Package ‘lgspline’

March 15, 2026

Type Package

Title Lagrangian Multiplier Smoothing Splines for Smooth Function Estimation

Version 1.0.1

Description Implements Lagrangian multiplier smoothing splines for flexible nonparametric regression and function estimation. Provides tools for fitting, prediction, and inference using a constrained optimization approach to enforce smoothness. Supports generalized linear models, Weibull accelerated failure time (AFT) models, quadratic programming constraints, and customizable working-correlation structures, with options for parallel fitting. The core spline construction builds on Ezhov et al. (2018) <doi:10.1515/jag-2017-0029>. Quadratic-programming and SQP details follow Goldfarb & Idnani (1983) <doi:10.1007/BF02591962> and Nocedal & Wright (2006) <doi:10.1007/978-0-387-40065-5>. For smoothing spline and penalized spline background, see Wahba (1990) <doi:10.1137/1.9781611970128> and Wood (2017) <doi:10.1201/9781315370279>. For variance-component and correlation-parameter estimation, see Searle et al. (2006) <ISBN:978-0470009598>. The default multivariate partitioning step uses k-means clustering as in MacQueen (1967).

License MIT + file LICENSE

Language en-US

Depends R (>= 3.5.0)

Imports Rcpp (>= 1.0.7), RcppArmadillo, FNN, RColorBrewer, plotly, quadprog, methods, stats

LinkingTo Rcpp, RcppArmadillo

Suggests testthat (>= 3.0.0), spelling, knitr, rmarkdown, parallel, survival, MASS, graphics

URL `https://github.com/matthewlouisdavisBioStat/lgspline`

BugReports `https://github.com/matthewlouisdavisBioStat/lgspline/issues`

Encoding UTF-8

RoxygenNote 7.3.2

Config/testthat/edition 3

NeedsCompilation yes

Author Matthew Davis [aut, cre] (ORCID:
<https://orcid.org/0000-0001-9714-1018>)

Maintainer Matthew Davis <matthewlouisdavis@gmail.com>

Repository CRAN

Date/Publication 2026-03-15 12:10:07 UTC

Contents

coef.lgspline . . . 3
coef.wald_lgspline . . . 4
confint.lgspline . . . 5
confint.wald_lgspline . . . 5
cox_dispersion_function . . . 6
cox_family . . . 7
cox_glm_weight_function . . . 7
cox_helpers . . . 9
cox_qp_score_function . . . 9
cox_schur_correction . . . 10
create_onehot . . . 11
Details . . . 12
equation . . . 34
find_extremum . . . 38
generate_posterior . . . 40
generate_posterior_correlation . . . 43
get_B_verification_examples . . . 46
integrate . . . 48
integrate.lgspline . . . 48
leave_one_out . . . 50
lgspline . . . 51
lgspline_cox . . . 83
lgspline_negbin . . . 84
lgspline_weibull . . . 86
logLik.lgspline . . . 87
loglik_cox . . . 89
loglik_negbin . . . 90
loglik_weibull . . . 91
matinvsqrt . . . 92
matsqrt . . . 94
negbin_dispersion_function . . . 95
negbin_family . . . 96
negbin_glm_weight_function . . . 97
negbin_helpers . . . 98
negbin_qp_score_function . . . 98
negbin_schur_correction . . . 99

negbin_theta 100
plot.lgspline 101
plot.wald_lgspline 104
predict.lgspline 105
print.lgspline 107
print.summary.lgspline 107
print.wald_lgspline 108
prior_loglik 109
process_qp 110
reml_grad_from_dV 114
std 116
summary.lgspline 117
summary.wald_lgspline 118
wald_univariate 118
weibull_dispersion_function 119
weibull_family 121
weibull_glm_weight_function 123
weibull_qp_score_function 124
weibull_scale 126
weibull_schur_correction 127

Index **131**

`coef.lgspline` *Extract Coefficients from a Fitted lgspline*

Description

Returns the per-partition polynomial coefficient lists from a fitted lgspline model.

Usage

```
## S3 method for class 'lgspline'
coef(object, ...)
```

Arguments

`object`	A fitted lgspline model object.
`...`	Not used.

Details

Coefficient names reflect the polynomial expansion terms, e.g.:

- intercept
- v: linear term for predictor v
- v_^2: quadratic term

- v^3: cubic term
- _v_x_w_: two-way interaction

Column/variable names replace numeric indices when available.

To get all coefficients as a single matrix: `Reduce('cbind', coef(model_fit))`.

Value

A list of per-partition coefficient vectors. Returns NULL with a warning if `object$B` is not found.

See Also

`lgspline`

Examples

```
set.seed(1234)
t <- runif(1000, -10, 10)
y <- 2*sin(t) + -0.06*t^2 + rnorm(length(t))
model_fit <- lgspline(t, y)

coefficients <- coef(model_fit)
print(coefficients[[1]])
print(Reduce('cbind', coefficients))
```

`coef.wald_lgspline` *Extract Coefficients from a wald_lgspline Object*

Description

Extract Coefficients from a wald_lgspline Object

Usage

```
## S3 method for class 'wald_lgspline'
coef(object, ...)
```

Arguments

`object`	A `"wald_lgspline"` object.
`...`	Not used.

Value

Named numeric vector of coefficient estimates, or NULL if no estimate column is available.

`confint.lgspline` *Confidence Intervals for lgspline Coefficients*

Description

Wald-based confidence intervals for regression coefficients and, when available, correlation parameters (on the working scale).

Usage

```
## S3 method for class 'lgspline'
confint(object, parm, level = 0.95, ...)
```

Arguments

`object`	A fitted lgspline object with `return_varcovmat = TRUE`.
`parm`	Optional vector of parameter indices or names. Default returns all regression parameters; working-scale correlation parameters are appended when available.
`level`	Confidence level. Default 0.95.
`...`	Additional arguments passed to `wald_univariate`.

Details

For Gaussian identity-link models, t-distribution quantiles are used with effective degrees of freedom $N - \mathrm{trace}(\mathbf{XUGX}^\top)$. All other families use normal quantiles.

Correlation parameter intervals (if `VhalfInv_params_estimates` and `VhalfInv_params_vcov` are present) are computed on the unbounded working scale via a Wald interval.

Value

A matrix with columns giving lower and upper confidence limits, named e.g. `2.5 %` and `97.5 %` for 95% intervals. When available, rows for working-scale correlation parameters are appended after the regression coefficients.

`confint.wald_lgspline` *Extract Confidence Intervals from a wald_lgspline Object*

Description

Extract Confidence Intervals from a wald_lgspline Object

Usage

```
## S3 method for class 'wald_lgspline'
confint(object, parm = NULL, level = NULL, ...)
```

Arguments

`object`	A `"wald_lgspline"` object.
`parm`	Parameter specification (ignored; all returned).
`level`	Confidence level (ignored; uses the object's critical value).
`...`	Not used.

Value

Matrix with columns lower and upper, or NULL if confidence limits are not available.

`cox_dispersion_function` *Cox PH Dispersion Function*

Description

Returns 1 unconditionally. Cox PH has no dispersion parameter; this function exists solely for interface compatibility with lgspline's `dispersion_function` argument.

Usage

```
cox_dispersion_function(
  mu,
  y,
  order_indices,
  family,
  observation_weights,
  VhalfInv,
  ...
)
```

Arguments

`mu`	Predicted values.
`y`	Observed survival times.
`order_indices`	Observation indices.
`family`	Family object.
`observation_weights`	Observation weights.
`VhalfInv`	Inverse square root of correlation matrix.
`...`	Additional arguments (including `status`).

Value

Scalar 1.

`cox_family` *Cox Proportional Hazards Family for lgspline*

Description

Creates a family-like object for Cox PH models. The link function is `log` (the linear predictor is log-relative-hazard), but unlike standard GLM families there is no dispersion parameter and no closed-form mean-variance relationship.

The family provides `$loglik` and `$aic` methods compatible with `logLik.lgspline`.

Usage

```
cox_family()
```

Details

Cox PH is semiparametric: the baseline hazard is unspecified. The partial log-likelihood depends only on the order of event times and the linear predictor $\eta = \mathbf{X}\beta$. Consequently:

- No dispersion parameter is estimated (`sigmasq_tilde` is fixed at 1).
- `dev.resids` returns martingale-style residuals for GCV tuning compatibility.
- The response variable is survival time (positive), and the link is `log`.

Value

A list with family components used by `lgspline`.

Examples

```
fam <- cox_family()
fam$family
fam$link
```

`cox_glm_weight_function` *Cox PH GLM Weight Function*

Description

Computes working weights for the Cox PH information matrix, used by lgspline when updating $\mathbf{G}$ after obtaining constrained estimates. The weights are a diagonal approximation built from the Breslow tied-event information contributions.

Usage

```
cox_glm_weight_function(
  mu,
  y,
  order_indices,
  family,
  dispersion,
  observation_weights,
  status
)
```

Arguments

`mu`	Predicted values (exp(eta), i.e., relative hazard).
`y`	Observed survival times.
`order_indices`	Observation indices in partition order.
`family`	Cox family object (unused, for interface compatibility).
`dispersion`	Dispersion parameter (fixed at 1 for Cox PH).
`observation_weights`	Observation weights.
`status`	Event indicator (1 = event, 0 = censored).

Details

For a tied event-time block g, the diagonal approximation uses

$$W_{jj}^{(g)} = d_g^{(w)} \frac{h_j}{S_g}\Big(1 - \frac{h_j}{S_g}\Big), \qquad j \in R_g$$

where $h_j = w_j \exp(\eta_j)$, $S_g = \sum_{k \in R_g} h_k$, and $d_g^{(w)} = \sum_{i \in D_g} w_i$.

When the natural weights are degenerate (all zero or non-finite), falls back to a vector of ones.

Value

Numeric vector of working weights, length N.

Examples

```
## Used internally by lgspline; see cox_helpers examples below.
```

`cox_helpers` *Cox Proportional Hazards Helpers for lgspline*

Description

Functions for fitting Cox proportional hazards regression models within the lgspline framework. Analogous to the Weibull AFT helpers, these provide the partial log-likelihood, score, information, and all interface functions needed by lgspline's unconstrained fitting, penalty tuning, and inference machinery.

`cox_qp_score_function` *Cox PH Score Function for Quadratic Programming and Blockfit*

Description

Computes the score (gradient of partial log-likelihood) in the format expected by lgspline's `qp_score_function` interface. The block- diagonal design matrix $\mathbf{X}$ and response $\mathbf{y}$ are in partition order; this function internally sorts by event time, computes the Cox score using the Breslow approximation for tied event times, and returns the result in the original partition order.

Usage

```
cox_qp_score_function(
  X,
  y,
  mu,
  order_list,
  dispersion,
  VhalfInv,
  observation_weights,
  status
)
```

Arguments

`X`	Block-diagonal design matrix (N x P).
`y`	Response vector (survival times, N x 1).
`mu`	Predicted values (N x 1), same order as X and y.
`order_list`	List of observation indices per partition.
`dispersion`	Dispersion (fixed at 1).
`VhalfInv`	Inverse square root correlation matrix (NULL for independent observations).
`observation_weights`	Observation weights.
`status`	Event indicator (1 = event, 0 = censored).

Value

Numeric column vector of length P (gradient w.r.t. coefficients).

---

cox_schur_correction *Cox PH Schur Correction*

Description

Returns zero corrections for all partitions. Cox PH has no nuisance dispersion parameter, so no Schur complement correction to the information matrix is needed. This function exists for interface compatibility with lgspline's `schur_correction_function`.

Usage

```
cox_schur_correction(
  X,
  y,
  B,
  dispersion,
  order_list,
  K,
  family,
  observation_weights,
  ...
)
```

Arguments

`X`	List of partition design matrices.
`y`	List of partition response vectors.
`B`	List of partition coefficient vectors.
`dispersion`	Dispersion (fixed at 1).
`order_list`	List of observation indices per partition.
`K`	Number of knots.
`family`	Family object.
`observation_weights`	Observation weights.
`...`	Additional arguments.

Value

List of K+1 zeros.

create_onehot *Create One-Hot Encoded Matrix*

Description

Converts a categorical vector into a one-hot encoded matrix where each unique value becomes a binary column.

Usage

```
create_onehot(x, drop_first = FALSE)
```

Arguments

x	A vector containing categorical values (factors, character, etc.)
drop_first	Logical; if TRUE and more than one dummy column is created, drop the first column.

Details

The function creates dummy variables for each unique value in the input vector using `model.matrix()` with dummy-intercept coding. Column names are cleaned by removing the 'x' prefix added by `model.matrix()`.

Value

A data frame containing the one-hot encoded binary columns with cleaned column names

Examples

```
## lgspline will not accept this format of "catvar", because inputting data
# this way can cause difficult-to-diagnose issues in formula parsing
# all variables must be numeric
df <- data.frame(numvar = rnorm(100),
                 catvar = rep(LETTERS[1:4],
                              25))
print(head(df))

## Instead, replace with dummy-intercept coding by
# 1) applying one-hot encoding
# 2) dropping the first column
# 3) appending to our data

dummy_intercept_coding <- create_onehot(df$catvar)[,-1]
df$catvar <- NULL
df <- cbind(df, dummy_intercept_coding)
print(head(df))
```

`Details` *Lagrangian Multiplier Smoothing Splines: Mathematical Details*

Description

This document provides the mathematical and implementation details for Lagrangian Multiplier Smoothing Splines as implemented in **lgspline**.

This package provides exhaustive resources for fitting multivariate smoothing smoothing splines with a monomial basis and analytical form cubic smoothing spline penalty.

The material is presented such that a programmer or statistician of reasonable experience and background can understand and implement the procedure from scratch, and also potentially critique some of the modelling choices that went into designing this package.

Informally, **lgspline** answers the following question: How can we best adapt a useful functionality of basis splines, under the alternative interpretation of smoothing as explicit external constraints instead?

The obvious benefit is a much more flexible and interpretable final model that for non-experienced users is simply easier to understand without post-hoc processing, and for experienced users can be used to customize models more easily.

The drawback is that the interpretation of constraints as external adds a new layer of complexity to each step of the model fitting process, whereas for implicit design matrix construction these complications are bypassed.

While it is true a B-spline can always be converted back into monomial form, tensor-product splines that generalize this to multiple dimensions often explodes the number and degree of interaction terms, the conversion may not be computationally stable, and it is not available in standard software.

Statistical Problem Formulation

Consider an $N \times q$ matrix of predictors $\mathbf{T} = (\mathbf{t}_1, \dots, \mathbf{t}_N)^\top$ and an $N \times 1$ response vector $\mathbf{y} = (y_1, \dots, y_N)^\top$. We assume the relationship follows a generalized linear model with unknown smooth function f:

$$y_i \sim \mathcal{D}(g^{-1}(f(\mathbf{t}_i)),\, \sigma^2)$$

where $\mathcal{D}$ is a distribution (e.g. exponential family or related) with mean $\mu_i = g^{-1}(f(\mathbf{t}_i))$, link function $g(\cdot)$, and dispersion parameter σ^2. For Gaussian response with identity link, observations are independently distributed as $y_i \mid \mathbf{t}_i, \sigma^2 \sim \mathcal{N}(f(\mathbf{t}_i), \sigma^2)$.

The objective is to estimate f by:

1. Partitioning the predictor space into $K + 1$ mutually exclusive regions.
2. Fitting local polynomial models within each partition.
3. Enforcing smoothness at partition boundaries via Lagrangian multipliers.
4. Penalizing the integrated squared second derivative to discourage roughness.

Unlike other smoothing spline formulations, no post-fitting algebraic rearrangement or disentanglement of a spline basis is needed to obtain interpretable models. The polynomial expansions are homogeneous across partitions, and the relationship between predictor and response is explicit at the coefficient level.

To anchor the notation, in the single-predictor cubic case one would write

$$\hat{f}(t_i) = \hat{\beta}_{(0)} + \hat{\beta}_{(1)}t_i + \hat{\beta}_{(2)}t_i^2 + \hat{\beta}_{(3)}t_i^3 = \mathbf{x}_i^\top \hat{\boldsymbol{\beta}},$$

where $\mathbf{x}_i = (1, t_i, t_i^2, t_i^3)^\top$. The LMSS formulation preserves exactly this kind of polynomial representation, but now does so within each partition and then forces neighboring pieces to agree in the smoothness conditions described below.

Core notation used throughout:

- $\mathbf{y}_{(N\times 1)}$: Response vector.
- $\mathbf{T}_{(N\times q)}$: Matrix of predictors.
- $\mathbf{X}_{(N\times P)}$: Block-diagonal matrix of polynomial expansions, with diagonal blocks $\mathbf{X}_k$ of dimension $n_k \times p$.
- $\boldsymbol{\Lambda}_{(P\times P)}$: Block-diagonal penalty matrix, with blocks $\boldsymbol{\Lambda}_k$ of dimension $p \times p$.
- $\hat{\boldsymbol{\beta}}_{(P\times 1)}$: Unconstrained penalized estimate.
- $\tilde{\boldsymbol{\beta}}_{(P\times 1)}$: Constrained coefficient estimates.
- $\mathbf{G}_{(P\times P)}$: Block-diagonal matrix with blocks $\mathbf{G}_k = (\mathbf{X}_k^\top \mathbf{W}_k \mathbf{D}_k \mathbf{X}_k + \boldsymbol{\Lambda}_k)^{-1}$, where $\mathbf{W}_k$ and $\mathbf{D}_k$ are defined below.
- $\mathbf{A}_{(P\times r)}$: Constraint matrix encoding smoothness conditions. Reduced to linearly independent columns via pivoted QR decomposition.
- $\mathbf{U}_{(P\times P)}$: $\mathbf{I} - \mathbf{G}\mathbf{A}(\mathbf{A}^\top \mathbf{G}\mathbf{A})^{-1}\mathbf{A}^\top$.
- $\mathbf{D}_{(N\times N)}$: Diagonal matrix of user-supplied observation weights (`observation_weights` or `weights`). Defaults to the identity. These play the role of prior precision on individual observations: a weight of 2 is equivalent to seeing that observation twice.
- $\mathbf{W}_{(N\times N)}$: Diagonal matrix of GLM working weights. In the implementation these diagonal entries are whatever is returned by `glm_weight_function`; by default this is `family$variance(mu)`, optionally multiplied by user-supplied observation weights. For Gaussian response with identity link, $\mathbf{W} = \mathbf{I}$. For other families, $\mathbf{W}$ depends on the current fitted values and is updated at each Newton-Raphson iteration. For the common canonical families used by default, this matches the familiar Fisher-scoring weighting role.
- $\mathbf{V}_{(N\times N)}$: Correlation matrix of errors. When no correlation structure is specified, $\mathbf{V} = \mathbf{I}$. Otherwise supplied via `VhalfInv` or estimated through `VhalfInv_fxn`.

