# Package 'steinsampling'

August 28, 2026

**Title** Kernelized Stein Discrepancy for Goodness-of-Fit Tests and Stein Sampling

**Version** 0.1.1

**Date** 2026-08-27

**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 constructing example targets, simulation, density evaluation, and score callbacks.

**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, mvtnorm (>= 0.9-9994)

**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]

**Maintainer** Junhao Gao <jug049@ucsd.edu>

**Repository** CRAN

**Date/Publication** 2026-08-28 11:10:02 UTC

# Contents

---

`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()`, `fssd_statistic()`, and `fssd_null_pvalue()`. They expose intermediate matrices and reusable calibration components. All public GOF statistic primitives return the scale used for null comparison: `n U_n`, `n V_n`, or `n FSSDhat^2`. Their bootstrap or simulated null draws use exactly the matching scale; unscaled U-statistics are private implementation details used only where an optimization objective requires them.

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() instead moves every particle by deterministic transport.

stein_kernel() and custom_stein_kernel() create kernel objects. 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, densitygmm() evaluation, and 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_tau *Compute the FSSD feature matrix*

---

## Description

Builds the row-feature matrix used by fssd_statistic() and fssd_null_pvalue(). The rows of V represent the test locations $L = \{v_1, \ldots, v_J\}$.

## Usage

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

## Arguments

| | |
|---|---|
| X | Numeric $\tilde{n} \times d$ sample matrix. |
| scores | Numeric $\tilde{n} \times d$ score matrix for X. |
| V | Numeric J x d matrix representing L, one test location per row. |
| kernel_obj | Stein kernel object. |

## Details

For observation $x_i$, target score $s_p(x_i)$, and test location $v_j$, define

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

With `d = ncol(X)` and `J = nrow(V)`, the returned row is

$$\tau_i = \tau(x_i) = \frac{1}{\sqrt{dJ}} \operatorname{vec}\{\xi_p(x_i, v_1), \ldots, \xi_p(x_i, v_J)\} \in \mathbb{R}^{dJ}.$$

The block of `d` columns associated with row `j` of `V` contains $\xi_p(x_i, v_j)/\sqrt{dJ}$.

This function only constructs the features. For the fixed-location null calibration, `L` and the kernel, including its scale, must be fixed independently of the rows in `X`, or selected using separate training data.

## Value

Numeric $\tilde{n} \times dJ$ matrix. Row `i` is `tau(x_i)`; columns are ordered by test location and then coordinate.

## Examples

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

---

# cross_kernel *Score-derivative coupling of a base kernel*

---

## Description

Score-derivative coupling of a base kernel

## Usage

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

## Arguments

| | |
|---|---|
| `obj` | A `SteinKernel` object. |
| `X` | Numeric $n_X \times d$ matrix. |
| `grads, grads_Y` | Score matrices matching `X` and `Y`. |
| `Y` | Optional numeric $n_Y \times d$ matrix; `NULL` uses `X`. |
| `precon` | Optional preconditioner overriding `obj$precon`. |
| `...` | Ignored. |

### Details

Derived from grad_x using $\nabla_y k(x, y) = \nabla_x k(y, x)$, which holds for every symmetric base kernel, so no kernel supplies this term.

### Value

Numeric $n_X \times n_Y$ matrix.

---

## custom_stein_kernel *Create a Stein kernel from callbacks*

---

### Description

Create a Stein kernel from callbacks

### Usage

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

### Arguments

| | |
|---|---|
| eval_fn, grad_x_fn, trace_mixed_fn | Callbacks (X, Y, precon). |
| grad_theta_v_fn | Optional analytic FSSD-opt callback (X, vj, grads_X, g_block, obj) returning grad_vj and, when the kernel exposes a scale, grad_param. |
| scale_init | Optional positive starting squared scale. Requires set_scale_fn. |
| set_scale_fn | Optional (obj, scale2) returning obj with its scale-dependent callbacks rebuilt. |
| custom_grad_mode | "analytic" or location-only "numeric". |

### Details

`eval_fn`, `grad_x_fn`, and `trace_mixed_fn` supply the kernel value, its first derivative, and its mixed-derivative trace. Each receives `X`, `Y`, and `precon`; `precon` may be `NULL` and each callback decides whether to use it. For $X \in \mathbb{R}^{n_X \times d}$ and $Y \in \mathbb{R}^{n_Y \times d}$, `eval_fn` returns the $n_X \times n_Y$ matrix $k(X, Y)$, `grad_x_fn` the $n_X \times n_Y \times d$ array $\nabla_x k(X, Y)$, and `trace_mixed_fn` the $n_X \times n_Y$ matrix $\mathrm{tr}\{\nabla_x \nabla_y^\top k(X, Y)\}$. Kernel symmetry supplies $\nabla_y k$ by reversing the arguments of `grad_x_fn`.

KSD uses all three callbacks; FSSD uses only `eval_fn` and `grad_x_fn`. For FSSD-opt a custom kernel must provide `grad_theta_v_fn` or use `custom_grad_mode = "numeric"`. A custom kernel can update its scale when `scale_init` and `set_scale_fn` are supplied; that path requires `grad_theta_v_fn`, because numeric mode differentiates the test locations only.

### Value

A custom kernel specification.

### Examples

```
rbf <- stein_kernel("gaussian_rbf", h = 1)
custom_stein_kernel(
  eval_fn        = function(X, Y = NULL, precon = NULL) eval_kernel(rbf, X, Y),
  grad_x_fn      = function(X, Y = NULL, precon = NULL) grad_x_kernel(rbf, X, Y),
  trace_mixed_fn = function(X, Y = NULL, precon = NULL) trace_mixed_kernel(rbf, X, Y)
)
```

---

## `densitygmm` *Evaluate a Gaussian mixture density*

---

### Description

Returns the density of a `gmm()` object at each supplied observation.

### Usage

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

### Arguments

| | |
|---|---|
| `model` | Gaussian mixture model returned by `gmm()`. |
| `X` | Finite numeric vector or matrix. For a one-dimensional model, a vector represents multiple observations. For a multivariate model, a length-d vector represents one observation. Matrix rows are observations. |

### Details

For each observation $x_i$,

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

The function returns ordinary density values, not log densities.

## Value

A numeric vector containing one ordinary density value per interpreted observation in `X`.

## Examples

```
model <- gmm()
X <- rgmm(model)
p <- densitygmm(model = model, X = X)
```

---

`eval_kernel` *Evaluate a base kernel*

---

## Description

Evaluate a base kernel

## Usage

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

## Arguments

| | |
|---|---|
| `obj` | A `SteinKernel` object. |
| `X` | Numeric $n_X \times d$ matrix. |
| `Y` | Optional numeric $n_Y \times d$ matrix; `NULL` uses `X`. |
| `precon` | Optional preconditioner overriding `obj$precon`. |
| `...` | Ignored. |

## Value

Numeric $n_X \times n_Y$ matrix $B_{ij} = k(x_i, y_j)$.

## Examples

```
eval_kernel(stein_kernel("gaussian_rbf", h = 1), matrix(c(-1, 0, 1), ncol = 1))
```

`find_median_distance` *Median-heuristic squared scale*

## Description

Computes the squared median Euclidean distance between sample rows.

## Usage

```
find_median_distance(Z)
```

## Arguments

| | |
|---|---|
| `Z` | Finite numeric vector, matrix, or data frame with at least two observations. Matrix rows are observations; a vector is treated as an `n x 1` sample. |

## Details

For rows $z_1, \ldots, z_n$, the returned scale is

$$\rho_{\mathrm{med}} = \left\{ \underset{i<j}{\mathrm{median}} \|z_i - z_j\|_2 \right\}^2 .$$

The median is taken before squaring. All $n(n-1)/2$ pairs are used, so the calculation requires quadratic time and memory. Coordinates are not standardized. If the median distance is zero, the function warns and returns the floor value $10^{-5}$.

