stein variational gradient descent following Riabiz et al. (2022) <doi:10.48550/arXiv.2005.03952>, Chen et al. (2018) <doi:10.48550/arXiv.1803.10161>, Chen et al. (2019) <doi:10.48550/arXiv.1905.03673>, and Liu and Wang (2016) <doi:10.48550/arXiv.1608.04471>. Gaussian mixture utilities are included for simulation, likelihoods, posterior probabilities, scores, and plots.

URL https://github.com/junhao7622/steinsampling

BugReports https://github.com/junhao7622/steinsampling/issues

License GPL (>= 2)

Encoding UTF-8

Depends R (>= 4.0)

Imports stats, graphics, mvtnorm

Suggests testthat, withr

Collate 'steinsampling-package.R' 'stein_helpers.R' 'kernel_classes.R' 'bootstrap.R' 'gmm_model.R' 'ksd_u_test.R' 'ksd_v_test.R' 'fssd_test.R' 'svgd.R' 'stein_thinning.R' 'stein_points.R' 'stein_point_mcmc.R'

Config/testthat/edition 3

NeedsCompilation no

Config/roxygen2/version 8.0.0

Author Junhao Gao [aut, cre], Ery Arias-Castro [aut]

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

Date/Publication 2026-07-21 10:30:02 UTC

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

RemoteRef HEAD

RemoteSha 1c47f5ccf81a5ca884188a7c40f2bb257c920ce6

Contents

steinsampling-package . . . 3
compute_fssd_null_pvalue . . . 4
compute_fssd_unbiased_stat . . . 5
compute_tau . . . 7
custom_adjusted_gradient . . . 8
custom_stein_kernel . . . 9
find_median_distance . . . 11
fmin_grid . . . 12
fmin_mc . . . 13
fmin_nm . . . 14
fssd_opt_test . . . 16
fssd_rand_test . . . 18
fssd_test . . . 19
get_score_evaluator . . . 22
gmm . . . 23
grw . . . 24
grwmetrop . . . 25
kernel_generics . . . 26
ksd_u_bootstrap . . . 28
ksd_u_statistic . . . 30
ksd_u_test . . . 31
ksd_uq_matrix . . . 33
ksd_v_bootstrap . . . 34
ksd_v_statistic . . . 35
ksd_v_test . . . 36
ksd_vq_matrix . . . 39
likelihoodgmm . . . 40
mala . . . 41
perturbgmm . . . 42
plotgmm . . . 43
posteriorgmm . . . 44
rgmm . . . 45
rwm . . . 46
scorefunctiongmm . . . 47

sp_mcmc . . . 48
sp_mcmc_criterion . . . 51
sp_mcmc_eval_candidates . . . 52
sp_mcmc_select_start . . . 54
sp_mcmc_state . . . 55
stein_codescent . . . 56
stein_kernel . . . 58
stein_kernel_imq_score . . . 59
stein_kernel_inverse_log . . . 61
stein_kernel_matrix . . . 62
stein_points . . . 63
stein_thinning . . . 66
svgd . . . 68
update_svgd . . . 70

Index **73**

`steinsampling-package` *steinsampling: Stein tests and Stein sampling tools*

Description

Score-based goodness-of-fit tests and Stein sampling tools: KSD and FSSD tests, Stein thinning, Stein Points, SP-MCMC, SVGD, and reusable Stein kernels.

Details

Most functions use the target score

$$s_p(x) = \nabla_x \log p(x).$$

Because this is the log-density gradient, many tests and samplers do not require the normalizing constant of `p(x)`.

The public API is layered. Complete methods occupy the top layer: `ksd_u_test()`, `ksd_v_test()`, `fssd_test()`, `stein_thinning()`, `stein_points()`, `sp_mcmc()`, and `svgd()`. The next layer exposes their reusable numerical pieces: `stein_kernel_matrix()`, `ksd_uq_matrix()`, `ksd_u_statistic()`, `ksd_u_bootstrap()`, `ksd_vq_matrix()`, `ksd_v_statistic()`, `ksd_v_bootstrap()`, `compute_tau()`, `compute_fssd_unbiased_stat()`, and `compute_fssd_null_pvalue()`. They allow users to inspect intermediate matrices, reuse bootstrap weights, or reproduce paper calculations stepwise.

For testing, `ksd_u_test()` implements the Liu, Lee, and Jordan independent-sample KSD test; its lower-level path is `ksd_uq_matrix()`, the off-diagonal `ksd_u_statistic()`, and centered-multinomial `ksd_u_bootstrap()`. `ksd_v_test()` keeps the same sample, score, and kernel layer but includes the diagonal and uses Rademacher or Markov signs through `ksd_vq_matrix()`, `ksd_v_statistic()`, and `ksd_v_bootstrap()`. `fssd_test()` instead replaces the full pairwise matrix with `compute_tau()`, finite Stein features evaluated at test locations.

The sampling and compression methods reuse this kernel layer differently. `stein_thinning()` returns row indices that compress an existing sample, usually MCMC output. `stein_points()` constructs points by searching continuous candidates with `fmin_grid()`, `fmin_mc()`, or `fmin_nm()`,

while sp_mcmc() uses short MCMC paths as finite candidate sets. svgd() and update_svgd() instead move every particle by deterministic transport.

stein_kernel() and custom_stein_kernel() create kernel objects; the generics eval_kernel(), grad_x_kernel(), trace_mixed_kernel(), cross_kernel(), and grad_theta_v_kernel() expose the pieces needed to assemble scalar Stein kernels k0 and finite-location FSSD features. Most users need only the top-level algorithms, but custom kernels and diagnostics use the same public interface as built-ins.

Finally, the example-model layer provides gmm() construction, rgmm() simulation, likelihoodgmm(), posteriorgmm(), and scorefunctiongmm() evaluation, plus get_score_evaluator() to create the function(X) score callback expected by Stein routines.

Author(s)

Maintainer: Junhao Gao <jug049@ucsd.edu>

Authors:

- Junhao Gao <jug049@ucsd.edu>
- Ery Arias-Castro <eariascastro@ucsd.edu>

See Also

Useful links:

- https://github.com/junhao7622/steinsampling
- Report bugs at https://github.com/junhao7622/steinsampling/issues

---

compute_fssd_null_pvalue

Simulate the FSSD null distribution

Description

Simulates the asymptotic null distribution used to turn an observed FSSD statistic into a p-value.

Usage

```
compute_fssd_null_pvalue(tau_matrix, fssd_stat, n_simulations = 2000)
```

Arguments

tau_matrix	Numeric FSSD feature matrix.
fssd_stat	Observed FSSD statistic.
n_simulations	Number of null draws to simulate.

Details

Let `lambda_1, ..., lambda_m` be the eigenvalues of the centered empirical feature covariance matrix

$$\hat{\Sigma} = n^{-1} \sum_i (\tau_i - \bar{\tau})(\tau_i - \bar{\tau})^T.$$

The null draws have the form

$$\sum_j \lambda_j (Z_j^2 - 1),$$

with independent standard normal `Z_j`. The observed value compared against those draws is `n * fssd_stat`. This matches the FSSD paper's null result for the scaled statistic `n * FSSDhat^2`, using the empirical plug-in covariance of `tau(X)` in place of the population covariance.

This is the final lower-level step used by `fssd_rand_test()` and `fssd_opt_test()`. It is exported so users can change the number of null simulations, inspect the eigenvalues, or compare the same observed statistic under different Monte Carlo seeds without recomputing `tau`.

Value

A list with three entries:

- `p_value`: fraction of simulated null draws at least as large as `test_score`.
- `test_score`: the observed statistic multiplied by `nrow(tau_matrix)`, matching the paper's asymptotic null scale.
- `eigenvalues`: nonnegative eigenvalues of the empirical covariance matrix used as weights in the simulated null distribution.

The three entries are returned together because they answer different diagnostic questions. `p_value` is the value used by `fssd_rand_test()` and `fssd_opt_test()` to make the hypothesis-test object. `test_score` records the exact scaled number compared with the null draws, which avoids ambiguity about whether the unscaled or `n`-scaled statistic was used. `eigenvalues` expose the estimated null distribution; if most eigenvalues are numerically zero, the finite feature set or sample size may be giving a nearly degenerate null simulation.

Examples

```
tau <- matrix(c(1, 2, 3), ncol = 1)
compute_fssd_null_pvalue(tau, fssd_stat = 0.1, n_simulations = 10)
```

---

compute_fssd_unbiased_stat

Estimate squared FSSD from a feature matrix

Description

Computes the unbiased U-statistic estimate of squared FSSD from the feature matrix returned by `compute_tau()`.

Usage

```
compute_fssd_unbiased_stat(tau_matrix)
```

Arguments

`tau_matrix`	Numeric FSSD feature matrix.

Details

If `tau_i` is row `i` of `tau_matrix`, the returned value is

$$\frac{1}{n(n-1)} \sum_{i \neq j} \tau_i^T \tau_j.$$

This is the sample version of the paper's `FSSD^2` for the selected test locations. The implementation uses column sums to avoid explicitly forming all pairwise inner products.

This function is separated from `compute_tau()` because the same feature matrix is also needed for the null simulation. Keeping the statistic as its own step makes the FSSD workflow explicit: features first, U-statistic second, null simulation third.

The diagonal terms `tau_i^T tau_i` are excluded for the same reason that a U-statistic excludes self-pairs: the target quantity is an expectation over two independent draws from the sample distribution. The implementation uses the identity

$$\sum_{i \neq j} \tau_i^T \tau_j = ||\sum_i \tau_i||^2 - \sum_i ||\tau_i||^2$$

so it can compute the statistic from column sums and row norms rather than forming the full `n x n` matrix of feature inner products.

Value

One numeric value: the unbiased estimate of squared FSSD for the supplied feature matrix. This is the value stored as `statistic` by `fssd_rand_test()` and `fssd_opt_test()` before it is scaled by `n` for the null comparison. It is returned as a plain number because the location matrix `V`, kernel choice, and null-simulation settings are carried by the higher-level test object.

Examples

```
tau <- matrix(c(1, 2, 3), ncol = 1)
compute_fssd_unbiased_stat(tau)
```

compute_tau *Compute the FSSD feature matrix*

Description

Builds the matrix of `tau(x_i)` features used by the FSSD statistic and its null simulation. This is the feature-construction step of the FSSD algorithm.

Usage

```
compute_tau(X, grads, V, kernel_obj)
```

Arguments

`X`	Numeric sample matrix.
`grads`	Score matrix for `X`.
`V`	Matrix of FSSD test locations.
`kernel_obj`	Stein kernel object.

Details

For sample `x` with target score `s_p(x)` and one test location `v`, the paper's feature is

$$\xi_p(x, v) = s_p(x)k(x, v) + \nabla_x k(x, v).$$

The output row stacks these feature vectors for all `J` locations and scales them by `1 / sqrt(d * J)`, where `d` is the sample dimension. The resulting matrix has one row per sample and `d * J` columns. More explicitly, columns `((j - 1) * d + 1):(j * d)` contain $\xi_p(x_i, v_j)/\sqrt{dJ}$ for all rows `i`. Inner products of these rows are the $\Delta(x, y) = \tau(x)^T\tau(y)$ terms used in the FSSD U-statistic.

This function is the FSSD analogue of building a Stein-kernel matrix. It keeps the feature representation instead of immediately taking all pairwise inner products. That design lets `compute_fssd_unbiased_stat()` compute the statistic and lets `compute_fssd_null_pvalue()` estimate the null covariance from the same `tau` matrix.

Moving sideways from `stein_kernel_matrix()` to `compute_tau()` changes the second argument of the kernel calculation from sample rows to fixed test locations. For each row `x_i` and location `v_j`, the function calls the base kernel and its x-gradient through `eval_kernel()` and `grad_x_kernel()`. It does not call `trace_mixed_kernel()` because FSSD does not directly assemble the full pairwise scalar `k0(x_i, x_l)`. Instead it applies one Stein operator at the sample point and stores the finite-location feature vector; `compute_fssd_unbiased_stat()` later forms the paper's one-sample second-order (pairwise) U-statistic by inner products of feature vectors from distinct observations.

Value

Numeric matrix with `nrow(X)` rows and `ncol(X) * nrow(V)` columns. Row `i` is the stacked and scaled feature vector `tau(x_i)`. Columns are ordered by test location first and coordinate second: all `d` coordinates for `V[1, ]`, then all `d` coordinates for `V[2, ]`, and so on. The matrix is intended to be passed to `compute_fssd_unbiased_stat()` and `compute_fssd_null_pvalue()`.

Examples

```
X <- matrix(c(-1, 0, 1), ncol = 1)
grads <- -X
V <- matrix(0, ncol = 1)
kernel <- stein_kernel(type = "gaussian_rbf", h = 1)
compute_tau(X, grads, V, kernel)
```

`custom_adjusted_gradient` *Apply a custom SVGD direction adjustment*

Description

Calls a user-supplied adjustment function for the scaling part of an SVGD update. It receives the raw SVGD direction and the running squared-gradient history, then returns the direction that `update_svgd()` will multiply by the step size.

Usage

```
custom_adjusted_gradient(
  adj_grad,
  grad,
  historical_grad,
  iter,
  theta,
  stepsize,
  alpha,
  fudge_factor
)
```

Arguments

`adj_grad`	User function that returns the adjusted update direction.
`grad`	Current raw SVGD direction.
`historical_grad`	Exponential moving average of `grad^2`, maintained by `update_svgd()`.
`iter`	Current iteration index.
`theta`	Current particle matrix.
`stepsize`	SVGD step size.
`alpha`	Exponential-decay/momentum coefficient used to update `historical_grad`.
`fudge_factor`	Small numerical stabilizer used by the default RMSProp-style rule.

Details

SVGD itself determines the raw direction `grad`. This helper deliberately touches only the rescaling step. Custom adjustment functions are useful when the raw direction should be rescaled in a way other than the package's default RMSProp-style rule. The returned matrix is multiplied by `stepsize` later inside `update_svgd()`.

This wrapper exists so custom adjustment functions receive a stable set of arguments. It keeps user code from depending on local variable names inside `update_svgd()`, while making clear that custom adjustments should return a direction, not a fully stepped particle matrix.

Value

Numeric matrix with the same shape as `grad` and `theta`. The matrix is the adjusted direction; `update_svgd()` multiplies it by `stepsize` and adds it to the particles.

Examples

```
adj <- function(grad, historical_grad, ...) grad / (1 + sqrt(historical_grad))
custom_adjusted_gradient(
  adj, grad = matrix(1), historical_grad = matrix(1),
  iter = 1, theta = matrix(0), stepsize = 0.1,
  alpha = 0.9, fudge_factor = 1e-6
)
```

`custom_stein_kernel` *Create a custom Stein kernel*

Description

Wraps user-supplied functions for a base kernel and its derivatives in a `SteinKernel` object. This lets custom kernels use the same package interface as the built-in kernels.

Usage

```
custom_stein_kernel(
  eval_fn,
  grad_x_fn,
  trace_mixed_fn,
  grad_theta_v_fn = NULL,
  custom_grad_mode = c("error", "analytic", "numeric")
)
```

Arguments

`eval_fn`	Function that evaluates the base kernel.
`grad_x_fn`	Function that evaluates the gradient with respect to `X`.
`trace_mixed_fn`	Function that evaluates the mixed derivative trace.

`grad_theta_v_fn`
: Optional function used to optimize FSSD test locations.

`custom_grad_mode`
: How to handle missing FSSD optimization gradients.

Details

A custom kernel must provide three functions. `eval_fn(X, Y, precon)` returns the base kernel matrix

$$K_{ij} = k(X[i,], Y[j,]).$$

`grad_x_fn(X, Y, precon)` returns an array whose `(i, j, l)` entry is

$$\partial k(X[i,], Y[j,])/\partial X[i,l].$$

`trace_mixed_fn(X, Y, precon)` returns the matrix

$$T_{ij} = \sum_l \partial^2 k(X[i,], Y[j,])/\partial X[i,l]\partial Y[j,l].$$

These are precisely the base-kernel pieces from which `stein_kernel_matrix()` builds the Liu et al. Stein kernel `k0`.

The constructor validates computation, not KSD theorem assumptions. Shape checks do not establish symmetry, positive definiteness, membership in the target's Stein class, the boundary/integration-by-parts identity, `C0`-universality, or required Lipschitz and moment bounds; these remain the caller's responsibility.

Liu et al.'s independent-sample U test requires a positive-definite base kernel in the relevant Stein class, plus the smoothness, integrability, and moment conditions supporting the Stein identity and degenerate bootstrap; its null limit specifically assumes a finite second moment of induced `k0`.

Chwialkowski et al.'s V test uses `C0`-universality so zero discrepancy identifies the target; its wild bootstrap requires `k0` symmetric, positive-definite, null-degenerate, Lipschitz, and satisfying $E[k0(Z,Z)^2] < \infty$. Dependent observations additionally require the stationary tau-mixing and tuning assumptions in `ksd_v_test()`. Correct matrix and array shapes alone therefore do not define a valid KSD test.

The cross term derives from `grad_x_fn(X,Y,precon)` and its swapped call, appropriate for symmetric two-argument kernels whose second-argument derivative follows by swapping and transposing. Nonsymmetric kernels need a full S3 class with `cross_kernel()`.

