Package: SpaNorm (via r-universe)

November 25, 2026

Title Spatially-aware normalisation for spatial transcriptomics data

Version 1.8.0

Description This package implements the spatially aware library size normalisation algorithm, SpaNorm. SpaNorm normalises out library size effects while retaining biology through the modelling of smooth functions for each effect. Normalisation is performed in a gene- and cell-/spot- specific manner, yielding library size adjusted data.

License GPL (>= 3)

Encoding UTF-8

Roxygen list(markdown = TRUE)

biocViews Software, GeneExpression, Transcriptomics, Spatial, CellBiology

Depends R (>= 4.4)

Imports edgeR, ggplot2, Matrix, matrixStats, methods, rlang, scran, SeuratObject, SingleCellExperiment, SpatialExperiment, stats, SummarizedExperiment, S4Vectors, utils, BiocParallel, BiocSingular, DelayedArray

Suggests testthat (>= 3.0.0), knitr, rmarkdown, prettydoc, pkgdown, covr, BiocStyle, scater, Seurat (>= 5.0.0), patchwork, ggforce, ggnewscale, torch, RhpcBLASctl

URL https://bhuvalab.github.io/SpaNorm

BugReports https://github.com/BhuvaLab/SpaNorm/issues

VignetteBuilder knitr

Config/testthat/edition 3

LazyData false

Config/roxygen2/version 8.0.0

Config/pak/sysreqs
libglpk-dev libmagick++-dev gsfonts libicu-dev libxml2-dev libssl-dev zlib1g-dev

Repository Bioconductor 3.24

Date/Publication 2026-11-23 22:53:16 UTC

RemoteUrl https://github.com/bioc/SpaNorm

RemoteRef RELEASE_3_24

RemoteSha 25b625953e81d65293367c88af9e038b5c007270

Contents

add_vec_mat_gpu . . . 3
calculateMu . . . 3
checkGPU . . . 5
diag_mat . . . 5
dnbinom_gpu . . . 6
fastSizeFactors . . . 6
filterGenes . . . 7
fitNB . . . 8
getBackendDevice . . . 10
getBackendDtype . . . 10
getGPUMemoryBudget . . . 11
gpuDtypeBytes . . . 11
hasBadValues . . . 12
HumanDLPFC . . . 12
invert_mat . . . 13
invert_mat_batched . . . 14
is_torch_tensor . . . 15
isPolished . . . 15
matmul_gpu . . . 16
mult_vec_mat_gpu . . . 16
nbBlockDesign . . . 17
nbDevianceMoments . . . 19
nbUnitDeviance . . . 20
plotCovariate . . . 21
plotSpatial . . . 22
polishNB . . . 23
polishSpaNorm . . . 27
qlDispersion . . . 31
qlMomentTable . . . 32
resetGPUCache . . . 33
rowSums_gpu . . . 34
setGPUMemoryBudget . . . 34
SpaNorm . . . 35
SpaNormFit . . . 38
SpaNormPCA . . . 39
SpaNormSVG . . . 41
tcrossprod_gpu . . . 42
toGPUMatrix . . . 43
toGPUVector . . . 44
topSVGs . . . 44

toRMatrix 45
tpsBasis 45

Index **48**

`add_vec_mat_gpu` *Broadcast-add a vector to a matrix, GPU-aware*

Description

Adds `vec` to `mat`, broadcasting per-row if `length(vec) == nrow(mat)` or per-column if `length(vec) == ncol(mat)` (matching the GPU path's broadcast orientation explicitly, rather than relying on R's column-major recycling). Runs on the accelerator when either argument is a torch tensor and an accelerator is active.

Usage

```
add_vec_mat_gpu(vec, mat, backend = c("auto", "cpu", "gpu"))
```

Arguments

`vec`	a numeric vector (or torch tensor).
`mat`	a matrix (or torch tensor).
`backend`	one of `"auto"` (default), `"cpu"` or `"gpu"`.

Value

`mat + vec`, broadcast as described above.

Examples

```
add_vec_mat_gpu(1:3, matrix(1, 3, 4), backend = "cpu")
```

`calculateMu` *Compute fitted means from a negative binomial GLM fit*

Description

Computes the fitted mean matrix `mu = exp(gmean + tcrossprod(alpha, W) + offset)` from the per-gene coefficients of a negative binomial GLM, with optional per-gene winsorisation of the log-means to `median +/- winsor * MAD` to bound the influence of extreme fitted values. Exposed so downstream packages (e.g. spiDE) can reconstruct fitted means from a `fitNB` result; for the generic (intercept-free) fit pass `gmean = rep(0, nrow(alpha))`.

Usage

```
calculateMu(
  gmean,
  alpha,
  W,
  winsor = DEFAULT_WINSOR,
  backend = c("cpu", "auto", "gpu"),
  offset = NULL
)
```

Arguments

gmean	a numeric vector of per-gene intercepts (length nrow(alpha)).
alpha	a genes x covariates matrix of coefficients.
W	a cells x covariates design matrix.
winsor	a numeric, the number of MADs at which per-gene log-means are winsorised (default 4); Inf disables winsorisation.
backend	one of "cpu" (default – always returns a base R matrix, matching this function's behaviour before GPU dispatch was added), "gpu" or "auto" (use an accelerator if one is available; the result is a torch tensor in that case, not a matrix).
offset	either NULL (the default) or a nrow(alpha) x nrow(W) matrix added to the linear predictor with a coefficient of 1, matching fitNB's offset argument. Note that winsor clamps the total log-mean, i.e. after the offset has been added.

Details

Runs on the accelerator when backend resolves to one (alpha/ W/gmean are pushed onto the active device via toGPUMatrix/toGPUVector); the result is a torch tensor in that case (convert back with toRMatrix if needed), or a base R matrix otherwise.

Value

a genes x cells matrix (or torch tensor, if backend resolves to an accelerator) of fitted means.

Examples

```
set.seed(1)
Y <- matrix(rpois(20 * 50, 5), 20, 50)
W <- cbind(1, scale(seq_len(50)))
fit <- fitNB(Y, W, verbose = FALSE)
mu <- calculateMu(fit$gmean, fit$alpha, W)
dim(mu)
```

checkGPU — *Is an accelerated (GPU) torch device in use?*

Description

Is an accelerated (GPU) torch device in use?

Usage

```
checkGPU()
```

Value

a logical, TRUE iff getBackendDevice() resolves to a non-"cpu" device.

Examples

```
checkGPU()
```

diag_mat — *Build a diagonal matrix from a vector, GPU-aware*

Description

As diag, but returns a torch tensor (dense, to match the CPU fallback's dense output) when an accelerator is active.

Usage

```
diag_mat(vec, backend = c("auto", "cpu", "gpu"))
```

Arguments

vec	a numeric vector (or torch tensor).
backend	one of "auto" (default), "cpu" or "gpu".

Value

a diagonal matrix (base matrix or torch tensor).

Examples

```
diag_mat(1:3, backend = "cpu")
```

dnbinom_gpu *Negative binomial density (mean/size parameterisation), GPU-aware*

Description

As dnbinom (with mu instead of prob), but runs the log-pmf on the accelerator when x or mu is a torch tensor and an accelerator is active. Targets the domain SpaNorm actually uses (non-negative integer x, mu > 0) – outside that domain this is not a drop-in replacement for stats::dnbinom (e.g. mu = 0 gives NaN rather than a point mass at 0). Exposed so downstream packages (e.g. spiDE) computing their own per-gene NB log-likelihoods reuse the same GPU/CPU-dual-path NB math SpaNorm's own fitting uses, rather than duplicating it.

Usage

```
dnbinom_gpu(x, mu, size, log = FALSE)
```

Arguments

x	quantiles (matrix or torch tensor).
mu	the mean (matrix or torch tensor, same shape as x).
size	the NB size parameter (1/dispersion); a length-nrow(x) vector broadcasts row-wise, matching how stats::dnbinom recycles it.
log	a logical, return the log-density (default FALSE).

Value

the (log-)density, as a matrix or torch tensor matching the input.

Examples

```
dnbinom_gpu(matrix(0:3, 2, 2), mu = matrix(1, 2, 2), size = c(2, 2))
```

fastSizeFactors *Filter genes based on expression*

Description

This function computes the size factors using a fast but inaccurate approach. Size factors are computed using the direct estimate of library sizes (sum of all counts). Though fast, this approach does not cater for compositional biases in the data and therefore is less accurate than scran-based estimates.

Usage

```
fastSizeFactors(spe)

## S4 method for signature 'SpatialExperiment'
fastSizeFactors(spe)
```

Arguments

spe	a SpatialExperiment, Seurat, or SpatialFeatureExperiment object containing count data.

Value

a SpatialExperiment, Seurat, or SpatialFeatureExperiment, containing size factors in the 'sizeFactor' column of the column annotation.

Examples

```
data(HumanDLPFC)
HumanDLPFC <- fastSizeFactors(HumanDLPFC)
head(HumanDLPFC$sizeFactor)
```

---

filterGenes *Filter genes based on expression*

Description

This function removes genes that are very lowly expressed.

Usage

```
filterGenes(spe, prop = 0.1)

## S4 method for signature 'SpatialExperiment'
filterGenes(spe, prop = 0.1)

## S4 method for signature 'Seurat'
filterGenes(spe, prop = 0.1)
```

Arguments

spe	a SpatialExperiment, Seurat, or SpatialFeatureExperiment object containing count data.
prop	a numeric, indicating the proportion of loci/cells where the gene should be expressed (default is 0.1, i.e., genes should be expressed in at least 10% of the loci/cells).

Value

a logical vector encoding which genes should be kept for further analysis.