## Value

The positive scalar $\rho_{\mathrm{med}}$.

## 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.

## 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()`.

## 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()`.

Nelder-Mead is useful when the objective is smooth enough for local improvement to beat the finite candidate sets of `fmin_grid()` and `fmin_mc()`.

## 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` (the selected point), `d_min` (its score), `f_min` (the Stein objective value that updates the running KSD diagnostic), and `n_eval` (objective evaluations charged to this search).

## Examples

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

---

`fssd_null_pvalue` *Simulate the FSSD null distribution*

---

## Description

Performs the plug-in null calibration for a statistic returned by `fssd_statistic()`.

## Usage

```
fssd_null_pvalue(tau_matrix, statistic, n_simulations = 2000)
```

## Arguments

| | |
|---|---|
| `tau_matrix` | Numeric $\tilde{n} \times dJ$ feature matrix returned by `compute_tau()`. |
| `statistic` | Observed statistic $S_{\tilde{n}} = \tilde{n}\widehat{FSSD}^2$. |
| `n_simulations` | Number of null draws to simulate. |

## Details

The fixed-location null calibration requires the test locations `L` and the kernel, including its scale, to be fixed independently of the observations represented by `tau_matrix`, or selected using separate training data. This function cannot check that condition or correct same-sample selection.

Let $\tilde{n}$ be the number of rows of `tau_matrix`, and define

$$\widehat{\Sigma}_\tau = \frac{1}{\tilde{n}} \sum_{i=1}^{\tilde{n}} (\tau_i - \bar{\tau})(\tau_i - \bar{\tau})^T,$$

and let $\widehat{\lambda}_1, \ldots, \widehat{\lambda}_{dJ}$ be its eigenvalues. Numerically negative eigenvalues are replaced by zero. For $b = 1, \ldots, B$, the function simulates

$$S_{\tilde{n}}^{*(b)} = \sum_{q=1}^{dJ} \widehat{\lambda}_q \{(Z_q^{(b)})^2 - 1\}, \qquad Z_q^{(b)} \sim N(0,1),$$

and returns

$$\widehat{p} = \frac{1}{B} \sum_{b=1}^{B} \mathbf{1}\{S_{\tilde{n}}^{*(b)} \geq S_{\tilde{n}}\}.$$

The observed statistic and null draws are on the same $S_{\tilde{n}} = \tilde{n}\widehat{FSSD}^2$ scale. The returned proportion does not use an add-one correction. The simulated law is the large-$\tilde{n}$ limit of $S_{\tilde{n}}$, so the test holds its nominal level only asymptotically.

## Value

A list with four entries:

- `p_value`: uncorrected simulated right-tail proportion.
- `statistic`: the supplied $S_{\tilde{n}}$.
- `null_samples`: simulated draws on exactly the same scale as `statistic`.
- `eigenvalues`: nonnegative eigenvalues of $\widehat{\Sigma}_\tau$ used in the null draws.

## Examples

```
tau <- matrix(c(1, 2, 3), ncol = 1)
statistic <- fssd_statistic(tau)
fssd_null_pvalue(
  tau, statistic = statistic, n_simulations = 10
)
```

---

`fssd_opt_test` *FSSD test with optimized test locations*

---

## Description

Selects the test locations $L = \{v_1, \ldots, v_J\}$ on training rows. For the built-in Gaussian RBF and IMQ kernels, and for custom kernels created with `set_scale_fn`, it also selects the squared kernel scale $h^2$. The FSSD statistic and null calibration use disjoint held-out rows.

## Usage

```
fssd_opt_test(
  X,
  score_function,
  J = 5,
  n_simulations = 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 in L. |
| n_simulations | Number of null draws. |
| kernel | "gaussian_rbf", "imq", or a SteinKernel object. |
| scaling | Starting positive squared kernel scale. NULL initializes it from a median-based training grid. Supplying a value skips that grid but does not hold $h^2$ fixed when the kernel provides set_scale_fn. A scale stored in a SteinKernel object is the starting value and takes precedence. |
| train_ratio | Fraction of rows requested for FSSD-opt training. |
| gamma | Positive regularizer in the FSSD-opt criterion. |
| maxit | Maximum L-BFGS-B iterations for FSSD-opt. |
| locs_bounds_frac | Location-bound width in fitted standard deviations. |
| scale_lower, scale_upper | Bounds for the optimized squared kernel scale. |
| seed | Optional RNG seed for splitting rows, drawing starting locations, and simulating held-out null draws. |
| imq_beta | Finite IMQ exponent $\beta < 0$. |

## Details

The function first splits X into training and held-out rows. On the training rows it selects L and, when applicable, $h^2$ by maximizing

$$C(L, h^2) = \frac{\widehat{FSSD}^2_{train}}{\sqrt{\widehat{\mathrm{Var}}_{H_1} + \gamma}},$$

where

$$\widehat{\mathrm{Var}}_{H_1} = 4\bar{\tau}^T \widehat{\Sigma}_\tau \bar{\tau}, \qquad \widehat{\Sigma}_\tau = \frac{1}{n_{train}} \sum_{i=1}^{n_{train}} (\tau_i - \bar{\tau})(\tau_i - \bar{\tau})^T.$$

Here $\bar{\tau}$ is the mean feature vector on the training rows, and $\gamma > 0$ keeps the denominator away from zero.

Starting locations are drawn from a Gaussian fitted to the training rows. When $h^2$ is selected and scaling = NULL, it is initialized by a five-value grid around the training-row median scale. A supplied scale skips that grid and is used as the starting value. Bounded L-BFGS-B then refines L and $h^2$ jointly.

After optimization, L and the scale are fixed. compute_tau(), fssd_statistic(), and fssd_null_pvalue() are evaluated only on the held-out rows. Conditional on the training rows, the selected L and scale are fixed independently of the test rows, so the fixed-location null calibration applies.

## Value

An object of class `htest`. `statistic` is $S_{n_{test}} = n_{test}\widehat{FSSD}^2$, computed on the held-out rows; `p.value` is obtained from the held-out plug-in null calibration. `info` contains `L`, the realized split sizes, optimized scale, objective value, objective-function count, and optimizer convergence code.

## Examples

```
X <- matrix(rnorm(40), ncol = 1)
score_function <- function(x) -as.matrix(x)
fssd_opt_test(X, score_function, J = 1, scaling = 1,
              train_ratio = 0.5, n_simulations = 10, maxit = 2)
```

---

# fssd_rand_test *FSSD test with random test locations*

---

## Description

Fits a Gaussian distribution to `X`, draws the test locations $L = \{v_1, \ldots, v_J\}$, and uses all rows of `X` to compute the FSSD statistic and its plug-in null approximation.

## Usage

```
fssd_rand_test(
  X,
  score_function,
  J = 5,
  n_simulations = 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 in `L`. |
| `n_simulations` | Number of null draws. |
| `kernel` | `"gaussian_rbf"`, `"imq"`, or a `SteinKernel` object. |
| `scaling` | Final positive squared kernel scale. `NULL` computes the squared exact all-pair median distance from the same `X`. A scale stored in a supplied `SteinKernel` object takes precedence. |
| `seed` | Optional RNG seed for drawing `V` and simulating null draws. |
| `imq_beta` | Finite IMQ exponent $\beta < 0$. |

## Details

Let $\bar{x}$ and $\widehat{\Sigma}_X$ be the mean and fitted covariance of `X`. The locations are drawn independently as

$$v_j \sim N_d(\bar{x}, \widehat{\Sigma}_X), \qquad j = 1, \ldots, J.$$

If `scaling = NULL`, the squared all-pair median distance is also computed from `X`. The same rows of `X` are then passed to `compute_tau()`, `fssd_statistic()`, and `fssd_null_pvalue()`. Thus `X` is used both to choose `L` and any default scale and to compute the statistic and null approximation. These choices are not independent of the tested rows, as required by the fixed-location null calibration, so the reported p-value is heuristic.

