Package 'steinsampling'

September 7, 2026

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

Version 0.1.2

Date 2026-08-29

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, https://arxiv.org/abs/2608.26450

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

Collate 'steinsampling-package.R' 'stein_helpers.R' 'kernel_classes.R' 'stein_points_alternative_kernel.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 2026-09-07 16:50:02 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

Many targets are known only through an unnormalized density. Stein methods use the target score

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

to test whether data match the target and to transport, construct, or select representative points. The normalizing constant of `p(x)` is not required.

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 internal intermediate values.

`ksd_u_test()` tests independent observations with an off-diagonal U-statistic. `ksd_v_test()` includes the diagonal and uses Rademacher or Markov signs. `fssd_test()` uses finite Stein features at test locations. For KSD-U, `ksd_uq_matrix()` builds the matrix, `ksd_u_statistic()` computes the statistic, and `ksd_u_bootstrap()` calibrates it. The KSD-V counterparts are `ksd_vq_matrix()`, `ksd_v_statistic()`, and `ksd_v_bootstrap()`. For FSSD, `compute_tau()` constructs the features, `fssd_statistic()` computes the statistic, and `fssd_null_pvalue()` simulates the null distribution.

`svgd()` transports an initial particle set. `stein_points()` constructs a point set by continuous search with `fmin_grid()`, `fmin_mc()`, or `fmin_nm()`. `sp_mcmc()` constructs points from states visited by short Markov chains. `stein_thinning()` selects rows from an existing sample.

`stein_kernel()` creates Gaussian RBF and IMQ kernels; `custom_stein_kernel()` creates one from callbacks. `eval_kernel()`, `grad_x_kernel()`, `trace_mixed_kernel()`, `cross_kernel()`, and `grad_theta_v_kernel()` provide the corresponding kernel calculations.

`gmm()` constructs a Gaussian mixture, `rgmm()` samples it, `densitygmm()` evaluates its density, and `get_score_evaluator()` returns its `function(X)` score callback.

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
- https://arxiv.org/abs/2608.26450
- 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. A Gaussian RBF kernel must have a fixed positive bandwidth.

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

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

Value

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

See Also

`fssd_statistic()`, `fssd_null_pvalue()`

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.

Examples

```
k <- stein_kernel("gaussian_rbf", h = 1)
X <- matrix(c(-1, 0, 1), ncol = 1)
S <- -X
all.equal(stein_kernel_matrix(k, X, S),
          tcrossprod(S) * eval_kernel(k, X) + cross_kernel(k, X, S) +
            trace_mixed_kernel(k, X))
```

---

custom_stein_kernel *Create a Stein kernel from callbacks*

Description

Create a Stein kernel from callbacks

Usage

```
custom_stein_kernel(
  eval,
  grad_x,
  trace_mixed,
  fssd_grad = NULL,
  scale2 = NULL,
  precon = NULL,
  custom_grad_mode = c("analytic", "numeric")
)
```

Arguments

`eval`, `grad_x`, `trace_mixed`	Callbacks (`k, X, Y, M`).
`fssd_grad`	Optional analytic FSSD-opt callback (`k, X, vj, grads_X, g_block, M`) returning `grad_vj` and, when the kernel exposes a scale, `grad_param`.
`scale2`	Optional positive squared scale.
`precon`	Optional symmetric positive-definite M.
`custom_grad_mode`	`"analytic"` or location-only `"numeric"`.

Details

`eval`, `grad_x`, and `trace_mixed` supply the kernel value, its first derivative, and its mixed-derivative trace. Each receives `k`, `X`, `Y`, and `M`; `M` 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` returns the $n_X \times n_Y$ matrix $k(X, Y)$, `grad_x` the $n_X \times n_Y \times d$ array $\nabla_x k(X, Y)$, and `trace_mixed` 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`.

KSD uses all three callbacks; FSSD uses only `eval` and `grad_x`. For FSSD-opt a custom kernel must provide `fssd_grad` or use `custom_grad_mode = "numeric"`. A custom kernel can update its scale when `scale2` is supplied; that path requires `fssd_grad`, because numeric mode differentiates the test locations only.