Examples

```
data(HumanDLPFC)
keep <- filterGenes(HumanDLPFC)
table(keep)
```

`fitNB` *Fit a per-gene negative binomial GLM*

Description

Fits a per-gene negative binomial regression over an arbitrary design matrix using SpaNorm's IRLS engine (iteratively reweighted least squares with per-gene dispersion estimated by `edgeR::estimateDisp`). This exposes the fitting machinery independently of SpaNorm's spatial model: the fitted model is `log(mu) = W %*% t(alpha)` with no built-in intercept, so encode one as a column of `W` when a per-gene baseline is required. It corresponds to the internal fit with `is.spanorm = FALSE` (no shared library-size column, ridge applied to every column of `W`).

Usage

```
fitNB(
  Y,
  W,
  idx = rep(TRUE, ncol(Y)),
  lambda.a = 0,
  winsor = DEFAULT_WINSOR,
  maxit.psi = 25,
  maxit.nb = 50,
  tol = 1e-04,
  ...,
  offset = NULL,
  psi = NULL,
  backend = c("auto", "cpu", "gpu"),
  verbose = TRUE
)
```

Arguments

`Y`	a genes x cells matrix of counts (dense, sparse, or DelayedArray).
`W`	a cells x covariates numeric design matrix.
`idx`	a logical vector (length `ncol(Y)`) selecting the cells used to fit the model (default: all cells).

`lambda.a` a numeric ridge penalty on the columns of `W`: a single value or a per-column vector (default 0, i.e. unregularised).

`winsor` a numeric, the number of MADs at which per-gene coefficients are winsorised during fitting (default 4). Must be a single positive number; `Inf` disables winsorisation entirely.

`maxit.psi` a numeric, the maximum number of dispersion iterations.

`maxit.nb` a numeric, the maximum number of NB mean IRLS iterations.

`tol` a numeric, the convergence tolerance.

`...` additional fitting parameters forwarded to the internal fitter, e.g. `maxn.psi` (dispersion-estimation subsample size) or `step.factor` (IRLS step-halving factor).

`offset` either `NULL` (the default, no offset) or a genes x cells numeric matrix with the same dimensions as `Y`, added to the linear predictor with a coefficient fixed at 1, so that each fitted log-mean `log(mu[g, c])` gains `offset[g, c]`. Use this (rather than an extra column of `W`) for a known, non-estimated effect – a fitted column's coefficient floats and can absorb signal that correlates with the known effect, whereas an offset cannot. The offset is subset by `idx` internally, alongside `Y`, and enters the dispersion estimation as well as the mean.

`psi` either `NULL` (the default: per-gene dispersions are estimated by the usual outer loop, via `edgeR::estimateDisp`) or a numeric vector of length `nrow(Y)` of per-gene NB dispersions (`size = 1/psi`). Supplied dispersions are used as-is – no re-estimation, no winsorisation – and the outer dispersion loop is bypassed entirely: the coefficients come from a single IRLS fit at the given `psi` (so `maxit.psi` is ignored, and the returned `loglik` has one element). The dispersions need not come from an identical design – values estimated on a design nested in (or equal to) the fitting design are appropriate, e.g. pooled across a coarser model, which errs conservative. `offset` and `psi` compose.

`backend` a character, the compute backend ('auto', 'cpu', or 'gpu').

`verbose` a logical, whether to print progress messages (default TRUE).

Value

a list with per-gene coefficients `alpha` (genes x covariates), dispersions `psi`, a `gmean` element (always zero – the generic fit has no intercept term), the `sampling` factor (with a supplied `psi` no dispersion subsample is drawn, so its `"dispersion"` level is absent: cells are `"glm"` if used for fitting, else `"all"`), and per-outer-iteration `loglik` (length 1 when `psi` is supplied).

Examples

```
set.seed(1)
Y <- matrix(rpois(20 * 50, 5), 20, 50)
W <- cbind(1, scale(seq_len(50))) # intercept + one covariate
fit <- fitNB(Y, W, verbose = FALSE)
str(fit$alpha)
```

getBackendDevice *Resolve the active torch backend device*

Description

Resolves (once per session, then cached – see `resetGPUCache`) the torch device to use for accelerated operations: `"cuda"` if a CUDA GPU is available, else `"mps"` on Apple Silicon, else `"cpu"`. Exposed for downstream packages (e.g. spiDE) that push their own tensors onto the same device SpaNorm is using.

Usage

```
getBackendDevice()
```

Value

a character, one of `"cuda"`, `"mps"` or `"cpu"`.

Examples

```
getBackendDevice()
```

getBackendDtype *Resolve the active torch backend dtype*

Description

The torch floating-point dtype to use for accelerated tensors: float64 everywhere except MPS, which cannot represent float64 and so is capped at float32 (its maximum precision).

Usage

```
getBackendDtype()
```

Value

a torch dtype (`torch::torch_float64()` or `torch::torch_float32()`).

Examples

```
if (checkGPU()) getBackendDtype()
```

getGPUMemoryBudget *Determine the accelerator memory budget for blocked fitting*

Description

Resolves (once per session, then cached – see `resetGPUCache`) the accelerator memory budget (bytes) to use for memory-aware gene-blocked fitting: free CUDA memory via `nvidia-smi`, a conservative fraction of total system memory for MPS (unified memory has no per-process query), or a fixed fallback if detection fails. A user-supplied `gpu.mem.budget` bypasses detection entirely – set it to `Inf` to disable blocking outright. Returns `Inf` when no accelerator is in use. Exposed so downstream packages (e.g. spiDE) doing their own GPU-memory-aware blocking share the same budget/cache as SpaNorm's own fitting.

Usage

```
getGPUMemoryBudget(gpu.mem.budget = NULL)
```

Arguments

`gpu.mem.budget`	`NULL` (default, auto-detect) or a single positive number (bytes).

Value

a numeric, the memory budget in bytes (`Inf` if no accelerator is in use).

Examples

```
getGPUMemoryBudget()
```

gpuDtypeBytes *Bytes per element for the active backend dtype*

Description

The element width (in bytes) of `getBackendDtype()`'s dtype – 4 for float32 (MPS), 8 for float64 (cuda/cpu). Used for sizing memory budgets against genes x cells tensors; exposed so downstream packages (e.g. spiDE) doing their own GPU memory-budget accounting don't need to duplicate this device/dtype mapping.

Usage

```
gpuDtypeBytes()
```

Value

a numeric, 4 or 8.

Examples

```
gpuDtypeBytes()
```

`hasBadValues` *Does an object hold any NA/NaN/Inf?*

Description

`TRUE` if `x` (tensor or matrix) holds any `NA`/`NaN`/ `Inf`. On a tensor this is a scalar reduction copied back, not a full GPU->CPU materialisation of `x`.

Usage

```
hasBadValues(x)
```

Arguments

`x`	a matrix or torch tensor.

Value

a logical.

Examples

```
hasBadValues(matrix(c(1, NA, 3, 4), 2, 2))
```

`HumanDLPFC` *Human dorsolateral prefrontal cortex (DLPFC) visium sample*

Description

Human DLPFC sample profiled using the 10x Visium platform. Lowly expressed genes were filtered out using the `filterGenes` function to retain genes expressed in at least 10% of samples. This version was obtained from the SpatialLIBD R/Bioconductor package.

Usage

```
data(HumanDLPFC)
```

Format

A SpatialExperiment containing region annotations in the `colData` column 'AnnotatedCluster'.

Source

SpatialLIBD R/Bioconductor package.

References

Maynard KR, Collado-Torres L, Weber LM, Uytingco C, Barry BK, Williams SR, Catallini JL, Tran MN, Besich Z, Tippani M, Chew J. Transcriptome-scale spatial gene expression in the human dorsolateral prefrontal cortex. Nature neuroscience. 2021 Mar;24(3):425-36.

`invert_mat` *Invert a symmetric positive-definite matrix*

Description

Inverts a (typically small) symmetric positive-definite matrix using a Cholesky factorisation, falling back to a general solve when the Cholesky fails. When a torch backend is active and `mat` is a tensor, the factorisation runs on the accelerator (or CPU for MPS) and the result is returned on the input device. Exposed for downstream packages (e.g. spiDE) that reuse SpaNorm's fitting machinery for Wald-type inference.

Usage

```
invert_mat(mat)
```

Arguments

`mat`	a symmetric positive-definite matrix (base matrix or torch tensor).

Value

the matrix inverse, in the same representation as `mat`.

Examples

```
m <- crossprod(matrix(rnorm(30), 10, 3))
invert_mat(m)
```

`invert_mat_batched` *Invert a batch of symmetric positive-definite matrices*

Description

Batched counterpart of `invert_mat`: inverts a `(batch, n, n)` stack of symmetric positive-definite matrices in a single Cholesky factorisation call rather than looping `invert_mat()` over each slice. SpaNorm's own model never needs this (its per-gene fit shares a single coefficient across all genes, so it only ever inverts one matrix at a time); it is exposed for downstream packages (e.g. spiDE) that need a genuinely per-gene/per-feature covariance – a different `n x n` matrix for every slice of the batch, varying with a per-gene working weight – where looping `invert_mat()`'s R-level `tryCatch`/Cholesky call once per gene is the dominant cost.

Usage

```
invert_mat_batched(mat)
```

Arguments

`mat`	a `(batch, n, n)` torch tensor, or a base R array of the same shape (`dim(mat) == c(batch, n, n)`).

Details

Unlike `invert_mat()`'s per-matrix `tryCatch`, a failure here (e.g. one non-SPD slice) falls back to a general solve for the *whole* batch, not just the failing slice – correctness is unaffected (the fallback solves every slice correctly regardless of conditioning), it is simply not the exact same numerical routine applied per-slice in that (rare) case.

Value

the batch of matrix inverses, in the same representation as `mat`.

Examples

```
m <- array(0, c(4, 3, 3))
for (i in 1:4) m[i, , ] <- crossprod(matrix(rnorm(9), 3, 3)) + diag(3)
invert_mat_batched(m)
```

is_torch_tensor *Is an object a torch tensor?*

Description

Is an object a torch tensor?

Usage

```
is_torch_tensor(x)
```

Arguments

x	an object to test.

Value

a logical.