## Value

An object of class `htest`. `statistic` is $S_n = n\widehat{FSSD}^2$; `p.value` is the same-sample plug-in p-value; and `info` contains the sampled `J x d` matrix `V` representing `L`.

## Examples

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

---

`fssd_statistic` *Compute the scaled FSSD test statistic*

---

## Description

Computes the scaled off-diagonal FSSD U-statistic from a feature matrix returned by `compute_tau()`.

## Usage

```
fssd_statistic(tau_matrix)
```

## Arguments

| | |
|---|---|
| `tau_matrix` | Numeric $\tilde{n} \times dJ$ feature matrix returned by `compute_tau()`. |

## Details

If `tau_i` is row `i` of `tau_matrix` and $\tilde{n}$ is its number of rows, the returned value is

$$S_{\tilde{n}} = \tilde{n}\widehat{FSSD}^2 = \frac{1}{\tilde{n}-1}\sum_{i \neq j} \tau_i^T \tau_j.$$

Diagonal terms are excluded because population FSSD uses two independent draws. Although population `FSSD^2` is nonnegative, its unbiased off-diagonal estimator and the returned $S_{\tilde{n}}$ can be negative in finite samples. This function does not compute a p-value.

## Value

One numeric value: $S_{\tilde{n}} = \tilde{n}\widehat{FSSD}^2$.

## Examples

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

---

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

---

## Description

Runs one of two Finite Set Stein Discrepancy tests. With `variant = "opt"`, the test locations $L = \{v_1, \ldots, v_J\}$ are selected on training rows. For the built-in Gaussian RBF and IMQ kernels, and for custom kernels created with `set_scale_fn`, the squared kernel scale $h^2$ is selected as well. The statistic and null calibration use held-out rows. With `variant = "rand"`, all rows of `X` are used both to select `L` and any default scale and to compute the statistic and null approximation, so its p-value is heuristic.

## Usage

```
fssd_test(
  X,
  score_function,
  variant = c("opt", "rand"),
  J = 5,
  n_simulations = 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` | `"opt"` (default) or `"rand"`. |

| | |
|---|---|
| J | Number of test locations in L. |
| n_simulations | Number of null draws. |
| kernel | "gaussian_rbf", "imq", or a SteinKernel object. |
| scaling | Positive squared kernel scale. It is final for FSSD-rand and an initial value for FSSD-opt. NULL uses a median-based value. A scale in a supplied SteinKernel takes precedence. |
| train_ratio | Fraction of rows requested for FSSD-opt training. |
| gamma | Positive regularizer in the FSSD-opt criterion. |
| maxit | Maximum L-BFGS-B iterations for FSSD-opt. |
| locs_bounds_frac | Location-bound width in fitted standard deviations. |
| scale_lower, scale_upper | Bounds for the optimized squared kernel scale. |
| seed | Optional RNG seed. |
| imq_beta | Finite IMQ exponent $\beta < 0$. |

## Details

variant = "opt" calls fssd_opt_test(), and variant = "rand" calls fssd_rand_test(). Both variants construct the FSSD features, compute

$$S_{\tilde{n}} = \tilde{n}\widehat{FSSD}^2,$$

and perform the plug-in null calibration implemented by fssd_null_pvalue(). Here $\tilde{n} = n_{test}$ for FSSD-opt and $\tilde{n} = n$ for FSSD-rand. The functions compute_tau(), fssd_statistic(), and fssd_null_pvalue() expose these calculations separately.

## Value

An htest object. statistic is $S_{\tilde{n}} = \tilde{n}\widehat{FSSD}^2$; p.value is its simulated right-tail p-value; info records L and variant-specific diagnostics.

## Examples

```
X <- matrix(rnorm(40), ncol = 1)
score_function <- function(x) -as.matrix(x)

fssd_test(X, score_function, J = 1, scaling = 1,
          train_ratio = 0.5, maxit = 2, n_simulations = 10)

fssd_test(X, score_function, variant = "rand", J = 1,
          scaling = 1, n_simulations = 10)
```

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

## Description

Returns a `function(X)` that evaluates the score of a `gmm()` object.

## Usage

```
get_score_evaluator(model)
```

## Arguments

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

## Details

Define the component responsibility

$$r_k(x) = \frac{w_k \phi(x; \mu_k, \Sigma_k)}{\sum_{\ell=1}^{K} w_\ell \phi(x; \mu_\ell, \Sigma_\ell)}.$$

The returned function evaluates

$$s_p(x) = \nabla_x \log p(x) = \sum_{k=1}^{K} r_k(x) \Sigma_k^{-1} (\mu_k - x).$$

## Value

A function with signature `function(X)`. For an `n x d` matrix, it returns the corresponding `n x d` score matrix. For vector input, it returns a vector: multiple one-dimensional scores when `model$d == 1`, or one length-`d` score when `model$d > 1`.

## 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

Constructs a Gaussian mixture distribution with nComp components.

## Usage

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

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

## 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. |
| x | A "gmm" object. |
| ... | Ignored. |

## Details

The model density is

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

where $w_k$, $\mu_k$, and $\Sigma_k$ are the weight, mean, and covariance of component $k$. Weights are normalized to sum to one.

By default, nComp = 5 and d = 1. If mu is omitted, centered random means are generated. If sigma is omitted, every component uses the identity covariance matrix.

## Value

An object of class "gmm" containing nComp; the d x nComp mean matrix mu; the d x d x nComp covariance array sigma; normalized weights; and dimension d.

## Examples

```
model <- gmm()

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)
```

---

# grad_theta_v_kernel *Gradient of the local FSSD-opt objective*

---

## Description

Gradient of the local FSSD-opt objective

## Usage

```
grad_theta_v_kernel(obj, X, vj, grads_X, g_block, precon = NULL, ...)
```

## Arguments

| | |
|---|---|
| obj | A SteinKernel object. |
| X | Numeric $n \times d$ matrix. |
| vj | Test location, a numeric vector of length $d$. |
| grads_X | Score matrix matching X. |
| g_block | Numeric $n \times d$ matrix with rows $g_i^\top$. |
| precon | Optional preconditioner overriding obj$precon. |
| ... | Ignored. |

## Details

With $\xi_p(x, v) = s_p(x)k(x, v) + \nabla_x k(x, v)$ and $\mathcal{L}_j(v_j) = \sum_i g_i^\top \xi_p(x_i, v_j)$, returns grad_vj equal to $\nabla_{v_j} \mathcal{L}_j$ and, when the kernel exposes a squared scale $\rho$, grad_param equal to $\partial \mathcal{L}_j / \partial \rho$.

## Value

List with grad_vj and grad_param.

## grad_x_kernel *Differentiate a base kernel in its first argument*

### Description

Differentiate a base kernel in its first argument

### Usage

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

### Arguments

| | |
|---|---|
| obj | A SteinKernel object. |
| X | Numeric $n_X \times d$ matrix. |
| Y | Optional numeric $n_Y \times d$ matrix; NULL uses X. |
| precon | Optional preconditioner overriding obj$precon. |
| ... | Ignored. |

### Value

Numeric $n_X \times n_Y \times d$ array $G_{ijr} = \partial k(x_i, y_j)/\partial x_r$.

## kernel_scale2 *Read or replace a kernel's squared scale*

### Description

Read or replace a kernel's squared scale

### Usage