Value

A `SteinKernel` object containing the supplied callbacks.

Examples

```
rbf <- stein_kernel("gaussian_rbf", h = 1)
custom_stein_kernel(
  eval        = function(k, X, Y, M) eval_kernel(rbf, X, Y),
  grad_x      = function(k, X, Y, M) grad_x_kernel(rbf, X, Y),
  trace_mixed = function(k, X, Y, M) 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, X)
```

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

One density value per 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 ρ_{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 a deterministic grid-search function for stein_points(). 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	Finite lower and upper bound vectors of equal length, with ub > lb componentwise.
n0	Positive integer grid size, recycled across dimensions, or a length-d vector of positive integer sizes.
grow	A single logical value; whether to increase grid size as points are selected.

Details

With `grow = TRUE`, the grid resolution increases 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()`. It is practical mainly in low dimension because the total number of candidates is the product of the grid sizes across dimensions.

Value

An optimizer function with signature `function(objective, X_curr, t)` that 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 its objective value, 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 draws a finite set of candidate points in the box and returns 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`	Finite lower and upper bound vectors of equal length, with `ub > lb` componentwise.
`n_mc`	Positive integer number of Monte Carlo candidates.
`mu0`, `Sigma0`	Initial Gaussian proposal mean and covariance. `mu0` must be a finite length-`d` vector and `Sigma0` a finite symmetric positive-definite `d x d` matrix.
`sigsq`	Finite positive local proposal variance after the delay period.
`delay`	Nonnegative integer 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.

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).

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`	Finite lower and upper bound vectors of equal length, with `ub` > `lb` componentwise.
`n_res`	Positive integer number of random restarts.
`mu0`, `Sigma0`	Initial Gaussian proposal mean and covariance. `mu0` must be a finite length-`d` vector and `Sigma0` a finite symmetric positive-definite `d x d` matrix.
`sigsq`	Finite positive local proposal variance after the delay period.
`delay`	Nonnegative integer number of optimization iterations before using local proposals.
`control`	List passed to `stats::optim()`.

Details

The returned optimizer performs a 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()`.

Value

An optimizer function with signature `function(objective, X_curr, t)` that returns `x_min` (the selected point), `d_min` (its score), `f_min` (its objective value), and `n_eval` (objective evaluations charged to the 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 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 $L = \{v_1, \ldots, v_J\}$ and, when the kernel exposes one, the squared kernel scale ρ on training rows. 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.
`n_simulations`	Number of null draws.
`kernel`	`"gaussian_rbf"`, `"imq"`, or a `SteinKernel` object.
`scaling`	Starting positive squared kernel scale (h^2 for Gaussian RBF, c^2 for IMQ). `NULL` uses a five-value grid around the training-row median squared scale when the kernel needs one. Supplying a value skips that grid but does not fix an optimizable squared scale. A finite squared scale in a supplied `SteinKernel` is the starting value and takes precedence; `scaling` is ignored when the kernel exposes no squared scale.
`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`	Number of training-set standard deviations added below each coordinate minimum and above each coordinate maximum.
`scale_lower`, `scale_upper`	Bounds applied to the squared kernel scale when that scale is optimized.
`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 training criterion to maximize is

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

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. Bounded L-BFGS-B refines `L` and any optimizable squared scale jointly; the `scaling` argument specifies how its starting value is chosen.

The selected locations and kernel are fixed for the held-out test. Conditional on the training rows, these choices are independent of the test rows, as required by the null calibration in `fssd_null_pvalue()`.

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 `V`, the realized split sizes, optimized squared 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.
n_simulations	Number of null draws.
kernel	"gaussian_rbf", "imq", or a SteinKernel object.
scaling	Final positive squared kernel scale (h^2 for Gaussian RBF, c^2 for IMQ). NULL uses the square of the median of all pairwise distances in X when the kernel needs a squared scale. A Gaussian RBF kernel carrying precon uses its preconditioned distance; otherwise the distance is Euclidean. A finite squared scale in a supplied SteinKernel takes precedence; scaling is ignored when the kernel exposes no squared scale.
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

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

The same rows are used for the statistic and null simulation. The locations, and any data-derived kernel scale, use the tested rows, so the reported p-value is heuristic. See fssd_null_pvalue() for the independence condition.