In the Gaussian identity case with unit weights and no correlation, $\mathbf{G}_k = (\mathbf{X}_k^\top \mathbf{X}_k + \boldsymbol{\Lambda}_k)^{-1}$ and most formulas simplify accordingly. When $\mathbf{D}$ or $\mathbf{W}$ appear in a formula, the product $\mathbf{WD}$ means "GLM working weights times observation weights"; whenever one of them is the identity it drops out.

Before these quantities reach the main fitting stage, the user-facing inputs are parsed, standardized, and organized by `process_input`. When the formula interface is used and `auto_encode_factors = TRUE`, that preprocessing step also relies on helpers such as `create_onehot` to encode factor levels before the design reaches `lgspline.fit()`. The notation in the remainder of this document therefore refers to the internal objects that actually enter `lgspline.fit()`, not necessarily the raw objects originally supplied by the user.

From the user side, many of the arguments that control these internal objects can be supplied either individually or through the grouped lists `penalty_args`, `tuning_args`, `expansion_args`, `constraint_args`, `qp_args`, `parallel_args`, `covariance_args`, `return_args`, and `glm_args`, as documented in `lgspline`. These grouped lists are unpacked before dispatch into the same fitting pipeline, so they are a convenience layer rather than a separate modeling abstraction. A closely related exploratory mode is `dummy_fit = TRUE` in `lgspline` or `lgspline.fit`, which runs the preprocessing, partition construction, expansion building, and penalty setup without solving for nonzero coefficients, making it a practical way to inspect objects such as `X`, `A`, the returned `make_partition_list` from `make_partitions`, and the assembled `penalties` from `compute_Lambda` before a full fit.

Model Formulation and Estimation

Piecewise Polynomial Structure: For K knots (one predictor) or $K+1$ partitions (multiple predictors) there are $K+1$ mutually exclusive partitions $\mathcal{P}_0, \dots, \mathcal{P}_K$. Each observation i belongs to exactly one partition. Within partition k, the function is represented as a polynomial of degree $p-1$ in each predictor:

$$\hat{f}_k(\mathbf{t}) = \mathbf{x}^\top \tilde{\boldsymbol{\beta}}_k$$

where $\mathbf{x}$ collects the polynomial basis terms (intercept, linear, quadratic, cubic, and optionally quartic and interaction terms) and $\tilde{\boldsymbol{\beta}}_k$ are the corresponding coefficients. In one predictor, the same idea can be written more explicitly as

$$\hat{f}(t_i) = \sum_{k=0}^{K} \mathbf{x}_{ik}^\top \hat{\boldsymbol{\beta}}_k \mathbf{1}(t_i \in \mathcal{P}_k),$$

which highlights that the unconstrained problem is just a collection of local polynomial regressions. The expansions are homogeneous across partitions, so coefficients are directly comparable. This is implemented via `get_polynomial_expansions`.

The exact contents of $\mathbf{x}$ are controlled by the basis-expansion arguments documented in `lgspline`: `include_quadratic_terms`, `include_cubic_terms`, `include_quartic_terms`, `include_2way_interactions`, `include_3way_interactions`, `include_quadratic_interactions`, `exclude_interactions_for`, `exclude_these_expansions`, and `custom_basis_fxn`. Likewise, `just_linear_with_interactions` and `just_linear_without_interactions` determine which predictors remain structurally linear even though they still participate in the same partition-wise polynomial bookkeeping described here.

Letting p denote the number of basis terms per partition, $P = p(K+1)$ is the total number of coefficients. The full design matrix $\mathbf{X}$ and penalty matrix $\boldsymbol{\Lambda}$ are block-diagonal with $K+1$ blocks, so unconstrained estimation reduces to $K+1$ independent penalized regressions, which appears as follows for the identity link case:

$$\hat{\boldsymbol{\beta}}_k = \mathbf{G}_k \mathbf{X}_k^\top \mathbf{W}_k \mathbf{D}_k \mathbf{y}_k, \quad \mathbf{G}_k = (\mathbf{X}_k^\top \mathbf{W}_k \mathbf{D}_k \mathbf{X}_k + \boldsymbol{\Lambda}_k)^{-1}.$$

For Gaussian identity with unit weights this reduces to the familiar $\mathbf{G}_k = (\mathbf{X}_k^\top \mathbf{X}_k + \boldsymbol{\Lambda}_k)^{-1}$. The block-diagonal structure means these can be computed in parallel across partitions. In the user-facing interface this is realized by supplying a cluster through `cl`, optionally controlling work splitting with `chunk_size`, and enabling stages such as `parallel_eigen` for the eigendecompositions and, in non-Gaussian Path 3, `parallel_unconstrained` for the partition-wise unconstrained fits; nearby stages can likewise use `parallel_penalty` and `parallel_make_constraint`. The eigenvalue decomposition and matrix square roots of each $\mathbf{G}_k$ are computed by `compute_G_eigen`, and can be returned in the fitted object as `G` and `Ghalf` when `return_G = TRUE` and `return_Ghalf = TRUE`.

Fitted values for the canonical Gaussian case appear as $\tilde{\mathbf{y}} = \mathbf{X}\tilde{\boldsymbol{\beta}} = \mathbf{H}\mathbf{y}$ for $\mathbf{H} = \mathbf{XUGX}^\top$.

Smoothing Constraints and the Constraint Matrix: Without further intervention the piecewise polynomial will be discontinuous. The central idea of LMSS is that smoothness is not hidden inside a special basis, but instead imposed directly where neighboring partitions meet. At each knot $t_{k,k+1}$ between neighboring partitions k and $k+1$, up to three smoothing constraints are imposed:

1. Continuity: $\mathbf{x}_{k,k+1}^\top \boldsymbol{\beta}_k = \mathbf{x}_{k,k+1}^\top \boldsymbol{\beta}_{k+1}$.
2. First-derivative continuity: $\mathbf{x}_{k,k+1}'^\top \boldsymbol{\beta}_k = \mathbf{x}_{k,k+1}'^\top \boldsymbol{\beta}_{k+1}$.
3. Second-derivative continuity: $\mathbf{x}_{k,k+1}''^\top \boldsymbol{\beta}_k = \mathbf{x}_{k,k+1}''^\top \boldsymbol{\beta}_{k+1}$.

where $\mathbf{x}'$ and $\mathbf{x}''$ are elementwise first and second derivatives of the basis with respect to $\mathbf{t}$. For the familiar cubic single-predictor basis $\mathbf{x} = (1, t, t^2, t^3)^\top$, these derivative vectors are

$$\mathbf{x}' = (0, 1, 2t, 3t^2)^\top, \qquad \mathbf{x}'' = (0, 0, 2, 6t)^\top.$$

With K knots this yields up to $3K$ scalar constraints (for a single predictor; more for multiple predictors with interactions), collected as linear equations $\mathbf{A}^\top \boldsymbol{\beta} = \mathbf{0}$ in a $P \times r$ matrix $\mathbf{A}$. The constraint matrix is built by `make_constraint_matrix` and returned in the fitted object as `A`.

In higher dimensions or with many partitions, the constraints can become over-specified and force the model toward a single global polynomial. In these cases it is recommended to drop second-derivative constraints or include quartic terms, allowing the model to fit a richer surface while maintaining perceived smoothness at knots. The appropriate constraint level can be controlled via `include_constrain_fitted`, `include_constrain_first_deriv`, and `include_constrain_second_deriv`. The companion flag `include_constrain_interactions` determines whether the analogous mixed-partial constraints are imposed for interaction terms, and `no_intercept` adds the special homogeneous equality constraint that fixes the intercept at zero (the same behavior triggered by using `0 +` in the formula interface).

Before computing the projection $\mathbf{U}$, the constraint matrix is reduced to a linearly independent subset of columns via pivoted QR decomposition. This avoids numerical instability from redundant constraints and ensures $\mathbf{A}^\top\mathbf{GA}$ is invertible.

Lagrangian Projection: The constrained estimate is derived via Lagrangian multipliers. Define the $P \times P$ projection matrix:

$$\mathbf{U} = \mathbf{I} - \mathbf{GA}(\mathbf{A}^\top\mathbf{GA})^{-1}\mathbf{A}^\top.$$

Then the constrained estimate is:

$$\tilde{\boldsymbol{\beta}} = \mathbf{U}\hat{\boldsymbol{\beta}}.$$

The matrix $\mathbf{U}$ has the property that $\mathbf{UGU}^\top = \mathbf{UG}$, which is used extensively in variance estimation and posterior draws. In words, the unconstrained penalized estimate is projected back into the coefficient space that satisfies the smoothness restrictions, and all subsequent uncertainty calculations inherit that same projected geometry. The projection is computed via get_U and, when requested, returned in the fitted object as U through return_U = TRUE.

When the constraints are inhomogeneous ($\mathbf{A}^\top\boldsymbol{\beta} = \mathbf{c}$ with $\mathbf{c} \neq \mathbf{0}$), a particular solution $\boldsymbol{\beta}_0$ satisfying $\mathbf{A}^\top\boldsymbol{\beta}_0 = \mathbf{c}$ is added back after projection, yielding the full Lagrangian solution $\mathbf{U}\hat{\boldsymbol{\beta}} + (\mathbf{I} - \mathbf{U})\boldsymbol{\beta}_0$. In lgspline and lgspline.fit, users realize this by supplying extra equality columns in constraint_vectors together with matching right-hand sides in constraint_values; null_constraint provides the alternate shorthand documented in lgspline when constraint_vectors is supplied and constraint_values is left empty.

In practice $\mathbf{U}$ is never explicitly formed during fitting. The constrained estimate is obtained from a transformed OLS residual problem (the $\mathbf{G}^{1/2}\mathbf{r}^*$ trick) in four steps:

1. Obtain the unconstrained partition-wise unconstrained estimate $\hat{\boldsymbol{\beta}}$.
2. Set $\mathbf{y}^* = \mathbf{G}^{-1/2}\hat{\boldsymbol{\beta}}$ and $\mathbf{X}^* = \mathbf{G}^{1/2}\mathbf{A}$.
3. Fit the linear model $\mathbb{E}[\mathbf{y}^*] = \mathbf{X}^*\boldsymbol{\gamma}$ by OLS using QR decomposition.
4. Compute the residuals $\mathbf{r}^* = \mathbf{y}^* - \mathbf{X}^*(\mathbf{X}^{*\top}\mathbf{X}^*)^{-1}\mathbf{X}^{*\top}\mathbf{y}^*$ from that transformed OLS fit and recover the constrained estimate by $\tilde{\boldsymbol{\beta}} = \mathbf{G}^{1/2}\mathbf{r}^*$.

A scaling factor $1/\sqrt{K+1}$ is applied to both $\mathbf{X}^*$ and $\mathbf{y}^*$ prior to the OLS call and divided out afterward, improving numerical conditioning when the constraint matrix has many rows.

The most expensive operation in this approach is the QR decomposition of the $P \times r$ matrix $\mathbf{X}^* = \mathbf{G}^{1/2}\mathbf{A}$, which is far cheaper than working with the full $P \times P$ system directly. Without correlation or SQP constraints, $\mathbf{G}$ is stored and operated upon as a list of $K+1$ small $p \times p$ matrices rather than the full $P \times P$ block-diagonal, saving substantial memory when K is large and allowing for parallelism.

When correlation is present, $\mathbf{X}^\top\mathbf{V}^{-1}\mathbf{X}$ is no longer block-diagonal, so the full-dimensional system must be handled directly unless the Woodbury acceleration (see .woodbury_decompose_V) applies. When additional inequality constraints are present, the code either augments the equality system with a partition-wise active-set refinement (block-separable case) or falls back to dense SQP via the Goldfarb-Idnani dual active-set method implemented in solve.QP.

GLM Extension and Iterative Updates

Working Quantities: For GLM responses with mean $\mu_i = g^{-1}(\eta_i)$ and working weights $w_i = [V(\mu_i)\{g'(\mu_i)\}^2]^{-1}$, the penalized information becomes $\mathbf{G}_k^{-1} = \mathbf{X}_k^\top\mathbf{W}_k\mathbf{X}_k + \boldsymbol{\Lambda}_k$ with $\mathbf{W}_k = \mathrm{diag}(w_i : i \in \mathcal{P}_k)$. Because $\mathbf{W}$ is diagonal, $\mathbf{X}^\top\mathbf{W}\mathbf{X}$ remains block-diagonal and the four-step procedure carries through with $\mathbf{G}_k = (\mathbf{X}_k^\top\mathbf{W}_k\mathbf{X}_k + \boldsymbol{\Lambda}_k)^{-1}$ in place of $(\mathbf{X}_k^\top\mathbf{X}_k + \boldsymbol{\Lambda}_k)^{-1}$. Under the default glm_weight_function, the diagonal entries reduce to family$variance(mu) (optionally scaled by observation weights), which for canonical families is the usual Fisher scoring weight.

The partition-wise unconstrained estimates are obtained by unconstrained_fit_fxn, by default unconstrained_fit_default, which initializes via an augmented ridge trick (appending $\boldsymbol{\Lambda}^{1/2}$ as pseudo-observations to glm.fit) then refines using damped_newton_r with nr_iterate. For non-canonical log-link gamma regression, the keep_weighted_Lambda = TRUE option correctly returns the augmented ridge estimate directly.

Three Fitting Paths:

Path 1: Correlation structure present. When a working correlation $\mathbf{V}$ is present, such as for marginal means models or generalized estimating equation (GEE)-like models, $\mathbf{V}^{-1}$ couples observations across partitions, so partition-wise fitting is not directly available. Two sub-paths handle this.

Path 1a (Gaussian identity + GEE) solves the whitened system directly via $\tilde{\mathbf{G}} = (\tilde{\mathbf{X}}^\top\tilde{\mathbf{X}} + \boldsymbol{\Lambda}_{\text{block}})^{-1}$ where $\tilde{\mathbf{X}} = \mathbf{V}^{-1/2}\mathbf{X}$, then applies the Lagrangian projection in the full P-space. See .get_B_gee_gaussian.

Path 1b (non-Gaussian GEE) uses a damped SQP iteration on the whitened system. The first iterate is a constrained Newton step from the projection matrix $\mathbf{U}$; subsequent iterates solve the quadratic subproblem via solve.QP. See .get_B_gee_glm.

Both sub-paths have Woodbury-accelerated variants described below.

Path 2: Gaussian identity, no correlation. The constrained estimate is obtained by a single Lagrangian projection from the per-partition penalized least-squares cross-products (the four steps in closed form). No outer iteration is needed. When $K = 0$ and there are no additional constraints, this reduces to the ordinary penalized closed form $\hat{\boldsymbol{\beta}} = \mathbf{G}\mathbf{X}^\top\mathbf{y}$. Implemented in .get_B_gaussian_nocorr.

Path 3: Non-Gaussian GLM, no correlation. Unconstrained estimates are obtained separately within each partition, then projected onto the constraint space. When the link is non-identity, the information $\mathbf{G}$ depends on the current fitted values through the working weights, so the projection must be iterated. The unconstrained anchor $\hat{\boldsymbol{\beta}}$ is held fixed while $\mathbf{G}$ is recomputed at each iterate's constrained estimate:

$$\tilde{\boldsymbol{\beta}}^{(s+1)} = \mathbf{U}^{(s)}\hat{\boldsymbol{\beta}}, \quad \mathbf{U}^{(s)} = \mathbf{I} - \mathbf{G}^{(s)}\mathbf{A}(\mathbf{A}^\top\mathbf{G}^{(s)}\mathbf{A})^{-1}\mathbf{A}^\top.$$

Iteration stops when the mean absolute coefficient change falls below `tol`, or when the update begins increasing (in which case the previous iterate is restored). The recomputation of weighted Gram matrices, Schur corrections, and square-root information factors at each step is handled by `.solver_recompute_G_at_estimate`. Implemented in .get_B_glm_nocorr.

Woodbury Acceleration for Structured Correlation: For structured correlation matrices (AR(1), exchangeable, banded), the perturbation $\mathbf{V}^{-1} - \mathbf{I}$ is sparse. Writing the GLS information matrix as

$$\mathbf{G}_V^{-1} = \underbrace{(\mathbf{X}^\top\mathbf{X} + \boldsymbol{\Lambda})}_{\text{block-diagonal}} + \underbrace{\mathbf{X}^\top(\mathbf{V}^{-1} - \mathbf{I})\mathbf{X}}_{\boldsymbol{\Delta}},$$

the block-diagonal part of $\boldsymbol{\Delta}$ (within-partition corrections) is absorbed into per-partition eigendecompositions, yielding a corrected block-diagonal inverse $\mathbf{G}_c$. The off-diagonal remainder $\boldsymbol{\Delta}_{\text{off}}$ captures cross-partition coupling and has effective rank r determined numerically by .woodbury_decompose_V. For example, AR(1) correlation partitioned by time gives $r \approx 2K$ since only observation pairs straddling knot boundaries contribute.

Factoring $\boldsymbol{\Delta}_{\text{off}} \approx \mathbf{L}\mathbf{S}\mathbf{L}^\top$ where $\mathbf{L}$ is $P \times r$ and $\mathbf{S}$ is $r \times r$ diagonal with entries ± 1, the Woodbury identity gives

$$\mathbf{G}_V = \mathbf{G}_c - \mathbf{G}_c\mathbf{L}(\mathbf{S}^{-1} + \mathbf{L}^\top\mathbf{G}_c\mathbf{L})^{-1}\mathbf{L}^\top\mathbf{G}_c,$$

where the inner matrix is only $r \times r$.