```
kernel_scale2(obj, value = NULL)
```

### Arguments

| | |
|---|---|
| obj | A SteinKernel object. |
| value | Optional new squared scale. |

### Details

The squared scale is $h^2$ for Gaussian RBF and $c^2$ for IMQ. Returns NULL for a kernel that exposes no scale, and NA_real_ for one whose scale exists but is not set.

### Value

The squared scale, or the updated kernel when value is supplied.

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

## Description

Build the Stein-kernel matrix for KSD-U

## Usage

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

## Arguments

| | |
|---|---|
| `X` | Numeric vector or matrix containing $n$ observations. Rows are observations; a vector is treated as an $n \times 1$ matrix. |
| `score_function` | Function that accepts the checked $n \times d$ sample matrix and returns an $n \times d$ score matrix. |
| `scaling` | Positive squared scale passed to the kernel. For the built-in RBF and IMQ kernels this is, respectively, $h^2$ and $c^2$. `NULL` uses the squared median pairwise distance. |
| `kernel` | Kernel choice: `"gaussian_rbf"`, `"imq"`, or a `SteinKernel` object. |
| `imq_beta` | Finite exponent $\beta < 0$ of the built-in IMQ kernel. |

## Details

For observations $X_1, \ldots, X_n$, this function returns the full matrix

$$K_{ij} = k_{0,p}(X_i, X_j).$$

The diagonal is included here. `ksd_u_statistic()` and `ksd_u_bootstrap()` exclude it.

## Value

Numeric $n \times n$ matrix $K$, including its diagonal.

## Examples

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

`ksd_u_bootstrap` *Centered multinomial bootstrap for KSD-U*

## Description

Centered multinomial bootstrap for KSD-U

## Usage

```
ksd_u_bootstrap(
  K0,
  nboot = 1000,
  W_mat = NULL,
  boot_method = "multinomial_centered"
)
```

## Arguments

| | |
|---|---|
| `K0` | Numeric $n \times n$ Stein-kernel matrix, such as the output of `ksd_uq_matrix()`. |
| `nboot` | Positive number of bootstrap draws. Ignored when `W_mat` is supplied. |
| `W_mat` | Optional numeric $n \times B$ matrix of centered weights. Row $i$ corresponds to observation $i$; column $b$ specifies bootstrap draw $b$. If `NULL`, the weights above are generated. |
| `boot_method` | Bootstrap method. Currently only `"multinomial_centered"` is supported. |

## Details

For bootstrap draw $b$, let

$$w_i^{(b)} = \frac{N_i^{(b)}}{n} - \frac{1}{n}, \qquad (N_1^{(b)}, \ldots, N_n^{(b)}) \sim \text{Multinomial}\left(n; \frac{1}{n}, \ldots, \frac{1}{n}\right).$$

The returned draw is

$$T_n^{*(b)} = n \sum_{i \neq j} w_i^{(b)} w_j^{(b)} K_{ij}.$$

The diagonal is excluded. The draws are on the same scale as `ksd_u_statistic()`. This function does not compute a p-value.

## Value

Numeric vector containing $T_n^{*(1)}, \ldots, T_n^{*(B)}$.

## 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*

### Description

Compute the KSD-U statistic

### Usage

```
ksd_u_statistic(K0)
```

### Arguments

| | |
|---|---|
| K0 | Numeric $n \times n$ Stein-kernel matrix, such as the output of ksd_uq_matrix(). It must contain at least two rows and columns. |

### Details

For $K_{ij} = k_{0,p}(X_i, X_j)$, this function returns

$$T_n = nU_n = \frac{1}{n-1} \sum_{i \neq j} K_{ij}.$$

The diagonal is excluded. The result can be negative. This function does not compute bootstrap draws or a p-value.

### Value

One numeric value, $T_n = nU_n$.

### 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-U goodness-of-fit test for independent observations*

### Description

Tests whether independent observations come from a target distribution specified by its score function. The target density need not be normalized.

## 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 containing $n$ observations. Rows are observations and columns are coordinates; a vector is treated as an $n \times 1$ matrix. |
| `score_function` | Function that accepts the checked $n \times d$ sample matrix and returns the $n \times d$ matrix with row $s_p(X_i)^\top$. |
| `boot_method` | Bootstrap method. Currently only `"multinomial_centered"` is supported. |
| `scaling` | Positive squared scale passed to the kernel. For the built-in RBF and IMQ kernels this is, respectively, $h^2$ and $c^2$. `NULL` uses the squared median pairwise distance. |
| `nboot` | Positive number of bootstrap draws $B$. |
| `kernel` | Kernel choice: `"gaussian_rbf"`, `"imq"`, or a `SteinKernel` object. |
| `return_raw_boot` | Logical. If `TRUE`, include the $B$ bootstrap draws in the result. |
| `block_size` | Positive number of rows per block. `NULL` uses `min(1024, n)` when `n` > `block_threshold` and `n` otherwise. Blocking changes memory use, not the test definition. |
| `block_threshold` | Positive sample-size threshold above which blockwise computation is used. |
| `imq_beta` | Finite exponent $\beta < 0$ of the built-in IMQ kernel. |

## Details

Let $s_p(x) = \nabla_x \log p(x)$ and

$$K_{ij} = k_{0,p}(X_i, X_j),$$

where $k_{0,p}$ is the Stein kernel defined in `stein_kernel_matrix()`. The test uses

$$U_n = \frac{1}{n(n-1)} \sum_{i \neq j} K_{ij}, \qquad T_n = nU_n.$$

The diagonal is excluded. Thus $U_n$ is unbiased for the population squared KSD, but its finite-sample value can be negative.

Null calibration uses the centered multinomial bootstrap implemented by ksd_u_bootstrap(). If $T_n^{*(1)}, \ldots, T_n^{*(B)}$ are the bootstrap draws, the reported right-tail p-value is

$$\widehat{p} = \frac{1 + \sum_{b=1}^{B} \mathbf{1}\{T_n^{*(b)} \geq T_n\}}{B + 1}.$$

## Value

An htest object. statistic is $T_n = nU_n$; p.value is the bootstrap p-value above. If return_raw_boot = TRUE, bootstrap_samples contains $T_n^{*(1)}, \ldots, T_n^{*(B)}$.

## Examples

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

---

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

---

## Description

Build the Stein-kernel matrix for KSD-V

## Usage

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

## Arguments

| | |
|---|---|
| X | Numeric vector or matrix containing $n$ observations. Rows are observations; a vector is treated as an $n \times 1$ matrix. |
| score_function | Function that accepts the checked $n \times d$ sample matrix and returns an $n \times d$ score matrix. |
| scaling | Positive squared scale passed to the kernel. For the built-in RBF and IMQ kernels this is, respectively, $h^2$ and $c^2$. NULL uses the squared median pairwise distance. |
| kernel | Kernel choice: "gaussian_rbf", "imq", or a SteinKernel object. |
| imq_beta | Finite exponent $\beta < 0$ of the built-in IMQ kernel. |

## Details

For observations $X_1, \ldots, X_n$, this function returns

$$K_{ij} = k_{0,p}(X_i, X_j).$$

The diagonal is included and is retained by `ksd_v_statistic()` and `ksd_v_bootstrap()`.

## Value

Numeric $n \times n$ matrix $K$, including its diagonal.

## Examples

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

---

`ksd_v_bootstrap` *Wild bootstrap for KSD-V*

---

## Description

Wild bootstrap for KSD-V

## Usage