For an ordinary `(X,Y)` kernel, `stein_kernel_matrix()` directly assembles the four Stein terms. Because the wrapper passes no target scores to the three functions, score-dependent kernels such as `stein_kernel_imq_score()` need a full S3 class with their own `stein_kernel_matrix()` method.

FSSD optimization additionally needs `grad_theta_v_fn`, or `custom_grad_mode = "numeric"` for slower numerical differentiation. The analytic function accepts `X`, `vj`, `grads_X`, `g_block`, and `obj`, and returns test-location derivative `grad_vj` plus optional scalar kernel-parameter derivative `grad_param`.

This functional alternative to a full S3 class stores explicit formulas and lets existing methods call them with built-in matrix shapes. Relative to `stein_kernel()`, only how the base kernel and derivatives are supplied changes: top-level algorithms still receive one `SteinKernel`, with samples and scores interpreted row-wise.

Value

A list with class "SteinKernel_custom" and "SteinKernel". It stores the eval_fn, grad_x_fn, trace_mixed_fn, optional grad_theta_v_fn, and custom_grad_mode. Like stein_kernel() output, it can enter the same public algorithms without exposing whether its implementation is built in or custom.

Examples

```
eval_fn <- function(X, Y = NULL, precon = NULL) {
  X <- as.matrix(X)
  Y <- if (is.null(Y)) X else as.matrix(Y)
  diff <- outer(X[, 1], Y[, 1], "-")
  exp(-0.5 * diff^2)
}
grad_fn <- function(X, Y = NULL, precon = NULL) {
  X <- as.matrix(X)
  Y <- if (is.null(Y)) X else as.matrix(Y)
  diff <- outer(X[, 1], Y[, 1], "-")
  array(-diff * exp(-0.5 * diff^2), dim = c(nrow(X), nrow(Y), 1))
}
trace_fn <- function(X, Y = NULL, precon = NULL) {
  X <- as.matrix(X)
  Y <- if (is.null(Y)) X else as.matrix(Y)
  diff <- outer(X[, 1], Y[, 1], "-")
  (1 - diff^2) * exp(-0.5 * diff^2)
}
custom_stein_kernel(eval_fn, grad_fn, trace_fn)
```

---

find_median_distance *Median squared distance between sample pairs*

Description

Computes the square of the median Euclidean distance between all sample-row pairs. This is the exact median-heuristic squared scale used by the built-in kernels when the user does not supply a scale.

Usage

```
find_median_distance(Z)
```

Arguments

Z	Numeric vector, matrix, or data frame of samples.

Details

If the sample rows are `z_1, ..., z_n` in `R^d`, the full calculation forms the vector of all off-diagonal Euclidean distances

$$D = \{||z_i - z_j||_2 : 1 \leq i < j \leq n\}.$$

If `d_ij = ||z_i - z_j||_2`, the return value is `median({d_ij : i < j})^2`. The median is taken before squaring, matching the bandwidth rule stated in the KSD-U and FSSD experiments. This order matters when the number of pairwise distances is even. For matrix input, rows are observations and columns are coordinates; a numeric vector is treated as an `n x 1` sample.

The value is a data-dependent bandwidth proxy. In this package it is used as a squared scale for built-in kernels when the user supplies `scaling = NULL`. For the Gaussian RBF kernel this means `h^2 = median(d_ij)^2` in `exp(-||x - y||^2 / (2 h^2))`. For the IMQ kernel it means `c^2 = median(d_ij)^2` in `(c^2 + ||x - y||^2)^beta`. The function does not standardize columns, so variables should already be on a comparable scale or the user should provide a kernel scale directly.

All $n(n-1)/2$ pairwise distances are used. The calculation is therefore deterministic but requires quadratic time and memory in the row count. For large samples, provide `scaling` explicitly to the calling KSD or FSSD routine, or provide a fixed bandwidth when constructing a kernel.

Repeated or nearly repeated rows can make the median equal to zero. A zero squared scale would make the built-in kernels ill-defined, so the function returns a small positive floor value and emits a warning in that case. This warning is usually a signal that the sample contains many duplicates or that the chain has not moved enough for a median-distance rule to be informative.

Value

A positive numeric scalar. It is the square of the median finite off-diagonal Euclidean distance. Use `sqrt(find_median_distance(Z))` only if a distance-scale bandwidth is needed. If all finite pairwise distances are zero, the returned value is `1e-5`; if no finite distance can be formed, the returned fallback is `1`.

Examples

```
find_median_distance(c(-1, 0, 2))

Z <- matrix(c(0, 0, 1, 0, 0, 2), ncol = 2, byrow = TRUE)
find_median_distance(Z)
```

---

fmin_grid *Create the grid search used by Stein Points*

Description

Creates the deterministic grid-search function used in the Stein Points appendix. It evaluates every point on a Cartesian grid and returns the smallest objective value.

Usage

```
fmin_grid(lb, ub, n0 = 100, grow = TRUE)
```

Arguments

lb, ub	Lower and upper bounds for candidate points.
n0	Initial grid size per dimension.
grow	Logical; whether to increase grid size as points are selected.

Details

With grow = TRUE, the grid size follows the paper's idea of increasing the grid resolution as the point set grows: the per-dimension size is n0 + round(sqrt(t)), where t is the optimizer iteration supplied by stein_points() or stein_codescent(). The method is simple and deterministic once the grid is fixed, but it is practical mainly in low dimension because the total number of candidates is the product of the grid sizes across dimensions.

This is the deterministic optimizer in the Stein Points family. It is useful for small-dimensional examples and for debugging because the same inputs lead to the same candidate set and the same selected point.

Value

An optimizer function with signature function(objective, X_curr, t). It evaluates the objective on a Cartesian grid and returns x_min, d_min, f_min, and n_eval. x_min is the grid row with the smallest objective, d_min is the target score at that row, f_min is the objective value used by stein_points() to update its KSD accumulator, and n_eval is the number of grid rows scored. Returning the same four fields as fmin_mc() and fmin_nm() makes the deterministic grid search interchangeable with the stochastic and local optimizers.

Examples

```
fmin_grid(lb = -1, ub = 1, n0 = 5, grow = FALSE)
```

---

fmin_mc *Create the Monte Carlo search used by Stein Points*

Description

Creates a candidate-search function for stein_points(). It implements the Monte Carlo search described in the Stein Points appendix: draw a finite set of candidate points in the box and return the candidate with the smallest supplied objective value.

Usage

```
fmin_mc(lb, ub, n_mc = 20, mu0 = NULL, Sigma0 = NULL, sigsq = 1, delay = 20)
```

Arguments

lb, ub	Lower and upper bounds for candidate points.
n_mc	Number of Monte Carlo candidates.
mu0, Sigma0	Initial Gaussian proposal mean and covariance.
sigsq	Local proposal variance after the delay period.
delay	Number of optimization iterations before using local proposals.

Details

The returned optimizer is a function used by stein_points() and stein_codescent(). Early iterations draw candidates from a broad Gaussian distribution truncated to [lb, ub]. Once t exceeds delay, the proposal becomes local: it chooses one of the current points and draws a Gaussian perturbation with variance sigsq, again keeping only candidates in the box. This mirrors the adaptive proposal used in the paper experiments.

The optimizer returned by fmin_mc() does not know anything about Stein kernels. It receives an objective from stein_points(), evaluates that objective on sampled candidate rows, and returns the best candidate plus its score and evaluation count. This design keeps stochastic search separate from the mathematical Stein objective.

Value

An optimizer function with signature function(objective, X_curr, t). It returns a list with x_min (best candidate row), d_min (score at that row), f_min (objective value), and n_eval (number of candidates scored). This is the interface expected by stein_points() and stein_codescent(). The four fields are separated because the main Stein Points loop needs all of them: x_min is appended to the design, d_min is cached so the score is not recomputed, f_min updates the running KSD sum, and n_eval records the search cost for budget comparisons.

Examples

```
fmin_mc(lb = -1, ub = 1, n_mc = 5)
```

---

fmin_nm *Create the multi-start Nelder-Mead search used by Stein Points*

Description

Creates a local search function for stein_points(). It draws several starting points in the box, runs Nelder-Mead from each one, and returns the best local solution found.

Usage

```
fmin_nm(
  lb,
  ub,
  n_res = 3,
  mu0 = NULL,
  Sigma0 = NULL,
  sigsq = 1,
  delay = 20,
  control = list(reltol = 0.001)
)
```

Arguments

lb, ub	Lower and upper bounds for candidate points.
n_res	Number of random restarts.
mu0, Sigma0	Initial Gaussian proposal mean and covariance.
sigsq	Local proposal variance after the delay period.
delay	Number of optimization iterations before using local proposals.
control	Control list passed to stats::optim().

Details

The Stein Points paper uses numerical optimization because the exact global search is usually unavailable. This helper performs a practical multi-start local search. A sine-squared transformation maps unconstrained Nelder-Mead parameters back into [lb, ub], so every objective evaluation stays inside the requested search box. Each restart begins from the same boxed proposal rule used by fmin_mc().

This optimizer is parallel to fmin_grid() and fmin_mc(): all three return the same optimizer interface, but they search differently. Nelder-Mead is useful when the objective is smooth enough for local improvement to beat a finite candidate set.

Value

An optimizer function with signature function(objective, X_curr, t). The returned function runs stats::optim() from n_res starting points and returns x_min, d_min, f_min, and n_eval. The common return contract lets stein_points() switch between optimizers without changing its main loop. x_min is the selected point, d_min is its score, f_min is the final Stein objective value used to update the running KSD diagnostic, and n_eval is the number of objective evaluations charged to this search.

Examples

```
fmin_nm(lb = -1, ub = 1, n_res = 2)
```

`fssd_opt_test` *FSSD test with optimized test locations*

Description

Implements the FSSD-opt variant: use a training split to find test locations that make the FSSD signal large relative to its estimated variability, then run the final goodness-of-fit test on disjoint held-out data.

Usage

```
fssd_opt_test(
  X,
  score_function,
  J = 5,
  nboot = 2000,
  kernel = c("gaussian_rbf", "imq"),
  scaling = NULL,
  train_ratio = 0.2,
  gamma = 1e-04,
  maxit = 100,
  locs_bounds_frac = 10,
  scale_lower = 0.1,
  scale_upper = 10000,
  seed = NULL,
  imq_beta = -0.5
)
```

Arguments

`X`	Numeric vector or matrix of samples (`n x d`).
`score_function`	Function returning the target score for each row of `X`; the output should have shape `n x d`.
`J`	Number of test locations.
`nboot`	Number of simulated null draws used for the p-value.
`kernel`	`"gaussian_rbf"`, `"imq"`, or a `SteinKernel` object.
`scaling`	Optional positive squared kernel scale. `NULL` uses the square of the median distance between all sample-row pairs. This exact calculation has quadratic time and memory cost; large-sample callers should provide `scaling` explicitly.
`train_ratio`	Requested fraction of samples used to tune FSSD-opt. The split is rounded down and adjusted so both training and test sets contain at least two rows.
`gamma`	Small positive regularizer used by the FSSD-opt objective.
`maxit`	Maximum number of L-BFGS-B optimizer iterations for FSSD-opt.

`locs_bounds_frac`
: Width of the box used to bound optimized test locations, measured in fitted standard deviations.

`scale_lower`, `scale_upper`
: Bounds for the optimized squared kernel scale.

`seed`
: Optional RNG seed for the data split, random locations, and simulated null draws.

`imq_beta`
: IMQ exponent.

Details

The FSSD paper chooses locations, and for Gaussian kernels also the bandwidth, by approximately maximizing test power under the alternative. This package uses the same idea with a small stabilizer:

$$\frac{\widehat{FSSD}^2}{\sqrt{\widehat{var}_{H1}} + \gamma},$$

where `gamma` keeps the denominator away from zero. The numerator is the training-sample FSSD estimate and the denominator is an empirical standard deviation proxy under the alternative.

The algorithm first splits `X` into disjoint training and test rows. Starting locations are drawn from a Gaussian fitted to the training rows. When `scaling = NULL`, built-in kernels try a short grid centered on the exact all-training-row median scale before bounded L-BFGS-B refinement. An explicit `scaling` value is used directly as the optimizer's starting scale and skips that median grid. The locations and squared kernel scale are then refined jointly when applicable. The final statistic and null simulation are always evaluated on the held-out test rows. This disjointness is part of the paper's procedure: locations and bandwidth are learned on the training set, then the goodness-of-fit test is performed on the test set to avoid feature-selection overfitting. With `n = nrow(X)`, the actual training size is `max(2, min(floor(train_ratio * n), n - 2))`; the remaining rows form the test set.

The optimized stage and the testing stage use the same feature definition but play different roles. During training, `compute_tau()` is evaluated repeatedly while `V` and the squared scale are changed, and `grad_theta_v_kernel()` supplies derivatives of the finite-location feature with respect to each test location. After optimization, `V` and the kernel scale are fixed; the final evaluation then follows the same lower-level path as `fssd_rand_test()`: `compute_tau()`, `compute_fssd_unbiased_stat()`, and `compute_fssd_null_pvalue()`.

Value

An object of class `htest`. The main fields are `statistic` (one number labeled `fssd`), `p.value`, `method`, `data.name`, and `parameter`. The `info` field is a list with `variant = "opt"`, optimized locations `V`, the kernel name, actual split sizes `n_train` and `n_test`, optimized squared scale `sigma2_opt` when applicable, optimized objective value `objective_opt`, objective-function count `function_evaluations`, and L-BFGS-B `convergence` code. The statistic is computed only on the held-out test rows.

The optimization diagnostics are returned because FSSD-opt is a two-stage method: first choose features, then test. If the optimizer stops early or chooses a boundary scale, those diagnostics matter for interpreting the final p-value.

Examples

```
X <- matrix(rnorm(12), ncol = 1)
score_function <- function(x) -as.matrix(x)
fssd_opt_test(X, score_function, J = 1, nboot = 10, maxit = 1,
              train_ratio = 0.5)
```

`fssd_rand_test` *FSSD test with random test locations*

Description

Implements the FSSD-rand variant: draw the finite set of test locations from a Gaussian fitted to the sample, then compute the FSSD statistic and p-value on all rows.

Usage

```
fssd_rand_test(
  X,
  score_function,
  J = 5,
  nboot = 2000,
  kernel = c("gaussian_rbf", "imq"),
  scaling = NULL,
  seed = NULL,
  imq_beta = -0.5
)
```

Arguments

`X`	Numeric vector or matrix of samples (`n x d`).
`score_function`	Function returning the target score for each row of `X`; the output should have shape `n x d`.
`J`	Number of test locations.
`nboot`	Number of simulated null draws used for the p-value.
`kernel`	`"gaussian_rbf"`, `"imq"`, or a `SteinKernel` object.
`scaling`	Optional positive squared kernel scale. `NULL` uses the square of the median distance between all sample-row pairs. This exact calculation has quadratic time and memory cost; large-sample callers should provide `scaling` explicitly.
`seed`	Optional RNG seed for the data split, random locations, and simulated null draws.
`imq_beta`	IMQ exponent.

Details

Having a density for the location distribution is only one condition in the paper's Theorem 1. Its almost-sure identification result also assumes that the state space is connected and open, the base kernel is C0-universal and real analytic, the required Stein-kernel and score-difference moments are finite, and the tail boundary condition for the Stein witness holds. The paper states that all of these conditions are assumed throughout and then specializes to the Gaussian kernel. This package's IMQ and custom-kernel options are extensions; for them, a location distribution with a density alone does not establish Theorem 1, and the caller is responsible for verifying the remaining kernel, moment, and boundary assumptions.

This implementation draws locations from a practical Gaussian proposal with mean and covariance estimated from X. The resulting location matrix V has J rows and ncol(X) columns, and it is stored in result$info$V for inspection or reuse. This variant does not split the sample and does not optimize the locations.

After the locations are drawn, compute_tau() builds the feature matrix, compute_fssd_unbiased_stat() computes the U-statistic estimate, and compute_fssd_null_pvalue() simulates the null distribution. This gives a compact baseline FSSD test: the randomness is only in the location draw and in the Monte Carlo approximation of the null distribution.

Value

An object of class htest. The main fields are statistic (one number labeled fssd), p.value, method, data.name, and parameter. The info field is a list with variant = "rand", the sampled J x d location matrix V, and the kernel name. The statistic is computed on all rows of X.

The sampled locations are returned because they define the finite set in "Finite Set Stein Discrepancy"; two runs with different V are two different finite-feature tests even if the sample and target score are unchanged.

Examples

```
X <- matrix(rnorm(8), ncol = 1)
score_function <- function(x) -as.matrix(x)
fssd_rand_test(X, score_function, J = 1, nboot = 10)
```

---

fssd_test *Finite Set Stein Discrepancy goodness-of-fit test*

Description

Runs the FSSD test of Jitkrittum et al. The test compares a sample with a target score by evaluating the Stein witness function at a small finite set of test locations. Use variant = "rand" for random locations and variant = "opt" to learn locations, and for built-in kernels the kernel scale, on a training split.