Examples

```
is_torch_tensor(matrix(1:4, 2, 2))
```

isPolished *Does a SpaNormFit carry a converged per-gene polish?*

Description

Does a SpaNormFit carry a converged per-gene polish?

Usage

```
isPolished(fit)
```

Arguments

fit	an object of class SpaNormFit.

Value

TRUE iff fit has a non-empty polish slot, i.e. it was returned by polishSpaNorm() (or otherwise had its polish slot set). FALSE for a fresh fitSpaNorm()/SpaNorm() result, and for an object saved before this slot existed (checked via methods::.hasSlot(), since fit$polish/fit@polish would error on such an object).

Examples

```
data(HumanDLPFC)

spe <- SpaNorm(HumanDLPFC, sample.p = 0.05, df.tps = 2, tol = 1e-2)
isPolished(S4Vectors::metadata(spe)$SpaNorm)
```

`matmul_gpu` *Tensor-aware matrix multiplication*

Description

Computes `x %*% y` using the accelerator if either argument is a torch tensor and an accelerator is active (vectors are reshaped so the result matches base R's `%*%` shape semantics); falls back to base `%*%` otherwise. Named explicitly (rather than overriding `%*%`) so base matrix multiplication semantics are preserved everywhere else. Exposed so downstream packages (e.g. spiDE) can build batched linear algebra (e.g. a shared design matrix against many per-gene/per-block weight vectors) without depending on torch internals.

Usage

```
matmul_gpu(x, y)
```

Arguments

`x, y`	a matrix, numeric vector, or torch tensor.

Value

`x %*% y`, as a matrix or torch tensor matching the input.

Examples

```
matmul_gpu(matrix(1:6, 2, 3), matrix(1:6, 3, 2))
```

`mult_vec_mat_gpu` *Broadcast-multiply a vector and a matrix, GPU-aware*

Description

As `add_vec_mat_gpu`, but multiplying rather than adding.

Usage

```
mult_vec_mat_gpu(vec, mat, backend = c("auto", "cpu", "gpu"))
```

Arguments

vec	a numeric vector (or torch tensor).
mat	a matrix (or torch tensor).
backend	one of "auto" (default), "cpu" or "gpu".

Value

mat * vec, broadcast as per add_vec_mat_gpu.

Examples

```
mult_vec_mat_gpu(1:3, matrix(1, 3, 4), backend = "cpu")
```

---

nbBlockDesign *A design with a per-group block, in compact form*

Description

Describes the design matrix

$$W = [\, X \mid Z_1 \mid Z_2 \mid \cdots \mid Z_G \,],$$

where X holds the ordinary (dense) columns and the per-group blocks Z_g are the rows of Z belonging to group g and zero elsewhere: every group has its own copy of the q columns of Z. This is, for example, a per-patient intercept and per-patient spatial library-size spline, Z = cbind(1, l * tpsBasis(x, y, df = c(3, 3))), with block the patient. The linear predictor of a gene with coefficients alpha is X %*% alpha_x + rowSums(Z * A[block,]), with A the G x q matrix of that gene's block coefficients.

Usage

```
nbBlockDesign(X, Z, block, rank.tol = 1e-10)

## S3 method for class 'nbBlockDesign'
dim(x)

## S3 method for class 'nbBlockDesign'
dimnames(x)

## S3 method for class 'nbBlockDesign'
as.matrix(x, ...)

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

Arguments

X	a numeric cells x p_x matrix of the non-absorbed columns (p_x >= 1).
Z	a numeric cells x q matrix: each cell's values of its own group's block columns.
block	the group of each cell, a factor (or a vector coerced with factor()) with no missing values. Every level is a group, including a level with no cells, whose coefficients are then identified by the penalty alone (or held at their start, see Degenerate blocks).
rank.tol	the tolerance below which a group's block is treated as rank-deficient (see Degenerate blocks).
x	an nbBlockDesign.
...	unused.

Details

The object is accepted as the design W by polishNB(), nbProfilePsi() and nbNewtonSolver(), which then never build the dense n x (p_x + G q) matrix. The G blocks are absorbed exactly by a Schur complement, as polishNB(absorb =) absorbs a per-column grouping of a dense design, so a Newton step costs one n x (p_x + q) gram whatever G is. The results are those of the dense design as.matrix(W) with the block columns absorbed by group (absorb = c(rep(NA, p_x), rep(seq_len(G), each = q))).

Coefficient layout. The implied design has p = p_x + G * q columns, in the order of as.matrix(): the columns of X, then group 1's q columns, then group 2's, and so on (groups in the order of levels(block)). alpha, lambda.a and start.cols follow that layout; the per-group coefficients of gene g are matrix(alpha[g, -(1:p_x)], G, q, byrow = TRUE). lambda.a and start.cols may also be given for [X | Z] only (length p_x + q), in which case the Z part is used for every group.

Degenerate blocks. Each group's block of the information matrix, $C_g = Z_g' \mathrm{diag}(w) Z_g + \mathrm{diag}(\lambda_g)$, is scaled to unit diagonal and Cholesky-factorised. A group whose block is rank-deficient or nearly so – fewer cells than columns, a column constant or zero over the group's cells, a level of block with no cells at all, any of these with a zero penalty – is recognised by its smallest squared Cholesky pivot falling below rank.tol (on the unit-diagonal scale that pivot is $1 - R^2$ of a column regressed on the columns before it, so the default 1e-10 flags a variance inflation above 10^{10}). Such a block is inverted by a truncated eigendecomposition instead (a generalised inverse; eigenvalues below rank.tol times the largest are dropped), and a column with no weight and no penalty in the group is dropped outright. The directions this drops are combinations of the group's coefficients that do not change the linear predictor on its cells and carry no penalty, so the objective is flat along them: the Newton step is the minimum-norm step in the unit-diagonal scaling, which leaves the coefficients along those directions at their starting values (an unpenalised column of a group with no cells keeps its start exactly). The likelihood, the fitted means, the dispersion, the dense-column coefficients and their covariance are those of the same design with the aliased columns removed; the individual block coefficients of an aliased group are not identified and are one solution among many. The dense grouped path has no such rule and reports a gene with an unpenalised aliased block as singular.

Value

an object of class nbBlockDesign: a list holding X, Z, block and precomputed indexing, with dim() and dimnames() methods that describe the implied design (so nrow(), ncol() and colnames() work) and an as.matrix() method that builds it densely (for checking, or for a small problem).

See Also

polishNB(), nbNewtonSolver(), tpsBasis().

Examples

```
set.seed(1)
n <- 300
patient <- factor(sample(c("P1", "P2", "P3"), n, replace = TRUE))
X <- cbind(niche = rnorm(n))
l <- rnorm(n)                                  # centred log library size
Z <- cbind(1, l)                               # per-patient intercept and slope
D <- nbBlockDesign(X, Z, patient)
dim(D)                                         # 300 x (1 + 3 * 2)
mu <- exp(1 + 0.3 * X[, 1] + 0.5 * l)
Y <- t(replicate(4, rnbinom(n, mu = mu, size = 5)))
a0 <- matrix(0, 4, ncol(D))
fit <- polishNB(Y, D, a0, rep(0.2, 4), lambda.a = c(0, 1e-3, 1e-3),
                start.cols = c(FALSE, TRUE, FALSE))
fit$alpha[, 1]                                 # the niche coefficient
```

---

nbDevianceMoments *First two moments of the NB unit deviance, by direct summation*

Description

Reproduces the definition edgeR's compute_weight() approximates with Chebyshev tables. The summation window is centred on mu and sized from the pmf's own quantiles rather than fixed at edgeR's 50 terms from zero: edgeR can truncate there because that branch is only reached when phi >= 4, where the pmf piles up near zero; this is reached with any dispersion.

Usage

```
nbDevianceMoments(mu, phi, eps = 1e-10, maxterms = 20000L)
```

Arguments

mu	vector of means.
phi	NB dispersion: scalar, or the same length as mu.
eps	tail probability left outside the summation window at each end.
maxterms	hard cap on the number of pmf terms summed per element.

Details

Elements are grouped by the width they need so a few high-mu cells do not impose their window on everything, and each group accumulates in two passes so memory stays O(length(mu)) rather than O(width * length(mu)).

Value

list with `w0` (deviance rescaling) and `w1` (effective df), each the length of `mu`.

Examples

```
nbDevianceMoments(mu = c(0.1, 1, 10), phi = 5)
```

`nbUnitDeviance` *NB unit deviance*

Description

The negative binomial unit deviance $2\{y\log(y/\mu) - (y+1/\phi)\log((y+1/\phi)/(\mu+1/\phi))\}$, elementwise over a genes x cells matrix, on the CPU or – when `y` or `mu` is a torch tensor – on that tensor's device.

Usage

```
nbUnitDeviance(y, mu, phi)
```

Arguments

`y`	counts, genes x cells (matrix or torch tensor).
`mu`	fitted means, same shape (matrix or torch tensor).
`phi`	NB dispersion: a scalar, or one value per gene (row).

Value

a genes x cells matrix, or a torch tensor when the input was one.

Examples

```
y <- matrix(rpois(20, 3), 4, 5)
nbUnitDeviance(y, mu = y + 0.5, phi = 0.1)
```

plotCovariate *Diagnostic plot of predicted expression for a covariate*

Description

This function can be used to spatially visualise the library size, biology or batch specific effect modelled for each gene.

Usage

```
plotCovariate(spe, covariate = c("biology", "ls", "batch"), ...)
```

Arguments

spe	a SpatialExperiment object.
covariate	a character, specifying the type of covariate to be plot: "biology" (default), "ls" to plot the library size effect, and "batch" to plot the batch-specific effect.
...	additional parameters to be passed to the plotSpatial function.

Value

a ggplot2 object

Examples

```
library(SpatialExperiment)
library(ggplot2)

data(HumanDLPFC)