```
ksd_v_bootstrap(
  K0,
  nboot = 1000,
  W_mat = NULL,
  boot_method = c("rademacher", "markov"),
  change_prob = NULL
)
```

## Arguments

| | |
|---|---|
| `K0` | Numeric $n \times n$ Stein-kernel matrix, such as the output of `ksd_vq_matrix()`. |
| `nboot` | Positive number of bootstrap draws. Ignored when `W_mat` is supplied. |
| `W_mat` | Optional numeric $n \times B$ sign matrix. Row $i$ corresponds to observation $i$; column $b$ specifies bootstrap draw $b$. If `NULL`, signs are generated from `boot_method`. |
| `boot_method` | Sign process used when `W_mat = NULL`: `"rademacher"` or `"markov"`. |
| `change_prob` | Markov sign-change probability $a_n$. It must be supplied and lie strictly between 0 and 1 when `boot_method = "markov"`. |

## Details

For sign vector $W^{(b)}$, the returned draw is

$$nV_n^{*(b)} = \frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{n} W_i^{(b)} W_j^{(b)} K_{ij}.$$

Rademacher signs are independent and take values $-1$ and $1$ with equal probability. Markov signs satisfy

$$W_1^{(b)} = 1, \qquad W_t^{(b)} = \begin{cases} -W_{t-1}^{(b)}, & \text{with probability } a_n, \\ W_{t-1}^{(b)}, & \text{with probability } 1 - a_n. \end{cases}$$

Here $a_n$ is `change_prob`. All matrix entries, including the diagonal, are used. The draws are on the same scale as `ksd_v_statistic()`. This function does not compute a p-value.

## Value

Numeric vector containing $nV_n^{*(1)}, \ldots, nV_n^{*(B)}$.

## 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*

---

## Description

Compute the KSD-V statistic

## Usage

```
ksd_v_statistic(K0)
```

## Arguments

| | |
|---|---|
| `K0` | Numeric $n \times n$ Stein-kernel matrix, such as the output of `ksd_vq_matrix()`. |

## Details

For $K_{ij} = k_{0,p}(X_i, X_j)$, this function returns

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

All ordered pairs, including $i = j$, are included. This function does not compute bootstrap draws or a p-value.

## Value

One numeric value, $nV_n$.

## 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-V goodness-of-fit test with wild-bootstrap calibration*

---

## Description

Tests observations against a target distribution specified by its score function. Use Rademacher calibration for independent observations and Markov calibration for ordered dependent observations.

## 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 containing $n$ observations. Rows are observations and columns are coordinates; a vector is treated as an $n \times 1$ matrix. For Markov calibration, rows must follow the dependence order. |
| `score_function` | Function that accepts the checked $n \times d$ sample matrix and returns the $n \times d$ matrix with row $s_p(X_i)^\top$. |
| `boot_method` | Calibration method: `"rademacher"` for independent observations or `"markov"` for ordered dependent observations. |
| `scaling` | Positive squared scale passed to the kernel. For the built-in RBF and IMQ kernels this is, respectively, $h^2$ and $c^2$. `NULL` uses the squared median pairwise distance. |
| `nboot` | Positive number of bootstrap draws $B$. |

`change_prob` Sign-change probability $a_n$ for Markov calibration. It must be supplied and lie strictly between 0 and 1. It is ignored for Rademacher calibration.

`kernel` Kernel choice: `"gaussian_rbf"`, `"imq"`, or a `SteinKernel` object.

`return_raw_boot` Logical. If `TRUE`, include the $B$ bootstrap draws in the result.

`block_size` Positive number of rows per block. `NULL` uses `min(1024, n)` when `n > block_threshold` and `n` otherwise. Blocking changes memory use, not the test definition.

`block_threshold` Positive sample-size threshold above which blockwise computation is used.

`imq_beta` Finite exponent $\beta < 0$ of the built-in IMQ kernel.

## Details

Let $s_p(x) = \nabla_x \log p(x)$ and

$$K_{ij} = k_{0,p}(X_i, X_j),$$

where $k_{0,p}$ is defined in `stein_kernel_matrix()`. The test uses

$$V_n = \frac{1}{n^2} \sum_{i=1}^{n} \sum_{j=1}^{n} K_{ij}, \qquad nV_n = \frac{1}{n} \sum_{i=1}^{n} \sum_{j=1}^{n} K_{ij}.$$

The diagonal is retained. For a positive-definite base kernel, $V_n$ is nonnegative but upward biased for the population squared KSD.

`boot_method = "rademacher"` uses independent signs and is intended for independent observations. `boot_method = "markov"` uses correlated signs and requires rows of `X` to be in dependence order. The latter calibration requires the dependence, moment, and kernel conditions of the wild-bootstrap result; this function cannot verify them. Asymptotically, the sign-change probabilities $a_n$ must satisfy $a_n \to 0$ and $na_n \to \infty$.

If $nV_n^{*(1)}, \ldots, nV_n^{*(B)}$ are the bootstrap draws, the reported right-tail p-value is

$$\widehat{p} = \frac{1 + \sum_{b=1}^{B} \mathbf{1}\{nV_n^{*(b)} \geq nV_n\}}{B + 1}.$$

## Value

An `htest` object. `statistic` is $nV_n$; `p.value` is the bootstrap p-value above. If `return_raw_boot = TRUE`, `bootstrap_samples` contains $nV_n^{*(1)}, \ldots, nV_n^{*(B)}$.

## Examples

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

`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. The proposal noise has covariance `h * Sigma`, so `h` is the square of the step size used by the authors' `mala.m`. |
| `Sigma` | Symmetric positive-definite proposal preconditioner, used for both the drift and the noise. If `NULL`, the identity matrix is used. |
| `m_iter` | Number of returned chain rows, including the initial state in row 1. The function therefore makes `m_iter - 1` Markov transitions. |

## Details

With the row-vector convention used by the code, the proposal from the current state `x` is

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

so `Sigma` preconditions the drift and the noise alike and 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 $y = x+(h/2)\nabla \log p(x)+\sqrt{h}z$. Appendix A.5 of the SP-MCMC paper prints $Sigma^{-1}$ in the drift while keeping $N(0, Sigma)$ noise; the authors' released `mala.m` uses the same matrix in both places, which is the standard preconditioned Langevin discretization, and this implementation follows the released code. Taking $Sigma$ to be an approximation of the target covariance therefore scales the drift correctly.

Two conventions are worth stating because they differ from the released MATLAB code and from the Stein kernel. First, `mala.m` parameterizes the chain by the square root of this step size, so its `h` corresponds to `sqrt(h)` here. Second, `Sigma` is a proposal preconditioner and is unrelated to the `precon` matrix of an IMQ Stein kernel: SP-MCMC Equation 6 uses $Lambda^{-1}$ inside `k0`, so a run that preconditions both with the same target covariance passes `Sigma = S` here and `precon = solve(S)` to `stein_kernel()`.

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)
```

---

# print.SteinKernel *Print a Stein kernel*

---

## Description

Print a Stein kernel

## Usage

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

## Arguments

| | |
|---|---|
| x | A SteinKernel object. |
| ... | Ignored. |

## Value

x, invisibly.

## Examples

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

## print.stein_points *Print a Stein point set*

### Description

Prints the point matrix size, evaluation count, and available diagnostic.

### Usage

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

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

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

### Arguments

| | |
|---|---|
| x | An object returned by stein_points(), stein_codescent(), or svgd(). |
| ... | Ignored. |

### Value

x, invisibly.