The four-step projection is preserved by expressing $\mathbf{G}_V^{1/2}$ through $\mathbf{G}_c^{1/2}$. Define $\mathbf{Q} = \mathbf{G}_c^{1/2}\mathbf{L}$ (computed block-diagonally) with thin SVD $\mathbf{Q} = \mathbf{O}_Q\boldsymbol{\Sigma}_Q\mathbf{R}_Q^\top$. Then

$$\mathbf{G}_V^{1/2} = \mathbf{G}_c^{1/2}\mathbf{F}^{1/2}, \quad \mathbf{F}^{1/2} = \mathbf{I}_P - \mathbf{O}_Q\mathbf{C}\mathbf{O}_Q^\top,$$

where $\mathbf{C}$ is an $r \times r$ diagonal matrix computed in .woodbury_halfsqrt_components. Every step of the projection decomposes as a block-diagonal operation through $\mathbf{G}_c^{1/2}$ plus an additive rank-r correction through $\mathbf{O}_Q$:

$$\mathbf{y}^* = \mathbf{G}_c^{1/2}(\mathbf{X}^\top\mathbf{V}^{-1}\mathbf{y}) - \mathbf{G}_c^{1/2}\mathbf{O}_Q\mathbf{C}\mathbf{O}_Q^\top\mathbf{G}_c^{1/2}(\mathbf{X}^\top\mathbf{V}^{-1}\mathbf{y}),$$

and similarly for $\mathbf{X}^*$ and the back-transformation. The OLS residual step itself is unchanged.

For the non-Gaussian Woodbury path (`.get_B_gee_glm_woodbury`), the perturbation $\mathbf{\Delta}_V = \mathbf{V}^{-1} - \mathbf{I}$ and the product $\mathbf{\Delta}_V\mathbf{X}$ are precomputed once and held fixed across Newton iterations. At each step, the weighted perturbation $\mathbf{\Delta}(\mathbf{W}) = \mathbf{X}^\top\mathbf{W}(\mathbf{V}^{-1} - \mathbf{I})\mathbf{X}$ is formed using the precomputed product, then split and re-factored via `.woodbury_redecompose_weighted`.

The Woodbury path is used when $r < P/3$; otherwise the code falls back to the dense whitened approach. If the correction matrix $\mathbf{F}$ is not positive definite at any point, the dense path is also used as fallback. See `.get_B_gee_woodbury` (Gaussian) and `.get_B_gee_glm_woodbury` (non-Gaussian).

Step Control: Different paths use different step-control strategies.

In the GEE paths (Paths 1a and 1b), the coefficient update is damped:

$$\boldsymbol{\beta}^{(s+1)} = (1 - \alpha_s)\boldsymbol{\beta}^{(s)} + \alpha_s\boldsymbol{\beta}^{(s)}_{\text{cand}},$$

with $\alpha_s = 2^{-d_s}$ and d_s increased whenever the candidate gives non-finite or larger deviance. The loop terminates after 10 consecutive rejections or 100 total iterations, and exits early when both coefficient change and deviance improvement fall below `tol` after a burn-in period.

In Path 3 (non-Gaussian, no correlation), the outer projection loop has no line search. The algorithm recomputes $\mathbf{G}$ at the current constrained estimate and applies a fresh projection. If the mean absolute coefficient change begins increasing, the previous iterate is restored and the loop stops. The partition-wise unconstrained estimates themselves are obtained by damped Newton-Raphson inside `unconstrained_fit_default`, where `damped_newton_r` computes a Newton direction once per iteration and halves the step size until the penalized log-likelihood improves.

Accommodating Correlation Structures

Parametric Correlation Structures: Suppose $\text{Cov}(\mathbf{y}) = \sigma^2\mathbf{V}(\boldsymbol{\theta})$ for a known parametric family indexed by $\boldsymbol{\theta}$ (e.g., AR(1) with $\theta = \rho$, Matern with $\boldsymbol{\theta} = (\ell, \nu)$, exchangeable with $\theta = \rho$). The penalized generalized least-squares problem becomes

$$\min_{\boldsymbol{\beta}} (\mathbf{y} - \mathbf{X}\boldsymbol{\beta})^\top\mathbf{V}^{-1}(\mathbf{y} - \mathbf{X}\boldsymbol{\beta}) + \boldsymbol{\beta}^\top\mathbf{\Lambda}\boldsymbol{\beta} \quad \text{s.t. } \mathbf{A}^\top\boldsymbol{\beta} = \mathbf{0}.$$

The correlation matrix $\mathbf{V}$ is supplied through the fitted-object components `Vhalf` and `VhalfInv`, either directly or via user functions `Vhalf_fxn` and `VhalfInv_fxn`. When both are non-`NULL`, `get_B` dispatches to the GEE paths (Path 1a or 1b). For built-in correlation structures, the required square-root matrices are assembled numerically using `matsqrt`, `matinvsqrt`, and `invert`.

Whitening and Permutation: Because the data are stored in partition ordering (all observations from partition 0, then partition 1, etc.) while $\mathbf{V}$ is in the original observation ordering, a permutation is applied internally: $\mathbf{V}^{-1/2}_{\text{perm}} = \mathbf{V}^{-1/2}[\boldsymbol{\pi}, \boldsymbol{\pi}]$, where $\boldsymbol{\pi} =$ `unlist(order_list)` maps original indices to partition-ordered indices. The whitened design and response are $\tilde{\mathbf{X}} = \mathbf{V}^{-1/2}_{\text{perm}}\mathbf{X}_{\text{block}}$ and $\tilde{\mathbf{y}} = \mathbf{V}^{-1/2}_{\text{perm}}\mathbf{y}$.

The $\mathbf{X}$ and $\mathbf{y}$ inputs to `lgspline.fit` are preserved in their unwhitened form. Whitening is applied inside `get_B` and `blockfit_solve` where the full $N \times P$ block-diagonal design is available, since applying $\mathbf{V}^{-1/2}$ to only the diagonal blocks of the partitioned design would silently discard cross-partition contributions and corrupt the Gram matrix.

Loss of Block-Diagonal Structure: Unlike the independent-errors case, $\mathbf{X}^\top\mathbf{V}^{-1}\mathbf{X}$ is not block-diagonal because $\mathbf{V}^{-1}$ introduces cross-partition coupling. The unconstrained estimator

$$\hat{\boldsymbol{\beta}} = (\mathbf{X}^\top\mathbf{V}^{-1}\mathbf{X} + \boldsymbol{\Lambda})^{-1}\mathbf{X}^\top\mathbf{V}^{-1}\mathbf{y}$$

requires a full $P \times P$ solve, and the constraint projection proceeds with $\mathbf{G} = (\mathbf{X}^\top\mathbf{V}^{-1}\mathbf{X} + \boldsymbol{\Lambda})^{-1}$. For structured correlation matrices (AR(1), exchangeable, banded), the perturbation $\mathbf{V}^{-1} - \mathbf{I}$ is sparse and the cross-partition coupling has low effective rank. In these cases the Woodbury-accelerated paths (`.get_B_gee_woodbury`, `.get_B_gee_glm_woodbury`) recover the partition-wise computational structure by decomposing the coupling into a block-diagonal correction plus a low-rank remainder, as described in the GLM Extension section. For dense or high-rank $\mathbf{V}^{-1}$, the code falls back to the full whitened system (`.get_B_gee_gaussian`, `.get_B_gee_glm`).

GEE Deviance Monitoring: For non-Gaussian models with correlation, the deviance used for convergence monitoring is computed in the whitened space by `.bf_gee_deviance`. When the family supplies `custom_dev.resids`, the raw deviance residuals $r_i = \text{sign}(d_i)\sqrt{|d_i|}$ are divided by $\sqrt{w_i}$ and pre=dmultiplied by $\mathbf{V}_{\text{perm}}^{-1/2}$ before squaring and averaging:

$$D_{\text{GEE}} = \frac{1}{N}\left\|\mathbf{V}_{\text{perm}}^{-1/2}\,\mathbf{w}^{-1/2}\mathbf{r}\right\|^2,$$

where $\mathbf{w}$ is the vector of working weights at the current iterate, clamped below at $\sqrt{\varepsilon_{\text{mach}}}$. When only `dev.resids` is available, the function falls back to the standard mean deviance; otherwise it uses mean squared error.

REML Estimation of Correlation Parameters: Correlation parameters $\boldsymbol{\theta}$ are estimated by minimizing a negative restricted log-likelihood (REML) objective. The criterion implemented in **lgspline** is a central-limit-theorem-based working approximation to a Laplace-style marginal likelihood criterion, applied here solely to correlation structure estimation rather than penalty parameter selection. Let $\mathbf{D} = \text{diag}(d_i)$ be the observation weight matrix, $\tilde{\mathbf{W}} = \text{diag}(\tilde{w}_i)$ the GLM working weight matrix at the current fitted values, $\mathbf{V}$ the correlation matrix parameterized by $\boldsymbol{\rho}$ (a vector on the unconstrained real line), and $\tilde{\sigma}^2$ the dispersion profiled at its restricted maximum likelihood estimate. The negative REML objective implemented in **lgspline**, scaled by $1/N$, is

$$-\ell_R(\boldsymbol{\rho}) = \frac{1}{N}\left[-\log|\mathbf{V}^{-1/2}| + \frac{N}{2}\log\tilde{\sigma}^2 + \frac{1}{2\tilde{\sigma}^2}(\mathbf{y}-\boldsymbol{\mu})^\top\mathbf{D}\tilde{\mathbf{W}}^{-1}\mathbf{V}^{-1}(\mathbf{y}-\boldsymbol{\mu}) + \frac{1}{2}\log|(\tilde{\sigma}^2\mathbf{U}\mathbf{G})^{-1}|^+\right],$$

where $|\cdot|^+$ denotes the generalized determinant (product of nonzero eigenvalues), and $\boldsymbol{\mu} = g^{-1}(\mathbf{X}\tilde{\boldsymbol{\beta}})$ are the fitted values on the response scale.

Gradients with respect to correlation parameters are available in closed form for all built-in structures except Matern, which uses finite-difference approximation due to the complexity of differentiating the modified Bessel function K_ν with respect to ν. See `reml_grad_from_dV` for the full gradient derivation and notation. Custom analytic gradients can be supplied through `REML_grad`, and a fully custom criterion can replace REML through `custom_VhalfInv_loss`. The Toeplitz example in `lgspline` demonstrates how to supply custom correlation structures with user-defined gradient functions. Optimization over these working correlation parameters is then carried out by the same quasi-Newton engine used elsewhere in the package, namely `efficient_bfgs` with fallback to `approx_grad` when needed. In the user-facing interface, this machinery is activated through `correlation_structure`, `correlation_id`, `spacetime`, `VhalfInv`, `Vhalf`, `VhalfInv_fxn`, `Vhalf_fxn`, `VhalfInv_par_init`, `REML_grad`, `custom_VhalfInv_loss`, and `VhalfInv_logdet`.

The gradient of the negative REML has three terms per parameter:

1. $\frac{1}{2}\mathrm{tr}(\mathbf{V}^{-1}\partial\mathbf{V}/\partial\theta_j)$: the log-determinant contribution.
2. $-\frac{1}{2\tilde{\sigma}^2}\mathbf{r}^\top(\partial\mathbf{V}/\partial\theta_j)\mathbf{r}$: the residual quadratic form contribution, where $\mathbf{r} = \mathrm{diag}(\sqrt{d_i}/\sqrt{\tilde{w}_i})\mathbf{V}^{-1/2}(\mathbf{y}-\boldsymbol{\mu})$.
3. $-\frac{1}{2}\mathrm{tr}\Big(\mathbf{M}^{+}\mathbf{X}_*^\top\mathbf{V}^{-1}(\partial\mathbf{V}/\partial\theta_j)\mathbf{V}^{-1}\tilde{\mathbf{W}}\mathbf{D}\mathbf{X}_*\Big)$: the REML correction, where $\mathbf{X}_* = \mathbf{X}\mathbf{U}$ is the constrained design and $\mathbf{M} = \mathbf{X}_*^\top\mathbf{V}^{-1}\tilde{\mathbf{W}}\mathbf{D}\mathbf{X}_* + \mathbf{U}^\top\boldsymbol{\Lambda}\mathbf{U}$ is the projected penalized information.

For each supported correlation family, the derivatives $\partial\mathbf{V}/\partial\theta_j$ are available in closed form, enabling analytic gradient computation for use with the quasi-Newton optimizer.

Connection to Standard Mixed Model REML: The quadratic penalty $\boldsymbol{\Lambda}$ acts as the inverse prior covariance of a Gaussian random effect on the spline coefficients, with the smoothing parameter satisfying $\lambda = \tau^2/\sigma^2$; this is the same mixed model representation used by **mgcv**, and Monte Carlo draws under the resulting Laplace-style approximation are available via `generate_posterior`. For Gaussian responses with identity link and no penalization, the implemented criterion coincides exactly with classical REML. For non-Gaussian responses, the criterion substitutes the Fisher information for the full penalized log-likelihood Hessian, exploiting the CLT approximation that $\tilde{\mathbf{W}}^{-1/2}(\mathbf{y}-\boldsymbol{\mu})$ is approximately Gaussian; this yields a method-of-moments style estimator for the correlation and variance parameters that depends only on mean-variance relationships through $\mathbf{W}$, and therefore generalizes naturally to quasi-likelihood and other settings where a fully specified log-likelihood is unavailable. The REML correction term $\log|(\tilde{\sigma}^2\mathbf{U}\mathbf{G})^{-1}|^{+}$ uses a generalized log-determinant so that only nonzero eigenvalues contribute when rank deficiency arises from smoothness constraints or identifiability conditions in $\mathbf{A}$. When $\boldsymbol{\Lambda}$ is full rank, the criterion coincides with the exact marginal likelihood from integrating out $\boldsymbol{\beta}$ under its Gaussian prior; when $\boldsymbol{\Lambda}$ is rank-deficient, unpenalized coefficients are projected out in the REML sense while penalized coefficients are integrated through their prior. During penalty tuning, the block-diagonal approximation is retained for GCV criteria and gradients; since GCV is rotation-invariant the practical effect on automatic selection is expected to be negligible, though this is not confirmed and the tuned penalties can always be overridden.

Built-In Correlation Structures: The package provides several built-in correlation structures for modeling spatial and temporal dependence. These are specified via `correlation_structure` with group membership in `correlation_id` and spatial or temporal coordinates in `spacetime` (an N-row matrix). Exchangeable correlation does not require `spacetime`.

All positive scale parameters are estimated on the log scale, with back-transform $\exp(\cdot)$. Parameters constrained to $(0, 1)$ use a double-exponential back-transform of the form $\exp(-\exp(\eta))$, so optimization still occurs on the unconstrained real line while the correlation remains bounded.

Exchangeable Aliases: `'exchangeable'`, `'cs'`, `'CS'`, `'compoundsymmetric'`, `'compound-symmetric'`.
A constant correlation ν between any two observations within the same cluster. Parameterization: $\nu = \exp(-\exp(\rho))$, so $\nu \in (0, 1)$. Only positive within-cluster correlation is supported under this parameterization.

Spatial Exponential Aliases: `'spatial-exponential'`, `'spatialexponential'`, `'exp'`, `'exponential'`.
Correlation decays exponentially with distance: $\exp(-\omega d)$ where d is Euclidean distance and $\omega > 0$. Parameterization: $\omega = \exp(\rho)$. Mathematically equivalent to the power correlation θ^d with $\theta = e^{-\omega}$, but with better numerical properties during optimization.

AR(1) Aliases: `'ar1'`, `'ar(1)'`, `'AR(1)'`, `'AR1'`.
Correlation depends on rank difference between observations: ν^r where r is the rank difference within cluster. Parameterization: $\nu = \exp(-\exp(\rho))$, so $\nu \in (0, 1)$. Only positive autocorrelation is supported.

Gaussian / Squared Exponential Aliases: `'gaussian'`, `'rbf'`, `'squared-exponential'`.
Smooth decay with squared distance: $\exp(-d^2/(2\ell^2))$ where ℓ is the length scale. Parameterization: $\ell = \exp(\rho)$.

Spherical Aliases: `'spherical'`, `'Spherical'`, `'cubic'`, `'sphere'`.
Polynomial decay with a hard cutoff at range r: $1 - 1.5(d/r) + 0.5(d/r)^3$ for $d \leq r$, and 0 otherwise. Parameterization: $r = \exp(\rho)$.

Matern Aliases: `'matern'`, `'Matern'`.
Flexible correlation with adjustable smoothness: $(2^{1-\nu}/\Gamma(\nu))(\sqrt{2\nu}\, d/\ell)^\nu K_\nu(\sqrt{2\nu}\, d/\ell)$. Two parameters: length scale $\ell = \exp(\rho_1)$ and smoothness $\nu = \exp(\rho_2)$. No analytical gradient is available for ν due to the difficulty of differentiating the modified Bessel function K_ν with respect to its order, so finite differences are used; this makes Matern slower and potentially less stable than other structures.

Gamma-Cosine Aliases: `'gamma-cosine'`, `'gammacosine'`, `'GammaCosine'`.
Oscillatory dependence: $(d^{\alpha-1}e^{-\gamma d})/(\Gamma(\alpha)/\gamma^\alpha) \cdot \cos(\omega d)$. Three parameters: shape $\alpha = \exp(\rho_1)$, rate $\gamma = \exp(\rho_2)$, frequency $\omega = \exp(\rho_3)$. Reduces to exponential when $\alpha = 1$ and $\omega \approx 0$.

Gaussian-Cosine Aliases: `'gaussian-cosine'`, `'gaussiancosine'`, `'GaussianCosine'`.
Smooth oscillatory correlation: $\exp(-d^2/(2\ell^2)) \cdot \cos(\omega d)$. Two parameters: length scale $\ell = \exp(\rho_1)$ and frequency $\omega = \exp(\rho_2)$. Reduces to Gaussian when $\omega \approx 0$.

Interpreting Estimated Correlation Parameters: Correlation parameters are estimated on transformed scales; they must be back-transformed for interpretation. When `confint.lgspline` is called and the inverse Hessian from BFGS is available, confidence intervals are returned on the untransformed (working) scale and should be back-transformed as described in the examples for `lgspline`.

Custom Correlation Structures: Custom correlation structures can be specified through:

- `VhalfInv_fxn`: Creates $\mathbf{V}^{-1/2}$.
- `Vhalf_fxn`: Creates $\mathbf{V}^{1/2}$. When omitted, the code computes it by explicit inversion of `VhalfInv`.
- `REML_grad`: Provides the analytical gradient of the REML objective.
- `VhalfInv_logdet`: Efficient log-determinant computation.
- `custom_VhalfInv_loss`: Replaces the REML objective entirely.