HumanDLPFC = SpaNorm(HumanDLPFC, sample.p = 0.05, df.tps = 2, tol = 1e-2)
# plot spatial region annotations
p1 <- plotCovariate(HumanDLPFC, covariate = "biology", colour = ENSG00000075624) +
  scale_colour_viridis_c(option = "F")
p1

p2 <- plotCovariate(HumanDLPFC, covariate = "ls", colour = ENSG00000075624) +
  scale_colour_viridis_c(option = "F")
p2
```

`plotSpatial` *Plot spatial transcriptomic annotations per spot*

Description

Plot spatial transcriptomic annotations per spot

Usage

```
plotSpatial(
  spe,
  what = c("annotation", "expression", "reduceddim"),
  ...,
  assay = SummarizedExperiment::assayNames(spe),
  dimred = SingleCellExperiment::reducedDimNames(spe),
  img = FALSE,
  crop = FALSE,
  imgAlpha = 1,
  rl = 1,
  circles = FALSE
)
```

Arguments

`spe`	a SpatialExperiment object.
`what`	a character, specifying what aspect should be plot, "annotation", "expression", or reduced dimension ("reduceddim").
`...`	additional aesthetic mappings or fixed parameters (e.g., shape = ".").
`assay`	a character or numeric, specifying the assay to plot (default is the first assay).
`dimred`	a character or numeric, specifying the reduced dimension to plot (default is the first reduced dimension).
`img`	a logical, indicating whether the tissue image (if present) should be plot (default = FALSE).
`crop`	a logical, indicating whether the image should be cropped to the spatial coordinates (default = FALSE).
`imgAlpha`	a numeric, specifying the alpha value for the image (default = 1).
`rl`	a numeric, specifying the relative size of the text (default = 1).
`circles`	a logical, indicating whether the spots should be plotted as circles (default = FALSE). This can be slower for large datasets.

Value

a ggplot2 object

Examples

```
library(SpatialExperiment)
library(ggplot2)

# load data
data(HumanDLPFC)

# plot spatial region annotations
p1 <- plotSpatial(HumanDLPFC, colour = AnnotatedCluster)
p1

# change colour scale
p1 + scale_colour_brewer(palette = "Paired")

# plot spatial expression
plotSpatial(HumanDLPFC, what = "expression", colour = ENSG00000075624) +
 scale_colour_viridis_c(option = "F")

# plot logcounts
logcounts(HumanDLPFC) <- log2(counts(HumanDLPFC) + 1)
plotSpatial(HumanDLPFC, what = "expression", colour = ENSG00000075624, assay = "logcounts") +
 scale_colour_viridis_c(option = "F")

# change point shape
plotSpatial(HumanDLPFC, what = "expression", colour = ENSG00000075624,
  assay = "logcounts", shape = 18) +
 scale_colour_viridis_c(option = "F")
```

`polishNB` *Converge every gene of a shared negative binomial fit to its own optimum*

Description

`fitNB()` fits every gene in one IRLS loop: it shares a single gene-averaged cell weight vector across genes, decides step-halving and convergence on the aggregate log-likelihood, and clamps coefficients across genes. For most genes the result is indistinguishable from the gene's own optimum, but a bright, cell-type-restricted gene can be left well short of it, with an inflated dispersion. `polishNB()` takes such a fit and converges each gene separately, by damped Newton on that gene's own penalised negative binomial log-likelihood

$$\sum_i \log f_{NB}(y_i; \mu_i, \psi) - \frac{1}{2}\sum_j \lambda_j \alpha_j^2, \qquad \log \mu = W\alpha + \text{offset}.$$

Usage

```
polishNB(
  Y,
```

```
  W,
  alpha,
  psi,
  lambda.a = 0,
  offset = NULL,
  absorb = NULL,
  absorb.batch = NULL,
  start.cols = NULL,
  psi.method = c("profile", "fixed"),
  psi.range = c(0.001, 1000),
  warm = FALSE,
  maxit = 50L,
  tol = 1e-08,
  engine = c("batch", "gene"),
  batch.size = NULL,
  block.size = NULL,
  backend = c("cpu", "auto", "gpu"),
  BPPARAM = BiocParallel::SerialParam(),
  verbose = FALSE
)
```

Arguments

`Y`	a genes x cells matrix of integer counts (dense, sparse or DelayedArray; densified one gene block at a time). The negative binomial likelihood is undefined on non-integer values, so a non-integer assay is refused.
`W`	a cells x p numeric design matrix, or an `nbBlockDesign()`: a design whose per-group block is given in compact form and absorbed group by group, without the dense matrix ever being built (CPU only; see the Block designs section).
`alpha`	a genes x p matrix of starting coefficients, typically `fitNB()$alpha`.
`psi`	the starting per-gene dispersions (length `nrow(Y)`, or one value for every gene), typically `fitNB()$psi`.
`lambda.a`	the ridge penalty, a single value or one per column of `W` (for an `nbBlockDesign`, also one per column of `[X | Z]`, the `Z` values then used for every group). It is applied as given: each gene's objective subtracts `0.5 * sum(lambda.a * alpha^2)`, with no scaling by the number of cells or genes. The caller owns the scaling, so a fit made with a scaled penalty must pass the scaled values here.
`offset`	`NULL` (the default), a log-scale offset with one value per cell (the same for every gene), or a genes x cells matrix of them, added to every linear predictor with its coefficient fixed at 1: `log(mu) = W alpha + offset`. It must be finite. A `Matrix` offset is made a base matrix; a torch tensor is refused (pass it on the host; on a device the engine moves it). A matrix is held dense, so prefer a vector when the offset is the same for every gene.
`absorb`	which columns of `W` the per-gene Newton solver absorbs by a Schur complement (exact; it makes a wide indicator block cost one dense-column gram per iteration). `NULL` (the default) absorbs nothing. A logical over the columns of `W` makes each marked column its own 1x1 block; the marked columns must be

0/1 indicators that partition the cells. A per-column grouping (integer, factor or character ids, NA for a dense column) makes the columns sharing an id one block; no cell may load on two blocks. See nbNewtonSolver().

absorb.batch — the absorption for the shared-factor batched solver, which is what runs on a device (backend = "gpu", or "auto" with a GPU found): NULL or a logical over the columns of W. That solver absorbs 1x1 blocks only, so it cannot take a grouping with multi-column blocks. NULL (the default) passes a logical absorb through unchanged and sends a grouping to the dense batched solver (exact, only slower); a caller that knows the 1x1 subset of its grouping (e.g. the nested indicators inside a random-slope fit's per-sample blocks) passes it here. Not used on the CPU.

start.cols — a logical over the columns of W marking the indicator columns (such as cell-type intercepts) that the sane start fills with the gene's log mean over that column's cells, or NULL to put the overall log mean on the first column. For an nbBlockDesign it may also be given over the columns of [X | Z] (e.g. marking the block's intercept column), and then applies to every group's copy.

psi.method — how the dispersion is set at the converged mean: "profile" (profile maximum likelihood per gene) or "fixed" (the input psi is kept and only the mean is converged).

psi.range — the search interval for the profile dispersion, two increasing positive numbers. A gene whose optimum falls on either end keeps its input dispersion.

warm — logical; alpha and psi are an already converged fit at a nearby penalty. A warm polish is a few damped Newton steps at the held dispersion, with no dispersion search and no restart check.

maxit, tol — the Newton iteration cap and the relative log-likelihood tolerance, per gene.

engine — "batch" (the default) runs a batch of genes through each Newton step so they share every read of the design; "gene" is the per-gene reference implementation. Both reach the same optimum; the batched profile dispersion is a bisection and the per-gene one optimize(), so they agree on psi to optimize()'s tolerance.

batch.size — genes per batched Newton (engine = "batch"), or NULL to size it from a per-worker memory budget, options(SpaNorm.polish.mem.budget = <bytes>) (default 1e9; the older spiDE.polish.mem.budget is read as a fallback).

block.size — genes per block, the unit of densification and dispatch, or NULL for at least one block per worker and at most 2,000 genes.

backend — "cpu" (the default), "auto" or "gpu". A device needs engine = "batch" and float64, and forces serial dispatch.

BPPARAM — a BiocParallelParam over gene blocks. With more than one worker and the RhpcBLASctl package installed, each worker runs its BLAS and OpenMP single-threaded, because forked workers inherit the parent's thread count and oversubscribing the cores costs an order of magnitude per gene. Without RhpcBLASctl the workers keep the thread count they inherit; install it, or set the BLAS threads to 1 (e.g. OPENBLAS_NUM_THREADS=1) before starting R, when using several workers.

verbose — logical; report progress.

Details

With psi.method = "profile" the dispersion is then re-estimated by profile maximum likelihood at the converged mean and the mean re-polished, twice; a gene whose dispersion optimum sits on the search bound keeps its input dispersion (psi_bound). A gene whose starting point is degenerate (a fitted log-mean below -10 at a cell with a positive count) or whose Newton diverges restarts from a sane point: the log mean over the cells of each start.cols column, or the overall log mean on the first column. A gene that cannot be polished from either start keeps its input coefficients and dispersion, with polished = FALSE.

Once the shared fit is made the genes are independent, so this stage is exact when blocked: the counts are densified one gene block at a time (block.size) and the blocks are dispatched with BPPARAM. The dispersion moderation in fitNB() works across genes, which is why that fit must see the whole gene set and this one need not.

The integer-count check reads only the first 20 genes. It is a cheap guard against an assay that is non-integer throughout (a back-transform such as 2^logcounts - 1, on which every gene's dispersion would silently run to its upper bound), not a scan of every value: a non-integer count in a later gene is not detected.

Value

a list with alpha (genes x p), psi and loglik (the penalised log-likelihood at the returned fit, NA for a gene that was not polished), and polish, a data frame with one row per gene (row names from alpha): iterations (Newton steps), psi_fitnb (the input dispersion), restarted (the sane start was used), capped (a Newton pass hit maxit), singular (a singular information matrix), psi_bound (the dispersion optimum was on its search bound, so the input value was kept) and polished (FALSE when the gene kept its input fit).

Block designs

A design with a per-group block – a per-patient intercept and a per-patient library-size spline, say – is [X | Z_1 | ... | Z_G] with Z_g non-zero only on group g's cells. Passed densely, with absorb grouping the block columns by group, every Newton step still reads the whole n x (p_x + G q) matrix. Passed as W = nbBlockDesign(X, Z, block) it is never formed: the linear predictor and the score are O(n (p_x + q)), the per-gene gram O(n (p_x + q)^2), and each group's q x q block is absorbed by a Schur complement. The result is the dense grouped result (the same optimum; tested to 1e-8), with alpha in the layout nbBlockDesign() documents. absorb and absorb.batch must be NULL (the blocks are absorbed by construction), and backend must resolve to the CPU. A group whose block is rank-deficient (too few cells, a level with no cells, an unpenalised column constant over the group) is solved by a generalised inverse, see nbBlockDesign().

See Also

fitNB(), nbNewtonSolver().

Examples