Usage

```
fssd_test(
  X,
  score_function,
  variant = c("opt", "rand"),
  J = 5,
  nboot = 2000,
  kernel = c("gaussian_rbf", "imq"),
  scaling = NULL,
  train_ratio = 0.2,
  gamma = 1e-04,
  maxit = 100,
  locs_bounds_frac = 10,
  scale_lower = 0.1,
  scale_upper = 10000,
  seed = NULL,
  imq_beta = -0.5
)
```

Arguments

`X`	Numeric vector or matrix of samples (`n x d`).
`score_function`	Function returning the target score for each row of `X`; the output should have shape `n x d`.
`variant`	`"rand"` draws test locations from a Gaussian fitted to the data. `"opt"` tunes test locations and the kernel scale on a training split.
`J`	Number of test locations.
`nboot`	Number of simulated null draws used for the p-value.
`kernel`	`"gaussian_rbf"`, `"imq"`, or a `SteinKernel` object.
`scaling`	Optional positive squared kernel scale. `NULL` uses the square of the median distance between all sample-row pairs. This exact calculation has quadratic time and memory cost; large-sample callers should provide `scaling` explicitly.
`train_ratio`	Requested fraction of samples used to tune FSSD-opt. The split is rounded down and adjusted so both training and test sets contain at least two rows.
`gamma`	Small positive regularizer used by the FSSD-opt objective.
`maxit`	Maximum number of L-BFGS-B optimizer iterations for FSSD-opt.
`locs_bounds_frac`	Width of the box used to bound optimized test locations, measured in fitted standard deviations.
`scale_lower, scale_upper`	Bounds for the optimized squared kernel scale.
`seed`	Optional RNG seed for the data split, random locations, and simulated null draws.
`imq_beta`	IMQ exponent.

Details

FSSD chooses J locations $v_1, \ldots, v_J$ and checks whether the Stein witness function vanishes there. Write the target density as p with score $s_p(x) = \nabla_x \log p(x)$. For each sample point x_i and location v_j, the Stein feature is

$$\xi_p(x_i, v_j) = s_p(x_i)k(x_i, v_j) + \nabla_x k(x_i, v_j).$$

Here k is the base kernel, and the gradient is with respect to sample argument x_i, not location argument v_j; each location contributes a length-d vector.

Let $\Xi(x)$ be the d x J matrix with column $\xi_p(x, v_j)/\sqrt{dJ}$ and let $\tau(x) = vec(\Xi(x))$ stack its columns into length d * J, the row representation returned by compute_tau(). Then

$$\Delta(x, y) = \tau(x)^T \tau(y) = \frac{1}{dJ} \sum_{j=1}^{J} \xi_p(x, v_j)^T \xi_p(y, v_j)$$

is the FSSD U-statistic kernel; feature normalization by $\sqrt{dJ}$ produces the paper's $1/(dJ)$ inner-product factor.

Averaging over distinct sample pairs gives

$$\widehat{FSSD}^2 = \frac{1}{n(n-1)} \sum_{i \neq l} \tau(x_i)^T \tau(x_l).$$

compute_tau() returns the n x (d * J) matrix with row i equal to tau(x_i); the unbiased statistic is computed without storing every pairwise inner product.

The p-value uses the paper's asymptotic null distribution. Estimate $\hat{\Sigma} = n^{-1} \sum_i (\tau(x_i) - \bar{\tau})(\tau(x_i) - \bar{\tau})^T$, compute its eigenvalues, and simulate null draws $\sum_j \lambda_j (Z_j^2 - 1)$. Large values indicate disagreement between sample and target score. The "opt" variant first uses part of the data to choose useful locations.

Unlike ksd_u_test() and ksd_v_test(), which build all pairwise k0(x_i,x_j) values through stein_kernel_matrix(), FSSD uses a finite location matrix V and compute_tau(). It retains the same X, target-score convention, and kernel layer, while adding J and, for "opt", a training stage that selects V and the built-in kernel scale.

Value

An object of class htest. For both variants the list contains:

- statistic: one number labeled fssd; this is the unbiased estimate of squared FSSD for the final test locations.
- p.value: right-tail p-value from simulated asymptotic null draws.
- method: text naming the FSSD variant and kernel.
- data.name: expression used for X.
- parameter: numeric vector with nboot, J, and the final squared kernel scaling; the optimized variant also records train_ratio and gamma.
- info: list with the variant label, final location matrix V, kernel name, and for the optimized variant, optimization diagnostics.

The htest fields provide standard hypothesis-test printing; info retains V because a location-free p-value is difficult to reproduce or interpret.

Examples

```
X <- matrix(rnorm(8), ncol = 1)
score_function <- function(x) -as.matrix(x)
fssd_test(X, score_function, variant = "rand", J = 1, nboot = 10)
```

`get_score_evaluator` *Create a score function for a fixed Gaussian mixture model*

Description

Returns a function of `X` only, ready to pass to routines that need a score function.

Usage

```
get_score_evaluator(model)
```

Arguments

`model`	Gaussian mixture model returned by `gmm()`.

Details

Many Stein routines ask for a score function with signature `function(X)`. This helper stores the model and its inverse covariance matrices once, then returns a function that evaluates the Gaussian mixture score for new samples.

This is the preferred bridge from the Gaussian mixture helpers to the main Stein algorithms. Instead of repeatedly passing both `model` and `X` to `scorefunctiongmm()`, create `score <- get_score_evaluator(model)` once and then pass `score` to `ksd_u_test()`, `fssd_test()`, `stein_points()`, `sp_mcmc()`, or `stein_thinning()`.

The closure caches the inverse covariance matrices because they depend only on the mixture model, not on the evaluation points. That design keeps the returned score function cheap enough to call repeatedly inside iterative algorithms such as SVGD, Stein Points, and SP-MCMC.

Value

A function with signature `function(X)`. For matrix input it returns an `n x d` score matrix for new samples, using cached inverse covariance matrices from `model`. For vector input it returns a numeric vector, matching the package's convention that a vector is a batch of one-dimensional samples when `model$d == 1` and one multivariate sample when `model$d > 1`. The main Stein routines call score functions with matrix inputs, so they receive the matrix form needed to align rows of samples and scores.

Examples

```
model <- gmm()
grad_log_prob <- get_score_evaluator(model)
X <- rgmm(model)
G <- grad_log_prob(X)
```

gmm *Create a Gaussian mixture model*

Description

Builds a Gaussian mixture model object for simulations and examples.

Usage

```
gmm(nComp = NULL, mu = NULL, sigma = NULL, weights = NULL, d = NULL)
```

Arguments

nComp	Number of mixture components. If NULL, five components are generated.
mu	Component means, stored as a d x nComp matrix. A vector is also accepted for a one-dimensional mixture or a single component.
sigma	Component covariance matrices, stored as a d x d x nComp array. Each matrix must be symmetric positive definite. A scalar or vector is treated as one-dimensional component variances; a d x d matrix is reused for every component.
weights	Optional nonnegative mixture weights, with at least one positive value. Values are normalized to sum to one.
d	Dimension of each sample. If omitted, it is inferred from mu or sigma, and otherwise defaults to one.

Details

A Gaussian mixture draws each observation in two steps. First it chooses a component k with probability weights[k]. Then it draws the observation from the Gaussian distribution with mean mu[, k] and covariance sigma[, , k]. Its density is

$$p(x) = \sum_{k=1}^{K} w_k \phi(x; \mu_k, \Sigma_k),$$

where w_k is the component weight and phi is the Gaussian density.

If nComp is NULL, the function creates a five-component mixture in dimension d, with d = 1 by default. If means or covariances are omitted, random means and identity covariance matrices are filled in.

The returned object is deliberately a simple list because the package uses it as an example target distribution rather than as a fitted statistical model. The fields are exactly the quantities needed by rgmm(), likelihoodgmm(), posteriorgmm(), scorefunctiongmm(), and get_score_evaluator(): weights for component probabilities, means and covariances for Gaussian densities, and dimension d for input checking.

Value

A list with components: nComp, the number of mixture components; mu, a d x nComp matrix whose column k is component mean mu_k; sigma, a d x d x nComp array whose slice sigma[, , k] is covariance matrix Sigma_k; weights, normalized component probabilities; and d, the dimension of each observation. The list is the common input object for the other Gaussian mixture helpers.

Examples

```
# Default one-dimensional Gaussian mixture model
model <- gmm()

# One-dimensional Gaussian mixture model with three components
model <- gmm(nComp = 3)

# Three-dimensional mixture with specified means, covariances, and weights
mu <- matrix(c(1, 2, 3, 2, 3, 4, 5, 6, 7), ncol = 3)
sigma <- array(diag(3), c(3, 3, 3))
model <- gmm(nComp = 3, mu = mu, sigma = sigma, weights = c(0.2, 0.4, 0.4), d = 3)
```

---

grw *Run the Gaussian random-walk transition used by SP-MCMC*

Description

This is a convenience wrapper around grwmetrop() using proposal covariance h * Sigma, with h = 1 when omitted.

Usage

```
grw(log_p, score_function, x0, h = NULL, Sigma = NULL, m_iter)
```

Arguments

log_p	Function returning log density values.
score_function	Unused score function argument kept for transition interface compatibility.
x0	Initial state vector.
h	Optional positive step-size multiplier.
Sigma	Symmetric positive-definite proposal covariance or preconditioning matrix.
m_iter	Number of returned chain rows, including the initial state in row 1. The function therefore makes m_iter - 1 Markov transitions.

Details

The SP-MCMC main loop passes the log density, score function, current state, step size, proposal matrix, and chain length to every transition. The argument `score_function` is therefore present only for interface compatibility with `mala()` and custom transition functions. The random-walk proposal itself is $N(x, hSigma)$ and does not use the score.

Use this wrapper, rather than `grwmetrop()`, when supplying a custom transition-like function to `sp_mcmc()` or when comparing GRW and MALA under the same call signature.

Value

Output from `grwmetrop()`, with chain states, log densities, acceptance indicators, and evaluation counts. The score matrix is absent by design; SP-MCMC computes candidate scores afterward when needed.

Examples

```
score <- function(X) -as.matrix(X)
log_p <- function(X) -0.5 * rowSums(as.matrix(X)^2)
grw(log_p, score, x0 = 0, h = 0.1, m_iter = 3)
```

`grwmetrop` *Run a Gaussian random-walk Metropolis chain*

Description

Runs the random-walk Metropolis kernel used as the other built-in SP-MCMC candidate-chain transition.

Usage

```
grwmetrop(log_p, x0, S, m_iter)
```

Arguments

`log_p`	Function returning log density values.
`x0`	Initial state vector.
`S`	Symmetric positive-definite Gaussian proposal covariance matrix.
`m_iter`	Number of returned chain rows, including the initial state in row 1. The function therefore makes `m_iter - 1` Markov transitions.

Details

From the current state `x`, the proposal is

$$y = x + z, \quad z \sim N(0, S).$$

The move is accepted with probability `min(1, exp(log_p(y) - log_p(x)))`. Unlike MALA, this transition does not use the score to propose moves, so any candidate scores needed by SP-MCMC are computed later by `sp_mcmc_eval_candidates()`.

This is the lower-level random-walk Metropolis kernel. `grw()` wraps it to match the transition signature used by `sp_mcmc()`, where every transition receives both `log_p` and `score_function`.

The first row of the returned chain is `x0`. Subsequent rows are accepted proposals or repeats of the previous state after rejection. The acceptance indicators therefore describe whether each transition moved, not whether a row is unique. `sp_mcmc()` removes duplicate candidate states later before scoring the path with the Stein Points objective.

Value

A list with `X` (chain states), `log_p` (log density values), `accept` (0/1 acceptance indicators), and evaluation-count fields. It does not return scores because the random-walk proposal does not use them. The evaluation counts are returned so `sp_mcmc()` can report how much of the total budget was spent proposing the candidate path before Stein scoring begins.

Examples

```
log_p <- function(X) -0.5 * rowSums(as.matrix(X)^2)
grwmetrop(log_p, x0 = 0, S = matrix(0.1), m_iter = 3)
```

---

kernel_generics *Building blocks for Stein kernels*

Description

These generic functions define the base kernel, its needed derivatives, and the score-coupling terms. New kernel classes can implement these methods and then work with the rest of the package.

Usage

```
eval_kernel(obj, X, Y = NULL, ...)

grad_x_kernel(obj, X, Y = NULL, ...)

trace_mixed_kernel(obj, X, Y = NULL, ...)

cross_kernel(obj, X, grads, Y = NULL, grads_Y = NULL, ...)

grad_theta_v_kernel(obj, X, vj, grads_X, g_block, ...)
```

Arguments

obj	A Stein kernel object.
X	Numeric matrix with samples in rows.
Y	Optional second sample matrix.
...	Additional arguments passed to methods.
grads	Score matrix for X.
grads_Y	Optional score matrix for Y.
vj	One FSSD test location.
grads_X	Score matrix for X.
g_block	Matrix used by the FSSD optimization gradient.

Details

These functions are the pieces used by the full Stein kernel calculation in stein_kernel_matrix(). For rows x_i = X[i,] and y_j = Y[j,], eval_kernel() gives the base kernel matrix

$$K_{ij} = k(x_i, y_j).$$

grad_x_kernel() gives first derivatives with respect to the first argument,

$$G_{ij\ell} = \partial k(x_i, y_j)/\partial x_\ell.$$

trace_mixed_kernel() gives the matched second-derivative sum

$$T_{ij} = \sum_{\ell=1}^{d} \partial^2 k(x_i, y_j)/\partial x_\ell \partial y_\ell.$$

cross_kernel() gives the two score-derivative terms

$$C_{ij} = s_p(x_i)^T \nabla_y k(x_i, y_j) + s_p(y_j)^T \nabla_x k(x_i, y_j).$$

Then stein_kernel_matrix() returns

$$(s_p(x_i)^T s_p(y_j))K_{ij} + C_{ij} + T_{ij}.$$

grad_theta_v_kernel() is used by FSSD optimization. It gives the derivative of the FSSD feature at one test location with respect to that location and, for built-in kernels, with respect to the squared kernel scale.

These generics are not parallel top-level algorithms; they are a method contract. A user who moves sideways from a built-in kernel to a custom S3 kernel must supply methods with the same shapes and the same row ordering. A user who creates a kernel with custom_stein_kernel() supplies the same information as functions instead of S3 methods. Either path lets the top-level algorithms call stein_kernel_matrix() and compute_tau() without changing their own code.

These generics are public because they are the contract for kernel extension. A user who only runs ksd_u_test() or stein_points() normally calls stein_kernel() or custom_stein_kernel() instead. A user implementing a new S3 kernel class can implement these methods and then reuse the package algorithms without changing those algorithms.

The returned shapes are part of the contract: eval_kernel() and trace_mixed_kernel() return matrices, grad_x_kernel() returns an nrow(X) x nrow(Y) x d array, cross_kernel() returns the matrix of the two score-derivative terms, and grad_theta_v_kernel() returns the derivatives needed by FSSD optimization.

Value

The return value depends on the generic:

- `eval_kernel()` returns the base-kernel matrix with entry `k(X[i, ], Y[j, ])`.
- `grad_x_kernel()` returns an array whose `(i, j, l)` entry is the derivative of `k(X[i, ], Y[j, ])` with respect to coordinate `l` of `X[i, ]`.
- `trace_mixed_kernel()` returns the matrix of mixed-derivative traces `sum_l d^2 k(x, y) / dx_l dy_l`.
- `cross_kernel()` returns the matrix of the two score-derivative terms that enter the Stein kernel.
- `grad_theta_v_kernel()` returns the derivatives used when FSSD-opt changes a test location, and for built-in kernels, the squared kernel scale.

These shapes are fixed because `stein_kernel_matrix()`, `compute_tau()`, and FSSD optimization assemble their algebra from the same pieces. Returning matrices and arrays rather than collapsed summaries lets higher-level algorithms combine the pieces in different ways without asking each kernel implementation to know about every algorithm.

Examples

```
X <- matrix(c(-1, 0, 1), ncol = 1)
grads <- -X
kernel <- stein_kernel(type = "gaussian_rbf", h = 1)
eval_kernel(kernel, X)
grad_x_kernel(kernel, X)
trace_mixed_kernel(kernel, X)
cross_kernel(kernel, X, grads)
grad_theta_v_kernel(kernel, X, vj = 0, grads_X = grads, g_block = grads)
```

---

ksd_u_bootstrap *Bootstrap the KSD U-statistic from a Stein-kernel matrix*

Description

Generates or accepts centered multinomial weights and applies them to the U-statistic matrix. This is the bootstrap part of the Liu et al. KSD test separated from matrix construction.

Usage

```
ksd_u_bootstrap(
  U_mat,
  nboot = 1000,
  W_mat = NULL,
  boot_method = "multinomial_centered"
)
```

Arguments

U_mat	Numeric square matrix of pairwise Stein kernel values.
nboot	Number of bootstrap samples.
W_mat	Optional n x B centered multinomial bootstrap weight matrix, with one row per observation and one column per bootstrap draw.
boot_method	Bootstrap method. Currently only "multinomial_centered" is supported.