### 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)
stein_points(score, kernel, n_points = 3, d = 1, optimizer = opt, x_init = 0)
```

## rgmm *Sample from a Gaussian mixture model*

### Description

Draws independent observations from a gmm() object. For each observation, a component is sampled according to model$weights, followed by a Gaussian draw with that component's mean and covariance.

### Usage

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

### Arguments

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

### Value

For model$d == 1, a numeric vector of length n. For model$d > 1, an n x d numeric matrix with one observation per row.

### Examples

```
model <- gmm()
X <- rgmm(model)
```

---

## rwm *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

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

### Arguments

| | |
|---|---|
| log_p | Function returning log density values. |
| x0 | Initial state vector. |
| h | Positive step-size multiplier. The proposal covariance is h * Sigma. |
| Sigma | Symmetric positive-definite proposal covariance scale. If NULL, the identity matrix is used. |
| 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

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

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().

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 the same fields as `mala()`: `X` (chain states), `D` (explicitly `NULL` because RWM does not evaluate scores), `log_p` (log density values), `accept` (0/1 acceptance indicators), `n_eval`, and `counts`. The explicit `D = NULL` tells `sp_mcmc()` to score the distinct candidate states after the path is generated.

## Examples

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

---

# sp_mcmc *Select Stein Points from short Markov chains*

---

## Description

Builds a point set sequentially. At each step, a short Markov chain produces the candidates, and the candidate minimizing the greedy Stein objective is selected.

## Usage

```
sp_mcmc(
  score_function,
  log_p,
  kernel,
  n_points,
  d,
  mcmc = c("rwm", "mala"),
  criterion = c("last", "rand", "infl"),
  m_seq,
  h,
  Sigma = NULL,
  x_init,
  seed = NULL,
  transition_fn = NULL,
  proposal_fn = NULL,
  criterion_args = list()
)
```

## Arguments

| | |
|---|---|
| `score_function` | Function taking an `n x d` matrix and returning the corresponding `n x d` matrix of target scores. |
| `log_p` | Function taking an `n x d` matrix and returning `n` log-density values. |
| `kernel` | Kernel used to construct $k_{0,p}$. It may be a built-in kernel object, a compatible kernel function, or a list with `k0_matrix`. A Gaussian RBF kernel must use a fixed positive bandwidth. |

| | |
|---|---|
| n_points | Positive total number of points, including x_init. |
| d | Positive state dimension. |
| mcmc | MCMC transition: "rwm" for Gaussian random-walk Metropolis or "mala" for MALA. |
| criterion | Rule for choosing where the next MCMC chain starts: "last", "rand", "infl", or a custom function. |
| m_seq | Number of path states before duplicate removal. Supply one value for all steps, one value per selected point, or one value for each step after the first point. |
| h | Step-size multiplier used by the built-in MCMC transitions. Both transitions give the proposal noise covariance h * Sigma. |
| Sigma | Symmetric positive-definite proposal preconditioner shared by the drift and the noise of mala(), and the proposal covariance scale of rwm(). If NULL, the identity matrix is used. This is the target-scale matrix itself, not its inverse; the precon matrix of an IMQ stein_kernel() is the inverse, so preconditioning both the transition and k0 with the same covariance S means Sigma = S and precon = solve(S). |
| x_init | Finite numeric vector of length d giving the 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 $\nu_j$. It must return a list with exactly that many rows in X, the supplied current state in row 1, D set to a matching score matrix or explicitly to NULL, and counts containing log_p, score, and their sum total. The remaining $\nu_j - 1$ 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. |
| criterion_args | Optional list of extra arguments passed to a custom criterion function. |

## Details

Let $k_{0,p}$ be the Stein kernel formed from kernel and $s_p(x) = \nabla_x \log p(x)$. Given selected points $x_1, \ldots, x_{j-1}$, define

$$Q_j(x) = \frac{1}{2} k_{0,p}(x, x) + \sum_{i=1}^{j-1} k_{0,p}(x_i, x).$$

At step $j$, the function generates a Markov-chain path $Y_{j,1}, \ldots, Y_{j,\nu_j}$, removes duplicate states, and selects

$$x_j \in \operatorname*{argmin}_{x \in \mathcal{C}_j} Q_j(x),$$

where $\mathcal{C}_j$ is the remaining candidate set. Thus stein_points() and sp_mcmc() use the same greedy objective; they differ in how candidates are obtained.

The first path state is chosen from the previously selected points. criterion = "last" uses the latest point, "rand" samples a selected point, and "infl" uses the point whose removal gives the largest remaining-set discrepancy. A path of length $\nu_j$ makes $\nu_j - 1$ transitions. Therefore m_seq = 1 supplies only the starting state and performs no search.

Built-in paths use random-walk Metropolis (`mcmc = "rwm"`) or MALA (`mcmc = "mala"`). `transition_fn`, `proposal_fn`, and a functional `criterion` provide optional extension points. A custom transition must return `X`, containing exactly the requested path rows; `D`, containing matching scores or `NULL`; and `counts`, containing nonnegative `log_p`, `score`, and `total` entries with `total = log_p + score`.

The returned discrepancy after step $j$ is

$$\mathrm{KSD}_j = \left\{ \frac{1}{j^2} \sum_{a=1}^{j} \sum_{b=1}^{j} k_{0,p}(x_a, x_b) \right\}^{1/2}.$$

## Value

An object of class `"sp_mcmc"` with:

- `X` and `D`: `n_points x d` matrices of selected points and their scores.
- `ksd`: running discrepancy defined above.
- `n_eval`, `cum_n_eval`, and `counts`: per-step, cumulative, and component evaluation counts.
- `selected_index`: row used to initialize each short path.
- `chain_d2_max`, `chain_d2_selected`, `chain_d2_last`, and `accept_rate`: short-path movement and acceptance summaries.
- `kernel`, `mcmc`, `criterion`, `m_seq`, `h`, and `Sigma`: settings used by the run.
- `call`: matched function call.

## 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, h = 0.1,
        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` | Start-rule name, custom 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"` starts at a most influential point, defined in the paper as one whose removal raises the KSD of the remaining points the most. Removing point a lowers the double sum $\sum_{b,c} k0(x_b, x_c)$ by

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

and the remaining points share one divisor, so maximizing their KSD is the same as taking the smallest `I_a`, which is what the rule computes. `I_a` can be negative, and the most negative one marks the point contributing most to the current KSD. A custom function can implement another rule while still using the same SP-MCMC main loop.

The result pairs a `select(state)` function with a `label` that `sp_mcmc()` copies into its output, so a run records whether LAST, RAND, INFL, or a custom rule produced it.

### Value

A list with `label`, stored in the final `sp_mcmc()` output, and `select`, a function taking an `sp_mcmc_state()` object and returning one row index into `state$X`.

### 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. objective_values 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.

## Value

A list with:

- objective_values: doubled greedy objective value for each candidate row. The row with the smallest value is the point sp_mcmc() appends.
- scores: 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_evaluations: 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.

Pairing it with sp_mcmc_state() and sp_mcmc_criterion() checks that a custom rule returns a legal index before the full algorithm is run. Inside sp_mcmc() the returned index is also recorded in selected_index.

### Value

One integer: the one-based row index in state$X that should initialize the next short candidate chain.

### 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` | Start-rule label stored in the state. |

## 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`.

`sp_mcmc()` builds this object internally before every short chain; building one directly lets a custom rule 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 a snapshot handed to a start-point criterion, not a result object.

## 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 optimize one existing point under 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 can improve the locations without increasing the point count.

The paper defines each coordinate update by a global `argmin`, for which the current row is itself feasible and the KSD therefore cannot increase. The numerical grid, Monte Carlo, and Nelder–Mead searches are approximate and need not return a better point. After each search, this implementation evaluates the old and proposed rows under the same coordinate objective and keeps the proposal only when its value is no larger. This incumbent fallback restores the non-increase property without adding a score evaluation because both scores are already cached. When `X0` contains a single row the interaction sum is empty and the objective reduces to $k0(x, x)$, matching the special branch in the authors' MATLAB code.

The optimizer interface is the one used by `stein_points()`, so a grid, Monte Carlo, or Nelder-Mead search serves both construction and refinement.

## Value

An object of class "stein_codescent" with:

- X: refined point matrix with the same dimensions as X0.
- D: target scores at the refined points.
- objective: numeric vector; objective[it] is the retained value of the coordinate objective at update it.
- n_eval: evaluation counts for each coordinate-descent update.
- cum_n_eval: cumulative evaluation counts.
- kernel: kernel used to define the Stein objective.

The objective values are not KSDs. They differ from $n^2 KSD^2$ by the kernel sum over the rows held fixed, which changes at every update, so they are comparable within an update but not across updates.

There is no ksd trajectory, because each update replaces an existing row rather than extending a prefix. For 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

Create a built-in Stein kernel

## Usage

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

### Arguments

| | |
|---|---|
| type | "gaussian_rbf" or "imq". |
| sigma, h | Gaussian RBF bandwidth $h > 0$. Supply at most one. If both are NULL, the kernel has no fixed bandwidth. |
| beta | Finite IMQ exponent $\beta < 0$. |
| c | Positive IMQ length scale $c$. |
| precon | Optional symmetric positive-definite $M$. |

### Details

With $r_M(x, y) = (x - y)^\top M(x - y)$ and M = precon (identity when precon = NULL), the base kernels are

$$k_{\mathrm{RBF}}(x, y) = \exp\{-r_M(x, y)/(2h^2)\}, \qquad k_{\mathrm{IMQ}}(x, y) = \{c^2 + r_M(x, y)\}^\beta.$$

### Value

A SteinKernel object.

### Examples