```
set.seed(1)
W <- cbind(1, rnorm(200))
Y <- t(replicate(5, rnbinom(200, mu = exp(1 + 0.3 * W[, 2]), size = 3)))
```

```
fit <- fitNB(Y, W, verbose = FALSE, backend = "cpu")
pol <- polishNB(Y, W, fit$alpha, fit$psi)
pol$polish
```

---

polishSpaNorm *Converge each gene of a SpaNorm fit to its own optimum*

Description

SpaNorm() fits every gene in one shared IRLS loop, with a gene-averaged cell weight vector and an aggregate convergence criterion, so a gene can be left short of its own optimum (bright, spatially restricted genes most of all). polishSpaNorm() takes the stored fit and converges each gene separately under SpaNorm's own model, stores the polished fit, and rewrites the normalised assay from it.

Usage

```
polishSpaNorm(
  spe,
  adj.method = c("auto", "logpac", "pearson", "medbio", "meanbio"),
  scale.factor = 1,
  psi.method = c("fixed", "profile"),
  ls = c("fixed", "joint"),
  cells = c("all", "fit"),
  null = TRUE,
  maxit = 50L,
  tol = 1e-08,
  engine = c("batch", "gene"),
  batch.size = NULL,
  backend = c("cpu", "auto", "gpu"),
  BPPARAM = BiocParallel::SerialParam(),
  overwrite = FALSE,
  verbose = TRUE,
  assay = NULL,
  ...
)

## S4 method for signature 'SpatialExperiment'
polishSpaNorm(
  spe,
  adj.method = c("auto", "logpac", "pearson", "medbio", "meanbio"),
  scale.factor = 1,
  psi.method = c("fixed", "profile"),
  ls = c("fixed", "joint"),
  cells = c("all", "fit"),
  null = TRUE,
  maxit = 50L,
```

```
  tol = 1e-08,
  engine = c("batch", "gene"),
  batch.size = NULL,
  backend = c("cpu", "auto", "gpu"),
  BPPARAM = BiocParallel::SerialParam(),
  overwrite = FALSE,
  verbose = TRUE,
  assay = NULL,
  ...
)

## S4 method for signature 'Seurat'
polishSpaNorm(
  spe,
  adj.method = c("auto", "logpac", "pearson", "medbio", "meanbio"),
  scale.factor = 1,
  psi.method = c("fixed", "profile"),
  ls = c("fixed", "joint"),
  cells = c("all", "fit"),
  null = TRUE,
  maxit = 50L,
  tol = 1e-08,
  engine = c("batch", "gene"),
  batch.size = NULL,
  backend = c("cpu", "auto", "gpu"),
  BPPARAM = BiocParallel::SerialParam(),
  overwrite = FALSE,
  verbose = TRUE,
  assay = NULL,
  ...
)
```

Arguments

`spe`	a SpatialExperiment or Seurat object that `SpaNorm()` has been run on, with the raw counts in its 'counts' assay (layer, for Seurat).
`adj.method`	a character, specifying the method used to rewrite the normalised data from the polished fit (default 'auto'), as in `SpaNorm()`.
`scale.factor`	a numeric, specifying the scaling factor for the adjusted counts, as in `SpaNorm()`.
`psi.method`	how each gene's dispersion is set: `"fixed"` (the default) keeps the fit's dispersion and converges the mean under it; `"profile"` re-estimates it by profile maximum likelihood at the converged mean and re-polishes (see `polishNB()`).
`ls`	the library-size coefficient shared by every gene: `"fixed"` (the default) holds it at the fit's value; `"joint"` re-estimates it jointly with the per-gene coefficients, at the optimum of the total penalised likelihood over the polished genes (see Details).
`cells`	the cells/spots each gene is converged over: `"all"` (the default), or `"fit"` for the ones `SpaNorm()` sampled to fit the model.

`null` a logical, specifying whether to also polish the null model stored by `SpaNormSVG()` ('SpaNormNull'), when there is one (default TRUE). The SVG test compares the two fits, so they must be estimated the same way.

`maxit`, `tol` the Newton iteration cap and the relative log-likelihood tolerance, per gene.

`engine`, `batch.size`, `backend`
passed to `polishNB()`: the batched or per-gene Newton engine, genes per batch, and the compute backend.

`BPPARAM` a BiocParallelParam object over gene blocks, for the polish (see `polishNB()`) and the normalisation (default `BiocParallel::SerialParam()`).

`overwrite` a logical. A fit that is already polished is refused unless this is TRUE (default FALSE); with TRUE it is polished again, starting from the unpolished fit kept as 'SpaNormUnpolished' (and 'SpaNormNullUnpolished'), never from the polished one.

`verbose` a logical, specifying whether to show update messages (default TRUE).

`assay` a character, specifying the assay holding the raw counts for Seurat objects (default NULL uses the object's default assay). Ignored for SpatialExperiment objects.

`...` further arguments passed to `polishNB()`, such as `block.size` or `psi.range`.

Details

SpaNorm's mean for gene g in cell/spot i is

$$\log \mu_{gi} = \bar{\mu}_g + a_1 w_{i1} + \sum_{j \geq 2} W_{ij} \alpha_{gj},$$

where w_{i1} is the log library size and its coefficient a_1 is shared by every gene. The polish maximises each gene's penalised negative binomial log-likelihood by damped Newton (see `polishNB()`), with a_1 held at the fit's value (`ls = "fixed"`), $\bar{\mu}_g$ a per-gene unpenalised intercept, and the other coefficients ridge-penalised as `SpaNorm()` penalises them: by `lambda.a` for the biology and library-size splines, not at all for batch, times the number of cells/spots.

The polished coefficients are the optimum of the unwinsorised penalised likelihood. The normalisation (and `SpaNormSVG()`) still applies SpaNorm's usual winsorisation, exactly as for an unpolished fit: it forms the mean through `calculateMu()`, which by default winsorises the upper tail of each gene's log mean, and caps the dispersions at `exp(median(log psi) + 4 MAD(log psi))`.

A gene with no counts in the polished cells/spots has no finite optimum (its intercept runs to minus infinity), and nor has a gene with no counts in every polished cell/spot of one batch level (that level's unpenalised coefficient runs to minus infinity), so neither is polished: it keeps its input coefficients and dispersion, and its diagnostics row reads `polished = FALSE` with `iterations = 0` and `held_out` naming the reason, `"all-zero"` or `"zero-batch-level"` (`NA` for a gene that was polished). Genes held out for a batch level are counted in a warning. The levels are the groups of polished cells/spots that share one row of the model's unpenalised columns, the intercept and the batch indicators: for a batch factor, its levels; for several batch variables, each combination of levels present, which can hold out a gene whose optimum under an additive batch design is finite. Groups are formed only when every unpenalised column other than the intercept is a 0/1 indicator; when one is not (a numeric batch covariate, or a spline left unpenalised by `lambda.a = 0`), only the all-zero rule applies. A gene the Newton engine cannot converge from either start also keeps its

input fit (see polishNB()); a warning counts these genes, and how many had a singular information matrix. A batch matrix given with every level's indicator makes the batch columns collinear with the intercept, and so every gene's information singular: this is detected once, up front, by a QR of the unpenalised columns, and no gene is passed to the engine, so the outcome does not depend on rounding. The fit is marked polished either way.

With ls = "joint" the shared coefficient a_1 is moved to the joint optimum of the total penalised log-likelihood over the polished genes: after the per-gene polish at the fit's a_1, Newton steps on the profile log-likelihood of a_1 (each gene's own coefficients profiled out; the information is the Fisher information of the polish) alternate with a warm re-polish of every gene at the candidate value, and a step is halved until the total does not fall. It stops when the standardised score $|U|/\sqrt{I}$ (a1's distance from its optimum in units of its own profiled SE) is below 1e-3, or after 10 steps; maxit and tol are the per-gene Newton's, in the cold pass and every re-polish alike. With psi.method = "profile" each re-polish also re-profiles the dispersion. A gene whose information is singular at a step is left out of that step's score, information and total. If the profiled information is not positive (a_1 not identified: the log library size is collinear with the model's other terms), the fit is the ls = "fixed" polish, with a warning. The joint estimate weights genes by their information, so bright genes dominate it, where SpaNorm()'s a_1 is an unweighted mean over genes. Genes that are not polished (held out, or not convergeable) do not inform a_1; they keep their input $\bar{\mu}_g$, other coefficients and dispersion exactly, but take the new shared a_1, since it is one value for every gene, so their fitted mean shifts by $(a_1^{new} - a_1^{old})w_{i1}$. With no polished gene at all, a_1 is left at the fit's value with a warning.

The polished fit replaces 'SpaNorm' in the object's metadata (@misc for Seurat) and the input fit is kept as 'SpaNormUnpolished'. With null = TRUE a stored 'SpaNormNull' is polished with the same settings and the input kept as 'SpaNormNullUnpolished'. SVG results in rowData were computed from the unpolished fit, so they are removed with a warning; rerun SpaNormSVG(). The normalised assay ('logcounts', or the 'data' layer for Seurat) is rewritten from the polished fit.

The polished fit's polish slot (see isPolished()) holds settings (psi.method, ls, cells, maxit, tol, the penalty vector pen, the library-size coefficient before (a1.input) and after (a1), with psi.method = "profile" the dispersion's search interval psi.range (as passed through ..., else polishNB()'s default), and the SpaNorm version) and genes, one row per gene: polishNB()'s diagnostics plus loglik, the penalised log-likelihood over the polished cells at the returned fit (NA for a gene that was not polished), and held_out (see above). The fit's loglik slot is left as the shared fit's iteration trace. With ls = "joint", settings also holds ls.iterations (accepted steps on a_1), ls.maxit and ls.tol (the cap on those steps and the standardised-score stop, 10 and 1e-3, the latter in units of a1's profiled SE), ls.score (the final pooled score U), ls.se ($1/\sqrt{I}$, the profiled standard error of a_1), ls.singular (genes left out of U and I at some step because their information was singular) and ls.converged, and the per-gene iterations, capped and singular include the warm re-polishes.

The negative binomial likelihood is defined on counts only, so a non-integer counts assay (for example a back-transform such as 2^logcounts - 1) is refused.

Value

a SpatialExperiment or Seurat object holding the polished fit(s), with the normalised data rewritten in 'logcounts' or 'data', respectively.

See Also

SpaNorm(), polishNB(), isPolished().

Examples