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.

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 a Finite Set Stein Discrepancy goodness-of-fit test. With `variant = "opt"` (the default), the test locations and any squared kernel scale are selected on training rows, and the statistic and null calibration use held-out rows; see `fssd_opt_test()`. With `variant = "rand"`, all rows of `X` serve both purposes, so the p-value is heuristic; see `fssd_rand_test()`.

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.
`n_simulations`	Number of null draws.
`kernel`	`"gaussian_rbf"`, `"imq"`, or a `SteinKernel` object.
`scaling`	Positive squared kernel scale: h^2 for Gaussian RBF, c^2 for IMQ; a custom kernel may expose its own. It is final for FSSD-rand and an initial value for FSSD-opt. `NULL` uses a median-based value when the kernel needs a squared scale. A finite squared scale in a supplied `SteinKernel` takes precedence; `scaling` is ignored when the kernel exposes no squared scale.
`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`	Number of training-set standard deviations added below each coordinate minimum and above each coordinate maximum.
`scale_lower`, `scale_upper`	Bounds applied to the squared kernel scale when that scale is optimized.
`seed`	Optional RNG seed.
`imq_beta`	Finite IMQ exponent $\beta < 0$.

Value

An `htest` object. `statistic` is $S_{\tilde{n}} = \tilde{n}\widehat{FSSD}^2$, where $\tilde{n}$ is the held-out size for FSSD-opt and n for FSSD-rand; `p.value` is its simulated right-tail p-value; `info$V` is the test-location matrix, and `info` contains variant-specific diagnostics.

References

Jitkrittum et al. (2017), A Linear-Time Kernel Goodness-of-Fit Test.

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, μ_k, and Σ_k are the weight, mean, and covariance of component k. Weights are normalized to sum to one.

With no arguments, gmm() creates a one-dimensional five-component mixture. If mu is omitted, its entries are drawn independently from Uniform(0, 10) and centered across components within each coordinate. 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 ρ, `grad_param` equal to $\partial \mathcal{L}_j / \partial \rho$.

Value

List with `grad_vj` and `grad_param`.

Examples

```
X <- matrix(c(-1, 0, 1), ncol = 1)
grad_theta_v_kernel(stein_kernel("gaussian_rbf", h = 1), X = X, vj = 1.5,
                    grads_X = -X, g_block = matrix(1, nrow(X), ncol(X)))
```

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 $G_{ijr} = \partial k(x_i, y_j)/\partial x_r$.

Examples

```
k <- stein_kernel("gaussian_rbf", h = 1)
X <- matrix(c(-1, 0, 1), ncol = 1)
G <- grad_x_kernel(k, X)
dim(G)        # n_X x n_Y x d
G[, , 1]
```

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.

Examples

```
k <- stein_kernel("gaussian_rbf", h = 2)
kernel_scale2(k)                              # h^2
kernel_scale2(kernel_scale2(k, 9))
kernel_scale2(stein_kernel("imq", c = 3))    # c^2
```

---

ksd_uq_matrix *Build the Stein-kernel matrix for the KSD tests*

Description

Build the Stein-kernel matrix for the KSD tests

Usage

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

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 unless a supplied SteinKernel already has one. For 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. ksd_u_statistic() and ksd_u_bootstrap() exclude it; ksd_v_statistic() and ksd_v_bootstrap() retain 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	Finite numeric $n \times n$ Stein-kernel matrix with $n \geq 2$, such as the output of ksd_uq_matrix().
nboot	Positive integer number of bootstrap draws. Ignored when W_mat is supplied.
W_mat	Optional finite numeric $n \times B$ multiplier matrix. If NULL, centered multinomial weights are generated. Supplied columns need not be centered.
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`	Finite numeric $n \times n$ Stein-kernel matrix with $n \geq 2$, such as the output of `ksd_uq_matrix()`.