Details

For each bootstrap weight vector w, the bootstrap statistic is computed as

$$\sum_{i,j} w_i U_{ij} w_j,$$

with the diagonal of U_mat set to zero. These values are on the same scale as ksd_u_statistic(). The default weights are centered multinomial proportions, N_i / n - 1 / n, matching Eq. 16 of Liu, Lee, and Jordan (2016). The paper compares n times the observed and bootstrap statistics; this package leaves both on the unmultiplied U-statistic scale, which gives the same p-value. Supplying W_mat lets users reuse exactly the same bootstrap weights across kernels or experiments.

This function sits below ksd_u_test() and above the raw weight generator. It exists so users can separate randomness in the bootstrap weights from the deterministic matrix calculation. Supplying W_mat is the design point: the same matrix of weights can be reused across several U_mat objects, which is useful for paired comparisons of kernels or bandwidths.

The parallel bootstrap helper is ksd_v_bootstrap(). The input matrix plays the same role, but the weights have different meanings: ksd_u_bootstrap() uses centered multinomial proportions for the degenerate U-statistic in Liu et al., whereas ksd_v_bootstrap() uses wild-bootstrap sign sequences for the V-statistic. To move from this function to the V version, keep a Stein-kernel matrix but supply sign weights or choose a sign-generation method instead of centered multinomial weights.

Value

Numeric vector of length nboot when W_mat is not supplied, or length ncol(W_mat) when weights are supplied. Each entry is one bootstrap U-statistic on the same scale as ksd_u_statistic(), so it can be compared directly with the observed statistic.

Examples

```
U <- matrix(c(1, 0.2, 0.3, 0.2, 1, 0.4, 0.3, 0.4, 1), 3, 3)
ksd_u_bootstrap(U, nboot = 5)
```

`ksd_u_statistic` *Compute the KSD U-statistic from its Stein-kernel matrix*

Description

Takes the pairwise Stein-kernel matrix and returns the off-diagonal average used as the observed KSD statistic in the Liu et al. test.

Usage

```
ksd_u_statistic(U_mat)
```

Arguments

`U_mat`	Numeric square matrix of pairwise Stein kernel values.

Details

If `U_mat[i, j] = k0(x_i, x_j)`, the diagonal is ignored and the returned value is

$$\frac{1}{n(n-1)} \sum_{i \neq j} U_{ij}.$$

This helper is useful when the matrix has already been built once and the user wants to inspect or reuse the observed statistic separately from the bootstrap. It is the matrix-compression layer between `ksd_uq_matrix()`, which knows about samples, scores, and kernels, and `ksd_u_bootstrap()`, which knows only about a fixed Stein-kernel matrix and bootstrap weights.

In the U-statistic form the diagonal is deliberately removed. Those diagonal entries are self-interactions `k0(x_i, x_i)`; they are present in the matrix returned by `ksd_uq_matrix()` because they are useful diagnostics, but they are not part of Liu et al.'s unbiased KSD estimator. This helper makes that convention explicit, so a user who has built a matrix by hand can apply the same statistic used by `ksd_u_test()`.

The function is deliberately small: it only validates that the input is a square matrix, removes the diagonal contribution, and returns the average. Keeping this step separate from `ksd_uq_matrix()` and `ksd_u_bootstrap()` makes it possible to compare kernels or bootstrap choices while holding the observed matrix fixed.

Value

One numeric value: the off-diagonal average of `U_mat`. This is the same number stored as `statistic` by `ksd_u_test()`, before any bootstrap p-value is computed. It is returned as a plain number because all reproducibility information, such as the kernel scale and bootstrap settings, belongs to the higher-level `ksd_u_test()` object rather than to this matrix-compression step.

Examples

```
U <- matrix(c(1, 0.2, 0.3, 0.2, 1, 0.4, 0.3, 0.4, 1), 3, 3)
ksd_u_statistic(U)
```

ksd_u_test *KSD goodness-of-fit test using the Liu et al. U-statistic*

Description

Runs the Kernel Stein Discrepancy goodness-of-fit test in the U-statistic form of Liu, Lee, and Jordan (2016). The input sample is compared with a target distribution through the target score, so the normalizing constant of the target density is not needed.

Usage

```
ksd_u_test(
  X,
  score_function,
  boot_method = "multinomial_centered",
  scaling = NULL,
  nboot = 1000,
  kernel = c("gaussian_rbf", "imq"),
  return_raw_boot = FALSE,
  block_size = NULL,
  block_threshold = 5000,
  imq_beta = -0.5
)
```

Arguments

X	Numeric vector or matrix of samples. A vector is treated as n x 1.
score_function	Function returning the target score for each row of X.
boot_method	Bootstrap method. Currently only "multinomial_centered" is supported.
scaling	Positive kernel scale. NULL uses the square of the exact all-pair median distance.
nboot	Number of bootstrap samples.
kernel	Kernel choice: "gaussian_rbf", "imq", or a SteinKernel object, including objects created by custom_stein_kernel().
return_raw_boot	Logical; if TRUE, return bootstrap samples.
block_size, block_threshold	Options for computing large kernel matrices in blocks.
imq_beta	IMQ kernel exponent.

Details

Let p be the target density, $s_p(x) = \nabla_x \log p(x)$ its score, and k0(x,y) the Stein kernel built from that score and the selected base kernel; the normalizing constant of p is unnecessary. Liu et al. call this kernel u_q because their target is q; the package consistently uses k0. See stein_kernel_matrix() for its four terms.

X is interpreted row-wise; vectors become n x 1 matrices, and score_function is evaluated once and must return an n x d matrix. With scaling = NULL, find_median_distance() uses all rows to return the squared median pairwise distance: an RBF squared bandwidth or IMQ squared length scale. This exact calculation has quadratic time and memory cost, so large samples should supply scaling. For a preconditioned RBF object without a fixed bandwidth, the rule uses preconditioned distances and retains that preconditioner.

For observations $x_1, \ldots, x_n$, the paper's estimator is

$$U = \frac{1}{n(n-1)} \sum_{i \neq j} k0(x_i, x_j).$$

It omits k0(x_i,x_i) and is degenerate under the target, so its null is not a plain normal approximation.

Following Liu et al., the test (1) builds the n x n Stein-kernel matrix, (2) zeros its diagonal for the U-statistic, (3) generates centered multinomial weights w_i=N_i/n-1/n, and (4) compares U with w^T K0 w. Observed and bootstrap values remain on the unmultiplied U-statistic scale; multiplying both by n leaves the p-value unchanged. Large positive values indicate disagreement with the target score in kernel Stein discrepancy.

The statistic and bootstrap reproduce Liu et al.; the scalar p-value follows the package convention

$$p = \frac{1 + \#\{b : T_b \geq T_{obs}\}}{B + 1}.$$

Algorithm 1 instead rejects using the uncorrected strict proportion $B^{-1}\#\{b : T_b > T_{obs}\}$. The +1 correction prevents zero Monte Carlo p-values but is neither a verbatim algorithm step nor a new finite-sample KSD-bootstrap guarantee.

This is Liu et al.'s independent-sample test; ksd_v_test() supplies the Chwialkowski et al. V-statistic with independent or Markov wild-bootstrap signs for ordered or dependent samples. Its lower-level pieces are ksd_uq_matrix() for the Stein-kernel matrix, ksd_u_statistic() for the off-diagonal average, and ksd_u_bootstrap() for centered multinomial weights. The V-statistic counterparts are ksd_vq_matrix(), ksd_v_statistic(), and ksd_v_bootstrap().

Value

An object of class htest, implemented as a list with:

- statistic: one number labeled ksd_u; this is the observed off-diagonal U-statistic.
- p.value: right-tail bootstrap p-value using the package's (1 + exceedances) / (nboot + 1) finite-bootstrap convention.
- method: text naming the KSD U-statistic test and kernel.
- data.name: expression used for X.
- parameter: numeric vector containing nboot, the squared kernel scaling used, and imq_beta when the IMQ kernel is used.
- bootstrap_samples: present only when return_raw_boot = TRUE; numeric vector of bootstrap U-statistics on the same scale as statistic.

The htest shape prints like other R tests; parameter and optional bootstrap_samples retain settings needed to reproduce or diagnose the p-value.

Examples

```
X <- matrix(rnorm(10), ncol = 1)
score_function <- function(x) -as.matrix(x)
ksd_u_test(X, score_function, nboot = 10)
```

---

ksd_uq_matrix *Build the Stein-kernel matrix for the KSD U-statistic*

Description

Computes the pairwise Stein kernel values that are later averaged by the U-statistic. This is the matrix-level part of the Liu et al. KSD test.

Usage

```
ksd_uq_matrix(
  X,
  score_function,
  scaling = NULL,
  kernel = c("gaussian_rbf", "imq"),
  imq_beta = -0.5
)
```

Arguments

X	Numeric vector or matrix of samples. A vector is treated as n x 1.
score_function	Function returning the target score for each row of X.
scaling	Positive kernel scale. NULL uses the square of the exact all-pair median distance.
kernel	Kernel choice: "gaussian_rbf", "imq", or a SteinKernel object, including objects created by custom_stein_kernel().
imq_beta	IMQ kernel exponent.

Details

Entry (i, j) of the returned matrix is k0(x_i, x_j). The statistic itself is not computed here. Use ksd_u_statistic() to take the paper's off-diagonal average, ksd_u_bootstrap() to make bootstrap draws from this fixed matrix, or ksd_u_test() to run the full test in one call. See stein_kernel_matrix() for the formula used to assemble k0.

This function is the first lower-level step under ksd_u_test(). It is exported because the Stein-kernel matrix is often the object one wants to inspect when diagnosing a KSD test: large diagonal terms, nearly zero off-diagonal terms, or a poor kernel scale are visible here before the statistic and bootstrap compress the matrix to one number.

The parallel V-statistic builder is ksd_vq_matrix(). Both builders evaluate the same Stein-kernel formula through stein_kernel_matrix() and require the same X, score_function, and kernel settings. The difference appears in the next step: ksd_u_statistic() discards the diagonal, while

ksd_v_statistic() keeps it and reports the scaled n V_n statistic. The q in ksd_uq_matrix() is a historical name inherited from the KSD literature's u_q notation; the function still uses the target score supplied by score_function, following the package-wide target-p convention in this manual. In other words, the returned object is the package's K0 matrix with entries k0(x_i, x_j), not an estimate involving a second distribution named q.

Value

Numeric n x n matrix. Row i, column j is k0(x_i, x_j) for the checked sample matrix. The diagonal is included in the matrix because it is useful for diagnostics, but ksd_u_statistic() ignores it when computing the U-statistic. The matrix can be passed directly to ksd_u_statistic() and ksd_u_bootstrap().

Examples

```
X <- matrix(rnorm(5), ncol = 1)
score_function <- function(x) -as.matrix(x)
ksd_uq_matrix(X, score_function)
```

---

ksd_v_bootstrap *Wild-bootstrap the KSD V-statistic*

Description

Applies independent or Markov sign weights to a fixed KSD V-statistic matrix. This is the bootstrap step of the Chwialkowski et al. test separated from the construction of the Stein-kernel matrix.

Usage

```
ksd_v_bootstrap(
  U_mat,
  W_mat = NULL,
  nboot = 1000,
  boot_method = c("rademacher", "markov"),
  change_prob = NULL
)
```

Arguments

U_mat	Numeric square matrix of pairwise Stein kernel values.
W_mat	Optional n x B wild-bootstrap sign matrix, with entries usually equal to -1 or 1.
nboot	Number of bootstrap samples.
boot_method	One of "rademacher" or "markov" when W_mat is NULL.
change_prob	Markov sign-flip probability when boot_method = "markov"; it must be strictly between 0 and 1 and supplied explicitly.

Details

For each bootstrap weight vector `w`, the bootstrap statistic is

$$\frac{1}{n}\sum_{i,j} w_i U_{ij} w_j.$$

This is on the same scaled `n V_n` scale as `ksd_v_statistic()`. Equivalently, Chwialkowski et al. define a bootstrap V-statistic with a `1 / n^2` factor and compare `n` times that value; the formula above is that scaled value. The weights are either independent signs or Markov signs, depending on `boot_method`. For the dependent-sample theory, the Markov flip probability is a sample-size-dependent sequence $p_n \to 0$ satisfying $np_n \to \infty$, together with the mixing, Lipschitz, and moment assumptions documented in `ksd_v_test()`. A fixed value of `0.5` gives independent signs and does not reproduce that dependent wild bootstrap. Supplying `W_mat` is useful when the same bootstrap sign sequences should be reused across kernels or repeated experiments.

The function is separated from `ksd_vq_matrix()` for the same reason as in the U-statistic API: matrix construction is deterministic once `X`, the score, and the kernel are fixed, while the bootstrap is random unless the user supplies `W_mat`. This split lets simulation studies control those two sources separately.

The parallel U-statistic helper is `ksd_u_bootstrap()`. Use this V helper when `U_mat` will be summarized by `ksd_v_statistic()` and when bootstrap weights should be sign sequences. Use the U helper when `U_mat` will be summarized by `ksd_u_statistic()` and the Liu et al. centered multinomial bootstrap is desired.

Value

Numeric vector of bootstrap statistics on the scaled `n V_n` scale. Its length is `nboot` when signs are generated internally, or `ncol(W_mat)` when a sign matrix is supplied.

Examples

```
U <- matrix(c(1, 0.2, 0.3, 0.2, 1, 0.4, 0.3, 0.4, 1), 3, 3)
ksd_v_bootstrap(U, nboot = 5)
```

`ksd_v_statistic` *Compute the KSD V-statistic from its Stein-kernel matrix*

Description

The returned statistic is `n * mean(U_mat)`, which is the same as summing all entries of the matrix and dividing by `n`.

Usage

```
ksd_v_statistic(U_mat)
```

Arguments

`U_mat`	Numeric square matrix of pairwise Stein kernel values.

Details

If `U_mat[i, j] = k0(x_i, x_j)`, the statistic is

$$\frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{n}U_{ij}.$$

This is the scaled statistic `n V_n` when $V_n = n^{-2}\sum_{i,j}U_{ij}$. The helper is separated out so users can reuse a precomputed Stein-kernel matrix or compare dense and blocked matrix calculations. It is the matrix-compression layer between `ksd_vq_matrix()`, which evaluates scores and kernels, and `ksd_v_bootstrap()`, which keeps the matrix fixed and changes only the wild-bootstrap sign sequences.

The diagonal is included here because the V-statistic treats every ordered pair of rows, including `(i, i)`, as part of the empirical average. This is the main algebraic difference from `ksd_u_statistic()`. Keeping the two compression functions separate prevents accidental mixing of U-statistic and V-statistic conventions when users inspect lower-level matrices.

This function performs no bootstrap and does not evaluate the score function. It is the deterministic compression from the full pairwise matrix to the one-number V-statistic used by `ksd_v_test()`.

Because the function only needs `U_mat`, it is useful for checking whether two ways of building the same Stein-kernel matrix, for example dense and blocked computation, lead to the same final statistic before any bootstrap randomness is introduced.

Value

One numeric value: the scaled V-statistic `n V_n`. This is the same scale as the values returned by `ksd_v_bootstrap()` and as the `statistic` component of `ksd_v_test()`. It is returned as a plain number because the kernel choice and wild-bootstrap settings are handled by the higher-level test object.

Examples

```
U <- matrix(c(1, 0.2, 0.3, 0.2, 1, 0.4, 0.3, 0.4, 1), 3, 3)
ksd_v_statistic(U)
```

`ksd_v_test` *KSD goodness-of-fit test using the Chwialkowski et al. V-statistic*

Description

Runs the Kernel Stein Discrepancy goodness-of-fit test in the V-statistic form used by Chwialkowski et al. (2016). It compares samples with a target distribution through the target score and uses a wild bootstrap to approximate the null distribution.

Usage

```
ksd_v_test(
  X,
  score_function,
  boot_method = c("rademacher", "markov"),
  scaling = NULL,
  nboot = 1000,
  change_prob = NULL,
  kernel = c("gaussian_rbf", "imq"),
  return_raw_boot = FALSE,
  block_size = NULL,
  block_threshold = 5000,
  imq_beta = -0.5
)
```

Arguments

X	Numeric vector or matrix of samples.
score_function	Function returning the target score for each row of X.
boot_method	One of "rademacher" or "markov".
scaling	Positive kernel scale. NULL uses the square of the exact all-pair median distance.
nboot	Number of bootstrap samples.
change_prob	Markov sign-flip probability used when boot_method = "markov". It must be strictly between 0 and 1, supplied explicitly, and tuned with sample size; see Details. Use boot_method = "rademacher" for independent signs.
kernel	Kernel choice: "gaussian_rbf", "imq", or a SteinKernel object, including objects created by custom_stein_kernel().
return_raw_boot	Logical; if TRUE, append raw bootstrap samples to the output.
block_size, block_threshold	Options for computing large kernel matrices in blocks.
imq_beta	IMQ kernel exponent.

Details

Let p be the target density, $s_p(x) = \nabla_x \log p(x)$ its score, and k0(x,y) the Stein kernel built from that score and the selected base kernel; the normalizing constant of p is unnecessary. See stein_kernel_matrix() for the four terms in k0.