These functions enter `lgspline` through `correlation_structure`, `VhalfInv_fxn`, `Vhalf_fxn`, `REML_grad`, and `custom_VhalfInv_loss`, and the fitted object retains the resulting correlation machinery in components such as `VhalfInv_fxn`, `Vhalf_fxn`, and `VhalfInv_params_estimates`. When `VhalfInv` is supplied but `Vhalf` is not, `Vhalf` is computed unconditionally as the inverse of `VhalfInv` for all family/link combinations, since both `get_B` and `blockfit_solve` require it for GEE estimation.

Variance and Dispersion Estimation

Once the constrained estimate has been obtained, the next questions are how much flexibility the fitted model effectively used and how uncertainty should be propagated through the same constrained geometry. The quantities in this section are therefore all built on the projected information matrices from the previous sections.

Effective Degrees of Freedom and Dispersion: The effective degrees of freedom is the trace of the hat matrix. In the Gaussian identity case with observation weights and correlation, the fitted linear operator is built from the dense GLS analogue $\mathbf{G}_{\text{correct}}$ and can be written schematically as

$$\mathbf{H} = \mathbf{V}^{-1/2}(\mathbf{WD})^{1/2}\mathbf{XUG}_{\text{correct}}\mathbf{X}^\top(\mathbf{WD})^{1/2}\mathbf{V}^{-1/2},$$

where for Gaussian identity $\mathbf{W} = \mathbf{I}$. In the no-correlation Gaussian case this reduces to the familiar $\mathbf{H} = \mathbf{XUGX}^\top\mathbf{D}$.

For Gaussian identity fits, the dispersion estimate is computed as a weighted mean squared residual, optionally scaled by $N/(N - \text{tr}(\mathbf{H}))$ when `unbias_dispersion = TRUE`:

$$\tilde{\sigma}^2 = \frac{1}{N - \text{tr}(\mathbf{H})}\|\mathbf{y} - \tilde{\mathbf{y}}_i\|^2.$$

More generally with weights, a correlation structure and non-linear link function:

$$\tilde{\sigma}^2 = \frac{1}{N - \text{tr}(\mathbf{H})}\|\mathbf{V}^{-1/2}\mathbf{W}^{-1/2}\mathbf{D}^{1/2}(\mathbf{y} - \tilde{\mathbf{y}})\|^2.$$

This estimated dispersion is returned as `sigmasq_tilde`, and the corresponding effective degrees of freedom trace is returned as `trace_XUGX`. For non-Gaussian families, the fitting code delegates dispersion estimation to `dispersion_function`; thus the package does not assume a single closed-form Pearson-style formula outside the Gaussian identity setting. The hat-matrix trace itself is assembled by `compute_trace_H` in the dense correlation-aware case and by the same blockwise products summarized by `trace_XUGX` in the simpler no-correlation paths. A concrete built-in non-Gaussian example is the Weibull AFT path, which pairs `weibull_family` with `weibull_dispersion_function`, `weibull_glm_weight_function`, and `weibull_schur_correction`. Users who want these quantities available for downstream inference should keep `estimate_dispersion = TRUE` and `return_varcovmat = TRUE` (the defaults), since `wald_univariate`, `confint.lgspline`, and the prediction-standard-error path in `predict.lgspline` all rely on the post-fit dispersion and covariance components documented here.

Variance-Covariance Matrix: The variance-covariance matrix of $\tilde{\boldsymbol{\beta}}$ is estimated as:

$$\text{Var}(\tilde{\boldsymbol{\beta}}) = \tilde{\sigma}^2(\mathbf{UG}^{1/2})(\mathbf{UG}^{1/2})^\top$$

using the outer-product form for numerical stability. The result is returned as `varcovmat` when `return_varcovmat = TRUE`. The algebraically equivalent expression $\tilde{\sigma}^2\mathbf{UG}$ is not used because $\mathbf{G}$ is only positive semi-definite when the penalty matrix $\boldsymbol{\Lambda}$ has zero eigenvalues (e.g., the intercept and linear terms under the smoothing spline penalty when `flat_ridge_penalty = 0`), which can introduce negative diagonal entries in finite precision arithmetic. The outer-product form also guarantees symmetry.

This is the Bayesian posterior covariance, treating the penalty as a Gaussian prior on the coefficients. When `exact_varcovmat = TRUE`, a frequentist correction is additionally computed:

$$\text{Var}_{\text{exact}}(\tilde{\boldsymbol{\beta}}) = \tilde{\sigma^2}\mathbf{UG}^{1/2}(\mathbf{X}^\top\mathbf{WDV}^{-1}\mathbf{X})\mathbf{G}^{1/2}\mathbf{U}^\top = \tilde{\sigma}^2\mathbf{UGU}^\top - \tilde{\sigma}^2\mathbf{UG\Lambda GU}^\top.$$

The first term is the Bayesian posterior covariance; the second is a frequentist correction such that for Gaussian identity link (with or without correlation), this is the exact variance-covariance matrix of the constrained estimator.

When a correlation structure is present (`VhalfInv` non-`NULL`), the block-diagonal $\mathbf{G}$ is replaced by the full weighted GLS analogue

$$\mathbf{G}_{\text{correct}} = \left(\mathbf{X}^{\top}\mathbf{W}\mathbf{D}\mathbf{V}^{-1}\mathbf{X} + \boldsymbol{\Lambda}\right)^{-1},$$

where $\mathbf{W} = \mathbf{I}$ in the Gaussian identity case. This dense matrix is what enters the correlation-aware $\mathbf{U}$, $\text{Var}(\tilde{\boldsymbol{\beta}})$, and $\text{Var}_{\text{exact}}(\tilde{\boldsymbol{\beta}})$ computations.

In user-facing terms, `return_varcovmat` controls whether this matrix is stored at all, while `exact_varcovmat` switches between the default posterior/Laplace approximation and the exact frequentist correction in the Gaussian-identity setting. The stored covariance is what powers `wald_univariate`, `confint.lgspline`, and `se.fit = TRUE` in `predict.lgspline`; the `critical_value` argument supplied at fit time is carried forward as the default cutoff for those interval-producing helpers.

Recomputation of G at Convergence: At the final iterate, $\mathbf{G}$ is recomputed to reflect the converged working weights and Schur corrections. The implementation computes the weighted design $\mathbf{X}_w^{(k)} = \mathbf{X}_k \cdot \text{diag}(\sqrt{\mathbf{w}_k})$, forms the weighted Gram matrix $\mathbf{X}_w^{(k)\top}\mathbf{X}_w^{(k)}$, adds the Schur correction, and performs eigendecomposition via `compute_G_eigen` to obtain $\mathbf{G}_k$, $\mathbf{G}_k^{1/2}$, and $\mathbf{G}_k^{-1/2}$. The relationship $\mathbf{G}_k = \mathbf{G}_k^{1/2}(\mathbf{G}_k^{1/2})^{\top}$ is enforced exactly by construction, and the fitted object can retain these as `G` and `Ghalf` when `return_G = TRUE` and `return_Ghalf = TRUE`. The square-root factors are numerically stabilized through the helper routines `matsqrt` and `matinvsqrt`, which are also used elsewhere in the package when dense analogues of $\mathbf{G}^{1/2}$ or $\mathbf{G}^{-1/2}$ are required.

Bayesian Interpretation

The penalty has a natural Gaussian-prior interpretation, so once the constrained estimator and its covariance are available, Bayesian-style posterior simulation follows almost immediately. This section records the interpretation that is already implicit in the fitted object and in the package's posterior simulation helpers.

A Bayesian interpretation follows from viewing the penalty as a Gaussian prior on the coefficients. Conditional on the fitted smoothing parameters, the code samples on the coefficient scale from

$$\boldsymbol{\beta}^{(m)} = \tilde{\boldsymbol{\beta}} + \sqrt{\tilde{\sigma}^2}, \mathbf{U}\mathbf{G}^{1/2}\mathbf{z}^{(m)}, \qquad \mathbf{z}^{(m)} \sim \mathcal{N}(\mathbf{0}, \mathbf{I}).$$

The coefficients are then back-transformed to the original response and predictor scales.

When inequality constraints are absent, these draws are i.i.d. Gaussian posterior draws around the fitted mode. The underlying coefficient-draw closure also contains an elliptical slice sampling route for active inequality constraints, using the same covariance factor to keep retained draws in the feasible region.

At the implementation level, standard Gaussian posterior draws may place positive mass on the infeasible region, so the constrained-draw closure instead targets the truncated posterior

$$\pi(\boldsymbol{\beta} \mid \mathbf{y}) \propto \exp\left(-\frac{1}{2}(\boldsymbol{\beta} - \tilde{\boldsymbol{\beta}})^{\top}\mathbf{G}^{-1}(\boldsymbol{\beta} - \tilde{\boldsymbol{\beta}})\right)\mathbf{1}(\mathbf{C}^{\top}\boldsymbol{\beta} \succeq \mathbf{c}),$$

yielding credible intervals that respect the constraint boundaries. The public `generate_posterior` wrapper forwards `enforce_qp_constraints` to the stored constrained-draw closure, so constrained draws can be requested directly from the user-facing interface. When a working correlation structure is present, the companion helper `generate_posterior_correlation` extends this idea by propagating uncertainty in the fitted correlation parameters through the same `VhalfInv_fxn`/`Vhalf_fxn` machinery described in the correlation section, rather than conditioning only on fixed covariance parameters. Correlation parameters are drawn from a multivariate normal distribution centered about their estimates with the inverse approxiamte BFGS Hessian of the REML optimization problem `VhalfInv_params_vcov` used by default (or a custom alternative as supplied to the argument `correlation_param_vcov_sc`).

Inequality Constraints via Sequential Quadratic Programming

Overview: Inequality constraints of the form $\mathbf{C}^\top\boldsymbol{\beta} \succeq \mathbf{c}$ handle shape restrictions such as monotonicity, convexity, or boundedness. In the monomial basis, these are linear in $\boldsymbol{\beta}$. Monotonicity at a grid of points requires the first derivative polynomial to be non-negative there; convexity requires the second derivative to be non-negative; range constraints bound the fitted values directly. All of these translate to linear inequality constraints on the coefficient vector.

The inequality pieces are assembled by `process_qp`, which returns the `qp_Amat`, `qp_bvec`, and `qp_meq` objects passed to `solve.QP`, together with a `quadprog` flag indicating whether any inequality constraints are active. For derivative-sign constraints, `process_qp` calls `.build_deriv_qp`, which uses `make_derivative_matrix` on the expansion-standardized design and maps the derivative rows into the full P-dimensional coefficient space partition by partition.

Partition-Wise Active-Set Method: The sparsity pattern of $\mathbf{C}$ is inspected automatically by `.detect_qp_global` (equivalently `.solver_detect_qp_global`). When every column of $\mathbf{C}$ has nonzero entries in only a single partition block, the constraint system is block-separable and an active-set method can replace the dense QP.

At each iteration, the active set $\mathcal{A}$ (constraints satisfied at equality) is appended to $\mathbf{A}$ as additional equality constraints:

$$\mathbf{A}_{\text{aug}} = [\mathbf{A} \mid \mathbf{C}_{\mathcal{A}}].$$

The constrained estimate is obtained by the same OLS projection with $\mathbf{A}_{\text{aug}}$ in place of $\mathbf{A}$, and since $\mathbf{A}_{\text{aug}}$ retains block-diagonal compatibility, all operations remain partition-wise.

The active set is updated by checking primal feasibility (adding the most-violated inactive constraint) and dual feasibility (dropping the active constraint with the most negative Lagrange multiplier) until the KKT conditions are satisfied. Lagrange multipliers for active inequalities are recovered from the OLS fit used in the Lagrangian projection: the fitted coefficients on $\mathbf{X}^* = \mathbf{G}^{1/2}\mathbf{A}_{\text{aug}}$ give the multipliers up to sign. Implemented in `.active_set_refine` and `.check_kkt_partitionwise`.

When any column of $\mathbf{C}$ spans multiple partition blocks (for example, cross-knot monotonicity constraints), the block-diagonal structure is broken and the dense SQP approach is required. The selection is made automatically. If the active-set method does not converge within its iteration limit (default 50), the code falls back to dense SQP as well.

Dense SQP Iteration: The dense SQP approach, implemented in `.qp_refine`, solves a sequence of quadratic subproblems approximating the penalized log-likelihood. At each iteration s:

1. Compute the information matrix $\mathbf{M}^{(s)} = \mathbf{X}^\top\mathbf{W}^{(s)}\mathbf{X} + \boldsymbol{\Lambda}_{\text{block}} + \mathbf{S}^{(s)}$, where $\mathbf{S}^{(s)}$ is the Schur complement correction.
2. Compute the score vector via `qp_score_function`.
3. Solve the QP with `solve.QP`, where the combined constraint matrix is $[\mathbf{A} \mid \mathbf{C}]$ with the first R columns treated as equalities.

4. Apply a damped update $\boldsymbol{\beta}^{(s+1)} = (1-\alpha)\boldsymbol{\beta}^{(s)} + \alpha\boldsymbol{\beta}_{\text{QP}}$, with $\alpha = 2^{-d}$ and d incremented upon deviance increase.

A rescaling factor $\text{sc} = \sqrt{\text{mean}(|\mathbf{M}^{(s)}|)}$ is applied to the Hessian and linear term before calling the QP solver. The equality-constrained estimate from the Lagrangian projection serves as a warm start. The qp_score_function defaults to the canonical GLM score $\mathbf{X}^{\top}(\mathbf{y} - \boldsymbol{\mu})$; for custom models a different score can be supplied.

Active Set and Lagrange Multipliers: The active set at the solution identifies binding inequality constraints. The Lagrange multipliers quantify the cost of each: a multiplier of zero means the constraint is not binding. The implementation stores active constraint indices, the corresponding submatrix of the constraint matrix, and the multiplier vector in the qp_info list returned alongside coefficient estimates, with components lagrangian, iact, and Amat_active. When return_lagrange_multipliers = TRUE, the fitted object also stores the final multiplier vector directly as lagrange_multipliers.

The original assembled inequality data are retained in the fitted object's quadprog_list component (containing qp_Amat, qp_bvec, and qp_meq from process_qp) so the final active set can be interpreted relative to the full specification. The final $\mathbf{U}$ used in constructing the posterior variance-covariance matrix is built from both the equality constraints and the active inequality constraints at the solution.

Built-In Constraints: Built-in inequality constraints include:

- Monotonicity: qp_monotonic_increase, qp_monotonic_decrease. Enforced by requiring consecutive fitted values to be non-decreasing (or non-increasing): $(\mathbf{x}_i - \mathbf{x}_{i-1})^{\top}\boldsymbol{\beta} \geq 0$. These are constructed by process_qp from the partition-stacked block design reordered to observation order.
- Derivative sign: qp_positive_derivative, qp_negative_derivative. Enforced through the first-derivative design matrix from make_derivative_matrix. May be TRUE/FALSE or a character/integer vector selecting specific predictors.
- Second-derivative sign: qp_positive_2ndderivative, qp_negative_2ndderivative. Same construction using the second-derivative design matrix.
- Response range: qp_range_lower, qp_range_upper. Bounds on the linear predictor; for non-identity links, the bounds are transformed to the link scale inside process_qp.
- Custom constraints via qp_Amat_fxn, qp_bvec_fxn, and qp_meq_fxn, which receive the design matrix structure and return the constraint matrix, bound vector, and number of equalities. These are commonly paired with a custom qp_score_function when the quadratic approximation uses a non-default likelihood. The low-level objects qp_Amat, qp_bvec, and qp_meq remain documented in lgspline but in the current implementation serve as activation markers rather than being merged into the constraint set assembled by process_qp.

All constraints can be thinned to a user-specified subset of rows via qp_observations, which process_qp applies before assembly.

Blockfit Backfitting for Linear Non-Interactive Effects

Motivation: When a model contains both spline terms (receiving $K+1$ partition-specific coefficient vectors constrained to smoothness) and non-interactive linear terms ("flat" terms, specified via just_linear_without_interactions, receiving a single shared coefficient vector $\mathbf{v}$ across all partitions), the standard solver carries $K+1$ copies of $\mathbf{v}$ linked by equality constraints.

Backfitting avoids this inflation by solving a lower-dimensional problem at each step. Write the partition-k design as $\mathbf{X}_k = [\mathbf{Z}_k \mid \mathbf{X}_{\text{flat}}^{(k)}]$, where $\mathbf{Z}_k$ contains the spline columns and $\mathbf{X}_{\text{flat}}^{(k)}$ the flat columns. This is invoked when `blockfit = TRUE`, flat columns are non-empty, and $K > 0$.

The design, penalty, and constraint matrices are split into spline and flat components by `.bf_split_components`, which extracts spline rows from $\mathbf{A}$, drops null columns, rank-reduces via QR, and detects mixed constraints (columns of $\mathbf{A}$ with nonzero entries on both spline and flat rows).

Block-Coordinate Descent: Spline step. Holding $\mathbf{v}$ fixed, the code forms the adjusted response $\mathbf{y}_k - \mathbf{X}_{\text{flat}}^{(k)}\mathbf{v}$ and applies the spline-only Lagrangian projection via `.bf_lagrangian_project`:

$$\tilde{\boldsymbol{\beta}}_{\text{spline}}^{(k)} = \mathbf{U}_{\text{spline}}\mathbf{G}_{\text{spline}}\mathbf{Z}_k^\top(\mathbf{y}_k - \mathbf{X}_{\text{flat}}^{(k)}\mathbf{v}).$$

Flat step. Holding spline coefficients fixed, the shared flat vector is updated by pooled penalized regression on residuals:

$$\mathbf{v} = \left(\sum_{k=0}^{K}\mathbf{X}_{\text{flat}}^{(k)\top}\mathbf{X}_{\text{flat}}^{(k)} + \boldsymbol{\Lambda}_{\text{flat}}\right)^{-1}\sum_{k=0}^{K}\mathbf{X}_{\text{flat}}^{(k)\top}(\mathbf{y}_k - \mathbf{Z}_k\tilde{\boldsymbol{\beta}}_{\text{spline}}^{(k)}).$$

When the constraint matrix $\mathbf{A}$ has mixed columns (nonzero entries on both spline and flat rows), the flat update instead solves a KKT system enforcing the residual equality constraint $\mathbf{A}_{\text{flat}}^\top\mathbf{v} = \mathbf{c} - \mathbf{A}_{\text{spline}}^\top\tilde{\boldsymbol{\beta}}_{\text{spline}}$ via `.bf_constrained_flat_update`.