```
stein_kernel("gaussian_rbf", h = 1)
stein_kernel("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()`.

`stein_points()` still takes `score_function` for the greedy objective; `hess_log_p` is needed in addition because differentiating $k(s_p(x), s_p(y))$ with respect to point locations uses derivatives of the score field. That requirement is also why this kernel cannot be built with `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`. 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.

The squared distance is the ordinary Euclidean one between sample rows. Unlike the IMQ constructor in `stein_kernel()`, this helper exposes no preconditioning matrix; it reproduces the position-distance inverse-log kernel of the Stein Points experiments. The result is an ordinary `SteinKernel`, so `stein_points()` and `sp_mcmc()` need the same inputs as for any other kernel: points, scores, and the kernel object.

## Value

A `SteinKernel` object for the inverse-log kernel.

## Examples

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

---

`stein_kernel_matrix` *Assemble the pairwise Stein-kernel matrix*

---

## Description

Assemble the pairwise Stein-kernel matrix

## Usage

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

## Arguments

| | |
|---|---|
| `kernel` | A `SteinKernel` object. |
| `X`, `Y` | Numeric point matrices; `Y = NULL` uses `X`. |
| `grads`, `grads_Y` | Score matrices matching `X` and `Y`. |
| `...` | Optional `precon` overriding `kernel$precon`. |

## Details

With $s_p(x) = \nabla_x \log p(x)$,

$$k_{0,p}(x, y) = s_p(x)^\top s_p(y) k(x, y) + s_p(x)^\top \nabla_y k(x, y) + s_p(y)^\top \nabla_x k(x, y) + \mathrm{tr}\{\nabla_x \nabla_y^\top k(x, y)\}.$$

### Value

Numeric $n_X \times n_Y$ matrix $K_{ij} = k_{0,p}(x_i, y_j)$.

### Examples

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

---

## stein_points *Select points by Stein discrepancy minimization*

---

### Description

Builds a point set sequentially. At each step, `optimizer` searches the continuous state space for the next point under a greedy or herding Stein objective.

### 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 taking an `n x d` matrix and returning the corresponding `n x d` matrix of target scores. |
| `kernel` | Kernel used to construct $k_{0,p}$. It may be a built-in kernel object, a compatible custom object, a function `kernel(X, S_X, Y, S_Y)`, or a list with `k0_matrix`. A Gaussian RBF kernel must use a fixed positive bandwidth. |
| `n_points` | Positive number of points to select. |
| `d` | Positive state dimension. |
| `optimizer` | Function taking `(objective, X_curr, t)` and returning `x_min`, its score `d_min`, the minimized objective `f_min`, and the evaluation count `n_eval`. |
| `method` | Point-selection rule: `"greedy"` or `"herding"`. |
| `log_p` | Optional log density used to choose the first point. |

| | |
|---|---|
| x_init | Optional finite numeric vector of length d giving the first point. If supplied, log_p is not used. |
| c2 | Positive truncation constant when truncation != "none". |
| truncation | Candidate-filtering rule. Use "none" for no filtering. |
| seed | Optional local RNG seed. |

## Details

Let $k_{0,p}$ be the Stein kernel formed from kernel and $s_p(x) = \nabla_x \log p(x)$. Given selected points $x_1, \ldots, x_{j-1}$, the greedy rule chooses

$$x_j \in \operatorname*{argmin}_{x \in \mathbb{R}^d} Q_j(x),$$

where

$$Q_j(x) = \frac{1}{2} k_{0,p}(x, x) + \sum_{i=1}^{j-1} k_{0,p}(x_i, x).$$

With method = "herding", the self-interaction is omitted and the objective is

$$Q_j^{\text{herd}}(x) = \sum_{i=1}^{j-1} k_{0,p}(x_i, x).$$

optimizer performs the numerical search for each $x_j$. The supplied constructors fmin_grid(), fmin_mc(), and fmin_nm() use grid, Monte Carlo, and Nelder–Mead search, respectively. The first point is x_init; if x_init is NULL, optimizer instead maximizes log_p.

The returned discrepancy after step $j$ is

$$\text{KSD}_j = \left\{ \frac{1}{j^2} \sum_{a=1}^{j} \sum_{b=1}^{j} k_{0,p}(x_a, x_b) \right\}^{1/2}.$$

When truncation != "none", the truncation rule filters candidates from steps 2 through n_points; it is not applied to the first point.

## Value

An object of class "stein_points" with:

- X and D: n_points x d matrices of selected points and their scores.
- ksd: running discrepancy defined above.
- n_eval and cum_n_eval: per-step and cumulative evaluation counts.
- method, kernel, truncation, and c2: settings used by the run.
- call: matched function call.

## 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 *Select existing samples by Stein thinning*

---

## Description

Selects an ordered sequence of row indices from an existing sample using the greedy Stein-thinning criterion of Riabiz et al. (2022). The function does not simulate or move sample points.

## 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 | Finite numeric vector or matrix containing $n$ existing samples. Rows are samples and columns are coordinates; a vector is treated as an $n \times 1$ matrix. |
| S | Optional finite numeric $n \times d$ score matrix with row $s_p(z_i)^\top$. Supply S or score_function. |
| m | Positive integer number of indices to select. Repetition is allowed, so m may exceed $n$. The default "sclmed" rule requires m > 1. |
| score_function | Optional function that accepts X and returns an $n \times d$ score matrix. When supplied, its result is used instead of S. |
| pre | Preconditioning rule: "sclmed", "med", or "smpcov". |
| kernel | Either "imq", "gaussian_rbf", or a SteinKernel object. The default is "imq". A supplied Gaussian RBF object must have a fixed positive bandwidth. |

`pre_subsample` For `"med"` and `"sclmed"`, either a positive integer giving the maximum number of rows used to estimate $\rho$, `Inf` for all rows, or an explicit vector of row indices. Ignored for `"smpcov"`.

`pre_subsample_method`
Row-selection method used when `pre_subsample` is scalar: `"first"`, `"even"`, or `"random"`.