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 unless a supplied `SteinKernel` already has one. For built-in RBF and IMQ kernels this is, respectively, h^2 and c^2. `NULL` uses the squared median pairwise distance.
`nboot`	Positive integer 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 integer 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 integer 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

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

Warns when off-diagonal Stein-kernel magnitudes are negligible relative to the diagonal, because the resulting bootstrap calibration is degenerate.

Value

An `htest` object. `statistic` is $T_n = nU_n$; `p.value` is the bootstrap p-value above. `parameter` records `nboot` and `scaling`, plus `imq_beta` for an IMQ kernel; `kernel` contains the resolved `SteinKernel` object. If `return_raw_boot = TRUE`, `bootstrap_samples` contains $T_n^{*(1)}, \ldots, T_n^{*(B)}$.

References

Liu, Lee, and Jordan (2016), A Kernelized Stein Discrepancy for Goodness-of-Fit Tests.

Examples

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

`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`	Finite numeric $n \times n$ Stein-kernel matrix with $n \geq 2$, such as the output of `ksd_vq_matrix()`.
`nboot`	Positive integer number of bootstrap draws. Ignored when `W_mat` is supplied.
`W_mat`	Optional finite numeric $n \times B$ multiplier matrix. If `NULL`, signs are generated from `boot_method`. Supplied entries need not be ± 1.
`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`	Finite numeric $n \times n$ Stein-kernel matrix with $n \geq 2$, 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 unless a supplied SteinKernel already has one. For built-in RBF and IMQ kernels this is, respectively, h^2 and c^2. NULL uses the squared median pairwise distance.
nboot	Positive integer 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 integer 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 integer 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 is valid only under the dependence, moment, and kernel assumptions of Chwialkowski et al. (2016); this function does not check them. The sign-change probabilities a_n must asymptotically 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

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

Warns when off-diagonal Stein-kernel magnitudes are negligible relative to the diagonal, because the resulting bootstrap calibration is degenerate.

Value

An `htest` object. `statistic` is nV_n; `p.value` is the bootstrap p-value above. `parameter` records `nboot`, `scaling`, and `change_prob`, plus `imq_beta` for an IMQ kernel; `kernel` contains the resolved `SteinKernel` object. If `return_raw_boot = TRUE`, `bootstrap_samples` contains $nV_n^{*(1)}, \ldots, nV_n^{*(B)}$.

References

Chwialkowski, Strathmann, and Gretton (2016), A Kernel Test of Goodness of Fit.

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 by SP-MCMC.

Usage

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

Arguments

`log_p`	Function taking an `n x d` matrix and returning `n` log-density values.
`score_function`	Function taking an `n x d` matrix and returning an `n x d` score matrix.
`x0`	Finite initial state vector.
`h`	Positive step-size multiplier; the proposal covariance is `h * Sigma`.
`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 `x0` in row 1.

Details

From the current state `x`, the proposal is

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

so the proposal covariance is `h * Sigma`. The function evaluates `log_p` at `y`; `-Inf` rejects the proposal. Otherwise it evaluates the score at `y`, forms the reverse proposal, and applies the Metropolis correction. Row 1 of the output is `x0`; `m_iter - 1` proposals follow.

Value

A list with:

- `X`: chain states.
- `D`: scores at those states, reused by SP-MCMC.
- `log_p`: log-density values at those states.
- `accept`: 0/1 indicators, with 0 in row 1.
- `counts`: numbers of rows evaluated by `log_p` and `score_function`; `n_eval` = `counts$total`.

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 = 1.5, beta = -0.75)
```

print.stein_points *Print a Stein point set*

Description

Prints the point matrix size and final KSD or coordinate-update count, plus the corresponding evaluation total.

Usage

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

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

Arguments

x	An object returned by stein_points() or stein_codescent().
...	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)
```

---

print.svgd *Print an SVGD fit*

Description

Print an SVGD fit

Usage

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

Arguments

x	An object returned by svgd().
...	Ignored.