Convergence is checked using the maximum absolute change across both spline and flat coefficients. In the weighted inner loop used by the GLM solvers, the flat-block change alone determines stopping.

Four Estimation Cases: `blockfit_solve` dispatches to one of four paths.

Case (a): Gaussian identity + GEE (`.bf_case_gauss_gee`). Whitening destroys block-diagonal structure, so the code skips backfitting and performs the same full-system Gaussian GEE projection used by `get_B` Path 1a. The result is split back into spline and flat components for downstream assembly.

Case (b): Gaussian identity, no correlation (`.bf_case_gauss_no_corr`). Standard block-coordinate descent as described above. The spline-only $\mathbf{G}_{\text{spline}}$ factors are precomputed once, and the pooled flat penalized inverse is reused across iterations.

Case (c): GLM + GEE (`.bf_case_glm_gee`). Two stages. Stage 1 forms a warm start by running damped Newton-Raphson on the unwhitened working response: each outer iteration computes working responses and weights at the current linear predictor, then calls `.bf_inner_weighted` for the inner backfitting loop. Stage 2 refines the warm start on the full whitened system using the damped SQP loop (`.bf_sqp_loop`), replicating the approach used by `.get_B_gee_glm`.

Case (d): GLM without GEE (`.bf_case_glm_no_corr`). A damped Newton-Raphson outer loop updates working responses and weights, while each inner iteration alternates between weighted spline and weighted flat updates via `.bf_inner_weighted`. Deviance is monitored across outer iterations for convergence and damping.

Inequality Constraints and Reassembly: Because flat coefficients are shared by construction, the corresponding equality constraints are satisfied exactly. Smoothness constraints on the spline block are enforced by the spline-only Lagrangian projection. After backfitting convergence, inequality constraints are enforced via the same partition-wise active-set or dense SQP refinement used by `get_B`. The method is selected automatically by `.solver_detect_qp_global`:

block-separable constraints use the active-set method through `.active_set_refine`, while cross-partition constraints trigger the dense SQP loop through `.bf_sqp_loop`. For GEE (Case c), inequality handling occurs inside Stage 2 on the whitened system.

After convergence, the shared flat vector $\mathbf{v}$ is copied into each partition's coefficient vector, yielding $\boldsymbol{\beta}_k = [\tilde{\boldsymbol{\beta}}^{(k)\top}_{\text{spline}}, \mathbf{v}^\top]^\top$ for compatibility with downstream inference. If `blockfit_solve` throws an error, a warning is issued and the code falls back to `get_B`.

Knot Selection and Partitioning

The topic of knot selection is not the main focus of the package, but the partition structure is central because every later design matrix, penalty, and smoothness constraint depends on it. The defaults in **lgspline** are therefore meant to be practical and transparent rather than theoretically final.

Univariate Case: For a single predictor, the default partitioning is now handled by `make_partitions` in the same k-means framework used more generally: $K + 1$ centers are fit on an internally standardized copy of the predictor, controlled by `standardize_predictors_for_knots`, and then returned on the raw scale. Custom knots can still be supplied via `custom_knots`, in which case partition assignment is built directly from those raw-scale breakpoints. The default number of knots K is chosen adaptively based on N, p, q, and the GLM family. For multivariate fits, the resulting partition metadata are returned as `make_partition_list` and can be re-used in later calls to `lgspline`. This is particularly useful when one wants to hold the partition geometry fixed across repeated fits, for example while varying penalties, families, or correlation structures.

Multivariate Case: For multiple predictors, $K + 1$ cluster centers are identified by k-means on an internally standardized predictor matrix via `make_partitions`. This is the partitioning mechanism used to determine the multivariate spline regions; see MacQueen (1967) for the classical clustering formulation and Kisi et al. (2025) for a recent applied example of k-means-driven partitioning in a nonlinear prediction setting. Midpoints between neighboring centers (those whose midpoint does not fall into a third cluster) serve as knot locations. Observations are assigned to the nearest cluster center using `get.knnx`, and the returned centers and knots are on the original predictor scale. The resulting partition structure is a type of Voronoi diagram and is stored in the fitted object as `make_partition_list`. The `do_not_cluster_on_these` argument can exclude certain predictors from clustering (e.g., a treatment indicator that should not drive partitioning). The lower-level clustering behavior can be further controlled by `cluster_args` and `neighbor_tolerance`, while `cluster_on_indicators` determines whether binary predictors are allowed to influence the partition geometry at all.

Standardizing Predictors: Higher-order polynomial terms can dramatically inflate or deflate the magnitude of basis expansions, introducing numerical instability. All polynomial basis expansions are scaled by $q_{0.69} - q_{0.31}$, where q_ζ is the ζ-th quantile of the expansion. For a standard normal distribution this quantity is approximately 1, so the scaling is close to one standard deviation for symmetric distributions. This fitting-stage rescaling is controlled by `standardize_expansions_for_fitting`, while knot construction is controlled separately by `standardize_predictors_for_knots`. The same scaling is applied to the constraint matrix to maintain smoothness, and coefficients are back-transformed to the original scale after fitting.

Smoothing Spline Penalty

Penalty Construction: The penalty matrix $\boldsymbol{\Lambda}_s$ penalizes the integrated squared total curvature of the fitted function over the observed predictor ranges. This is the step that makes the piecewise polynomial fit genuinely behave like a smoothing spline rather than merely a constrained regression spline. The package computes this penalty directly from the monomial structure of the basis rather than by appealing to a pre-tabulated spline basis. For a single partition k with basis expansion $\mathbf{x}_k = (\phi_1(\mathbf{t}), \dots, \phi_p(\mathbf{t}))^\top$ where each $\phi_i(\mathbf{t}) = \prod_{j=1}^{q} t_j^{\alpha_{ij}}$ is a multivariate monomial:

$$\boldsymbol{\beta}_k^\top \boldsymbol{\Lambda}_s \boldsymbol{\beta}_k = \int_{\mathbf{a}}^{\mathbf{b}} \|\tilde{f}_k''(\mathbf{t})\|^2 \, d\mathbf{t},$$

where $\mathbf{a}$ and $\mathbf{b}$ are the observed predictor minimums and maximums (computed globally from the data, not partition-specific), and $\tilde{f}_k(\mathbf{t}) = \mathbf{x}_k^\top \boldsymbol{\beta}_k$ is the fitted function for partition $\mathcal{P}_k$.

Total curvature operator. The integrated squared second derivative decomposes into q curvature operators, one per predictor. For predictor v, the curvature operator D_v is defined as

$$D_v = \frac{\partial^2}{\partial t_v^2} + \sum_{s \neq v} \frac{\partial^2}{\partial t_v \, \partial t_s}.$$

That is, D_v captures both the pure second derivative with respect to t_v and all mixed second partial derivatives involving t_v. The penalty matrix entries are then

$$[\boldsymbol{\Lambda}_s]_{ij} = \sum_{v=1}^{q} \int_{\mathbf{a}}^{\mathbf{b}} D_v(\phi_i) \, D_v(\phi_j) \, d\mathbf{t}.$$

Monomial derivative rule. For a monomial $\phi(\mathbf{t}) = \prod_j t_j^{\alpha_j}$, the derivatives entering D_v have closed forms. The pure second derivative is

$$\frac{\partial^2}{\partial t_v^2} \prod_j t_j^{\alpha_j} = \alpha_v(\alpha_v - 1) \, t_v^{\alpha_v - 2} \prod_{j \neq v} t_j^{\alpha_j},$$

which is zero when $\alpha_v < 2$. The mixed second derivative is

$$\frac{\partial^2}{\partial t_v \, \partial t_s} \prod_j t_j^{\alpha_j} = \alpha_v \alpha_s \, t_v^{\alpha_v - 1} t_s^{\alpha_s - 1} \prod_{j \neq v,s} t_j^{\alpha_j},$$

which is zero when $\alpha_v < 1$ or $\alpha_s < 1$. Applying D_v to a monomial ϕ_i produces a sum of monomials with known coefficients and exponent vectors.

Factorized integration. Because every $D_v(\phi_i)$ is polynomial, the product $D_v(\phi_i) \, D_v(\phi_j)$ is also polynomial and the multivariate integral factorizes over predictors:

$$\int_{\mathbf{a}}^{\mathbf{b}} \prod_{j=1}^{q} t_j^{e_j} \, d\mathbf{t} = \prod_{j=1}^{q} \frac{b_j^{e_j+1} - a_j^{e_j+1}}{e_j + 1}.$$

Crucially, this integral runs over *all* q predictor ranges, including predictors that do not appear in the integrand (for which $e_j = 0$ and the factor reduces to $b_j - a_j$). This ensures that the penalty is properly scaled relative to the volume of the predictor space.

Single-predictor verification. For $q = 1$ with expansion $\mathbf{x} = (1, t, t^2, t^3)^\top$ on $[a, b]$, the curvature operator reduces to $D_1 = \partial^2/\partial t^2$ (no mixed partials exist), and the penalty matrix reduces to

$$\boldsymbol{\Lambda}_s = \int_a^b \mathbf{x}''\mathbf{x}''^\top \, dt = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 4(b-a) & 6(b^2 - a^2) \\ 0 & 0 & 6(b^2 - a^2) & 12(b^3 - a^3) \end{pmatrix},$$

equivalent to the classical cubic smoothing spline penalty originally proposed by Reinsch.

Handling of non-spline predictors. Predictors specified via `just_linear_without_interactions` or `just_linear_with_interactions` do not receive higher-order polynomial expansions in the design matrix. To ensure their curvature contributions are still correctly computed (particularly through interaction terms), the implementation temporarily appends phantom higher-order columns (with zero data) for these predictors, computes the full curvature penalty on the augmented basis, and then subsets the result back to the original $p \times p$ dimensions. This ensures that interaction terms involving non-spline predictors receive appropriate penalty contributions without affecting the rest of the estimation pipeline.

Parallel computation. Because the total penalty is an additive sum over predictors ($\mathbf{\Lambda}_s = \sum_{v=1}^{q} \mathbf{\Lambda}_{s,v}$), the computation can be parallelized by distributing the per-predictor curvature matrices across workers via `parallel::parLapply` and summing the results. This is controlled by the `parallel_penalty` argument and is beneficial when q is large.

The penalty is computed by `get_2ndDerivPenalty` (single predictor or subset) and `get_2ndDerivPenalty_wrapper` (full assembly with optional parallelism and non-spline handling).

Because the smoothing penalty has zero eigenvalues for the intercept and linear terms (whose second derivatives vanish), an optional ridge penalty on lower-order terms is added for computational stability. The full penalty block for partition k is:

$$\mathbf{\Lambda}_k = \lambda_w \big(\mathbf{\Lambda}_s + \lambda_r \mathbf{\Lambda}_r + \sum_{m=1}^{M} \xi_{mk} \mathbf{L}_{mk}\big)$$

where λ_w is the global wiggle penalty (`wiggle_penalty`), λ_r is ridge penalty on linear and intercept terms (`flat_ridge_penalty`) multiplied by the wiggle penalty, and ξ_{mk} and $\mathbf{L}_{mk}$ denote optional additional penalty multipliers and matrices, including the predictor- and partition-specific components activated through `unique_penalty_per_predictor`, `unique_penalty_per_partition`, `predictor_penalties`, and `partition_penalties`. This assembly is handled by `compute_Lambda`.

The penalty matrix $\mathbf{\Lambda}$ is stored as a list of $K+1$ $p \times p$ square, symmetric, positive semi-definite matrices.

Penalty Optimization via Generalized Cross-Validation: After the structural pieces of the model are fixed, the main remaining question is how much smoothing to apply. In **lgspline**, that tuning is performed with generalized cross-validation, but it is carried out using the same constrained estimator that will be used in the final model fit. Penalty parameters are estimated on the log scale via exponential parameterization ($\lambda = \exp(\theta)$, $\theta \in \mathbb{R}$), ensuring positivity. The chain rule factor $\partial \exp(\theta)/\partial\theta = \exp(\theta) = \lambda$ is applied throughout. User-facing arguments (`initial_wiggle`, `initial_flat`, `predictor_penalties`, `partition_penalties`) accept values on the raw, natural scale; conversion to log scale is handled internally. The final tuned values and assembled penalty pieces are returned in the fitted object's `penalties` component.

The total penalty matrix $\mathbf{\Lambda}$ is constructed as:

$$\mathbf{\Lambda} = \lambda_w \mathbf{L}_w + \lambda_r \mathbf{L}_r + \sum_j \nu_j \mathbf{L}_j^{(\text{pred})} + \sum_k \tau_k \mathbf{L}_k^{(\text{part})},$$

where $\mathbf{L}_w$ is the integrated squared second-derivative penalty (i.e., $\mathbf{\Lambda}_s$ above), $\mathbf{L}_r$ is a ridge penalty on intercept and linear coefficients, $\mathbf{L}_j^{(\text{pred})}$ are predictor-specific penalties, and $\mathbf{L}_k^{(\text{part})}$ are partition-specific penalties. The scalars λ_w (`wiggle_penalty`), λ_r (`flat_ridge_penalty`),

$\{\nu_j\}$ (`predictor_penalties`), and $\{\tau_k\}$ (`partition_penalties`) are tuned. The unbiased generalized cross-validation criterion is

$$\mathrm{GCV}_u = \frac{\sum_{i=1}^N D_{ii}\, r_i^2}{N(1-\bar{W})^2},$$

where r_i are residuals on the link scale and $\bar{W} = \mathrm{tr}(\mathbf{H})/N$ is the mean of the hat-matrix diagonal. For identity link, $r_i = y_i - \hat{\eta}_i$. For non-identity links, the residuals are $r_i = g((y_i + \delta)/(1 + 2\delta)) - (\hat{\eta}_i + \delta)/(1 + 2\delta)$, where $\delta \geq 0$ is a pseudocount that stabilizes the link transformation, automatically tuned within `tune_Lambda` if not supplied to `delta`. Non-Gaussian families with observation weights ω_i have their residuals scaled by ω_i. When the family provides a custom deviance residual function, that function is used in place of the link-scale residuals.

Pseudocount selection. The pseudocount δ is chosen to make the transformed response distribution most closely approximate a t-distribution with $N-1$ degrees of freedom, in the sense of minimizing the (optionally weighted) mean absolute deviation between the sorted standardized transformed responses and the corresponding t-quantiles. This is solved via Brent's method over $[10^{-64}, 1]$. When the link is identity, or when the response naturally lies in the domain of the link function, $\delta = 0$. This behavior is exposed through the `delta` argument in `lgspline`: supplying a fixed numeric value bypasses the internal search, while leaving it `NULL` allows the tuning code to choose the stabilizing pseudocount automatically when needed.

Meta-penalty regularization. A regularization term pulls the predictor- and partition-specific penalty parameters toward 1 on the raw scale:

$$P_{\mathrm{meta}}(\lambda_w, \nu_j, \tau_k) = \frac{1}{2} c_{\mathrm{meta}} \sum_j (\nu_j - 1)^2 + \frac{1}{2} \cdot 10^{-32} (\lambda_w - 1)^2,$$

where c_{meta} is a user-specified coefficient (`meta_penalty`). The gradient of P_{meta} on the log scale, incorporating the exp chain rule, is $\partial P_{\mathrm{meta}}/\partial\theta_j = c_{\mathrm{meta}}(\nu_j - 1)\nu_j$ and $\partial P_{\mathrm{meta}}/\partial\theta_1 = 10^{-32}(\lambda_w - 1)\lambda_w$. The total objective is $\mathrm{GCV}_u + P_{\mathrm{meta}}$.

Closed-Form Gradient of GCV: The gradient of GCV_u with respect to $\theta_1 = \log \lambda_w$ is computed analytically via the quotient rule:

$$\frac{\partial \mathrm{GCV}_u}{\partial \theta_1} = \frac{1}{D^2}\left(\frac{\partial \mathcal{N}}{\partial \theta_1} D - \mathcal{N} \frac{\partial D}{\partial \theta_1}\right),$$

where $\mathcal{N} = \sum r_i^2$ (numerator) and $D = N(1-\bar{W})^2$ (denominator). The key intermediates are:

- $\partial\mathbf{G}/\partial\lambda_w$, computed from the matrix identity $\partial(\mathbf{X}^\top\mathbf{X} + \mathbf{\Lambda})^{-1}/\partial\lambda = -\mathbf{G}(\partial\mathbf{\Lambda}/\partial\lambda)\mathbf{G}$.
- $\partial\mathbf{G}^{1/2}/\partial\lambda_w$, derived from $\partial\mathbf{G}/\partial\lambda_w$ via the eigendecomposition chain rule.
- $\partial\bar{W}/\partial\lambda_w$, the derivative of the trace of the hat matrix $\mathbf{H} = \mathbf{XUGX}^\top$, which depends on both $\partial\mathbf{G}/\partial\lambda_w$ and $\partial\mathbf{G}^{1/2}/\partial\lambda_w$.
- $\partial\mathcal{N}/\partial\theta_1 = -2\mathbf{r}^\top\mathbf{X}(\partial(\mathbf{UG})/\partial\lambda_w)\mathbf{X}^\top\mathbf{y} \cdot \lambda_w$, via the chain rule applied to the residual vector.
- $\partial D/\partial\theta_1 = 2(1-\bar{W})(-\partial\bar{W}/\partial\lambda_w) \cdot \lambda_w$.

In the implementation, these quantities are assembled by a small set of helper routines: `compute_dG_dlambda` for $\partial\mathbf{G}/\partial\lambda$, `compute_dGhalf` for $\partial\mathbf{G}^{1/2}/\partial\lambda$, `compute_dW_dlambda_wrapper` for derivatives of the effective degrees-of-freedom term, `compute_trace_UGXX_wrapper` for the trace pieces entering GCV, and `compute_dG_u_dlambda_xy` for the derivative of the fitted-value quadratic form.