`verbose_rbf_warning`
Logical; when `TRUE` (the default), supplying a Gaussian RBF kernel warns that Riabiz et al. (2022) use IMQ as the Stein thinning default. Set `FALSE` to silence it.

`...` Kernel parameters used when `kernel` is a character string: `c` and `beta` for IMQ, or `h` and `sigma` for Gaussian RBF. The compatibility argument `sigma2` is also accepted for RBF when neither `h` nor `sigma` is supplied.

## Details

Let rows of `X` be $z_1^\top, \ldots, z_n^\top$ and

$$K_{ab} = k_{0,p}(z_a, z_b).$$

Scores may be supplied through `S` or computed once using `score_function`. If the first $j-1$ selected indices are $\pi(1), \ldots, \pi(j-1)$, the next index is

$$\pi(j) \in \underset{i \in \{1,\ldots,n\}}{\arg\min} \left\{ K_{ii} + 2 \sum_{\ell=1}^{j-1} K_{\pi(\ell),i} \right\}.$$

The implementation stores one half of this objective,

$$\frac{1}{2} K_{ii} + \sum_{\ell=1}^{j-1} K_{\pi(\ell),i},$$

which has the same minimizer. Each step adds one Stein-kernel row to the running objective, giving selection cost $O(nmd)$ after the initial diagonal calculation.

Ties use the first minimum. An index may be selected more than once, so `m` may exceed $n$; repeated indices represent repeated mass on the corresponding input rows.

Let $M =$ `precon` and

$$r_M(x, y) = (x - y)^\top M (x - y).$$

For the median rules, let $\rho$ be the median pairwise Euclidean distance among the rows selected by `pre_subsample`. The rules are

$$M_{\text{med}} = \frac{1}{\rho^2} I, \qquad M_{\text{sclmed}} = \frac{\log(m)}{\rho^2} I, \qquad M_{\text{smpcov}} = \widehat{\text{Cov}}(X)^{-1}.$$

The default `"sclmed"` requires $m > 1$. Both median rules require at least two preconditioning rows. If $\rho = 0$, the package replaces $\rho^2$ by 1 and warns. `"smpcov"` uses all rows and requires a nonsingular empirical covariance matrix.

When `pre_subsample` is scalar, `"first"`, `"even"`, and `"random"` use initial, evenly spaced, and randomly sampled rows. An explicit vector of row indices is used directly.

The default base kernel is

$$k(x, y) = \{1 + r_M(x, y)\}^{-1/2}.$$

A Gaussian RBF kernel is supported but must have a fixed positive bandwidth; by default its use produces a warning. For supplied built-in RBF or IMQ objects, `pre` replaces the object's existing preconditioner. Custom callbacks may use the `precon` argument passed by Stein thinning.

## Value

Integer vector $(\pi(1), \ldots, \pi(m))$ containing one-based row indices in selection order. It is not sorted and may contain repeated indices. The selected sample is `X[idx, , drop = FALSE]`.

## Examples

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

---

# svgd *Transport particles with Stein variational gradient descent*

---

## Description

Moves a fixed ensemble of particles toward a target distribution specified by its score function. SVGD uses a base kernel and its first derivative; it does not construct the pairwise Stein kernel $k_{0,p}$.

## Usage

```
svgd(
  x0,
  score_function,
  kernel = stein_kernel(type = "gaussian_rbf"),
  n_iter = 1000,
  step_size = 0.001,
  alpha = 0.9,
  adj_grad = NULL,
  trace_iters = NULL
)
```

## Arguments

| | |
|---|---|
| `x0` | Finite numeric vector or matrix containing the initial particles. Rows are particles and columns are coordinates; a vector is treated as an $m \times 1$ matrix. |
| `score_function` | Function that accepts the current $m \times d$ particle matrix and returns an $m \times d$ matrix with row $s_p(x_i)^\top$. |
| `kernel` | A `SteinKernel` object describing the base kernel. SVGD calls `eval_kernel()` and `grad_x_kernel()`, not `stein_kernel_matrix()`. |

| | |
|---|---|
| `n_iter` | Nonnegative integer number of particle updates. |
| `step_size` | Positive finite update size $\epsilon$. |
| `alpha` | Number in $[0, 1)$ controlling the running squared-gradient average in the default adjustment. |
| `adj_grad` | Optional adjustment function. It receives `grad`, `historical_grad`, `iter`, `theta`, `step_size`, `alpha`, and `fudge_factor`, and must return a finite numeric matrix with the same dimensions as the particles. |
| `trace_iters` | Optional integer vector of iterations to retain. Each requested matrix is stored after that iteration's update. |

## Details

Let $x_1^{(t)}, \ldots, x_m^{(t)}$ be the particles at iteration $t$ and $s_p(x) = \nabla_x \log p(x)$. The raw direction for particle $i$ is

$$g_i^{(t)} = \frac{1}{m} \sum_{j=1}^{m} \left\{ k_\rho(x_j^{(t)}, x_i^{(t)}) s_p(x_j^{(t)}) + \nabla_{x_j} k_\rho(x_j^{(t)}, x_i^{(t)}) \right\}.$$

The first term moves particles toward regions of larger target density; the second repels nearby particles. Each update costs $O(m^2 d)$.

The update is

$$x_i^{(t+1)} = x_i^{(t)} + \epsilon \widetilde{g}_i^{(t)},$$

where $\epsilon$ is `step_size`. By default, operations being entrywise,

$$H^{(1)} = g^{(1)} \odot g^{(1)}, \qquad H^{(t)} = \alpha H^{(t-1)} + (1 - \alpha) g^{(t)} \odot g^{(t)},$$

and

$$\widetilde{g}^{(t)} = \frac{g^{(t)}}{10^{-6} + \sqrt{H^{(t)}}}.$$

Supplying `adj_grad` replaces this adjustment but not the outer multiplication by `step_size`. Use `adj_grad = function(grad, ...) grad` for the unadjusted direction.

For a Gaussian RBF kernel without a fixed bandwidth, let $\rho_t$ be the median pairwise Euclidean distance between the current particles. The bandwidth is recomputed at every iteration as

$$h_t = \frac{\rho_t}{\sqrt{2 \log m}}.$$

For one particle, $h_t = 1$. A zero or non-finite median requires a fixed positive bandwidth. Fixed-bandwidth RBF and other kernels retain their supplied parameters.

## Value

An object of class `"svgd"` containing:

- `X`: final $m \times d$ particle matrix.
- `D`: target scores evaluated at `X`.
- `trace`: requested post-update matrices named by iteration, or `NULL`.
- `n_eval`, `cum_n_eval`: per-iteration and cumulative score-evaluation counts for the update loop. The final evaluation used for `D` is not included.
- `kernel`, `n_iter`, and `step_size`: supplied update settings.
- `method` and `call`: method label and matched call.

## Examples

```
x0 <- matrix(seq(-2, 2, length.out = 5), ncol = 1)
target_score <- function(x) -x
svgd(x0, target_score, n_iter = 5, step_size = 0.05)
```

---

`trace_mixed_kernel` *Mixed-derivative trace of a base kernel*

---

## Description

Mixed-derivative trace of a base kernel

## Usage

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

## Arguments

| | |
|---|---|
| `obj` | A `SteinKernel` object. |
| `X` | Numeric $n_X \times d$ matrix. |
| `Y` | Optional numeric $n_Y \times d$ matrix; `NULL` uses `X`. |
| `precon` | Optional preconditioner overriding `obj$precon`. |
| `...` | Ignored. |

## Value

Numeric $n_X \times n_Y$ matrix $H_{ij} = \mathrm{tr}\{\nabla_x \nabla_y^\top k(x_i, y_j)\}$.

# Index