Rows of X are samples, and score_function(X) must return one score vector per row. With scaling = NULL, find_median_distance() uses all pairs to return a squared median distance: h^2 for RBF or c^2 for IMQ. This package default is not prescribed by Chwialkowski et al. and costs quadratic time and memory, so large samples should provide scaling. For a preconditioned RBF object without fixed bandwidth, the rule uses preconditioned distances and retains that preconditioner.

For observations $x_1, \ldots, x_n$, this test uses

$$nV_n = \frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{n} k0(x_i, x_j).$$

Some papers instead write $V_n = n^{-2}\sum_{i,j} k0(x_i, x_j)$. This package reports and bootstraps scaled `n V_n`, the form compared with the wild-bootstrap distribution, and includes `k0(x_i,x_i)` unlike `ksd_u_test()`.

The algorithm (1) builds the full Stein-kernel matrix including its diagonal, (2) computes scaled `n V_n`, (3) generates wild-bootstrap signs, and (4) recomputes the quadratic form with those signs. Independent samples use independent Rademacher signs. Chwialkowski et al.'s dependent-sample result requires more than ordered rows: a stationary tau-mixing process satisfying

$$\sum_{t=1}^{\infty} t^2\sqrt{\tau(t)} < \infty,$$

Lipschitz `k0`, and $E[k0(Z,Z)^2] < \infty$. Geometrically ergodic Markov chains meet the mixing requirement only under additional moment assumptions.

Markov signs form a two-state chain that switches between neighboring observations with probability `change_prob`. Asymptotically the paper uses $p_n = 1/w_n$, where $w_n \to \infty$ and $w_n = o(n)$, equivalently $p_n \to 0$ and $np_n \to \infty$. Thus `change_prob` depends on sample size and dependence strength. `0.5` yields independent signs–the paper's i.i.d. setting–and can miscalibrate dependent rows. Large positive values indicate disagreement with the target score.

The package reports

$$\frac{1 + \#\{b : T_b \geq T_{obs}\}}{B + 1}.$$

Chwialkowski et al. state their decision through the empirical bootstrap quantile. The observed and bootstrap statistics follow the paper, but this corrected finite-simulation p-value adds no finite-sample validity guarantee for dependent data.

Like `ksd_u_test()`, this function starts from `X`, `score_function`, and a Stein kernel. The U version is off-diagonal and uses centered multinomial weights; this V version keeps self-interactions, reports `n V_n`, and uses wild signs. Its lower-level pieces are `ksd_vq_matrix()`, `ksd_v_statistic()`, and `ksd_v_bootstrap()`.

Value

An object of class `htest`, implemented as a list with:

- `statistic`: one number labeled `ksd_v`; this is the observed scaled statistic `n V_n`.
- `p.value`: right-tail wild-bootstrap p-value using the package's `(1 + exceedances) / (nboot + 1)` finite-bootstrap convention.
- `method`: text naming the bootstrap method and kernel.
- `data.name`: expression used for `X`.
- `parameter`: numeric vector containing `nboot`, the squared kernel `scaling` used, and `change_prob` when `boot_method = "markov"`.

- bootstrap_samples: present only when return_raw_boot = TRUE; numeric vector of wild-bootstrap statistics on the same n V_n scale.

The htest shape behaves like a standard R test; parameter records the bootstrap and kernel choices because both affect the simulated null.

Examples

```
X <- matrix(rnorm(20), ncol = 1)
score_function <- function(x) -as.matrix(x)
ksd_v_test(X, score_function, nboot = 10)
```

---

ksd_vq_matrix *Build the Stein-kernel matrix for the KSD V-statistic*

Description

Computes the pairwise Stein kernel values used by the V-statistic version of the KSD test.

Usage

```
ksd_vq_matrix(
  X,
  score_function,
  scaling = NULL,
  kernel = c("gaussian_rbf", "imq"),
  imq_beta = -0.5
)
```

Arguments

X	Numeric vector or matrix of samples.
score_function	Function returning the target score for each row of X.
scaling	Positive kernel scale. NULL uses the square of the exact all-pair median distance.
kernel	Kernel choice: "gaussian_rbf", "imq", or a SteinKernel object, including objects created by custom_stein_kernel().
imq_beta	IMQ kernel exponent.

Details

Entry (i, j) is k0(x_i, x_j), including diagonal entries. Use ksd_v_statistic() to turn the matrix into the scaled statistic n V_n, ksd_v_bootstrap() to generate wild-bootstrap draws from the fixed matrix, or ksd_v_test() to run the full test. See stein_kernel_matrix() for the formula used to assemble k0.

This is the V-statistic analogue of ksd_uq_matrix(). It is useful when the sample comes from an ordered or dependent process and the user wants to keep matrix construction separate from the wild-bootstrap step. The diagonal is retained because the V-statistic includes self-interactions.

To move sideways from this function to ksd_uq_matrix(), keep the same X, score_function, and kernel settings. The matrix entries are still k0(x_i, x_j). The difference is the statistic and bootstrap that should be used afterward: U-statistic helpers drop the diagonal and use centered multinomial weights; V-statistic helpers keep the diagonal and use wild-bootstrap signs. The q in ksd_vq_matrix() follows the literature's Stein-kernel notation; it does not introduce a second user-supplied distribution. The target is the density whose score is returned by score_function, and the returned object is the package's K0 matrix with entries k0(x_i, x_j).

Value

Numeric n x n matrix with entry k0(x_i, x_j), including the diagonal. Pass this matrix to ksd_v_statistic() to get n V_n, or to ksd_v_bootstrap() to generate wild-bootstrap replicates.

Examples

```
X <- matrix(rnorm(5), ncol = 1)
score_function <- function(x) -as.matrix(x)
ksd_vq_matrix(X, score_function)
```

---

likelihoodgmm *Compute the mixture density for samples*

Description

Evaluates the Gaussian mixture density at each supplied sample.

Usage

```
likelihoodgmm(model = NULL, X = NULL)
```

Arguments

model	Gaussian mixture model returned by gmm().
X	Numeric vector or matrix of samples.

Details

For observation x_i, the returned value is

$$p(x_i) = \sum_{k=1}^{K} w_k \phi(x_i; \mu_k, \Sigma_k).$$

The component densities are combined on the log scale and then converted back to ordinary density values.

This function returns the density itself, not a log density. Use it for plots or simple likelihood checks. For MCMC transitions such as mala() and grwmetrop(), pass a separate log_p function because those transitions work on the log-density scale.

The function sits beside posteriorgmm() and scorefunctiongmm(). All three evaluate the same mixture at supplied sample rows, but they answer different questions: likelihoodgmm() returns the marginal density p(x), posteriorgmm() returns component responsibilities conditional on x, and scorefunctiongmm() returns the gradient of log p(x) needed by Stein algorithms.

Value

Numeric vector of length nrow(X) containing p(x_i) for each sample row. Values are nonnegative density values and do not sum to one over the sample. They are pointwise density evaluations, so their scale depends on the dimension and covariance matrices of the mixture.

Examples

```
# compute likelihood for a default 1-d gaussian mixture model
# and dataset generated from it
model <- gmm()
X <- rgmm(model)
p <- likelihoodgmm(model = model, X = X)
```

---

mala *Run a Metropolis-adjusted Langevin chain*

Description

Runs the MALA transition used as one of the short candidate-chain kernels in SP-MCMC. MALA uses the target score to drift proposals toward high-density regions and then applies a Metropolis correction.

Usage

```
mala(log_p, score_function, x0, h, Sigma = NULL, m_iter)
```

Arguments

log_p	Function returning log density values.
score_function	Function returning score values.
x0	Initial state vector.
h	Positive step size.
Sigma	Symmetric positive-definite proposal covariance or preconditioning matrix.
m_iter	Number of returned chain rows, including the initial state in row 1. The function therefore makes m_iter - 1 Markov transitions.

Details

From the current state `x`, MALA proposes a point near

$$x + (h/2)s_p(x)Sigma^{-1}.$$

More explicitly, with the row-vector convention used by the code,

$$y = x + (h/2)s_p(x)Sigma^{-1} + \sqrt{h}z, \quad z \sim N(0, Sigma).$$

Thus the proposal noise has covariance `h * Sigma`. The acceptance step compares the target log density and the two proposal densities, so accepted states leave the distribution described by `log_p` invariant. The returned score matrix `D` is included because SP-MCMC can reuse these scores when scoring the candidate path.

When `Sigma` is the identity matrix this reduces to the basic MALA proposal written in the SP-MCMC appendix, $y = x + (h/2)\nabla \log p(x) + \sqrt{h}z$. The general `Sigma` argument is the matrix used by this implementation's proposal calculation; it is separate from the `M` or `precon` matrix that may be stored inside an IMQ Stein kernel for computing `k0`.

MALA is a transition kernel, not a full sampler interface. It is exported so SP-MCMC users can inspect or replace the candidate-chain step. `sp_mcmc()` calls it when `mcmc = "mala"` and then passes its chain output to `sp_mcmc_eval_candidates()`.

Value

A list with `X` (chain states, one row per iteration), `D` (scores at those states), `log_p` (log density values), `accept` (0/1 acceptance indicators), and evaluation-count fields. `D` is returned because SP-MCMC can reuse those scores instead of calling the score function again for every candidate.

Examples

```
score <- function(X) -as.matrix(X)
log_p <- function(X) -0.5 * rowSums(as.matrix(X)^2)
mala(log_p, score, x0 = 0, h = 0.1, m_iter = 3)
```

---

perturbgmm *Perturb the component means of a Gaussian mixture model*

Description

Adds independent standard normal noise to each component mean while keeping the covariances and weights unchanged.

Usage

```
perturbgmm(model = NULL)
```

Arguments

`model`	Gaussian mixture model returned by `gmm()`.

Details

This helper is useful for examples where a fitted or proposed mixture should be close to an existing one but not exactly equal to it. If the original means are `mu_k`, the new means are `mu_k + z_k`, with each `z_k` drawn from a standard normal distribution in the same dimension.

It changes only the means because that gives a simple alternative target for goodness-of-fit examples: the modes move, but the number of components, component weights, and covariance shapes stay comparable.

The function is meant to create a nearby but different model, not to perform statistical fitting. For example, one can draw data from `model`, build `noisymodel <- perturbgmm(model)`, and then test whether the sample looks consistent with the perturbed score. Keeping weights and covariances fixed makes the difference easy to interpret: failures are driven by shifted component locations rather than by a completely different mixture shape. The perturbation scale is intentionally simple: each coordinate of each component mean receives one standard-normal draw.

Value

A Gaussian mixture list with the same structure as `gmm()`. The `mu` field is perturbed; `sigma`, `weights`, `nComp`, and `d` are copied from the input model. Returning the full model object, rather than only the perturbed means, lets the result be passed directly to `rgmm()`, `likelihoodgmm()`, `posteriorgmm()`, `scorefunctiongmm()`, or `get_score_evaluator()`.

Examples

```
# Add noise to default 1-d gaussian mixture model
model <- gmm()
noisymodel <- perturbgmm(model)
```

`plotgmm` *Plot a one-dimensional Gaussian mixture sample*

Description

Draws a histogram with a kernel density overlay. Optional component means are shown as vertical reference lines.

Usage

```
plotgmm(data, mu = NULL)
```

Arguments

`data`	Numeric vector of one-dimensional samples.
`mu`	Optional component means to mark on the plot.

Details

The function is intended as a quick visual check for one-dimensional simulation examples. The histogram shows the empirical sample distribution, the smooth line shows a kernel density estimate, and the optional vertical lines mark supplied component means.

This plotting helper is deliberately separate from the mixture-density functions. It visualizes sampled data; it does not evaluate the exact mixture density and it returns no numerical object for downstream Stein calculations.

Value

The function is called for its plotting side effect and returns `NULL` invisibly. It does not return density values, posterior probabilities, or scores; use `likelihoodgmm()`, `posteriorgmm()`, or `scorefunctiongmm()` for those numerical quantities.

Examples

```
# Plot a histogram for a given dataset
model <- gmm()
X <- rgmm(model)
plotgmm(data = X)

# Plot the histogram with lines indicating the component means
model <- gmm()
mu <- model$mu
X <- rgmm(model)
plotgmm(data = X, mu = mu)
```

`posteriorgmm` *Compute posterior component probabilities for a Gaussian mixture*

Description

For each observation, this function computes the probability that the observation came from each mixture component.

Usage

```
posteriorgmm(model = NULL, X = NULL)
```

Arguments

`model`	Gaussian mixture model created by `gmm()`.
`X`	Numeric vector or matrix of samples.

Details

The posterior probability for component k at observation x_i is

$$P(Z_i = k|x_i) = \frac{w_k \phi(x_i; \mu_k, \Sigma_k)}{\sum_l w_l \phi(x_i; \mu_l, \Sigma_l)}.$$

The calculation is done on the log scale first, which avoids numerical underflow when densities are very small. If every component log density is -Inf for a row, the function stops instead of returning arbitrary responsibilities.

These posterior probabilities are not used to classify observations in the package; they are an intermediate quantity for scorefunctiongmm(). The mixture score is a posterior-weighted average of component Gaussian scores. Returning the full responsibility matrix, instead of only the most likely component label, preserves the uncertainty needed for that weighted average.

Value

Numeric matrix with nrow(X) rows and model$nComp columns. Entry (i, k) is the posterior probability that sample row i came from component k; each row sums to one up to numerical rounding. The column order matches the component order in model$mu, model$sigma, and model$weights, so the matrix can be multiplied component-by-component with Gaussian score contributions.

Examples

```
model <- gmm()
X <- rgmm(model, n = 5)
posteriorgmm(model, X)
```

---

rgmm *Sample from a Gaussian mixture model*

Description

Draws independent observations from the mixture object returned by gmm(). For each observation, the function samples a component label using the model weights and then draws from that component's Gaussian distribution.

Usage

```
rgmm(model = NULL, n = 100)
```

Arguments

model	Gaussian mixture model returned by gmm().
n	Number of samples to draw.

Details

This is the simulation counterpart of likelihoodgmm() and scorefunctiongmm(). It is included so examples can generate data from the same mixture object used as a target model for Stein tests and samplers.

The returned observations are independent draws from the mixture. The sampled component labels are used only internally; they are not returned because the public Stein routines work with observations and scores, not latent mixture labels. If labels are needed for a simulation study, reproduce the two-step sampling logic with the same model$weights, model$mu, and model$sigma fields.

Value

For model$d == 1, a numeric vector of length n. For model$d > 1, an n x d numeric matrix with one simulated observation per row. The different shape for one-dimensional mixtures keeps simple examples readable, while the package's validation helpers convert vectors back to n x 1 matrices when needed.

Note

Multivariate sampling uses the package dependency mvtnorm.

Examples

```
# Generate 100 samples from default gaussian mixture model
model <- gmm()
X <- rgmm(model)

# Generate 300 samples from 3-d gaussian mixture model
model <- gmm(d = 3)
X <- rgmm(model, n = 300)
```

---

rwm *Alias for the Gaussian random-walk transition*

Description

Calls grw() with the same arguments. It is provided because the SP-MCMC paper refers to the Gaussian random-walk Metropolis transition as RWM.

Usage

```
rwm(log_p, score_function, x0, h = NULL, Sigma = NULL, m_iter)
```

Arguments

log_p	Function returning log density values.
score_function	Unused score function argument kept for transition interface compatibility.
x0	Initial state vector.
h	Optional positive step-size multiplier.
Sigma	Symmetric positive-definite proposal covariance or preconditioning matrix.
m_iter	Number of returned chain rows, including the initial state in row 1. The function therefore makes m_iter - 1 Markov transitions.

Details

This function is intentionally thin. The package's implementation name for the transition is grw(), short for Gaussian random walk, while the paper commonly uses RWM, short for random-walk Metropolis. Keeping both exported names avoids forcing users to translate between the paper notation and the package notation.

No additional computation happens here: rwm() forwards the arguments to grw(), which in turn calls grwmetrop() with proposal covariance h * Sigma. The return object therefore has exactly the same fields as grw().

Value

Output from grw(): chain states, log-density values, acceptance indicators, and evaluation counts. The alias exists for naming compatibility with the SP-MCMC paper; it does not change the transition or add fields.

Examples

```
score <- function(X) -as.matrix(X)
log_p <- function(X) -0.5 * rowSums(as.matrix(X)^2)
rwm(log_p, score, x0 = 0, h = 0.1, m_iter = 3)
```

---

scorefunctiongmm *Compute the score of a Gaussian mixture density*

Description

The score is the gradient of the log mixture density with respect to x.

Usage

```
scorefunctiongmm(model = NULL, X = NULL)
```

Arguments

model	Gaussian mixture model returned by gmm().
X	Numeric vector or matrix of samples.

Details

For a mixture, the score is a weighted average of the component scores:

$$\nabla_x \log p(x_i) = \sum_{k=1}^{K} P(Z_i = k|x_i) \left[-\Sigma_k^{-1}(x_i - \mu_k)\right].$$

The weights in this average are the posterior component probabilities returned by `posteriorgmm()`.

This is the Gaussian mixture helper most directly connected to the Stein routines. `ksd_u_test()`, `fssd_test()`, `stein_points()`, and related functions need `function(X)` returning the score matrix; this function computes that matrix for one supplied batch.