The full gradient is scaled by N before adding the meta-penalty gradient.

For the ridge penalty and predictor-/partition-specific penalties, a trace-ratio heuristic is used:

$$\frac{\partial \mathrm{GCV}_u}{\partial \lambda_l} \approx \frac{\mathrm{mean}(\mathrm{diag}(\mathbf{L}_l))}{\mathrm{mean}(\mathrm{diag}(\mathbf{\Lambda}))} \frac{\partial \mathrm{GCV}_u}{\partial \lambda_w},$$

and analogously for predictor- and partition-specific penalties, where $\mathbf{L}_j^{(\mathrm{pred})}$ or $\mathbf{L}_k^{(\mathrm{part})}$ replaces $\mathbf{L}_l$ in the numerator. This follows from a chain-rule argument: by the Leibniz rule and the inverse derivative, $\partial \lambda_w / \partial \lambda_l = (\partial \mathbf{\Lambda} / \partial \lambda_l)(\partial \mathbf{\Lambda} / \partial \lambda_w)^{-1}$. Since the derivative appears as a matrix rather than a scalar, the mean-diagonal ratio provides a scalar summary. Once the derivative for λ_w is in hand, the derivatives of other penalties are cheap to compute. The exp chain rule is then applied: $\partial/\partial\theta = (\partial/\partial\lambda) \cdot \lambda$.

Optimization Procedure: **Grid search initialization.** The GCV_u criterion is evaluated over a grid of candidate values for (λ_w, λ_r) on the log scale. All combinations of user-supplied candidate vectors (`initial_wiggle` and `initial_flat`) are formed, and the combination yielding the smallest finite GCV_u is selected as the starting point for BFGS optimization. Grid points producing non-finite GCV_u are discarded. If all grid points fail, an error is raised advising the user to check the data or adjust the grid.

Damped BFGS optimizer. A custom damped BFGS quasi-Newton optimizer, implemented in `efficient_bfgs`, minimizes $\mathrm{GCV}_u + P_{\mathrm{meta}}$. When analytic gradients are not usable, the fallback finite-difference helper is `approx_grad`.

Iterations 1-2: steepest descent. The first two iterations use steepest descent with a damping factor α: $\boldsymbol{\phi}^{(t+1)} = \boldsymbol{\phi}^{(t)} - \alpha \nabla_{\boldsymbol{\phi}}$.

Iterations 3+: BFGS. From iteration 3, an inverse Hessian approximation $\mathbf{J}^{(t)}$ is maintained via the standard secant update. Let $\mathbf{s}^{(t)} = \boldsymbol{\phi}^{(t)} - \boldsymbol{\phi}^{(t-1)}$ and $\mathbf{v}^{(t)} = \nabla^{(t)} - \nabla^{(t-1)}$. The BFGS update is:

$$\mathbf{J}^{(t+1)} = (\mathbf{I} - \mathbf{u}\mathbf{s}\mathbf{v}^\top)\mathbf{J}^{(t)}(\mathbf{I} - \mathbf{u}\mathbf{v}\mathbf{s}^\top) + \mathbf{u}\mathbf{s}\mathbf{s}^\top, \qquad \mathbf{u} = (\mathbf{v}^\top\mathbf{s})^{-1}.$$

When $|\mathbf{v}^\top\mathbf{s}| < 10^{-64}$, the approximation is reset to $\mathbf{I}$ and the iteration is flagged for restart. The search direction is $\mathbf{d}^{(t)} = -\mathbf{J}^{(t)}\nabla^{(t)}$.

Step acceptance. A step is accepted if $\mathrm{GCV}_u^{(\mathrm{new})} \leq \mathrm{GCV}_u^{(\mathrm{old})}$. On rejection, α is halved. If $\alpha < 2^{-10}$ (early iterations) or $\alpha < 2^{-12}$ (later iterations), the optimizer terminates with the best solution found.

Convergence. The optimizer terminates when $|\mathrm{GCV}_u^{(t)} - \mathrm{GCV}_u^{(t-1)}| < \epsilon$ or $\|\boldsymbol{\phi}^{(t)} - \boldsymbol{\phi}^{(t-1)}\|_\infty < \epsilon$, provided at least 10 iterations have elapsed, for penalties $\boldsymbol{\phi} = \lambda_w, \lambda_r, \nu_j, \tau_j, \ldots$.

Alternative. A base-R `optim` call with method `"BFGS"` and finite-difference gradients can be used used instead via `use_custom_bfgs=FALSE`, which uses `stats::optim`.

Post-optimization inflation. After optimization, the penalty parameters are inflated by a factor $((N+2)/(N-2))^2$ to counteract the in-sample bias toward underpenalization inherent in GCV-type criteria.

The tuning loop is implemented in `tune_Lambda`.

Incorporating Non-Spline Effects

Multiple fixed effects are accommodated naturally in the LMSS framework because spline effects, linear effects, and many interaction terms all live in the same partition-wise polynomial expansion. The distinction is therefore not whether a term is "allowed" by the solver, but whether it receives full spline treatment or remains structurally linear across partitions.

The constrained framework naturally accommodates non-spline terms. If only linear terms are included for a predictor (via `just_linear_without_interactions` or `just_linear_with_interactions`), the first-derivative smoothing constraint forces the linear coefficient to be identical across all partitions, since the derivative of a linear function is its slope. This is not an algorithmic modification but a natural consequence of the constraint structure.

For example, a model with one spline effect and a linear treatment indicator interaction will naturally keep the treatment-time interaction coefficient constant across partitions while allowing the time effect to vary nonlinearly. This conveniently extends to arbitrary combinations of spline and linear terms without requiring special handling.

When `blockfit = TRUE` is specified alongside `just_linear_without_interactions`, the flat-block path provides an alternative enforcement mechanism. Rather than relying on constraint projection, flat coefficients are pooled structurally across partitions during backfitting. The two approaches agree at the point estimate but differ in their uncertainty quantification; see the Blockfit section above.

Integration

Because the fitted object retains an explicit polynomial representation in each partition, numerical integration can be carried out in a fairly direct way. The package wraps that calculation in a user-facing S3 method so the user does not need to manage knot boundaries or partition membership by hand.

In the user-facing interface, numerical integration is applied through `integrate.lgspline`, which applies Gauss-Legendre quadrature to predictions from the fitted model produced by `predict.lgspline`.

Implementation: For a user-supplied rectangular domain, `integrate.lgspline` constructs a tensor-product grid of Gauss-Legendre nodes, evaluates the fitted model at those points, and forms the weighted sum. This works for both univariate and multivariate models, respects the fitted partition structure automatically, and avoids requiring the user to keep track of knot boundaries by hand.

The `vars` argument selects which predictors are integrated over. Predictors not listed in `vars` are held fixed at `initial_values` when supplied, or otherwise at the midpoint of their observed training range. The optional `B_predict` argument makes it possible to integrate posterior draws or other alternate coefficient sets, and `n_quad` controls the number of Gauss-Legendre nodes used per integrated dimension.

Integration is performed on the response scale by default. Setting `link_scale = TRUE` instead integrates the linear predictor $\eta = f(\mathbf{t})$. For identity-link Gaussian models the two scales coincide.

Lagrange Multipliers

When `return_lagrange_multipliers = TRUE`, the multiplier vector

$$\boldsymbol{\lambda} = (\mathbf{A}^\top \mathbf{G}\mathbf{A})^{-1}\mathbf{A}^\top \hat{\boldsymbol{\beta}}$$

is returned. These quantify the sensitivity of the penalized objective to relaxing each smoothness or user-supplied equality constraint. When constraint target values are nonzero ($\mathbf{A}^\top \boldsymbol{\beta}_0 \neq \mathbf{0}$), the modified formulation is used:

$$\boldsymbol{\lambda} = (\mathbf{A}^\top \mathbf{G}\mathbf{A})^{-1}\mathbf{A}^\top (\hat{\boldsymbol{\beta}} - \boldsymbol{\beta}_0)$$

where $\mathbf{A}^\top\boldsymbol{\beta}_0$ is the vector of constraint target values. Multipliers are `NULL` when no constraints are active ($\mathbf{A}$ is `NULL` or $K = 0$).

For inequality constraints, multipliers are returned as computed by `solve.QP`. The Lagrange multipliers for active inequality constraints can be used diagnostically to identify which shape constraints are most costly in terms of goodness of fit.

S3 Methods

Standard S3 methods are provided for objects of class `lgspline`:

- `print.lgspline` and `summary.lgspline`: Provide concise model summaries, with `print.summary.lgspline` formatting coefficient tables in a familiar regression-style layout.
- `logLik.lgspline`: Returns a standard `logLik` object. For Gaussian responses with identity link, the exact log-likelihood is computed. When a correlation structure is present via `VhalfInv`, the log-likelihood includes the $\log|\mathbf{V}^{-1/2}|$ adjustment and the corresponding whitened quadratic form. For other families, the method falls back to `family$aic` or a deviance-based approximation. An `include_prior` argument (default `TRUE`) optionally adds the Gaussian prior penalty interpretation of the smoothing spline penalty $-\frac{1}{2\sigma^2}\tilde{\boldsymbol{\beta}}^\top\boldsymbol{\Lambda}\tilde{\boldsymbol{\beta}}$ to obtain a penalized MAP log-likelihood.
- `predict.lgspline`: Produces fitted values and related quantities (e.g., derivatives and standard errors through `se.fit = TRUE`), lets `new_predictors` override `newdata`, accepts alternate coefficient lists through `B_predict`, and supports prediction on new predictor matrices consistent with the original spline expansions.
- `coef.lgspline`: Extracts partition-specific coefficient vectors.
- `confint.lgspline`: Extracts confidence intervals. When the inverse Hessian from BFGS optimization is available for correlation parameters, intervals for those correlation parameters are returned on the working (transformed) scale and should be back-transformed as described in the correlation section.
- `plot.lgspline`: For one-dimensional fits, produces base R graphics showing the fitted function (with optional partition-wise formulas) and supports overlay via `add = TRUE`. For two or more predictors, an interactive **plotly**-based visualization is returned. Specific predictors may be selected via `vars`.
- `integrate.lgspline`: Computes definite integrals of the fitted surface over rectangular domains by Gauss-Legendre quadrature.

Additional user-facing helpers include `wald_univariate` for coefficient-wise Wald inference, `generate_posterior` for posterior and posterior-predictive sampling, `generate_posterior_correlation` for correlation-aware posterior simulation, `equation` for closed-form display of the fitted partition formulas, and `find_extremum` for optimizing the fitted surface or a custom acquisition function built from it.

References

Buse, A. and Lim, L. (1977). Cubic Splines as a Special Case of Restricted Least Squares. *Journal of the American Statistical Association*, 72, 64-68.

Eilers, P. H. and Marx, B. D. (1996). Flexible Smoothing with B-splines and Penalties. *Statistical Science*, 11(2), 89-121.

Ezhov, N., Neitzel, F. and Petrovic, S. (2018). Spline Approximation, Part 1: Basic Methodology. *Journal of Applied Geodesy*, 12(2), 139-155.

Goldfarb, D. and Idnani, A. (1983). A Numerically Stable Dual Method for Solving Strictly Convex Quadratic Programs. *Mathematical Programming*, 27(1), 1-33.

Harville, D. A. (1977). Maximum Likelihood Approaches to Variance Component Estimation and to Related Problems. *Journal of the American Statistical Association*, 72(358), 320-338.

Hastie, T. J. and Tibshirani, R. J. (1990). *Generalized Additive Models*. Chapman & Hall/CRC.

Kisi, O., Heddam, S., Parmar, K. S., Petroselli, A., Kulls, C. and Zounemat-Kermani, M. (2025). Integration of Gaussian Process Regression and K Means Clustering for Enhanced Short Term Rainfall Runoff Modeling. *Scientific Reports*, 15, 7444.

MacQueen, J. B. (1967). Some Methods for Classification and Analysis of Multivariate Observations. In *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability*, Volume 1, 281-297. University of California Press.

McCullagh, P. and Nelder, J. A. (1989). *Generalized Linear Models*. Chapman & Hall, 2nd edition.

Murray, I., Adams, R. P. and MacKay, D. J. C. (2010). Elliptical Slice Sampling. *Proceedings of the 13th International Conference on Artificial Intelligence and Statistics (AISTATS)*, 9, 541-548.

Nocedal, J. and Wright, S. J. (2006). *Numerical Optimization* (2nd ed.). Springer.

Patterson, H. D. and Thompson, R. (1971). Recovery of Inter-Block Information When Block Sizes Are Unequal. *Biometrika*, 58, 545-554.

Pya, N. and Wood, S. N. (2015). Shape Constrained Additive Models. *Statistics and Computing*, 25(3), 543-559.

Reinsch, C. H. (1967). Smoothing by Spline Functions. *Numerische Mathematik*, 10, 177-183.

Ruppert, D., Wand, M. P. and Carroll, R. J. (2003). *Semiparametric Regression*. Cambridge University Press.

Searle, S. R., Casella, G. and McCulloch, C. E. (2006). *Variance Components*. Wiley.

Wahba, G. (1990). *Spline Models for Observational Data*. SIAM.

Wood, S. N. (2006). On Confidence Intervals for Generalized Additive Models Based on Penalized Regression Splines. *Australian & New Zealand Journal of Statistics*, 48(4), 445-464.

Wood, S. N. (2011). Fast Stable Restricted Maximum Likelihood and Marginal Likelihood Estimation of Semiparametric Generalized Linear Models. *Journal of the Royal Statistical Society: Series B*, 73(1), 3-36.

Wood, S. N. (2017). *Generalized Additive Models: An Introduction with R*. CRC Press, 2nd edition.

`equation` *Print Closed-Form Fitted Equation from lgspline Model*

Description

Displays the closed-form polynomial equation for each partition of a fitted lgspline model, along with partition boundary or cluster center information. Optionally prints the first derivative, second derivative, or antiderivative of the fitted equation with respect to a single specified variable.

Usage

```
equation(object, ...)

## S3 method for class 'lgspline'
equation(
  object,
  digits = 4,
  scientific = FALSE,
  show_bounds = TRUE,
  predictor_names = NULL,
  response_name = NULL,
  collapse_zero = TRUE,
  first_derivative = NULL,
  second_derivative = NULL,
  antiderivative = NULL,
  ...
)

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

## S3 method for class 'lgspline'
equation(
  object,
  digits = 4,
  scientific = FALSE,
  show_bounds = TRUE,
  predictor_names = NULL,
  response_name = NULL,
  collapse_zero = TRUE,
  first_derivative = NULL,
  second_derivative = NULL,
  antiderivative = NULL,
  ...
)

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

Arguments

`object`	A fitted lgspline model object.
`...`	Not used.
`digits`	Integer; decimal places for coefficient display. Default 4.
`scientific`	Logical; use scientific notation for coefficients with absolute value < 1e-3 or > 1e4. Default FALSE.

`show_bounds`	Logical; display partition bounds (1D) or knot midpoint boundaries (multi-D). Default TRUE.
`predictor_names`	Character vector; custom names for predictor variables. If NULL (default), uses original column names or "_j_" labels.
`response_name`	Character; label for response. If NULL (default), uses "y" for identity link Gaussian, or "link(E[y])" otherwise.
`collapse_zero`	Logical; omit terms with coefficient exactly 0. Default TRUE.
`first_derivative`	Default: NULL. Character name or integer index of the predictor variable with respect to which the first derivative is printed. Only one variable at a time is supported. When non-NULL, the printed equations show df/dx_j for each partition.
`second_derivative`	Default: NULL. Character name or integer index of the predictor variable with respect to which the second derivative is printed. Only one variable at a time is supported. When non-NULL, the printed equations show d^2f/dx_j^2 for each partition. Ignored if `first_derivative` is also non-NULL.
`antiderivative`	Default: NULL. Character name or integer index of the predictor variable with respect to which the antiderivative (indefinite integral) is printed. Only one variable at a time is supported. When non-NULL, the printed equations show $\int f\,dx_j$ for each partition, with an unspecified constant of integration C. Ignored if `first_derivative` or `second_derivative` is also non-NULL.
`x`	An object returned by `equation()` for printing.

Details

For 1D models with K knots, partition boundaries are displayed as intervals on the predictor scale. For multi-predictor models, partition boundaries are computed as the midpoints between adjacent cluster centers along each predictor dimension. When the model's `make_partition_list` contains `knots` (midpoint boundaries between clusters), those are used directly. Otherwise, cluster centers are displayed.

Coefficients are displayed on the original (unstandardized) predictor scale. For GLMs with non-identity link, the left-hand side shows the link function applied to the expected response.

Derivative and antiderivative modes. Only one of `first_derivative`, `second_derivative`, or `antiderivative` may be non-NULL. If more than one is supplied, the priority order is: first derivative, second derivative, antiderivative.

Derivatives and antiderivatives are computed symbolically from the polynomial coefficients. For a term ax^n, the first derivative is nax^{n-1}, the second derivative is $n(n-1)ax^{n-2}$, and the antiderivative is $ax^{n+1}/(n+1)$. Cross-terms (interactions) involving the target variable are differentiated or integrated with respect to that variable only, treating all other variables as constants.

A warning is emitted if the user attempts to differentiate or integrate with respect to more than one variable simultaneously. Multi-variable calculus operations should be performed one variable at a time by calling `equation()` repeatedly.

Value

Invisibly returns a list with components:

formulas Character vector of equation strings per partition.

bounds Matrix or list of partition boundary information.

link Character; link function name.

mode Character; one of "equation", "first_derivative", "second_derivative", or "antiderivative".

variable Character; the variable name for the calculus operation, or NULL if mode is "equation".

See Also

`lgspline`, `plot.lgspline`, `coef.lgspline`

Examples

```
## 1D example
set.seed(1234)
t <- runif(500, -5, 5)
y <- 2*sin(t) + 0.1*t^2 + rnorm(length(t), 0, 0.5)
fit <- lgspline(t, y, K = 2)
equation(fit)
equation(fit, digits = 2, predictor_names = "time")