Value

x, invisibly.

---

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, 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 Gaussian random-walk Metropolis transition used by SP-MCMC.

Usage

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

Arguments

log_p	Function taking an n x d matrix and returning n log-density values.
x0	Finite 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 x0 in row 1.

Details

From the current state x, the proposal is

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

The function evaluates log_p(y) and accepts with probability min(1, exp(log_p(y) - log_p(x))); -Inf is rejected. Row 1 of the output is x0; m_iter - 1 proposals follow. RWM does not evaluate the score. sp_mcmc_eval_candidates() computes candidate scores later.

Value

A list with:

- X: chain states.
- log_p: log-density values at those states.
- accept: 0/1 indicators, with 0 in row 1.
- D = NULL, because RWM computes no scores.
- counts: row counts with counts$score = 0; n_eval = counts$total = counts$log_p.

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 supplies candidates, and the state that adds the least to the Stein-kernel sum is kept.

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
)
```

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	A SteinKernel object. A Gaussian RBF kernel must have a fixed positive bandwidth.
n_points	Total number of points, including x_init.
d	State dimension.
mcmc	MCMC transition: "rwm" for Gaussian random-walk Metropolis or "mala" for MALA.
criterion	Path-start rule. "last" uses the newest point, "rand" samples uniformly, and "infl" uses the point whose removal maximizes the remaining-set KSD. A custom rule is a list containing select and an optional label.
m_seq	Number of path states before duplicate removal. A scalar is reused; a vector of length n_points - 1 configures each added point.
h	Positive multiplier in the proposal covariance h * Sigma.
Sigma	Symmetric positive-definite matrix used by the built-in transitions. If NULL, the identity matrix is used.
x_init	Initial finite numeric vector of length d.
seed	Optional RNG seed.
transition_fn	Optional function replacing rwm() or mala().
proposal_fn	Optional per-step update of h or Sigma; see Extensions.

Details

At step j, the function generates a path, removes repeated rows, and appends the row minimizing the greedy objective

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

where $k_{0,p}$ is the Stein kernel formed from kernel and the target score. The returned ksd is the running discrepancy defined in stein_points().

criterion chooses the selected point where each path starts. A path with m_seq states makes m_seq - 1 transitions. Built-in paths use rwm() or mala() with proposal covariance h * Sigma; MALA also uses Sigma in its drift. To use a covariance S for both the path and kernel distance, set Sigma = S and the kernel's precon = solve(S).

Value

An "sp_mcmc" object containing selected points X, scores D, and ksd; per-step and cumulative evaluation counts; path-start, distance, acceptance, and repeated-point diagnostics; recorded settings; and the matched call. Initial diagnostics are NA. Recorded h and Sigma are the original inputs, not step-specific replacements.

Extensions

Supply a custom start rule as `list(select = function(state) ..., label = "custom")`. `select` returns one integer index into `state$X`; `label` defaults to `"custom"`. `state` contains `j`, `X`, `D`, `K0`, recorded diagnostics, and run settings.

A custom `transition_fn(log_p, score_function, x0, h, Sigma, m_iter)` returns `X`, `D`, and `counts`; `X` starts at `x0`, `D` may be `NULL`, and optional `accept` is a length-`m_iter` vector of 0/1 indicators with initial entry 0. `counts` contains nonnegative integer `log_p`, `score`, and `total` counts, with `total = log_p + score`.

`proposal_fn(j, X_curr, h, Sigma, mcmc)` returns a list containing `h` and/or `Sigma`; an empty list keeps both inputs. Changes apply only at step `j`.

References

Chen et al. (2019), Stein Point Markov Chain Monte Carlo.

See Also

`stein_points()`, `sp_mcmc_eval_candidates()`, `rwm()`, `mala()`

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_eval_candidates

Score candidate points with the greedy Stein objective

Description

Scores candidate rows using the greedy objective minimized by `sp_mcmc()`.

Usage

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

Arguments

kernel	A SteinKernel object. A Gaussian RBF kernel must have a fixed positive bandwidth.
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 value used for comparison is

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

where the sum runs over rows in X_curr. objective_values contains one comparison 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 function evaluates score_function on the candidate rows and records how many rows were scored.

Value

A list with:

- objective_values: one comparison value per candidate row.
- scores: candidate scores, reused from cand_D when supplied.
- k0_diag: Stein-kernel diagonal for the candidate rows.
- score_evaluations: rows scored here; zero when cand_D is supplied.

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)
```

stein_codescent *Refine Stein Points by coordinate descent*

Description

Refines a completed Stein point set one row at a time while holding the others fixed. A replacement is accepted only if the point-set KSD does not increase.