```
data(HumanDLPFC)

top <- order(-Matrix::rowSums(SummarizedExperiment::assay(HumanDLPFC, "counts")))[1:50]
spe <- SpaNorm(HumanDLPFC[top, ], sample.p = 0.05, df.tps = 2, tol = 1e-2)
spe <- polishSpaNorm(spe)
isPolished(S4Vectors::metadata(spe)$SpaNorm)
```

---

qlDispersion *Per-gene quasi-likelihood dispersion*

Description

The edgeR v4 quasi-likelihood dispersion of each gene's NB GLM: the sum of its unit deviances, each rescaled by the observation's deviance moments, over the effective residual degrees of freedom. With `moments = "table"` (the default) the moments are evaluated once on a shared (log mu, log phi) table and interpolated, so the per-gene work is elementwise and runs on the accelerator when `y` and `mu` are torch tensors.

Usage

```
qlDispersion(
  y,
  mu,
  phi,
  design = NULL,
  p = NULL,
  prior = 1,
  leverage = c("trace", "exact"),
  moments = c("table", "grid", "cell"),
  ngrid = 256L,
  nphi = 64L,
  table = NULL
)
```

Arguments

`y`	counts, genes x cells (matrix or torch tensor).
`mu`	fitted means, same shape (matrix or torch tensor).
`phi`	NB dispersion: scalar or one value per gene.
`design`	the design matrix (cells x p); needed for `leverage = "exact"`, otherwise only its column count is used.

p	the number of design columns, in place of design.
prior	the average quasi-dispersion edgeR divides through by; 1 leaves the parameterisation alone.
leverage	"trace" spreads the p degrees of freedom evenly, which is exact to O(p/n) and is what n » p designs want; "exact" forms per-observation hat values (O(n p^2) per gene, CPU only) and is what reproduces edgeR on its own small-n designs.
moments	"table" evaluates the moments on a shared (log mu, log phi) table and interpolates onto every gene and cell (both backends); "grid" uses a per-gene log-mu grid; "cell" evaluates at every cell. The last two are CPU only.
ngrid	grid points along log mu.
nphi	maximum grid points along log phi for moments = "table".
table	a moments table from qlMomentTable, built once by a caller that scores genes in blocks; when NULL the table is built from this call's own range of means and dispersions.

Value

a list of per-gene deviance (adjusted), df (effective) and s2 = deviance / df.

Examples

```
set.seed(1)
mu <- matrix(exp(rnorm(400, 0, 1)), 8, 50)
y <- matrix(rnbinom(400, mu = mu, size = 2), 8, 50)
qlDispersion(y, mu, phi = 0.5, p = 3)
```

---

qlMomentTable *A shared table of NB deviance moments*

Description

The deviance moments w0 and w1 (see nbDevianceMoments) evaluated once on a uniform grid over log mean and log dispersion, for qlDispersion to interpolate onto every gene and cell. A caller that scores genes in blocks builds one table over the whole range and passes it to every block, so the result is invariant to how the genes are split; lookups outside the range are clamped to the edge, where the moments are asymptotically flat.

Usage

```
qlMomentTable(
  lmu_range,
  lphi_range,
  step_mu = 0.08,
  step_phi = 0.12,
  ngrid = NULL,
  nphi = NULL
)
```

Arguments

`lmu_range`	range of log means the table must cover, `c(lo, hi)`.
`lphi_range`	range of log dispersions, `c(lo, hi)`; a single value gives a one-row table.
`step_mu, step_phi`	grid spacing along log mean and log dispersion; bilinear interpolation error scales with the square of the spacing.
`ngrid, nphi`	explicit grid sizes, overriding the steps.

Value

a list holding the two moment matrices (log-phi rows, log-mu columns) and the grid geometry, for `qlDispersion(table = )`.

Examples

```
tab <- qlMomentTable(log(c(1e-6, 1e3)), log(c(0.1, 10)))
dim(tab$w1)
```

`resetGPUCache` *Clear the cached GPU device/memory-budget state*

Description

Clears the session-level cache of the resolved torch device and memory budget (set by `getBackendDevice`/`getGPUMemoryBud`
Useful in tests, or if the available device/memory changes within a session.

Usage

```
resetGPUCache()
```

Value

`NULL`, invisibly.

Examples

```
resetGPUCache()
```

rowSums_gpu *Row sums, GPU-aware*

Description

As rowSums2, but runs on the accelerator when mat is a torch tensor and an accelerator is active.

Usage

```
rowSums_gpu(mat)
```

Arguments

mat	a matrix or torch tensor.

Value

a numeric vector (or torch tensor) of row sums.

Examples

```
rowSums_gpu(matrix(1:6, 2, 3))
```

setGPUMemoryBudget *Set the accelerator memory budget for blocked fitting*

Description

Explicitly sets the session-cached GPU memory budget (see getGPUMemoryBudget), bypassing auto-detection for the rest of the session, or until resetGPUCache clears it. Useful when auto-detection is unreliable – e.g. on a MIG-partitioned GPU, where nvidia-smi-based detection may not resolve the correct instance on every driver/scheduler combination – so the correct budget (e.g. read from nvidia-smi -L's reported MIG instance size) can be set once rather than passed via gpu.mem.budget on every call.

Usage

```
setGPUMemoryBudget(bytes)
```

Arguments

bytes	a single positive number (bytes), or Inf to disable blocking outright.

Value

NULL, invisibly.

Examples

```
setGPUMemoryBudget(1e9)
getGPUMemoryBudget()
resetGPUCache()
```

SpaNorm *Spatially-dependent normalisation for spatial transcriptomics data*

Description

Performs normalisation of spatial transcriptomics data using spatially-dependent spot- and gene-specific size factors.

Usage

```
SpaNorm(
  spe,
  sample.p = 0.25,
  gene.model = c("nb"),
  adj.method = c("auto", "logpac", "pearson", "medbio", "meanbio"),
  scale.factor = 1,
  df.tps = 6,
  lambda.a = 1e-04,
  batch = NULL,
  tol = 1e-04,
  step.factor = 0.5,
  maxit.nb = 50,
  maxit.psi = 25,
  maxn.psi = 500,
  overwrite = FALSE,
  backend = c("auto", "cpu", "gpu"),
  BPPARAM = BiocParallel::SerialParam(),
  verbose = TRUE,
  assay = NULL,
  ...
)

## S4 method for signature 'SpatialExperiment'
SpaNorm(
  spe,
  sample.p = 0.25,
  gene.model = c("nb"),
  adj.method = c("auto", "logpac", "pearson", "medbio", "meanbio"),
  scale.factor = 1,
  df.tps = 6,
  lambda.a = 1e-04,
```

```
  batch = NULL,
  tol = 1e-04,
  step.factor = 0.5,
  maxit.nb = 50,
  maxit.psi = 25,
  maxn.psi = 500,
  overwrite = FALSE,
  backend = c("auto", "cpu", "gpu"),
  BPPARAM = BiocParallel::SerialParam(),
  verbose = TRUE,
  assay = NULL,
  ...
)