## First derivative with respect to predictor
equation(fit, first_derivative = 1)

## Second derivative
equation(fit, second_derivative = 1)

## Antiderivative
equation(fit, antiderivative = 1)

## 2D example with named predictors
x1 <- runif(300, 0, 10)
x2 <- runif(300, 0, 10)
y <- x1 + 0.5*x2 + 0.1*x1*x2 + rnorm(300)
fit2d <- lgspline(cbind(x1, x2), y, K = 3)
equation(fit2d, predictor_names = c("Length", "Width"))

## Derivative w.r.t. first variable only
equation(fit2d, first_derivative = "Length",
         predictor_names = c("Length", "Width"))

## GLM example
y_bin <- rbinom(500, 1, plogis(0.5*t))
fit_glm <- lgspline(t, y_bin, family = binomial(), K = 1)
equation(fit_glm)
```

find_extremum *Find the Extremum of a Fitted lgspline*

Description

Finds the global maximum or minimum of a fitted lgspline using L-BFGS-B, with options for partition-based heuristics, stochastic exploration, and custom objective functions (e.g., acquisition functions for Bayesian optimization).

Usage

```
find_extremum(
  object,
  vars = NULL,
  quick_heuristic = TRUE,
  initial = NULL,
  B_predict = NULL,
  minimize = FALSE,
  stochastic = FALSE,
  stochastic_draw = function(mu, sigma, ...) {
     N <- length(mu)
     rnorm(N, mu,
    sigma)
 },
  sigmasq_predict = object$sigmasq_tilde,
  custom_objective_function = NULL,
  custom_objective_derivative = NULL,
  ...
)
```

Arguments

`object`	A fitted lgspline model object.
`vars`	Integer or character vector; indices or names of predictors to optimize over. Default NULL optimizes all predictors.
`quick_heuristic`	Logical; if TRUE (default) searches only the best-performing partition. If FALSE, initiates searches from all partition local maxima.
`initial`	Numeric vector; optional starting values. Useful for fixing binary predictors. Default NULL.
`B_predict`	List; optional coefficient list for prediction, e.g. from `generate_posterior`. Default NULL uses `object$B`.
`minimize`	Logical; find minimum instead of maximum. Default FALSE.
`stochastic`	Logical; add noise during optimization for exploration. Default FALSE.

`stochastic_draw`	Function; generates noise for stochastic optimization. Takes `mu`, `sigma`, and `...`. Default `rnorm(length(mu), mu, sigma)`.
`sigmasq_predict`	Numeric; variance for stochastic draws. Default `object$sigmasq_tilde`.
`custom_objective_function`	Function; optional custom objective. Takes `mu`, `sigma`, `y_best`, `...`. Default NULL.
`custom_objective_derivative`	Function; optional gradient of `custom_objective_function`. Takes `mu`, `sigma`, `y_best`, `d_mu`, `...`. Default NULL.
`...`	Additional arguments passed to internal optimization routines.

Value

A list with elements:

- **t** Numeric vector; predictor values at the extremum.
- **y** Numeric; objective value at the extremum.

See Also

`lgspline`, `generate_posterior`

Examples

```
set.seed(1234)
t <- runif(1000, -10, 10)
y <- 2*sin(t) + -0.06*t^2 + rnorm(length(t))
model_fit <- lgspline(t, y)
plot(model_fit)

max_point <- find_extremum(model_fit)
min_point <- find_extremum(model_fit, minimize = TRUE)
abline(v = max_point$t, col = 'blue')
abline(v = min_point$t, col = 'red')

## Expected improvement acquisition function
ei_obj <- function(mu, sigma, y_best, ...) {
  d <- y_best - mu
  d * pnorm(d/sigma) + sigma * dnorm(d/sigma)
}
ei_deriv <- function(mu, sigma, y_best, d_mu, ...) {
  d <- y_best - mu
  z <- d/sigma
  d_z <- -d_mu/sigma
  pnorm(z)*d_mu - d*dnorm(z)*d_z + sigma*z*dnorm(z)*d_z
}

post_draw <- generate_posterior(model_fit)
acq <- find_extremum(model_fit,
                     stochastic = TRUE,
                     B_predict = post_draw$post_draw_coefficients,
                     sigmasq_predict = post_draw$post_draw_sigmasq,
                     custom_objective_function = ei_obj,
                     custom_objective_derivative = ei_deriv)
abline(v = acq$t, col = 'green')
```

`generate_posterior` *Generate Posterior Samples from a Fitted lgspline*

Description

Draws from the posterior distribution of model coefficients, with optional dispersion sampling, posterior predictive draws, and propagation of uncertainty in estimated correlation parameters.

Usage

```
generate_posterior(
  object,
  new_sigmasq_tilde = object$sigmasq_tilde,
  new_predictors = NULL,
  theta_1 = 0,
  theta_2 = 0,
  posterior_predictive_draw = function(N, mean, sqrt_dispersion, ...) {
     rnorm(N,
    mean, sqrt_dispersion)
 },
  draw_dispersion = TRUE,
  include_posterior_predictive = FALSE,
  num_draws = 1,
  enforce_qp_constraints = TRUE,
  draw_correlation = FALSE,
  correlation_param_mean = NULL,
  correlation_param_vcov = NULL,
  correlation_VhalfInv_fxn = NULL,
  correlation_Vhalf_fxn = NULL,
  correlation_param_vcov_scale = NULL,
  include_warnings = TRUE,
  ...
)
```

Arguments

`object` A fitted `lgspline` model object.

`new_sigmasq_tilde`
Numeric; dispersion $\tilde{\sigma}^2$ used as the point estimate when `draw_dispersion = FALSE`. Default `object$sigmasq_tilde`.

`new_predictors`	Matrix; predictor matrix for posterior predictive sampling. Default uses in-sample predictors.
`theta_1`	Numeric; shape increment for the inverse-gamma prior on σ^2. Default 0.
`theta_2`	Numeric; rate increment for the inverse-gamma prior. Default 0.
`posterior_predictive_draw`	Function; sampler for posterior predictive realisations. Must accept `N`, `mean`, `sqrt_dispersion`, `...`. Default `rnorm`.
`draw_dispersion`	Logical; sample σ^2 from its posterior. Default TRUE.
`include_posterior_predictive`	Logical; generate posterior predictive draws at `new_predictors`. Default FALSE.
`num_draws`	Positive integer; number of draws. Default 1.
`enforce_qp_constraints`	Logical; if TRUE, enforce active QP inequality constraints during posterior sampling via the stored elliptical-slice constrained sampler. Default TRUE.
`draw_correlation`	Logical; propagate correlation parameter uncertainty. Requires `VhalfInv_fxn` and `VhalfInv_params_estimates` in the fitted object. Default FALSE.
`correlation_param_mean`	Numeric vector; mean of the approximate normal posterior for correlation parameters on the unbounded (working) scale. Default: `object$VhalfInv_params_estimates`.
`correlation_param_vcov`	Matrix; variance-covariance for correlation parameter draws. Default: inverse Hessian from BFGS (`object$VhalfInv_params_vcov`).
`correlation_VhalfInv_fxn`	Function; maps correlation parameter vector to $\mathbf{V}^{-1/2}$. Default `object$VhalfInv_fxn`.
`correlation_Vhalf_fxn`	Function or NULL; maps to $\mathbf{V}^{1/2}$. Passed through to `generate_posterior_correlation`; the current correlation-aware posterior path only requires `correlation_VhalfInv_fxn`.
`correlation_param_vcov_scale`	NULL or numeric; if supplied, divides a user-supplied `correlation_param_vcov` by this value before passing it to `generate_posterior_correlation`. When NULL, no additional scaling is applied.
`include_warnings`	Logical; emit warnings for degenerate draws, constraint violations, etc. Default TRUE.
`...`	Additional arguments forwarded to the GLM weight function, dispersion function, and `posterior_predictive_draw`.

Details

Uses a Laplace approximation centred at the MAP estimate for non-Gaussian responses.

Dispersion posterior. When `draw_dispersion = TRUE`, σ^2 is drawn from

$$\sigma^2 \mid \mathbf{y} \sim \mathrm{InvGamma}(\alpha_1, \alpha_2),$$

where

$$\alpha_1 = \theta_1 + \tfrac{1}{2}(N - s \cdot \mathrm{tr}(\mathbf{H})), \quad \alpha_2 = \theta_2 + \tfrac{1}{2}(N - s \cdot \mathrm{tr}(\mathbf{H}))\tilde{\sigma}^2,$$

$\mathbf{H} = \mathbf{XUGX}^\top$ is the hat matrix, $s = 1$ when `unbias_dispersion = TRUE` (else $s = 0$), and $\theta_1 = \theta_2 = 0$ recovers an improper uniform prior.

Correlation parameter posterior. When `draw_correlation = TRUE` and the fitted model contains an estimated correlation structure, each draw first samples $\boldsymbol{\rho}$ from

$$\boldsymbol{\rho}^{(m)} \sim \mathcal{N}(\hat{\boldsymbol{\rho}}_{\mathrm{REML}}, \mathbf{H}^{-1}_{\mathrm{BFGS}}),$$

rebuilds the posterior covariance under the drawn correlation structure (reusing all pre-computed design matrices, constraints, and penalty matrices) and then draws coefficients from the updated posterior. Knot placement, partitioning, coefficient re-estimation, and GCV tuning are skipped entirely. Draws producing non-positive-definite correlation matrices are rejected and redrawn (up to 50 attempts).

When `draw_correlation = FALSE` (default), correlation parameters are fixed at their estimated values.

Inequality constraints. Active QP inequalities can be enforced during posterior sampling via elliptical slice sampling, producing draws from the corresponding truncated multivariate normal posterior on the coefficient scale. The public `enforce_qp_constraints` argument is forwarded to the stored sampler for both the standard and correlation-aware posterior paths.

Value

When `num_draws = 1`, a named list:

post_draw_coefficients List of length K+1; per-partition coefficient vectors on the original scale.

post_draw_sigmasq Drawn (or fixed) dispersion.

post_pred_draw Posterior predictive vector (only when `include_posterior_predictive = TRUE`).

post_draw_correlation_params Drawn correlation parameters on the working scale (only when `draw_correlation = TRUE`).

When `num_draws > 1`, each element becomes a list of length `num_draws`, and `post_pred_draw` (if requested) is an $N_{\mathrm{new}} \times M$ matrix, where $M = \mathrm{num\_draws}$.

See Also

`lgspline`, `generate_posterior_correlation`, `wald_univariate`

Examples

```
set.seed(1234)
n_blocks <- 100; block_size <- 5; N <- n_blocks * block_size
rho_true <- 0.3
t <- seq(-5, 5, length.out = N)
true_mean <- sin(t)
errors <- Reduce("rbind",
  lapply(1:n_blocks, function(i) {
    sigma <- diag(block_size) + rho_true *
```

```
      (matrix(1, block_size, block_size) - diag(block_size))
    matsqrt(sigma) %*% rnorm(block_size)
  })
)
y <- true_mean + errors * 0.5

model_fit <- lgspline(t, y,
  K = 3,
  correlation_id = rep(1:n_blocks, each = block_size),
  correlation_structure = "exchangeable",
  include_warnings = FALSE
)

## Propagate correlation uncertainty across 50 draws
post <- generate_posterior(model_fit,
  draw_correlation = TRUE, num_draws = 50,
  include_warnings = FALSE
)

## Fixed correlation parameters for comparison
post_fixed <- generate_posterior(model_fit, num_draws = 50)

corr_draws <- unlist(post$post_draw_correlation_params)
rho_draws <- exp(-exp(corr_draws))
print(summary(rho_draws))
```

---

generate_posterior_correlation

Generate Posterior Samples Propagating Correlation Parameter Uncertainty

---

Description

Called internally by generate_posterior when draw_correlation = TRUE, but can be used directly for finer control. For each draw, samples the correlation parameter vector from its approximate normal posterior, rebuilds the posterior covariance under that drawn correlation structure without re-solving for a new coefficient mode, then draws coefficients from the updated posterior.

Usage

```
generate_posterior_correlation(
  object,
  new_sigmasq_tilde = object$sigmasq_tilde,
  new_predictors = NULL,
  theta_1 = 0,
  theta_2 = 0,
  posterior_predictive_draw = function(N, mean, sqrt_dispersion, ...) {
      rnorm(N,
    mean, sqrt_dispersion)
 },
  draw_dispersion = TRUE,
  include_posterior_predictive = FALSE,
  num_draws = 1,
  enforce_qp_constraints = TRUE,
  correlation_param_mean = NULL,
  correlation_param_vcov_sc = NULL,
  correlation_VhalfInv_fxn = NULL,
  correlation_Vhalf_fxn = NULL,
  include_warnings = TRUE,
  ...
)
```

Arguments

`object`	A fitted `lgspline` object with a correlation structure (i.e., `VhalfInv_fxn` and `VhalfInv_params_estimates` present, or supplied via override arguments).
`new_sigmasq_tilde`	Numeric; dispersion starting value when `draw_dispersion = FALSE`. Default `object$sigmasq_tilde`.
`new_predictors`	Matrix or NULL; predictor matrix for posterior predictive sampling. Default uses in-sample predictors.
`theta_1`	Numeric; shape increment for the inverse-gamma prior. Default 0.
`theta_2`	Numeric; rate increment for the inverse-gamma prior. Default 0.
`posterior_predictive_draw`	Function; sampler for posterior predictive realisations. Default `rnorm`.
`draw_dispersion`	Logical; sample σ^2 within each draw. Default TRUE.
`include_posterior_predictive`	Logical; generate posterior predictive draws. Default FALSE.
`num_draws`	Positive integer; number of draws (each requires one correlation parameter sample and one covariance rebuild). Default 1.
`enforce_qp_constraints`	Logical; if TRUE, enforce active QP inequality constraints during each coefficient draw via the stored elliptical-slice constrained sampler. Default TRUE.
`correlation_param_mean`	Numeric vector or NULL; mean of the approximate normal posterior on the working scale. Default: `object$VhalfInv_params_estimates`. Supplying this allows correlation draws for models fit with a fixed (non-optimised) correlation structure.
`correlation_param_vcov_sc`	Matrix or NULL; variance-covariance on the working scale. Default: `object$VhalfInv_params_vcov`. No further scaling is applied within this function.
`correlation_VhalfInv_fxn`	Function or NULL; maps parameter vector to $\mathbf{V}^{-1/2}$. Default `object$VhalfInv_fxn`.

`correlation_Vhalf_fxn`
: Function or NULL; maps to $\mathbf{V}^{1/2}$. Not consumed in the current method body but resolved for interface consistency and potential future use.

`include_warnings`
: Logical; emit warnings. Default TRUE.

`...`
: Additional arguments forwarded to the GLM weight function, dispersion function, and `posterior_predictive_draw`.

Details

Each draw proceeds in three steps:

1. **Draw correlation parameters.** $\boldsymbol{\rho}^{(m)} \sim \mathcal{N}(\hat{\boldsymbol{\rho}}_{\mathrm{REML}}, \mathbf{H}_{\mathrm{BFGS}}^{-1})$ on the unbounded working scale. Draws producing a non-PD correlation matrix are rejected and redrawn (up to 50 attempts); if all fail, the point estimate is used with a warning.
2. **Rebuild posterior covariance.** Using the already-expanded $\mathbf{X}_k$, $\mathbf{A}$, and $\boldsymbol{\Lambda}$ from the original fit, recompute only the covariance-side quantities implied by the drawn correlation structure:

   $$\mathbf{G}_{\mathrm{correct}}^{(m)} = \left(\mathbf{X}^{\top}\mathbf{W}\mathbf{D}\mathbf{V}^{-1}(\boldsymbol{\rho}^{(m)})\mathbf{X} + \boldsymbol{\Lambda}\right)^{-1},$$

   and from this the updated constraint projection $\mathbf{U}^{(m)}$ and effective degrees of freedom $\mathrm{trace}(\mathbf{H}^{(m)})$. The coefficient mode $\hat{\boldsymbol{\beta}}_{\mathrm{raw}}$ and fitted mean $\tilde{\mathbf{y}}$ are held fixed at the original fit values. Knot placement, partitioning, polynomial expansion, penalty tuning, and coefficient re-estimation are all skipped entirely.
3. **Draw coefficients.** Updated quantities (`U`, `Ghalf_correct`, `VhalfInv`, `sigmasq_tilde`, `trace_XUGX`) are passed to the stored closure via `override_*` arguments so that the draw is centred at the original mode but uses the covariance implied by the drawn correlation structure. The stored mode `object$B_raw` is passed as `override_B_raw` so it is not recomputed.

Why the mode is held fixed. Re-solving for a new MAP estimate under each drawn correlation structure is expensive, requires iterative solvers, and risks convergence failures on draws far from the REML estimate. The posterior draw is centred at the original mode, which remains a reasonable approximation when the REML surface is not sharply peaked. The covariance update captures the primary effect of correlation uncertainty on posterior width and shape.

BFGS inverse Hessian caveat. The BFGS inverse Hessian approximation for the correlation parameter covariance is asymptotically valid but may be poor for small samples, near-boundary estimates, or multimodal REML surfaces. It is not guaranteed to converge to the observed information matrix. Users should inspect `object$VhalfInv_params_vcov` before relying on these draws.

Value

When `num_draws = 1`, a named list:

post_draw_coefficients List of length K+1; per-partition coefficient vectors on the original scale.

post_draw_sigmasq Drawn dispersion.

post_pred_draw Posterior predictive vector (only when `include_posterior_predictive = TRUE`).

post_draw_correlation_params Drawn correlation parameters on the working scale.

When num_draws > 1:

post_draw_coefficients List of num_draws lists of K+1 coefficient vectors.

post_draw_sigmasq List of num_draws scalars.

post_pred_draw $N_{\text{new}} \times M$ matrix, where $M = \text{num\_draws}$ (only when include_posterior_predictive = TRUE).

post_draw_correlation_params List of num_draws vectors.

See Also

generate_posterior, lgspline, lgspline.fit

Examples

```
## See ?generate_posterior for a complete worked example.
```

---

get_B_verification_examples
Verification Examples for get_B

Description

Simple, self-contained examples that reviewers can run to verify that get_B produces correct output. These exercise Path 2 (Gaussian, no correlation) and Path 3 (binomial GLM).

Examples