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	A SteinKernel object.
n_iter	Number of coordinate-descent updates.
optimizer	Optimizer function used for each coordinate update.
seed	Optional RNG seed.

Details

At iteration it, row r = ((it - 1) %% nrow(X0)) + 1 is replaced by minimizing

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

over x. This is the greedy objective of stein_points(), except the set size stays fixed. With a single row the sum is empty and the objective is $k0(x,x)$.

Because the optimizers are approximate, each proposal is compared with the current row under the same objective before it is accepted. The optimizer interface is the same as in stein_points().

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: retained value of the coordinate objective at each update. These are not KSDs and are not comparable across updates, because the objective omits the kernel sum over the fixed rows, which changes as the points move.
- n_eval: optimizer-reported evaluations per update; the first entry also counts the nrow(X0) initial scores.
- cum_n_eval = cumsum(n_eval).

- `kernel`: kernel used to define the Stein objective.

No running `ksd` is returned. Compute the final KSD as `sqrt(sum(stein_kernel_matrix(kernel, out$X, out$D))) / 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 = 2)
stein_kernel("imq", c = 2, beta = -0.25)
```

`stein_kernel_imq_score` *Create a score-distance IMQ Stein kernel*

Description

Constructs an IMQ kernel from distances between target-score vectors.

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 an `n x d x d` Hessian array.

Details

With $s_p(x) = \nabla_x \log p(x)$, the base kernel is

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

Its Stein derivatives require `hess_log_p`, which returns an `n x d x d` array containing one Hessian of the target log density per row of `X`. Preconditioning is not supported.

Value

A `SteinKernel` object for the score-distance IMQ kernel.

Examples

```
hess_log_p <- function(X) array(-1, dim = c(nrow(as.matrix(X)), 1, 1))
stein_kernel_imq_score(alpha = 1.5, beta = -0.25,
                       hess_log_p = hess_log_p)
```

stein_kernel_inverse_log *Create an inverse-log Stein kernel*

Description

Creates the inverse-log base kernel 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 default $\beta = -1$ gives the plain inverse-log kernel. Compared with a Gaussian RBF kernel, this kernel decays much more slowly as points move apart and retains interactions between distant points.

This kernel uses Euclidean distance and does not support preconditioning.

Value

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

Examples