## S4 method for signature 'Seurat'
SpaNorm(
  spe,
  sample.p = 0.25,
  gene.model = c("nb"),
  adj.method = c("auto", "logpac", "pearson", "medbio", "meanbio"),
  scale.factor = 1,
  df.tps = 6,
  lambda.a = 1e-04,
  batch = NULL,
  tol = 1e-04,
  step.factor = 0.5,
  maxit.nb = 50,
  maxit.psi = 25,
  maxn.psi = 500,
  overwrite = FALSE,
  backend = c("auto", "cpu", "gpu"),
  BPPARAM = BiocParallel::SerialParam(),
  verbose = TRUE,
  assay = NULL,
  ...
)
```

Arguments

spe	a SpatialExperiment or Seurat object, with the count data stored in 'counts' or 'data' assays respectively.
sample.p	a numeric, specifying the (maximum) proportion of cells/spots to sample for model fitting (default is 0.25).
gene.model	a character, specifying the model to use for gene/protein abundances (default 'nb'). This should be 'nb' for count based datasets.
adj.method	a character, specifying the method to use to adjust the data (default 'auto', see details)

scale.factor	a numeric, specifying the sample-specific scaling factor to scale the adjusted count.
df.tps	a numeric, of length 1 or 2, specifying the maximum degrees of freedom for the thin-plate spline for the biology and library size effects, respectively (default is 6, see details).
lambda.a	a numeric, of length 1 or 2, specifying the smoothing parameter for regularizing regression coefficients (default is 0.0001, see details). Actual lambda.a used is lambda.a * ncol(spe).
batch	a vector or numeric matrix, specifying the batch design to regress out (default NULL, representing no batch effects). See details for more information on how to define this variable.
tol	a numeric, specifying the tolerance for convergence (default is 1e-4).
step.factor	a numeric, specifying the multiplicative factor to decrease IRLS step by when log-likelihood diverges (default is 0.5).
maxit.nb	a numeric, specifying the maximum number of IRLS iteration for estimating NB mean parameters for a given dispersion parameter (default is 50).
maxit.psi	a numeric, specifying the maximum number of IRLS iterations to estimate the dispersion parameter (default is 25).
maxn.psi	a numeric, specifying the maximum number of cells/spots to sample for dispersion estimation (default is 500).
overwrite	a logical, specifying whether to force recomputation and overwrite an existing fit (default FALSE). Note that if df.tps, batch, lambda.a, or gene.model are changed, the model is recomputed and overwritten.
backend	a character, specifying the backend to use for computations (default 'auto', see details). If 'gpu', GPU-based computations are used if available, otherwise CPU-based computations are used.
BPPARAM	a BiocParallelParam object specifying how to parallelise the normalisation step over gene-blocks (default `BiocParallel::SerialParam()`, i.e. no parallelisation). Pass e.g. `BiocParallel::MulticoreParam()` to speed up the logpac transform on large datasets.
verbose	a logical, specifying whether to show update messages (default TRUE).
assay	a character, specifying the assay to use for Seurat objects (default NULL uses the object's default assay). As SpaNorm models raw counts, set this to the assay holding the raw counts (e.g. 'Spatial') when the default assay is a transformed one such as 'SCT'. Ignored for SpatialExperiment objects.
...	other parameters fitting parameters.

Details

SpaNorm works by first fitting a spatial regression model for library size to the data. Normalised data can then be computed using various adjustment approaches. When a negative binomial gene-model is used, the data can be adjusted using the following approaches: 'logpac', 'pearson', 'medbio', and 'meanbio'.

The `df.tps` parameter specifies the degrees of freedom for the thin-plate spline. If only 1 value is provided, it specifies the degrees of freedom of the biology with the degrees of freedom of the

library size being half of that. If 2 values are provided, the first value specifies the degrees of freedom of the biology and the second value specifies the degrees of freedom of the library size. For rectangular tissues, df.tps specifies the degrees of freedom along the length, with the degrees of freedom along the width calculated ceiling(width / length * df.tps).

Similarly, the `lambda.a` parameter specifies the smoothing parameter for regularizing regression coefficients. If only 1 value is provided, it specifies the lambda.a for both the biology and library size functions. If 2 values are provided, the first value specifies the lambda.a for the biology and the second value specifies the lambda.a for the library size. Batch effects are not regularised.

If the counts assay is a `DelayedArray` (e.g. disk-backed via `HDF5Array`), the normalisation step is automatically performed block-wise so the full matrix is never realised in memory at once; the results are identical to the in-memory path. The block size follows `DelayedArray`'s global auto block size, which can be tuned with `DelayedArray::setAutoBlockSize()`.

Batch effects can be specified using the `batch` parameter. If this parameter is a vector, a design matrix will be created within the function using `model.matrix`. If a custom design is provided in the form of a numeric matrix, this should ideally be created using `model.matrix`. The batch matrix should be created with an intercept term. The SpaNorm function will automatically detect the intercept term and remove the relevant column. Alternatively, users can subset the model matrix to remove this column manually. Please note that the model formula should include the intercept term and that the intercept column should be subset out after.

Value

a SpatialExperiment or Seurat object with the adjusted data stored in 'logcounts' or 'data', respectively.

Examples

```
data(HumanDLPFC)

SpaNorm(HumanDLPFC, sample.p = 0.05, df.tps = 2, tol = 1e-2)
```

`SpaNormFit` *An S4 class to store a SpaNorm model fit*

Description

An S4 class to store a SpaNorm model fit

Usage

```
## S4 method for signature 'SpaNormFit'
x$name
```

Arguments

`x`	an object of class SpaNormFit.
`name`	a character, specifying the name of the slot to retrieve.

Value

Return value varies depending on method.

Slots

`ngenes` a numeric, specifying the number of genes in the dataset.

`ncells` a numeric, specifying the number of cells/spots in the dataset.

`gene.model` a character, specifying the gene-specific model to used (see `getGeneModels()`).

`df.tps` an integer, specifying the degrees of freedom to used for the thin plate spline.

`sample.p` a numeric, specifying the proportion of samples used to approximated the model.

`lambda.a` a numeric, specifying the shinkage parameter used.

`batch` a vector or matrix, specifying the batch design used (if any).

`W` a matrix, specifying the covariate matrix of the linear model.

`alpha` a matrix, specifying the coefficients of the linear model.

`gmean` a numeric, specifying the mean estimate for each gene in the linear model.

`psi` a numeric, specifying the over-dispersion parameter for each gene if a negative binomial model was used (or a vector of NAs if another gene model is used).

`wtype` a factor, specifying the covariate types of columns in the covariate matrix, W. These could be "biology", "ls", or "batch".

`loglik` a numeric, specifying the log-likelihood of the model at each external iteration.

`sampling` a factor, specifying the cells/spots used for dispersion estimation ('dispersion'), GLM fitting ('glm' and 'dispersion'), all other cells/spots ('all').

`polish` a list, specifying the convergence diagnostics from the per-gene polish stage (see `polishSpaNorm()`): `settings` (the polish call's configuration) and `genes` (a per-gene diagnostics data frame). An empty list for a fit that has not been polished; see `isPolished()`.

Examples

```
example(SpaNorm)
```

`SpaNormPCA` *GLM-based (SpaNorm) PCA*

Description

GLM-based PCA using the SpaNorm model. The null model is considered to consist of the library size effects, batch effects, and the gene mean. GLM-PCA is approximated by regressing the null model from the data, and performing PCA on the residuals (Pearson or deviance).

Usage

```
SpaNormPCA(
  spe,
  nsvgs = 3000,
  ncomponents = 50,
  svg.fdr = 1,
  BSPARAM = bsparam(),
  BPPARAM = SerialParam(),
  residuals = c("deviance", "pearson"),
  name = "PCA"
)

## S4 method for signature 'SpatialExperiment'
SpaNormPCA(
  spe,
  nsvgs = 3000,
  ncomponents = 50,
  svg.fdr = 1,
  BSPARAM = bsparam(),
  BPPARAM = SerialParam(),
  residuals = c("deviance", "pearson"),
  name = "PCA"
)

## S4 method for signature 'Seurat'
SpaNormPCA(
  spe,
  nsvgs = 3000,
  ncomponents = 50,
  svg.fdr = 1,
  BSPARAM = bsparam(),
  BPPARAM = SerialParam(),
  residuals = c("deviance", "pearson"),
  name = "PCA"
)
```

Arguments

spe	a SpatialExperiment or Seurat object, with the count data stored in 'counts' or 'data' assays respectively, and a SpaNorm model fit.
nsvgs	the number of SVGs to use for PCA.
ncomponents	the number of components to compute.
svg.fdr	the FDR threshold for SVG calling.
BSPARAM	a BiocSingularParam object specifying which algorithm should be used to perform the PCA.
BPPARAM	a BiocParallelParam object specifying whether the PCA should be parallelized.
residuals	the type of residuals to use for PCA. Either "deviance" (default) or "pearson".

name the name of the reducedDim to store the PCA results.

Details

SpaNorm PCA works by using the SpaNorm model fit for data normalisation to approximate a GLM-based PCA as described in Townes et al. (Genome Biology, 2019). The model used for normalisation represents the library size effects and the gene mean. Regressing these covariates, we remain with the deviance or Pearson residuals, upon which PCA can be performed to approximate the GLM-PCA.

Value

a SpatialExperiment or Seurat object with PCA results. For SpatialExperiment objects, these are stored in the reducedDims.

Examples

```
library(SpatialExperiment)
library(ggplot2)

data(HumanDLPFC)

HumanDLPFC = SpaNorm(HumanDLPFC, sample.p = 0.05, df.tps = 2, tol = 1e-2)
HumanDLPFC = SpaNormSVG(HumanDLPFC)
HumanDLPFC = SpaNormPCA(HumanDLPFC)
reducedDims(HumanDLPFC)
```

SpaNormSVG *Model-based spatially variable gene (SVG) calling*

Description

Spatially variable gene (SVG) calling using the SpaNorm model.

Usage

```
SpaNormSVG(spe, backend = c("auto", "cpu", "gpu"), verbose = TRUE)

## S4 method for signature 'SpatialExperiment'
SpaNormSVG(spe, backend = c("auto", "cpu", "gpu"), verbose = TRUE)

## S4 method for signature 'Seurat'
SpaNormSVG(spe, backend = c("auto", "cpu", "gpu"), verbose = TRUE)
```

Arguments

spe	a SpatialExperiment or Seurat object, with the count data stored in 'counts' or 'data' assays respectively, and a SpaNorm model fit.
backend	a character, specifying the backend to use for computations. Options are "auto" (default), "cpu", or "gpu". If "auto", it will use GPU if available, otherwise CPU.
verbose	a logical, specifying whether to show update messages (default TRUE).

Details

SpaNorm SVG calling works by using the SpaNorm model fit for data normalisation to perform a likelihood ratio test (LRT). The model used for normalisation is considered to be the full model. A second nested model is fit without the splines representing biology. These nested models are then compared using a LRT to identify genes where the splines representing biology contain strong signal.

Value

a SpatialExperiment or Seurat object with F-statistics, false discovery rates (FDRs). For SpatialExperiment objects, these are stored in the rowData.

Examples

```
library(SpatialExperiment)
library(ggplot2)

data(HumanDLPFC)