If the target were a single Gaussian, the score would be one linear term `-Sigma^{-1}(x - mu)`. For a mixture, the function computes that linear score for every component and averages the component scores using the posterior probabilities from `posteriorgmm()`. This is why both the density calculation and the inverse covariance matrices appear in the implementation.

Value

Numeric matrix with `nrow(X)` rows and `model$d` columns. Row `i` is `nabla_x log p(x_i)`, the score vector used by Stein discrepancies and score-based samplers. The matrix has the same row order as the supplied sample matrix, which is important because KSD, FSSD, Stein thinning, and Stein Points pair each sample row with its corresponding score row.

Examples

```
# Compute the score for a Gaussian mixture model and dataset
model <- gmm()
X <- rgmm(model)
score <- scorefunctiongmm(model = model, X = X)
```

---

sp_mcmc *Generate Stein Points from short MCMC candidate paths*

Description

Implements Stein Point Markov Chain Monte Carlo (SP-MCMC). At each Stein Points step, the algorithm runs a short MCMC chain to produce a manageable candidate set, then appends the candidate that most reduces the greedy KSD objective.

Usage

```
sp_mcmc(
  score_function,
  log_p,
  kernel,
  n_points,
  d,
```

```
  mcmc = c("grw", "mala", "rwm"),
  criterion = c("last", "rand", "infl"),
  m_seq,
  h = NULL,
  Sigma = NULL,
  x_init,
  seed = NULL,
  transition_fn = NULL,
  proposal_fn = NULL,
  n_eval_fn = NULL,
  criterion_args = list()
)
```

Arguments

score_function	Function returning target scores for sample rows.
log_p	Function returning log density values for sample rows.
kernel	A stein_kernel() object, compatible kernel function, or list with a k0_matrix function. A Gaussian RBF kernel must have a fixed positive bandwidth h; SP-MCMC evaluates one fixed Stein kernel across all short candidate paths.
n_points	Total number of points to generate.
d	State dimension.
mcmc	MCMC transition: "grw" for Gaussian random-walk Metropolis, "mala" for MALA, or alias "rwm".
criterion	Rule for choosing where the next MCMC chain starts: "last", "rand", "infl", or a custom function.
m_seq	Number of Markov transitions, and hence candidates before duplicate removal, at each selection step. Supply one value, one value per point, or one value per update after the first point.
h	Step-size multiplier used by the built-in MCMC transitions.
Sigma	Symmetric positive-definite proposal covariance or preconditioning matrix. If NULL, the identity matrix is used.
x_init	Fixed first point.
seed	Optional RNG seed.
transition_fn	Optional custom MCMC transition function. It receives the log density, score function, current state, step size, proposal matrix, and a requested row count of m_seq[j] + 1. It must return exactly that many rows in X, with the supplied current state in row 1. The remaining m_seq[j] rows are states after successive Markov transitions.
proposal_fn	Optional function that changes h or Sigma at each step. It should return a list containing replacement h and/or Sigma.
n_eval_fn	Optional function that overrides the per-step evaluation count.
criterion_args	optional list of extra arguments passed to a custom criterion function.

Details

SP-MCMC replaces Stein Points' global state-space search by a short Markov chain. At step `j` it starts from an already selected point, makes `m_seq[j]` Markov transitions, removes duplicate states among the resulting `m_seq[j]` candidates, and scores each remaining candidate with the greedy objective. The initial state is not a candidate. With target score $s_p(x) = \nabla_x \log p(x)$ and the `k0` from `stein_kernel_matrix()`, the prefix diagnostic is

$$KSD_m^2 = \frac{1}{m^2} \sum_{a=1}^{m} \sum_{b=1}^{m} k0(x_a, x_b).$$

The paper minimizes

$$\frac{1}{2} k0(x, x) + \sum_{i=1}^{j-1} k0(x_i, x).$$

The equivalent implementation form is

$$k0(x, x) + 2 \sum_{i=1}^{j-1} k0(x_i, x),$$

whose smallest candidate is appended.

`x_init` fixes the first point, leaving `n_points - 1` short chains. Scalar `m_seq` is recycled; a length-`n_points - 1` vector supplies each subsequent length. Built-in transitions return states, log densities, acceptance indicators, and reusable MALA scores. A cached `K0[a,b]=k0(x_a,x_b)` lets start rules, candidate scores, and KSD diagnostics share selected-point interactions.

Start rules are `"last"` (latest point), `"rand"` (random selected point), and `"infl"` (smallest current Stein-matrix influence). Built-in transitions are the paper's Gaussian random-walk Metropolis and MALA. Custom `criterion`, `transition_fn`, and `proposal_fn` expose these pieces without replacing the main loop. Diagnostics split MCMC and candidate-scoring evaluations in `counts`, accumulate them in `cum_n_eval`, and record acceptance rates and squared-distance movement for each short path.

`sp_mcmc_state()` builds start-rule state; `sp_mcmc_criterion()` turns LAST/RAND/INFL or a custom function into a reusable selector; `sp_mcmc_select_start()` validates its row index; and `sp_mcmc_eval_candidates()` applies the Stein Points objective. Thus moving from `stein_points()` to `sp_mcmc()` retains the greedy objective but replaces its optimizer with `grw()`, `rwm()`, `mala()`, or another MCMC transition.

Value

A list of class `"sp_mcmc"` and `"stein_points"` with:

- `X`: `n_points x d` matrix of selected point locations.
- `D`: target scores at the selected points.
- `ksd`: running KSD diagnostic after each selected point, with `ksd[j]` equal to the square root of the empirical double average of `k0` over the first `j` selected rows.
- `n_eval`, `cum_n_eval`: per-step and cumulative evaluation counts.
- `counts`: matrix splitting each step's count into `log_p`, `score`, `candidate_score`, `transition_total`, and `total`.

- `method`, `kernel`, `mcmc`, `criterion`, `m_seq`, `h`, `Sigma`: algorithm settings used for the run.
- `selected_index`: index of the previously selected point used to start each short chain. This records the output of the LAST/RAND/INFL or custom start rule before the MCMC transition is run.
- `chain_d2_max`, `chain_d2_selected`, `chain_d2_last`: squared Euclidean distances from the chain start to the farthest candidate, selected candidate, and last candidate. `chain_d2_first_last` is the squared distance between the first and last candidates, as used in Figure 2 of the SP-MCMC paper.
- `accept_rate`: Metropolis acceptance rate for each short chain, useful for interpreting whether poor point choices came from the Stein objective or from a candidate chain that barely moved.

Examples

```
score <- function(X) -as.matrix(X)
log_p <- function(X) -0.5 * rowSums(as.matrix(X)^2)
kernel <- stein_kernel(type = "gaussian_rbf", h = 1)
sp_mcmc(score, log_p, kernel, n_points = 2, d = 1, m_seq = 2, x_init = 0)
```

`sp_mcmc_criterion` *Create an SP-MCMC start-point rule*

Description

`criterion` may be one of `"last"`, `"rand"`, `"infl"`, or a function that accepts an `sp_mcmc_state` object and returns a one-based index into `state$X`.

Usage

```
sp_mcmc_criterion(
  criterion = c("last", "rand", "infl"),
  criterion_args = list()
)
```

Arguments

`criterion`	Criterion name, function, or criterion object.
`criterion_args`	Extra arguments passed to a custom criterion function.

Details

The SP-MCMC paper studies three rules for choosing where the next candidate chain starts. `"last"` starts at the most recently selected point. `"rand"` chooses a selected point uniformly at random. `"infl"` computes each point's current influence from the Stein kernel matrix,

$$I_a = \sum_b K0_{ab} + \sum_b K0_{ba} - K0_{aa},$$

and starts at the point with the smallest `I_a`. A custom function can implement another rule while still using the same SP-MCMC main loop.

The returned object is a small strategy object: it stores a human-readable `label` and a `select(state)` function. `sp_mcmc()` uses that object at every iteration. This design keeps the choice of starting point independent of the MCMC transition and the candidate-scoring rule.

The label is not just cosmetic. It is copied into the final `sp_mcmc()` output so that simulation results can record whether LAST, RAND, INFL, or a custom rule was used. The `select` function is the executable part of the rule. Splitting those two pieces lets a custom rule behave like the built-in rules in both computation and reporting.

Value

A list with `label` and `select`. `label` is stored in the final `sp_mcmc()` output. `select` is a function that takes an `sp_mcmc_state()` object and returns one row index into `state$X`. Returning this small object, rather than returning an index immediately, is necessary because the state changes at every SP-MCMC step.

Examples

```
sp_mcmc_criterion("last")
```

---

sp_mcmc_eval_candidates

Score SP-MCMC candidate points

Description

Scores candidate rows from a short MCMC path using the same greedy Stein Points objective used inside `sp_mcmc()`.

Usage

```
sp_mcmc_eval_candidates(
  kernel,
  score_function,
  X_curr,
  D_curr,
  cand_X,
  cand_D = NULL
)
```

Arguments

`kernel`	Stein kernel object or compatible custom kernel.
`score_function`	Function returning scores for candidate rows.
`X_curr`	Current selected point matrix.

D_curr	Current score matrix.
cand_X	Candidate point matrix.
cand_D	Optional candidate score matrix.

Details

For each candidate x, the objective in the paper is

$$\frac{1}{2}k0(x,x) + \sum_i k0(x_i,x)$$

in the notation of the Stein Points paper. This helper returns the doubled equivalent

$$k0(x,x) + 2\sum_i k0(x_i,x),$$

where the sum runs over rows in X_curr. The returned f_vec contains one objective value per candidate row, in the same order as cand_X. If the MCMC transition already returned candidate scores, pass them as cand_D; otherwise this helper evaluates score_function on the candidate rows and records how many score evaluations were needed.

This function is the bridge between the MCMC transition and the Stein Points objective. MCMC proposes a finite candidate set; this helper evaluates those candidates using the current Stein Points objective and returns the pieces needed for sp_mcmc() to append the best one.

Value

A list with:

- f_vec: doubled greedy objective value for each candidate row. The row with the smallest value is the point sp_mcmc() appends.
- D_new: score matrix for the candidate rows, either reused from cand_D or computed from score_function; it is returned so the selected row can be appended without another score call.
- score_eval: number of candidate score evaluations performed inside this helper.

Examples

```
score <- function(X) -as.matrix(X)
kernel <- stein_kernel(type = "gaussian_rbf", h = 1)
X_curr <- matrix(0, ncol = 1)
D_curr <- score(X_curr)
cand_X <- matrix(c(-0.5, 0.5), ncol = 1)
sp_mcmc_eval_candidates(kernel, score, X_curr, D_curr, cand_X)
```

`sp_mcmc_select_start` *Choose the next SP-MCMC chain start*

Description

Applies a criterion object to the current state and checks that the returned value is a valid row index.

Usage

```
sp_mcmc_select_start(criterion_obj, state)
```

Arguments

`criterion_obj`	Criterion object from `sp_mcmc_criterion()`.
`state`	Current SP-MCMC state.

Details

This helper is the validation step after `sp_mcmc_criterion()`. It keeps custom start rules honest by requiring a single one-based row index into the current point set. The selected row becomes the first state of the short MCMC chain for the next SP-MCMC update.

It is exported mainly for custom criterion development. A user can build a state with `sp_mcmc_state()`, build a criterion with `sp_mcmc_criterion()`, and check that the criterion returns a legal index before running the full SP-MCMC algorithm.

Inside `sp_mcmc()`, the returned index decides which already selected point becomes the initial state of the next short candidate chain. That index is also recorded in `selected_index` in the final output, because the start rule is part of the algorithm's behavior and affects the candidate set that is searched.

Value

One integer: the one-based row index in `state$X` that should initialize the next short candidate chain. The function returns only the checked index because the state object already contains the point matrix, scores, and diagnostic history.

Examples

```
state <- sp_mcmc_state(j = 2, X = matrix(0, ncol = 1),
                       D = matrix(0, ncol = 1), K0 = matrix(1))
sp_mcmc_select_start(sp_mcmc_criterion("last"), state)
```

`sp_mcmc_state` *Store the current SP-MCMC state for a start rule*

Description

Creates the state object passed to custom SP-MCMC start-point criteria. The criterion uses this object to decide which selected point should initialize the next short MCMC chain.

Usage

```
sp_mcmc_state(
  j,
  X,
  D,
  K0,
  ksd = NULL,
  n_eval = NULL,
  counts = NULL,
  selected_index = NULL,
  kernel = NULL,
  mcmc = NULL,
  criterion = NULL
)
```

Arguments

`j`	Current greedy index.
`X`	Current selected point matrix.
`D`	Current score matrix.
`K0`	Current Stein kernel matrix.
`ksd`	Optional running KSD values.
`n_eval`	Optional evaluation counts.
`counts`	Optional detailed evaluation-count matrix.
`selected_index`	Optional selected start indices.
`kernel`	Kernel used by SP-MCMC.
`mcmc`	MCMC transition name.
`criterion`	Criterion label.

Details

The state contains the current point set, scores, and Stein kernel matrix, as well as diagnostic vectors accumulated so far. The matrix field satisfies `K0[a, b] = k0(X[a, ], X[b, ])`, using the same target scores stored in `D`. In the paper, the start rule is one of LAST, RAND, or INFL. This object makes

those rules explicit: LAST uses `state$X`, RAND samples from its rows, and INFL uses `state$K0`. A custom rule should inspect the fields it needs and return one row index from `state$X`.

This constructor is exported for users who write custom SP-MCMC start rules. The main `sp_mcmc()` loop creates this object internally before every short chain. By exposing the same object, the package lets custom rules be tested outside the full algorithm.

Value

A list of class `"sp_mcmc_state"` with the fields supplied as arguments: current points `X`, scores `D`, Stein-kernel matrix `K0`, current step `j`, running diagnostics, and method labels. It is not a result object; it is a snapshot passed to a start-point criterion so the criterion can decide where the next short MCMC candidate chain should start without recomputing scores or pairwise kernel values.

Examples

```
sp_mcmc_state(j = 2, X = matrix(0, ncol = 1), D = matrix(0, ncol = 1),
              K0 = matrix(1))
```

`stein_codescent` *Refine Stein Points by coordinate descent*

Description

Runs the budget-constrained refinement described with Stein Points: keep the number of points fixed, then repeatedly replace one existing point by a point that reduces the same KSD objective.

Usage

```
stein_codescent(X0, score_function, kernel, n_iter, optimizer, seed = NULL)
```

Arguments

`X0`	Initial point matrix.
`score_function`	Function returning scores for candidate rows.
`kernel`	Stein kernel object or compatible custom kernel.
`n_iter`	Number of coordinate-descent updates.
`optimizer`	Optimizer function used for each coordinate update.
`seed`	Optional RNG seed.

Details

At iteration `it`, row `((it - 1) %% nrow(X0)) + 1` is optimized while the other rows are held fixed. For the row being replaced, the objective is the greedy Stein Points objective built from all remaining rows:

$$k0(x, x) + 2 \sum_{i \neq r} k0(x_i, x),$$

where `r` is the row currently being replaced. This is the same doubled objective used by the greedy branch of `stein_points()`, but the selected set size is fixed instead of growing by one point. The update is useful after a fixed budget of points has already been produced, because it improves the locations without increasing the point count.

This function is below `stein_points()` in the workflow. `stein_points()` builds an initial sequence; `stein_codescent()` treats that sequence as a fixed-size design and improves one coordinate row at a time. It uses the same optimizer interface as `stein_points()`, so a grid, Monte Carlo, or Nelder-Mead search can be reused for both construction and refinement.

Value

A list with:

- `X`: refined point matrix with the same dimensions as `X0`.
- `D`: target scores at the refined points.
- `n_eval`: evaluation counts for each coordinate-descent update.
- `cum_n_eval`: cumulative evaluation counts.
- `kernel`: kernel used to define the Stein objective.

The return shape keeps the same core pieces as `stein_points()` that are needed downstream: refined locations, their target scores, the kernel, and evaluation budgets. It does not store a full KSD trajectory because each coordinate-descent step replaces an existing row rather than appending a new prefix. To compute the final KSD, build `K0 <- stein_kernel_matrix(kernel, out$X, out$D)` and evaluate `sqrt(sum(K0)) / nrow(out$X)`.

Examples

```
score <- function(X) -as.matrix(X)
kernel <- stein_kernel(type = "gaussian_rbf", h = 1)
opt <- fmin_grid(lb = -1, ub = 1, n0 = 3, grow = FALSE)
X0 <- matrix(c(-0.5, 0.5), ncol = 1)
stein_codescent(X0, score, kernel, n_iter = 1, optimizer = opt)
```

`stein_kernel` *Create a built-in Stein kernel*

Description

Creates one of the built-in base kernels and stores the parameters needed to assemble the Stein kernel `k0`. The returned object can be passed to KSD, FSSD, SVGD, Stein Points, SP-MCMC, and Stein thinning functions.

Usage

```
stein_kernel(
  type = c("gaussian_rbf", "imq"),
  sigma = NULL,
  h = NULL,
  beta = -0.5,
  c = 1,
  precon = NULL
)
```

Arguments

`type`	Kernel type, `"gaussian_rbf"` or `"imq"`.
`sigma`, `h`	Gaussian RBF bandwidth. Provide at most one.
`beta`	IMQ exponent.
`c`	IMQ length scale.
`precon`	Optional positive definite matrix used to rescale distances.