```
stein_kernel_inverse_log(alpha = 2, beta = -0.5)
```

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 *Construct 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	A `SteinKernel` object used to construct $k_{0,p}$. 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` and its score `d_min` as length-`d` vectors, `f_min` as the value of the supplied objective at `x_min`, and a nonnegative integer `n_eval`. The returned vectors and objective value must be finite. During initialization from `log_p`, only `x_min` and `n_eval` are used; the selected point is scored separately. With truncation, the optimizer must select a feasible candidate; the selected point is checked before it is appended. An infeasible result raises an error. The selected point's objective is recomputed before updating the running KSD.
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	Finite positive denominator in the truncation radius. Ignored when `truncation = "none"`.
truncation	Candidate filter defined above: `"none"`, `"upper"`, `"lower"`, or `"linear"`.
seed	Optional local RNG seed.

Details

Let $k_{0,p}$ be the Stein kernel formed from `kernel` and $s_p(x) = \nabla_x \log p(x)$. The first point is `x_init`; if `x_init` is `NULL`, `optimizer` instead maximizes `log_p`.

Given selected points $x_1, \ldots, x_{j-1}$, the greedy optimizer minimizes

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

This is the amount added to the kernel sum used by the running KSD. With `method = "herding"`, the optimizer instead minimizes

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

The supplied `fmin_grid()`, `fmin_mc()`, and `fmin_nm()` constructors use grid, Monte Carlo, and Nelder-Mead search, respectively.

The returned discrepancy after step j is

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

Truncation keeps candidates satisfying

$$k_{0,p}(x, x) \leq R_j^2.$$

Write $U^2 = 2 \log\{\max(n_p oints, 2)\}/c2$ and $L_j^2 = 2 \log(j)/c2$. `"upper"` uses U^2, `"lower"` uses L_j^2, and `"linear"` uses $L_j^2 + (U^2 - L_j^2)(j-1)/\max(n_p oints - 1, 1)$. Larger `c2` gives a smaller radius. Truncation starts at step 2; `"none"` disables it.

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`: evaluation count reported by `optimizer` at each step; see `fmin_grid()`, `fmin_mc()`, and `fmin_nm()` for what each one counts. The first entry is 1 when `x_init` is supplied; otherwise it adds 1 for scoring the selected first point. `cum_n_eval = cumsum(n_eval)`.
- `method`, `kernel`, `truncation`, and `c2`: settings used by the run.
- `call`: matched function call.

References

Chen et al. (2018), Stein Points.

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 a greedy Stein-discrepancy criterion. 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")
)
```

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.
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	Preconditioner: "sclmed", "med", "smpcov", or a symmetric positive-definite d x d matrix used directly.
kernel	Either "imq", "gaussian_rbf", or a SteinKernel object. The default is "imq". The two character forms use fixed kernel parameters; to set them, pass a stein_kernel() object instead. 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 ρ, Inf for all rows, or an explicit vector of row indices. Ignored for "smpcov" and matrix pre.
pre_subsample_method	Row-selection method used when pre_subsample is scalar: "first" uses the initial rows, "even" uses evenly spaced rows, and "random" samples rows randomly.

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

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

Before the next selection, the chosen Stein-kernel row is added to the running objective. Selection costs $O(nmd)$ after the initial diagonal calculation. Ties use the first minimum. Every row remains eligible, so an index may be selected more than once and `m` may exceed n; repeated indices represent repeated mass on the corresponding rows.

`pre` chooses the matrix M in the kernel distance $r_M(x,y) = (x-y)^\top M (x-y)$. With ρ the median pairwise Euclidean distance among the rows selected by `pre_subsample`, the three rules are

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

`"med"` and `"sclmed"` require at least two preconditioning rows, and the default `"sclmed"` also requires $m > 1$; if $\rho = 0$, ρ^2 is replaced by 1 with a warning. `"smpcov"` uses all rows and requires a nonsingular empirical covariance matrix.

The default base kernel is

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

Using a Gaussian RBF kernel emits a warning; its bandwidth must be fixed and positive. For supplied built-in RBF or IMQ objects, `pre` replaces a stored preconditioner and warns when the matrices differ. Custom callbacks receive the thinning preconditioner as `M`.

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]`.

References

Riabiz et al. (2022), Optimal Thinning of MCMC Output.

Examples

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

`summary.gmm` *Summarize a Gaussian mixture model*

Description

Lays out one row per mixture component with its weight, mean, and marginal standard deviations.

Usage

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

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

Arguments

`object`	A `"gmm"` object.
`...`	Ignored.
`x`	A `"summary.gmm"` object.

Details

`components` has one row per component: the weight w_k, the entries of μ_k, and $\sqrt{\mathrm{diag}(\Sigma_k)}$. Off-diagonal covariance entries are not shown; read them from `object$sigma` when the components are correlated.

Value

`summary()` returns an object of class `"summary.gmm"` with components `nComp`, `d`, and `components`. `print()` renders it and returns `x` invisibly.