HumanDLPFC = SpaNorm(HumanDLPFC, sample.p = 0.05, df.tps = 2, tol = 1e-2)
HumanDLPFC = SpaNormSVG(HumanDLPFC)
head(rowData(HumanDLPFC))
```

---

tcrossprod_gpu *Tensor-aware* tcrossprod

Description

Computes `x %*% t(y)` (or `x %*% t(x)` if `y` is `NULL`) using the accelerator if either argument is a torch tensor and an accelerator is active; falls back to `tcrossprod` otherwise. Exposed so downstream packages (e.g. spiDE) can build their own GPU-blocked linear algebra with the same dual (tensor-or-matrix) semantics SpaNorm's own fitting uses.

Usage

```
tcrossprod_gpu(x, y = NULL)
```

Arguments

x	a matrix or torch tensor.
y	a matrix or torch tensor, or NULL (default).

Value

x %*% t(y), as a matrix or torch tensor matching the input.

Examples

```
m <- matrix(rnorm(12), nrow = 3)
tcrossprod_gpu(m)
```

---

toGPUMatrix *Convert a matrix to a torch tensor on the active backend*

Description

Converts a base R matrix to a torch tensor on the resolved backend device and dtype (getBackendDevice/getBackendDtype) if an accelerator is available and requested; otherwise returns mat unchanged (already a tensor, or backend = "cpu", or no accelerator present). Exposed so downstream packages (e.g. spiDE) share the exact same device/dtype resolution SpaNorm's own fitting uses.

Usage

```
toGPUMatrix(mat, ..., backend = c("auto", "cpu", "gpu"))
```

Arguments

mat	a base R matrix (or an existing torch tensor, returned as-is).
...	ignored.
backend	one of "auto" (default), "cpu" or "gpu".

Value

a torch tensor, or mat unchanged.

Examples

```
toGPUMatrix(matrix(1:4, 2, 2), backend = "cpu")
```

toGPUVector *Convert a vector to a torch tensor on the active backend*

Description

As toGPUMatrix, for a numeric vector; optionally broadcasts a length-1 vec to length n.

Usage

```
toGPUVector(vec, n = NULL, backend = c("auto", "cpu", "gpu"))
```

Arguments

vec	a numeric vector (or an existing torch tensor, returned as-is).
n	if supplied, the target length; a length-1 vec is broadcast to length n.
backend	one of "auto" (default), "cpu" or "gpu".

Value

a torch tensor, or vec unchanged.

Examples

```
toGPUVector(1:4, backend = "cpu")
```

topSVGs *Export top SVG results to a data frame*

Description

Export top SVG results to a data frame

Usage

```
topSVGs(spe, n = 10, fdr = 1)
```

Arguments

spe	a SpatialExperiment object with SVG results from SpaNormSVG.
n	a numeric, specifying the number of top SVGs to call.
fdr	a numeric, specifying the false discovery rate (FDR) threshold for calling SVGs.

Value

A data frame containing the top SVGs from F-test results including F-statistics, p-values and FDR.

Examples

```
library(SpatialExperiment)
library(ggplot2)

data(HumanDLPFC)

HumanDLPFC = SpaNorm(HumanDLPFC, sample.p = 0.05, df.tps = 2, tol = 1e-2)
HumanDLPFC = SpaNormSVG(HumanDLPFC)
topSVGs = topSVGs(HumanDLPFC, n = 10)
```

---

toRMatrix *Convert a torch tensor (or R object) to a base R matrix*

Description

Converts a torch tensor to a base R matrix (moving it to the CPU first if needed); non-tensor input is coerced via `as.matrix`. Exposed so downstream packages (e.g. spiDE) doing their own GPU-blocked computation can convert results back to plain R without depending on torch internals.

Usage

```
toRMatrix(x)
```

Arguments

`x`	a torch tensor or an R object coercible via `as.matrix`.

Value

a base R matrix.

Examples

```
toRMatrix(matrix(1:4, 2, 2))
```

---

tpsBasis *Thin-plate-style tensor spline basis over spatial coordinates*

Description

The spatial basis SpaNorm builds its biology and library-size functions from, exported so that a downstream model can use the same smooth functions of position. It is the tensor product of a natural cubic spline basis (`splines::ns()`) along each axis, with its columns centred.

Usage

```
tpsBasis(x, y, df = 6, ref.x = x, ref.y = y, center = TRUE)
```

Arguments

x, y	numeric vectors of the coordinates to evaluate the basis at.
df	the degrees of freedom. A single positive integer applies SpaNorm's aspect-ratio rule to the reference ranges: with gap = max(range_x, range_y) / df, the axes get ceiling(range_x / gap) and ceiling(range_y / gap) degrees of freedom, so the longer axis gets df. Two positive integers give the x and y degrees of freedom directly. The basis has df_x * df_y columns (for example 9 at df = c(3, 3)).
ref.x, ref.y	numeric vectors of the reference coordinates that define the knots, the per-axis degrees of freedom and the centring (for example the whole tissue section). Default: x and y.
center	logical; subtract the reference column means (default TRUE).

Details

The basis is defined by a set of *reference* coordinates (by default the evaluation coordinates themselves) and can be evaluated anywhere:

- the per-axis degrees of freedom come from the reference ranges (see df);
- each axis's interior knots sit at the quantiles of the reference coordinates along that axis and its boundary knots at their range, exactly as splines::ns(ref, df =) places them; a point outside the reference range is extrapolated linearly, as natural splines do;
- **centring**: when center = TRUE, every column has the mean of that column over the *reference* cells subtracted (the scale(center = TRUE, scale = FALSE) of the reference basis). The columns therefore average to zero over the reference cells, not over the evaluation cells: evaluated at a subset of the reference, each column is the same function of position that the whole reference sees, so two subsets of one section share one basis. The subtracted means are returned in the "scaled:center" attribute.

With the default reference (ref.x = x, ref.y = y) and a single df, the result is identical to the internal basis SpaNorm's fit uses (bs.tps()). Note that SpaNorm() rescales each axis to unit range before building its basis, so there the aspect-ratio rule below gives df knots-worth of degrees of freedom on both axes; to reproduce that on raw coordinates pass df = c(df, df).

Value

a length(x) by df_x * df_y numeric matrix. Column (i - 1) * df_y + j is the product of the i-th x-axis and the j-th y-axis natural spline. Attributes: "df.tps", the per-axis degrees of freedom c(df_x, df_y), and, when centred, "scaled:center", the subtracted reference means.

Examples

```
set.seed(1)
x <- runif(200, 0, 2)
y <- runif(200, 0, 1)
B <- tpsBasis(x, y, df = 4)          # 4 x 2 degrees of freedom, 8 columns
dim(B)
# the same basis, built on the whole section and evaluated on one region
sub <- x < 0.5
Bs <- tpsBasis(x[sub], y[sub], df = 4, ref.x = x, ref.y = y)
all.equal(Bs, B[sub, ], check.attributes = FALSE)
```

Index

∗ **datasets**
    HumanDLPFC, 12
$,SpaNormFit-method *(*SpaNormFit*)*, 38

add_vec_mat_gpu, 3, *16, 17*
as.matrix, *45*
as.matrix.nbBlockDesign *(*nbBlockDesign*)*, 17

calculateMu, 3
calculateMu(), *29*
checkGPU, 5

diag, *5*
diag_mat, 5
dim.nbBlockDesign *(*nbBlockDesign*)*, 17
dimnames.nbBlockDesign *(*nbBlockDesign*)*, 17
dnbinom, *6*
dnbinom_gpu, 6

fastSizeFactors, 6
fastSizeFactors,SpatialExperiment-method *(*fastSizeFactors*)*, 6
filterGenes, 7
filterGenes,Seurat-method *(*filterGenes*)*, 7
filterGenes,SpatialExperiment-method *(*filterGenes*)*, 7
fitNB, *3, 4*, 8, *23*, *26*

getBackendDevice, *5*, 10, *33*, *43*
getBackendDtype, 10, *11*, *43*
getGPUMemoryBudget, 11, *33, 34*
gpuDtypeBytes, 11

hasBadValues, 12
HumanDLPFC, 12

invert_mat, 13, *14*
invert_mat_batched, 14
is_torch_tensor, 15
isPolished, 15
isPolished(), *30, 31*

matmul_gpu, 16
mult_vec_mat_gpu, 16

nbBlockDesign, 17, *24*, *26*
nbDevianceMoments, 19, *32*
nbNewtonSolver, *18, 19*, *25, 26*
nbProfilePsi, *18*
nbUnitDeviance, 20

plotCovariate, 21
plotSpatial, *21*, 22
polishNB, *18, 19*, 23
polishNB(), *28–31*
polishSpaNorm, 27
polishSpaNorm,Seurat-method *(*polishSpaNorm*)*, 27
polishSpaNorm,SpatialExperiment-method *(*polishSpaNorm*)*, 27
print.nbBlockDesign *(*nbBlockDesign*)*, 17

qlDispersion, 31, *32*
qlMomentTable, 32, *32*

resetGPUCache, *10, 11*, 33, *34*
rowSums2, *34*
rowSums_gpu, 34

setGPUMemoryBudget, 34
SpaNorm, 35
SpaNorm(), *27–31*
SpaNorm,Seurat-method *(*SpaNorm*)*, 35
SpaNorm,SpatialExperiment-method *(*SpaNorm*)*, 35
SpaNormFit, 38
SpaNormFit-class *(*SpaNormFit*)*, 38
SpaNormPCA, 39

SpaNormPCA,Seurat-method *(*SpaNormPCA*)*, 39
SpaNormPCA,SpatialExperiment-method *(*SpaNormPCA*)*, 39
SpaNormSVG, 41
SpaNormSVG(), *29, 30*
SpaNormSVG,Seurat-method *(*SpaNormSVG*)*, 41
SpaNormSVG,SpatialExperiment-method *(*SpaNormSVG*)*, 41

tcrossprod, *42*
tcrossprod_gpu, 42
toGPUMatrix, *4*, 43, *44*
toGPUVector, *4*, 44
topSVGs, 44
toRMatrix, *4*, 45
tpsBasis, *19*, 45