Details

The Gaussian RBF base kernel is

$$k(x, y) = \exp(-r(x, y)/(2h^2)),$$

where `h` is the bandwidth and `h^2` is the squared scale stored in the object. The IMQ base kernel is

$$k(x, y) = (c^2 + r(x, y))^{\beta},$$

where positive `c` is the length scale, `c^2` the squared offset, and `beta` is negative. Both feed the same Stein-kernel machinery: `eval_kernel()`, `grad_x_kernel()`, `trace_mixed_kernel()`, `cross_kernel()`, and `stein_kernel_matrix()` dispatch to the corresponding formulas.

RBF is common in KSD, FSSD, and SVGD; exponential squared-distance decay can make off-diagonal entries nearly zero when `h` is too small. IMQ has polynomial decay for `beta < 0`, is the Stein-thinning default, and is common in Stein Points/SP-MCMC because their papers study its convergence. Switching from RBF to IMQ keeps `X`, score, and algorithm but replaces `h^2` by `c^2` and selects negative `beta`.

By default,

$$r(x, y) = ||x - y||^2.$$

If precon is supplied, distances become

$$r(x, y) = (x - y)^T precon(x - y).$$

The base kernel, first derivatives, and mixed trace all use this r. Top-level KSD/FSSD pass squared scaling: h^2 for RBF and c^2 for IMQ. With scaling = NULL, find_median_distance() returns the squared exact all-pair median; its quadratic time and memory cost favors fixed scales for large samples.

An S3 object stores parameters so the base kernel, first derivative, mixed trace, and FSSD optimization derivative dispatch consistently through stein_kernel_matrix(), compute_tau(), and the kernel generics.

Specialized Stein Points constructors at this layer are stein_kernel_inverse_log() for position-distance inverse-log and stein_kernel_imq_score() for score-distance IMQ.

Value

A list with class "SteinKernel_gaussian_rbf" or "SteinKernel_imq" plus the common class "SteinKernel". For Gaussian RBF kernels it stores type, optional squared bandwidth h2, and optional precon; h2 = NULL invokes the median squared-distance rule where supported. IMQ objects store type, beta, c, and optional precon. Pass the result to stein_kernel_matrix(), ksd_u_test(), fssd_test(), stein_points(), sp_mcmc(), and stein_thinning(). For adaptive RBF, KSD/FSSD freeze the input-data median scale, whereas SVGD recomputes the paper's dynamic median each iteration. Stein Points, SP-MCMC, coordinate descent, and Stein thinning require explicit fixed h because their KSD objectives need one kernel for all candidate and diagonal values.

Examples

```
stein_kernel(type = "gaussian_rbf", h = 1)
stein_kernel(type = "imq", c = 1, beta = -0.5)
```

---

stein_kernel_imq_score

Create a score-distance IMQ Stein kernel

Description

Creates the score-distance IMQ kernel used in the Stein Points experiments. This kernel measures distance between score vectors rather than distance between point locations.

Usage

```
stein_kernel_imq_score(alpha = 1, beta = -0.5, hess_log_p)
```

Arguments

alpha	Positive offset parameter.
beta	Exponent in (-1, 0).
hess_log_p	Function returning a Hessian array with shape n x d x d.

Details

The base kernel is an IMQ kernel applied to differences between score vectors, rather than differences between sample positions:

$$k(x, y) = (\alpha + ||s_p(x) - s_p(y)||^2)^\beta.$$

Here $s_p(x) = \nabla_x \log p(x)$ is the target score. Two points are close under this kernel when the target score field behaves similarly at the two points, even if the points themselves are not close in Euclidean distance. This is a different design choice from the IMQ option in stein_kernel(), which applies the IMQ kernel to ||x - y||^2 or its preconditioned version.

Because the base kernel depends on $s_p(x)$, its derivatives with respect to x depend on the Hessian of the target log density. The hess_log_p argument supplies that information. It should return an array whose first dimension indexes rows of X and whose remaining two dimensions contain the d x d Hessian matrix for each row. The package uses those Hessians when computing k0(x, y) and the diagonal self-interaction terms used by stein_points() and sp_mcmc().

This is the third Stein Points kernel family implemented here. It is parallel to the position-distance IMQ kernel from stein_kernel() and the inverse-log kernel from stein_kernel_inverse_log(), but it requires more target information. The top-level stein_points() call still takes score_function, because the greedy objective needs scores at selected and candidate points. This constructor additionally needs hess_log_p, because differentiating $k(s_p(x), s_p(y))$ with respect to point locations uses derivatives of the score field.

This constructor is separate from stein_kernel() because it requires a target-specific Hessian function, not only a kernel bandwidth or length scale. After construction it still behaves like a SteinKernel object, so stein_kernel_matrix() dispatches to its complete score-aware method. This is also why it cannot be represented by custom_stein_kernel(), whose leaf functions receive point matrices but not target-score matrices.

Value

A list with class "SteinKernel_imq_score" and "SteinKernel". It stores the IMQ parameters alpha and beta and the Hessian evaluator hess_log_p. These fields are kept together because evaluating this kernel requires the score-distance formula and the Hessian-based derivative terms; callers should pass the object to stein_points(), sp_mcmc(), or stein_kernel_matrix() rather than extracting the fields manually.

Examples

```
hess_log_p <- function(X) array(-1, dim = c(nrow(as.matrix(X)), 1, 1))
stein_kernel_imq_score(alpha = 1, beta = -0.5, hess_log_p = hess_log_p)
```

`stein_kernel_inverse_log` *Create an inverse-log Stein kernel*

Description

Creates the inverse-log base kernel used in the Stein Points experiments and wraps it as a `SteinKernel` object.

Usage

```
stein_kernel_inverse_log(alpha = 1, beta = -1)
```

Arguments

`alpha`	Positive offset parameter.
`beta`	Negative exponent.

Details

The base kernel has the form

$$k(x, y) = (\alpha + \log(1 + ||x - y||^2))^{\beta}.$$

The parameter `alpha` must be positive and `beta` must be negative; the inverse-log kernel in the Stein Points paper is the special case $\beta = -1$. Compared with a Gaussian RBF kernel, this kernel decays much more slowly as points move apart. That slower decay is useful in the Stein Points setting because the greedy objective needs to see interactions between already selected points and candidates that may be far away.

This constructor is parallel to `stein_kernel()`, but it is kept separate because the inverse-log kernel is mainly part of the Stein Points paper's specialized kernel family. Once constructed, the object still follows the same `SteinKernel` interface: `stein_points()`, `stein_codescent()`, `stein_kernel_matrix()`, and the kernel generics can call the corresponding S3 methods without knowing that the kernel came from this constructor.

The squared distance in the formula is the ordinary Euclidean squared distance between sample rows. Unlike the built-in IMQ constructor in `stein_kernel()`, this helper does not expose a pre-conditioning matrix; it is intended to reproduce the position-distance inverse-log kernel used in the Stein Points experiments.

This is the second of the three Stein Points kernel choices implemented in the package. The first is the position-distance IMQ kernel from `stein_kernel()`, with formula `(c^2 + r(x, y))^beta`. The third is `stein_kernel_imq_score()`, which replaces position distance by score-vector distance and therefore needs Hessians. Moving from IMQ to inverse-log keeps the same required inputs for `stein_points()` and `sp_mcmc()`: points, scores, and the kernel object. Only the base-kernel formula changes.

Value

A list with class "SteinKernel_inverse_log" and "SteinKernel". It stores alpha and beta, which are the only parameters needed by the S3 methods for eval_kernel(), grad_x_kernel(), trace_mixed_kernel(), and cross_kernel(). Returning the parameters in a kernel object, instead of returning only a function, lets the rest of the package evaluate the base kernel and all Stein-kernel derivative terms consistently.

Examples

```
stein_kernel_inverse_log(alpha = 1, beta = -1)
```

---

stein_kernel_matrix *Compute pairwise Stein kernel values*

Description

Combines a base kernel, its derivatives, and the target scores to form the Stein kernel k0. This is the common matrix object used by the package's KSD tests, FSSD features, Stein thinning, Stein Points, and SP-MCMC.

Usage

```
stein_kernel_matrix(kernel, X, grads, Y = NULL, grads_Y = NULL, ...)
```

Arguments

kernel	A Stein kernel object.
X	Numeric matrix with samples in rows.
grads	Score matrix for X.
Y	Optional second sample matrix. If NULL, Y = X.
grads_Y	Optional score matrix for Y. Required when Y is supplied.
...	Additional arguments passed to kernel methods.

Details

Liu, Lee, and Jordan call this u_q(x,x') for target q; here the same quantity is k0(x,y) for target p, since it also serves Stein thinning and Stein Points. Given score s_p(x)=\nabla_x\log p(x) and base kernel k, matrix entries are

$$k0(x,y) = s_p(x)^T s_p(y) k(x,y) + s_p(x)^T \nabla_y k(x,y) + s_p(y)^T \nabla_x k(x,y) + \nabla_x \cdot \nabla_y k(x,y).$$

These are, respectively, score-score coupling through k, two score-kernel derivative couplings, and the mixed-derivative trace obtained by applying the Langevin Stein operator in both arguments.

Points and scores must share row order: X[i,] pairs with grads[i,], and supplied Y[j,] with grads_Y[j,]. Omitting Y reuses X and its scores on both sides. This function never estimates scores; top-level functions such as ksd_u_test() evaluate score_function first.

KSD averages these entries; Stein Points and SP-MCMC use them to construct support points; Stein thinning scores which fixed sample row to retain; and FSSD applies the same Stein operator at finite locations through `compute_tau()`. `Y` produces a rectangular `X`-by-`Y` matrix; omitting it produces the square `X` matrix.

This central layer prevents higher-level routines from reimplementing `k0`. Gaussian RBF and position-distance IMQ use fused versions of the same formula. `custom_stein_kernel()` objects use the default method, assembling `k0` from three leaf functions when the base kernel depends only on `(X,Y)`. A target-dependent kernel whose value or derivatives require scores instead needs a full `stein_kernel_matrix.<class>()` method; `stein_kernel_imq_score()` is the built-in example.

In contrast, `compute_tau()` stores FSSD feature vectors `s_p(x_i)k(x_i,v_j)+\nabla_x k(x_i,v_j)` before inner products, rather than scalar pairwise `k0(x_i,y_j)`; both use the same base methods and score convention.

Value

Numeric matrix. If `Y = NULL`, the matrix is `nrow(X) x nrow(X)` and entry `(i, j)` is `k0(X[i, ], X[j, ])`. If `Y` is supplied, the matrix is `nrow(X) x nrow(Y)` and entry `(i, j)` is `k0(X[i, ], Y[j, ])`. The matrix supports each algorithm's use of the pairwise object: KSD averaging, Stein Points/SP-MCMC accumulation, or Stein-thinning row choice.

Examples

```
X <- matrix(c(-1, 0, 1), ncol = 1)
grads <- -X
kernel <- stein_kernel(type = "gaussian_rbf", h = 1)
stein_kernel_matrix(kernel, X, grads)
```

`stein_points` *Build a Stein Points sequence by greedy KSD minimization*

Description

Implements the Stein Points construction of Chen et al. The function builds a point set whose empirical distribution is meant to approximate the target by making the kernel Stein discrepancy small. The point-selection rule itself introduces no randomness, but reproducibility depends on the numerical search: grid search is deterministic, whereas Monte Carlo and randomly initialized Nelder-Mead searches are stochastic unless their seed is fixed.

Usage

```
stein_points(
  score_function,
  kernel,
  n_points,
  d,
  optimizer,
  method = c("greedy", "herding"),
```

```
  log_p = NULL,
  x_init = NULL,
  c2 = NULL,
  truncation = c("none", "upper", "lower", "linear"),
  seed = NULL
)
```

Arguments

score_function	Function returning target scores for candidate rows.
kernel	Built-in Stein kernel, compatible custom kernel, function `kernel(X, S_X, Y, S_Y)`, or list with `k0_matrix`. A Gaussian RBF kernel must have a fixed positive bandwidth `h`; unlike SVGD, Stein Points uses one fixed Stein kernel throughout the point-selection objective.
n_points	Number of points to select.
d	Dimension of each point.
optimizer	Function taking `(objective, X_curr, t)` and returning `x_min`, `d_min`, `f_min`, and `n_eval`.
method	Point-selection rule: `"greedy"` or `"herding"`.
log_p	Optional log density used to choose the first point.
x_init	Optional first point. If supplied, `log_p` is not used.
c2	Positive truncation constant when `truncation != "none"`.
truncation	Optional filter that removes candidates whose self-kernel value is too large. Use `"none"` for the basic Stein Points algorithm.
seed	Optional local RNG seed.

Details

For selected points $x_1, \ldots, x_m$, the paper's KSD is

$$KSD_m^2 = \frac{1}{m^2} \sum_{a=1}^{m} \sum_{b=1}^{m} k0(x_a, x_b).$$

`stein_kernel_matrix()` assembles `k0` from the base kernel and target score $s_p(x) = \nabla_x \log p(x)$. The first point is `x_init`, or maximizes `log_p` to follow the paper's default mode-near start. Given $x_1, \ldots, x_{j-1}$, greedy selection minimizes

$$\frac{1}{2} k0(x, x) + \sum_{i=1}^{j-1} k0(x_i, x).$$

The implementation uses the equivalent doubled form

$$k0(x, x) + 2 \sum_{i=1}^{j-1} k0(x_i, x),$$

which has the same minimizer and equals the increment to $\sum_{a,b\le j} k0(x_a, x_b)$. Herding instead minimizes

$$\sum_{i=1}^{j-1} k0(x_i, x).$$

Its diagnostic still adds both cross interactions and the new self term, so `ksd` remains comparable between greedy and herding.

Global search is separated from the Stein objective. `optimizer` receives the objective, current matrix `X_curr`, and iteration `t`, and returns `x_min` (selected row), `d_min` (its score), `f_min` (objective), and `n_eval` (target or candidate evaluations). `fmin_grid()`, `fmin_mc()`, and `fmin_nm()` implement the paper experiments' grid, Monte Carlo, and Nelder-Mead searches.

The output stores selected `X`, matching scores `D[j, ]=s_p(X[j, ])`, and

$$ksd[j] = \left\{ \frac{1}{j^2} \sum_{a=1}^{j} \sum_{b=1}^{j} k0(x_a, x_b) \right\}^{1/2}.$$

With `truncation != "none"`, the bound is checked when choosing points 2 through `n_points`. The first point is kept as supplied, or chosen by its density, so check it separately if the same bound must also hold there.

Thus the Stein objective knows the current points and score, while the optimizer only searches for the next candidate; this is why the helper constructors return optimizer functions rather than points.

The three Stein Points kernel families are

$$k_{\mathrm{IMQ}}(x, y) = (c^2 + r(x, y))^{\beta}, \quad r(x, y) = (x - y)^T M (x - y),$$

from `stein_kernel()`, usually with $\beta \in (-1, 0)$;

$$k_{\log}(x, y) = (\alpha + \log(1 + ||x - y||^2))^{\beta},$$

from `stein_kernel_inverse_log()`, with the paper's inverse-log choice recovered at $\beta = -1$; and

$$k_{\mathrm{score}}(x, y) = (\alpha + ||s_p(x) - s_p(y)||^2)^{\beta},$$

from `stein_kernel_imq_score()`. All retain the same greedy/herding loop but use, respectively, position distance, logarithmic position distance, or score-vector distance before common Stein-kernel assembly. Score distance also needs a Hessian evaluator because derivatives of $s_p(x) = \nabla_x \log p(x)$ enter `k0`.

Value

A list with:

- `X`: `n_points x d` matrix of selected point locations.
- `D`: `n_points x d` matrix of target scores evaluated at `X`.
- `ksd`: numeric vector; `ksd[j]` is the square root of the empirical KSD double average for the first `j` selected points, using the same `k0` that drove selection.
- `n_eval`: number of objective or score evaluations charged at each selection step. This is kept separate from `ksd` because different optimizers can reach similar KSD values with very different budgets.

- cum_n_eval: cumulative sum of n_eval, used for budget-vs-accuracy comparisons across fmin_grid(), fmin_mc(), fmin_nm(), and SP-MCMC.
- method: "greedy" or "herding".
- kernel: kernel object or compatible kernel supplied by the user.
- truncation, c2: truncation rule and constant used for candidate filtering.

The full list supports later analysis: stein_codescent() can refine X, D avoids rescoring, ksd records discrepancy, and n_eval/cum_n_eval compare grid, Monte Carlo, Nelder-Mead, and SP-MCMC at equal budgets.

Examples

```
score <- function(X) -as.matrix(X)
log_p <- function(X) -0.5 * rowSums(as.matrix(X)^2)
kernel <- stein_kernel(type = "gaussian_rbf", h = 1)
opt <- fmin_grid(lb = -1, ub = 1, n0 = 3, grow = FALSE)
stein_points(score, kernel, n_points = 2, d = 1, optimizer = opt,
             log_p = log_p)
```

---

stein_thinning *Compress existing samples with Stein thinning*

Description

Implements Algorithm 1 of Riabiz et al. (2022). Given an existing MCMC output X, the function returns m row indices that define a compressed empirical measure with small kernel Stein discrepancy with respect to the target score.