See Also

`gmm()`

Examples

```
model <- gmm(nComp = 2, mu = cbind(c(-2, 0), c(2, 0)),
             sigma = array(diag(2), c(2, 2, 2)), weights = c(0.3, 0.7),
             d = 2)
summary(model)
```

summary.sp_mcmc *Summarize an SP-MCMC point set*

Description

Adds SP-MCMC transition, selection, acceptance, and evaluation diagnostics to summary.stein_points().

Usage

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

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

Arguments

object	A "sp_mcmc" object.
...	Ignored.
x	A "summary.sp_mcmc" object.

Details

accept_rate is weighted by the number of transitions in each path; paths without acceptance diagnostics are omitted. n_repeated counts selected points that repeat an earlier one; a candidate path contains its own starting state, so repeats are expected. h is the initial step size, before any per-step proposal changes. counts contains the log_p, score, and candidate_score column totals; n_eval_total reports their total once.

Value

A "summary.sp_mcmc" object extending "summary.stein_points" with transition, criterion, m_seq, h, accept_rate, n_repeated, and counts. The print method returns x invisibly.

See Also

summary.stein_points(), sp_mcmc()

Examples

```
score_function <- function(x) -as.matrix(x)
log_p <- function(x) -0.5 * rowSums(as.matrix(x)^2)
fit <- sp_mcmc(score_function, log_p,
               stein_kernel(type = "gaussian_rbf", h = 1),
               n_points = 4, d = 1, m_seq = 2, h = 0.5, x_init = 0)
summary(fit)
```

`summary.stein_points` *Summarize a Stein point set*

Description

Summarizes the point-set size, kernel, method-specific settings, and evaluation cost for `stein_points()` or `stein_codescent()`.

Usage

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

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

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

Arguments

`object`	A `"stein_points"` or `"stein_codescent"` object.
`...`	Ignored.
`x`	A `"summary.stein_points"` object.

Details

`evaluation_label` names what `n_eval_total` counts for this class. Coordinate-descent objectives are not displayed because they change with the row being updated and are not point-set KSDs.

Value

A `"summary.stein_points"` object with components `method`, `kernel`, `n`, `d`, `evaluation_label`, `n_eval_total`, and `call`. Stein Points adds `ksd_last` and `truncation`, plus `c2` when truncation is enabled. Coordinate descent adds `n_iter`. The print method returns `x` invisibly.

See Also

`print.stein_points()`

Examples

```
score_function <- function(x) -as.matrix(x)
kernel <- stein_kernel(type = "gaussian_rbf", h = 1)
fit <- stein_points(score_function, kernel, n_points = 5, d = 1,
                    optimizer = fmin_grid(lb = -3, ub = 3, n0 = 20),
                    log_p = function(x) -0.5 * rowSums(as.matrix(x)^2))
summary(fit)
```

summary.svgd *Summarize an SVGD fit*

Description

Summarize an SVGD fit

Usage

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

Arguments

object	An object returned by svgd().
...	Ignored.

Value

A "summary.stein_points" object containing the point-set size, kernel, iteration settings, update-loop score-evaluation total, and call.

svgd *Transport particles with Stein variational gradient descent*

Description

Transports an initial set 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 ϵ.
`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 ϵ is `step_size`. By default, $\widetilde{g}$ is the AdaGrad-style rescaling used by the original implementation of Liu and Wang (2016): entrywise,

$$H^{(1)} = g^{(1)} \odot g^{(1)}, \qquad H^{(t)} = \alpha H^{(t-1)} + (1 - \alpha) g^{(t)} \odot g^{(t)}, \qquad \widetilde{g}^{(t)} = \frac{g^{(t)}}{10^{-6} + \sqrt{H^{(t)}}}.$$

Supplying `adj_grad` replaces this rescaling but not the outer multiplication by `step_size`; `adj_grad = function(grad, ...) grad` gives $\widetilde{g}^{(t)} = g^{(t)}$, the plain update as stated by Liu and Wang (2016).

For a Gaussian RBF kernel without a fixed bandwidth, let ρ_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, $h_t = 1$. If the median is zero or non-finite, `svgd()` stops; supply a fixed positive RBF 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.

References

Liu and Wang (2016), Stein Variational Gradient Descent.

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

Examples

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

Index