Usage

```
stein_thinning(
  X,
  S = NULL,
  m,
  score_function = NULL,
  pre = c("sclmed", "med", "smpcov"),
  kernel = "imq",
  pre_subsample = 1000L,
  pre_subsample_method = c("first", "even", "random"),
  verbose_rbf_warning = TRUE,
  ...
)
```

Arguments

`X`	Numeric matrix with samples in rows.
`S`	Optional score matrix with the same shape as `X`.
`m`	Positive integer number of points to select. The default `pre = "sclmed"` additionally requires `m > 1`.
`score_function`	Optional `function(X)` returning scores.
`pre`	Preconditioning rule: `"sclmed"`, `"med"`, or `"smpcov"`.
`kernel`	`"imq"` (default), `"gaussian_rbf"`, or a `SteinKernel` object, including objects made by `custom_stein_kernel()`. A supplied Gaussian RBF object must have a fixed positive bandwidth `h`; Stein thinning uses one fixed kernel throughout Algorithm 1.
`pre_subsample`	Maximum number of rows used for median-based scaling, or a vector of row indices to use directly.
`pre_subsample_method`	How to choose rows when `pre_subsample` is a scalar: `"first"`, `"even"`, or `"random"`.
`verbose_rbf_warning`	Warn when using RBF instead of the usual IMQ Stein thinning default.
`...`	Kernel parameters forwarded to `stein_kernel()`, such as `c`, `beta`, `h`, `sigma`, or `sigma2` for RBF.

Details

Stein thinning post-processes rows `X[1, ],...,X[n, ]` already produced by a sampler: it neither simulates a chain, moves particles, nor optimizes over continuous space. It returns indices for subsetting those rows. Score matrix `S` must match `X`, with row `S[i, ]` equal to $s_p(X[i,]) = \nabla_x \log p(x)|_{x=X[i,]}$. Alternatively, `score_function` is evaluated once to create `S`; the Stein kernel `k0` then uses these scores between rows of `X`.

Starting from an empty sequence, after selecting $s_1, \ldots, s_{j-1}$ the method minimizes

$$\frac{1}{2}k0(x_i, x_i) + \sum_{l=1}^{j-1} k0(x_{s_l}, x_i).$$

Although algebraically shaped like the marginal KSD decrease in greedy Stein Points, this selects indices from fixed support $\{x_1, \ldots, x_n\}$ rather than generating locations. For returned $idx = (s_1, \ldots, s_m)$, the diagnostic is

$$KSD_m^2 = \frac{1}{m^2} \sum_{a=1}^{m} \sum_{b=1}^{m} k0(x_{s_a}, x_{s_b}).$$

Each selected row's interactions are added to a running objective, equivalent to recomputing the sum but avoiding that work. If several rows have the same smallest value, the code takes the first one; the paper says any one may be used. Indices form a sequence, not necessarily a set: rows may repeat when this best lowers the objective.

The integer vector directly represents the paper's support points and multiplicities: use `X[idx, , drop = FALSE]` and `S[idx, , drop = FALSE]`. A repeated index unambiguously gives the same original support point repeated mass.

Thus this is compression, not another point generator. It shares the kernel layer with `stein_points()` and `sp_mcmc()`, but those methods propose new locations; Stein thinning keeps a short indexed subsequence of an existing, typically MCMC-produced support set.

Built-in kernels receive the paper's positive definite preconditioner `M` directly in

$$r(x, y) = (x - y)^T M(x - y).$$

If `med2` is the median squared Euclidean distance in the chosen preconditioning subsample, `"med"` uses

$$M = I/med2.$$

For zero median the paper recommends a positive exception such as length scale `ell = 1`; accordingly, the implementation warns and replaces `med2 = 0` by `1`. This remains positive definite but is no longer data-adaptive; zero commonly indicates repeated states or poor MCMC mixing and should be diagnosed, not treated as a successful estimate.

The paper's default scaled-median rule, `"sclmed"`, is

$$M = \log(m)I/med2,$$

equivalent to $\Gamma = \ell^2 I$, $\ell = med/\sqrt{\log(m)}$ and $M = \Gamma^{-1}$. This rule requires `m > 1`. It heuristically balances aggregate kernel interactions, but changes the kernel with `m`; Riabiz et al. therefore state that their preceding fixed-kernel theory does not cover `"sclmed"`.

For both median rules, unaffected consistency assumes a fixed `n0` MCMC states construct the preconditioner. `pre_subsample` controls this here; using all rows or letting its size grow lies outside that statement. `"smpcov"` uses the inverse empirical covariance of `X`.

Value

Length-`m` integer vector of one-based row indices in selection order. It is unsorted and may repeat; `X[idx, , drop = FALSE]` is the thinned sample, with repeats representing repeated support points in the empirical measure.

Examples

```
X <- matrix(rnorm(6), ncol = 1)
S <- -X
stein_thinning(X, S = S, m = 2, pre_subsample = 3)
```

`svgd` *Create an SVGD updater*

Description

Creates a small object for Stein Variational Gradient Descent, the particle update method of Liu and Wang (2016). SVGD moves a fixed set of particles so that their empirical distribution better approximates the target described by a score function.

Usage

```
svgd(kernel = stein_kernel(type = "gaussian_rbf"))
```

Arguments

kernel	SteinKernel object used by SVGD. Defaults to Gaussian RBF.

Details

SVGD represents its approximation by particles rather than a parametric density. Each update takes target scores $s_p(x) = \nabla_x \log p(x)$, combines them through a pairwise kernel, and adds a derivative term that separates nearby particles. Using scores removes the need for the target's normalizing constant.

The object stores a Stein kernel and two functions: `svgd_kernel()` computes its particle kernel matrix and derivative term, while `update()` calls `update_svgd()` to move particles. States and scores are matching `n x d` matrices, with particles in rows and coordinates in columns.

For the default Gaussian RBF kernel,

$$k(x, y) = \exp(-||x - y||^2/(2h^2)).$$

Without fixed `h`, each update takes the median `med` of all $n(n-1)/2$ current-particle Euclidean distances and sets

$$h = \frac{med}{\sqrt{2 \log n}}.$$

The paper's parameterization is $\exp(-||x - y||^2/h_{paper})$, with $h_{paper} = med^2/\log n$; the conversion is exact because this package uses denominator $2h^2$. The median is not subsampled and, as in the experiments, is recomputed each iteration, costing quadratic time and storage. For one particle the undefined rule uses `h = 1`, immaterial because RBF repulsion is zero. For multiple repeated particles, a zero median is not regularized; supply a fixed positive bandwidth.

The small list stores only the kernel and closures, not particles. Pass particles into each call and receive a new matrix, avoiding hidden mutable state and making identical `x0`, score, and settings directly comparable. Use `obj$update(...)` for the state-free convenience call or `update_svgd()` when supplying kernel or trace options explicitly.

Other Stein methods share only its score/kernel layer: KSD tests scalar `k0(x,y)`, FSSD builds finite-location $\tau(x)$, Stein thinning uses `k0` to choose existing rows, and Stein Points/SP-MCMC construct support points. SVGD transports every particle, using `eval_kernel()` and `grad_x_kernel()` but not full scalar `k0` from `stein_kernel_matrix()`. It keeps the target score and kernel choice while replacing a test sample or candidate optimizer by initial particles `x0` and an iteration count.

Value

A list with three entries:

- `kernel`: the stored `SteinKernel` object.
- `svgd_kernel(theta, kernel_obj = NULL)`: function returning the pairwise kernel matrix `Kxy` and summed derivative term `dxkxy` for particles `theta`.
- `update(...)`: function that calls `update_svgd()` using the stored kernel unless another kernel is supplied.

These closures keep the chosen kernel while particles remain explicit.

Examples

```
obj <- svgd()
x0 <- matrix(seq(-2, 2, length.out = 5), ncol = 1)
target_score <- function(theta) -theta
obj$update(x0, target_score, n_iter = 5, stepsize = 0.05)
```

`update_svgd` *Run SVGD particle updates*

Description

Applies the SVGD update repeatedly to a particle matrix. The target enters only through `lnprob`, which should return the score at each particle.

Usage

```
update_svgd(
  object,
  x0,
  lnprob,
  n_iter = 1000,
  stepsize = 0.001,
  kernel = NULL,
  alpha = 0.9,
  adj_grad = NULL,
  trace_iters = NULL
)
```

Arguments

`object`	SVGD object created by `svgd()`.
`x0`	Numeric matrix of initial particles. A vector is treated as one-dimensional particles.
`lnprob`	Function returning the target score at the current particles, despite the historical name. It should return $\nabla_x \log p(x)$, not `log p(x)`, with the same dimensions as `x0`.
`n_iter`	Nonnegative integer number of SVGD iterations.
`stepsize`	Positive finite SVGD step size. Custom `adj_grad` functions should not multiply by this value; it is applied by `update_svgd()`.
`kernel`	Optional `SteinKernel` overriding `object$kernel`.
`alpha`	Exponential-decay/momentum coefficient in `[0, 1)` used by the package's default RMSProp-style scaling. It is an implementation setting, not a parameter of the raw direction in Algorithm 1.

adj_grad	Optional function for custom scaling of the SVGD direction. It receives the arguments documented for custom_adjusted_gradient() and must return a matrix with the same dimensions as theta.
trace_iters	Optional integer vector of iterations to save. If NULL, return only the final particles.

Details

The name lnprob is kept for compatibility with common SVGD code, but the function must return the score $\nabla_x \log p(x)$, not the scalar log density. Rows of theta are particles and columns are coordinates, so lnprob(theta) should return an n x d matrix with the same shape.

At each iteration the particles theta are moved by

$$\theta \leftarrow \theta + \text{stepsize} \times \text{direction}.$$

Before the optional adaptive scaling, the raw SVGD direction for particle theta_i is the empirical version of the paper's optimal Stein direction:

$$g(\theta_i) = \frac{1}{n} \sum_{j=1}^{n} \left[k(\theta_j, \theta_i) s_p(\theta_j) + \nabla_{\theta_j} k(\theta_j, \theta_i) \right].$$

Here lnprob returns the score at particle j. The first term is kernel-smoothed attraction toward high density; the derivative term repels particles to prevent mode collapse. This raw direction is Algorithm 1, which leaves step size ϵ_l unspecified. The experiments mention AdaGrad, and the Bayesian neural-network experiment AdaGrad with momentum, but the paper gives no coordinatewise recurrence matching this package. Default scaling is therefore an experimental/implementation choice, not part of the analytic SVGD direction. In code, kxy %*% lnpgrad represents $\sum_j k(\theta_j, \theta_i) s_p(\theta_j)$, and dxkxy stores $\sum_j \nabla_{\theta_j} k(\theta_j, \theta_i)$ for destination i.

For raw direction g_l, the default keeps coordinatewise

$$H_1 = g_1^2, \qquad H_l = \alpha H_{l-1} + (1-\alpha) g_l^2$$

and uses $g_l / (\delta + \sqrt{H_l})$, $\delta = 10^{-6}$, before fixed outer stepsize. This resembles RMSProp or AdaGrad with momentum more than classical cumulative AdaGrad. For unscaled Algorithm 1, pass adj_grad = function(grad, ...) grad; other adj_grad functions replace only scaling. trace_iters saves particle copies after requested iterations.

Value

If trace_iters = NULL, an n x d numeric matrix containing the final particle locations after n_iter updates. If trace_iters is supplied, a list with:

- theta: final n x d particle matrix.
- trace: list of saved particle matrices; the element names are the iteration numbers requested in trace_iters.

The default final matrix supports ordinary transport use; traces are opt-in because particle matrices can be large. theta serves downstream algorithms, while selected trace states serve diagnostics and plots.

Examples

```
obj <- svgd()
x0 <- matrix(seq(-2, 2, length.out = 5), ncol = 1)
target_score <- function(theta) -theta
update_svgd(obj, x0, target_score, n_iter = 5, stepsize = 0.05)
```

Index

compute_fssd_null_pvalue, 4
compute_fssd_null_pvalue(), *3*, *7*, *17*, *19*
compute_fssd_unbiased_stat, 5
compute_fssd_unbiased_stat(), *3*, *7*, *17*, *19*
compute_tau, 7
compute_tau(), *3*, *5, 6*, *17*, *19*, *21*, *27, 28*, *59*, *63*
cross_kernel *(*kernel_generics*)*, 26
cross_kernel(), *4*, *58*
custom_adjusted_gradient, 8
custom_adjusted_gradient(), *71*
custom_stein_kernel, 9
custom_stein_kernel(), *4*, *27*, *31*, *33*, *37*, *39*, *60*, *63*, *67*

eval_kernel *(*kernel_generics*)*, 26
eval_kernel(), *4*, *7*, *58*, *69*

find_median_distance, 11
find_median_distance(), *32*, *37*, *59*
fmin_grid, 12
fmin_grid(), *3*, *15*, *65, 66*
fmin_mc, 13
fmin_mc(), *3*, *13*, *15*, *65, 66*
fmin_nm, 14
fmin_nm(), *3*, *13*, *65, 66*
fssd_opt_test, 16
fssd_opt_test(), *5, 6*
fssd_rand_test, 18
fssd_rand_test(), *5, 6*, *17*
fssd_test, 19
fssd_test(), *3*, *22*, *48*, *59*

get_score_evaluator, 22
get_score_evaluator(), *4*, *23*, *43*
gmm, 23
gmm(), *4*, *22*, *40*, *42, 43*, *45*, *47*
grad_theta_v_kernel *(*kernel_generics*)*, 26
grad_theta_v_kernel(), *4*, *17*
grad_x_kernel *(*kernel_generics*)*, 26
grad_x_kernel(), *4*, *7*, *58*, *69*
grw, 24
grw(), *26*, *46, 47*, *50*
grwmetrop, 25
grwmetrop(), *24, 25*, *40*, *47*

kernel_generics, 26
ksd_u_bootstrap, 28
ksd_u_bootstrap(), *3*, *30*, *32–35*
ksd_u_statistic, 30
ksd_u_statistic(), *3*, *29*, *32–36*
ksd_u_test, 31
ksd_u_test(), *3*, *21, 22*, *27*, *29, 30*, *33*, *38*, *48*, *59*, *62*
ksd_uq_matrix, 33
ksd_uq_matrix(), *3*, *30*, *32*, *39, 40*
ksd_v_bootstrap, 34
ksd_v_bootstrap(), *3*, *29*, *32*, *36*, *38–40*
ksd_v_statistic, 35
ksd_v_statistic(), *3*, *32*, *34, 35*, *38–40*
ksd_v_test, 36
ksd_v_test(), *3*, *10*, *21*, *32*, *35, 36*, *39*
ksd_vq_matrix, 39
ksd_vq_matrix(), *3*, *32, 33*, *35, 36*, *38*

likelihoodgmm, 40
likelihoodgmm(), *4*, *23*, *43, 44*, *46*

mala, 41
mala(), *25*, *40*, *50*

perturbgmm, 42
plotgmm, 43
posteriorgmm, 44
posteriorgmm(), *4*, *23*, *41*, *43, 44*, *48*

rgmm, 45
rgmm(), *4*, *23*, *43*
rwm, 46

rwm(), *50*

scorefunctiongmm, 47
scorefunctiongmm(), *4*, *22, 23*, *41*, *43–46*
sp_mcmc, 48
sp_mcmc(), *3, 4*, *22*, *25, 26*, *42*, *52–54*, *56*, *59–61*, *68*
sp_mcmc_criterion, 51
sp_mcmc_criterion(), *50*, *54*
sp_mcmc_eval_candidates, 52
sp_mcmc_eval_candidates(), *26*, *42*, *50*
sp_mcmc_select_start, 54
sp_mcmc_select_start(), *50*
sp_mcmc_state, 55
sp_mcmc_state(), *50*, *52*, *54*
stein_codescent, 56
stein_codescent(), *13, 14*, *61*, *66*
stein_kernel, 58
stein_kernel(), *4*, *10, 11*, *27*, *49*, *60, 61*, *65*, *67*
stein_kernel_imq_score, 59
stein_kernel_imq_score(), *10*, *59*, *61*, *63*, *65*
stein_kernel_inverse_log, 61
stein_kernel_inverse_log(), *59, 60*, *65*
stein_kernel_matrix, 62
stein_kernel_matrix(), *3*, *7*, *10*, *21*, *27, 28*, *31*, *33*, *37*, *39*, *50*, *58–61*, *64*, *69*
stein_points, 63
stein_points(), *3*, *13–15*, *22*, *27*, *48*, *50*, *57*, *59–61*, *68*
stein_thinning, 66
stein_thinning(), *3*, *22*, *59*
steinsampling *(*steinsampling-package*)*, 3
steinsampling-package, 3
svgd, 68
svgd(), *3, 4*, *70*

trace_mixed_kernel *(*kernel_generics*)*, 26
trace_mixed_kernel(), *4*, *7*, *58*

update_svgd, 70
update_svgd(), *4*, *8, 9*, *69*