```
## Not run:
## Example 1: Path 2 - Gaussian identity, with knots
set.seed(42)
t <- runif(200, -5, 5)
y <- sin(t) + rnorm(200, 0, 0.5)
fit1 <- lgspline(t, y, K = 3, opt = FALSE, wiggle_penalty = 1e-4)
stopifnot(inherits(fit1, "lgspline"))
stopifnot(length(fit1$B) == 4)  # K+1 = 4 partitions
cat("Example 1 passed: Gaussian identity, K=3\n")

preds1 <- predict(fit1, new_predictors = rnorm(10))
stopifnot(all(is.finite(preds1)))
cat("  Predictions finite: OK\n")

## Example 2: Path 2 - Gaussian identity, K=0 (no constraints)
fit2 <- lgspline(t, y, K = 0, opt = FALSE, wiggle_penalty = 1e-4)
stopifnot(inherits(fit2, "lgspline"))
stopifnot(length(fit2$B) == 1)
preds2 <- predict(fit2, new_predictors = rnorm(10))
stopifnot(all(is.finite(preds2)))
cat("Example 2 passed: Gaussian identity, K=0\n")
```

```
## Example 3: Path 3 - Binomial GLM
y_bin <- rbinom(200, 1, plogis(sin(t)))
fit3 <- lgspline(t, y_bin, K = 2, family = binomial(),
                 opt = FALSE, wiggle_penalty = 1e-3)
stopifnot(inherits(fit3, "lgspline"))
preds3 <- predict(fit3, new_predictors = rnorm(10))
stopifnot(all(preds3 >= 0 & preds3 <= 1))
cat("Example 3 passed: Binomial GLM, K=2\n")

## Example 4: Path 2 with QP constraints (monotonic increase)
t_sorted <- sort(runif(100, -3, 3))
y_mono <- t_sorted + 0.5 * sin(t_sorted) + rnorm(100, 0, 0.3)
fit4 <- lgspline(t_sorted, y_mono, K = 2,
                 qp_monotonic_increase = TRUE,
                 opt = FALSE, wiggle_penalty = 1e-4)
preds4 <- predict(fit4, new_predictors = cbind(sort(rnorm(50))))
stopifnot(all(diff(preds4) >= -sqrt(.Machine$double.eps)))
cat("Example 4 passed: Monotonic increase QP constraint\n")

## Example 5: Path 2 with range constraints
fit5 <- lgspline(t, y, K = 3,
                 qp_range_lower = -2, qp_range_upper = 2,
                 opt = FALSE, wiggle_penalty = 1e-4)
stopifnot(all(fit5$ytilde >= -2 - 0.01))
stopifnot(all(fit5$ytilde <= 2 + 0.01))
cat("Example 5 passed: Range constraints\n")

## Example 6: Multi-predictor
set.seed(123)
x1 <- rnorm(300)
x2 <- rnorm(300)
y6 <- sin(x1) + cos(x2) + rnorm(300, 0, 0.5)
fit6 <- lgspline(cbind(x1, x2), y6, K = 4,
                 opt = FALSE, wiggle_penalty = 1e-5)
stopifnot(inherits(fit6, "lgspline"))
stopifnot(fit6$q_predictors == 2)
cat("Example 6 passed: 2D predictor, K=4\n")

## Example 7: Coefficient consistency (determinism)
set.seed(999)
t7 <- runif(150, -4, 4)
y7 <- 2 * cos(t7) + rnorm(150, 0, 0.4)
fit7a <- lgspline(t7, y7, K = 2, opt = FALSE, wiggle_penalty = 1e-5)
set.seed(999)
fit7b <- lgspline(t7, y7, K = 2, opt = FALSE, wiggle_penalty = 1e-5)
max_diff <- max(abs(unlist(fit7a$B) - unlist(fit7b$B)))
stopifnot(max_diff < 1e-12)
cat("Example 7 passed: Deterministic coefficient reproduction\n")

cat("\nAll verification examples passed.\n")

## End(Not run)
```

`integrate` *Generic for Numerical Integration*

Description

S3 generic that dispatches to `integrate.lgspline` for fitted `lgspline` objects and falls back to `integrate` for ordinary functions.

Usage

```
integrate(f, ...)

## Default S3 method:
integrate(f, ...)
```

Arguments

`f`	A fitted model object or a function.
`...`	Arguments passed to methods.

`integrate.lgspline` *Definite Integral of a Fitted lgspline*

Description

Given a fitted `lgspline` object, computes the definite integral of the fitted surface over a rectangular domain using Gauss–Legendre quadrature on `predict()`.

Usage

```
## S3 method for class 'lgspline'
integrate(
  f,
  lower,
  upper,
  vars = NULL,
  initial_values = NULL,
  B_predict = NULL,
  link_scale = FALSE,
  n_quad = 50L,
  ...
)
```

Arguments

`f`	A fitted `lgspline` object.
`lower`	Numeric vector of lower bounds, one per integration variable. Scalar values are recycled.
`upper`	Numeric vector of upper bounds, one per integration variable. Scalar values are recycled.
`vars`	Default: NULL. Character or integer vector identifying which predictor(s) to integrate over. When NULL all numeric predictors are integrated simultaneously.
`initial_values`	Default: NULL. Numeric vector of length q supplying fixed values for predictors not among the integration variables. When NULL, non-integration predictors are held at the midpoint of their training range.
`B_predict`	Default: NULL. Optional list of coefficient vectors, one per partition. When NULL the fitted coefficients are used.
`link_scale`	Default: FALSE. Logical; when TRUE the integral is computed on the link (linear predictor) scale η rather than the response scale μ.
`n_quad`	Default: 50. Number of Gauss–Legendre nodes per integration dimension.
`...`	Additional arguments (currently unused; present for S3 method compatibility).

Value

A numeric scalar: the estimated definite integral.

Method

The integration domain is discretised into a tensor-product grid of Gauss–Legendre quadrature nodes. Predicted values at each node come from the model's `predict()` method, which correctly handles partition assignment and piecewise polynomial evaluation. The integral is the weighted sum

$$\int_{a_1}^{b_1} \cdots \int_{a_d}^{b_d} \hat{f}(\mathbf{t})\, \mathrm{d}t_1 \cdots \mathrm{d}t_d \;\approx\; \sum_{i=1}^{M} w_i\, \hat{f}(\mathbf{t}_i)$$

where $M = n_{\text{quad}}^d$ and each weight incorporates the Jacobian $(b_j - a_j)/2$ for the affine map from $[-1, 1]$ to $[a_j, b_j]$. Nodes and weights on $[-1, 1]$ are computed via the Golub–Welsch algorithm (eigenvalues of the symmetric tridiagonal Jacobi matrix).

For smooth polynomials, 30–50 nodes per dimension is typically sufficient; highly partitioned models (large K) may benefit from more. Total evaluation points scale as n_{quad}^d, so problems with $d \geq 4$ may require reducing `n_quad`.

Integration scale

By default (`link_scale = FALSE`), integration is on the response scale $\mu = g^{-1}(\eta)$. Setting `link_scale = TRUE` integrates the linear predictor $\eta = \mathbf{x}^\top \boldsymbol{\beta}$ directly, which is useful when the quantity of interest is the area under the link-transformed surface rather than the response. For the identity link the two coincide.

Examples

```
## 1-D: integral of fitted sin(t) over [-pi, pi] should be near 0
set.seed(1234)
t <- seq(-pi, pi, length.out = 1000)
y <- sin(t) + rnorm(length(t), 0, 0.01)
fit <- lgspline(t, y, K = 4, opt = FALSE)
integrate(fit, lower = -pi, upper = pi)

## Base R integrate still works as expected
integrate(sin, lower = -pi, upper = pi)

## 2-D: volume under fitted volcano surface
data(volcano)
vlong <- cbind(
  rep(seq_len(nrow(volcano)), ncol(volcano)),
  rep(seq_len(ncol(volcano)), each = nrow(volcano)),
  as.vector(volcano)
)
colnames(vlong) <- c("Length", "Width", "Height")
fit_v <- lgspline(vlong[, 1:2], vlong[, 3], K = 18,
                  include_quadratic_interactions = TRUE, opt = FALSE)
integrate(fit_v, lower = c(1, 1), upper = c(87, 61))
```

`leave_one_out` *Compute Leave-One-Out Cross-Validated Predictions for Gaussian Response/Identity Link under Constraint*

Description

Computes the leave-one-out cross-validated predictions from a model fit, assuming Gaussian-distributed response with identity link.

The LOO closed-formula for observation i is $\hat{y}_{(-i)} = y_i - \frac{1}{1-H_{ii}}(y_i - \hat{y}_i)$ where $\mathbf{H}$ is the effective hat matrix under smoothing constraints, adjusted for weights and correlation structure if present.

Observations with leverage at or above `leverage_threshold` are flagged in a warning, since extreme hat values can make the shortcut numerically unreliable. The default `leverage_threshold = 100` is intentionally permissive, so users who want diagnostic warnings for large H_{ii} should set a smaller threshold explicitly.

For related discussion of prediction-sum-of-squares calculations under linear restrictions, see Tarpey (2000), who studies the PRESS statistic for restricted least squares. That setting is closely related to the constraint-adjusted hat-matrix shortcut used here.

Usage

```
leave_one_out(model_fit, leverage_threshold = 100)
```

Arguments

`model_fit`	A fitted lgspline model object.
`leverage_threshold`	Numeric scalar. Observations with $H_{ii} \geq$ `leverage_threshold` are treated as high-leverage for the warning below. Default `100`.

Value

A vector of leave-one-out cross-validated predictions

References

Tarpey, T. (2000). A note on the prediction sum of squares statistic for restricted least squares. *The American Statistician*, 54(2), 116–118. doi:10.2307/2686028

Examples

```
## Basic usage with Gaussian response, computing PRESS
set.seed(1234)
t <- rnorm(50)
y <- sin(t) + rnorm(50, 0, .25)
model_fit <- lgspline(t, y)
loo <- leave_one_out(model_fit)
press <- mean((y - loo)^2, na.rm = TRUE)

plot(loo, y,
   main = "LOO Cross-Validation Prediction vs. Observed Response",
   xlab = 'Prediction', ylab = 'Response')
abline(0, 1)
```

`lgspline` *Fit Lagrangian Multiplier Smoothing Splines*

Description

A comprehensive software package for fitting a variant of smoothing splines as a constrained optimization problem, avoiding the need to algebraically disentangle a spline basis after fitting, and allowing for interpretable interactions and non-spline effects to be included.

`lgspline` fits piecewise polynomial regression splines constrained to be smooth where they meet, penalized by the squared, integrated, second-derivative of the estimated function with respect to predictors, using a monomial basis.

The method of Lagrangian multipliers is used to derive a polynomial regression spline that enforces the following smoothing constraints:

- Equivalent fitted values at knots
- Equivalent first derivatives at knots, with respect to predictors

- Equivalent second derivatives at knots, with respect to predictors

The coefficients are penalized by a closed-form of the traditional cubic smoothing spline penalty, as well as tunable modifications that allow for unique penalization of multiple predictors and partitions.

This package supports model fitting for multiple spline and non-spline effects, GLM families, Weibull accelerated failure time (AFT) models, Cox proportional-Hazards models, negative-binomial regression, arbitrary correlation structures, shape constraints, and extensive customization for user-defined models and constraints.

In addition, parallel processing capabilities and comprehensive tools for visualization, frequentist, and Bayesian inference are provided.

Usage

```
lgspline(
  predictors = NULL,
  y = NULL,
  formula = NULL,
  response = NULL,
  standardize_response = TRUE,
  standardize_predictors_for_knots = TRUE,
  standardize_expansions_for_fitting = TRUE,
  family = gaussian(),
  glm_weight_function = function(mu, y, order_indices, family, dispersion,
    observation_weights, ...) {
     if (any(!is.null(observation_weights))) {

    family$variance(mu) * observation_weights
     }
     else {

    family$variance(mu)
     }
 },
 schur_correction_function = function(X, y, B, dispersion, order_list, K, family,
    observation_weights, ...) {
     lapply(1:(K + 1), function(k) 0)
 },
  need_dispersion_for_estimation = FALSE,
 dispersion_function = function(mu, y, order_indices, family, observation_weights,
    VhalfInv, ...) {
     if (!is.null(VhalfInv)) {
         VhalfInv <-
    VhalfInv[order_indices, order_indices]
         c(mean((tcrossprod(VhalfInv, t(y -
    mu)))^2/family$variance(mu)))
     }
     else {
         c(mean((y -
    mu)^2/family$variance(mu)))
     }
 },
  K = NULL,
  custom_knots = NULL,
  cluster_on_indicators = FALSE,
  make_partition_list = NULL,
  previously_tuned_penalties = NULL,
  smoothing_spline_penalty = NULL,
  opt = TRUE,
  use_custom_bfgs = TRUE,
  delta = NULL,
  tol = 10 * sqrt(.Machine$double.eps),
  initial_wiggle = c(1e-10, 1e-05, 0.1),
  initial_flat = c(0.1, 10),
  wiggle_penalty = 2e-07,
  flat_ridge_penalty = 0.5,
  unique_penalty_per_partition = TRUE,
  unique_penalty_per_predictor = TRUE,
  meta_penalty = 1e-08,
  predictor_penalties = NULL,
  partition_penalties = NULL,
  include_quadratic_terms = TRUE,
  include_cubic_terms = TRUE,
  include_quartic_terms = NULL,
  include_2way_interactions = TRUE,
  include_3way_interactions = TRUE,
  include_quadratic_interactions = FALSE,
  offset = c(),
  just_linear_with_interactions = NULL,
  just_linear_without_interactions = NULL,
  exclude_interactions_for = NULL,
  exclude_these_expansions = NULL,
  custom_basis_fxn = NULL,
  include_constrain_fitted = TRUE,
  include_constrain_first_deriv = TRUE,
  include_constrain_second_deriv = TRUE,
  include_constrain_interactions = TRUE,
  cl = NULL,
  chunk_size = NULL,
  parallel_eigen = TRUE,
  parallel_trace = FALSE,
  parallel_aga = FALSE,
  parallel_matmult = FALSE,
  parallel_unconstrained = TRUE,
  parallel_find_neighbors = FALSE,
  parallel_penalty = FALSE,
  parallel_make_constraint = FALSE,
  unconstrained_fit_fxn = unconstrained_fit_default,
  keep_weighted_Lambda = FALSE,
  iterate_tune = TRUE,
  iterate_final_fit = TRUE,
  blockfit = TRUE,
  qp_score_function = function(X, y, mu, order_list, dispersion, VhalfInv,
    observation_weights, ...) {
     if (!is.null(observation_weights)) {

    crossprod(X, cbind((y - mu) * observation_weights))
     }
     else {

    crossprod(X, cbind(y - mu))
     }
 },
  qp_observations = NULL,
  qp_Amat = NULL,
  qp_bvec = NULL,
  qp_meq = 0,
  qp_positive_derivative = FALSE,
  qp_negative_derivative = FALSE,
  qp_positive_2ndderivative = FALSE,
  qp_negative_2ndderivative = FALSE,
  qp_monotonic_increase = FALSE,
  qp_monotonic_decrease = FALSE,
  qp_range_upper = NULL,
  qp_range_lower = NULL,
  qp_Amat_fxn = NULL,
  qp_bvec_fxn = NULL,
  qp_meq_fxn = NULL,
  constraint_values = cbind(),
  constraint_vectors = cbind(),
  return_G = TRUE,
  return_Ghalf = TRUE,
  return_U = TRUE,
  estimate_dispersion = TRUE,
  unbias_dispersion = NULL,
  return_varcovmat = TRUE,
  exact_varcovmat = FALSE,
  return_lagrange_multipliers = FALSE,
  custom_penalty_mat = NULL,
  cluster_args = c(custom_centers = NA, nstart = 10),
  dummy_dividor = 1.2345672152894e-22,
  dummy_adder = 2.234567210529e-18,
  verbose = FALSE,
  verbose_tune = FALSE,
  dummy_fit = FALSE,
  auto_encode_factors = TRUE,
  observation_weights = NULL,
  do_not_cluster_on_these = c(),
  neighbor_tolerance = 1 + 1e-08,
  null_constraint = NULL,
  critical_value = qnorm(1 - 0.05/2),
  data = NULL,
  weights = NULL,
  no_intercept = FALSE,
  correlation_id = NULL,
  spacetime = NULL,
  correlation_structure = NULL,
  VhalfInv = NULL,
  Vhalf = NULL,
  VhalfInv_fxn = NULL,
  Vhalf_fxn = NULL,
  VhalfInv_par_init = c(),
  REML_grad = NULL,
  custom_VhalfInv_loss = NULL,
  VhalfInv_logdet = NULL,
  include_warnings = TRUE,
  penalty_args = NULL,
  tuning_args = NULL,
  expansion_args = NULL,
  constraint_args = NULL,
  qp_args = NULL,
  parallel_args = NULL,
  covariance_args = NULL,
  return_args = NULL,
  glm_args = NULL,
  ...
)
```

Arguments

`predictors`	Default: NULL. Numeric matrix or data frame of predictor variables, or a formula when using the formula interface.
`y`	Default: NULL. Numeric response variable vector.
`formula`	Default: NULL. Optional statistical formula for model specification, supporting `spl()` (and the alias `s()`) for spline terms.
`response`	Default: NULL. Alternative name for response variable.
`standardize_response`	Default: TRUE. Logical indicator controlling whether the response variable should be centered and scaled before model fitting. Only offered for identity link functions.

`standardize_predictors_for_knots`
: Default: TRUE. Logical flag controlling whether predictors are internally standardized for partitioning / knot placement. The exact transformation is handled inside `make_partitions` and depends on the effective clustering dimension.

`standardize_expansions_for_fitting`
: Default: TRUE. Logical switch to standardize polynomial basis expansions during model fitting. Design matrices, variance-covariance matrices, and coefficients are backtransformed after fitting. $\mathbf{U}$ and $\mathbf{G}$ remain on the transformed scale; `B_raw` corresponds to coefficients on the expansion-standardized scale.

`family`
: Default: `gaussian()`. GLM family specifying the error distribution and link function. Minimally requires: family name, link name, linkfun, linkinv, variance.

`glm_weight_function`
: Default: function returning `family$variance(mu)`, optionally multiplied by `observation_weights`. Codes the mean-variance relationship (the diagonal $\mathbf{W}$ matrix) used for updating $\mathbf{G}$ after obtaining constrained estimates.

`schur_correction_function`
: Default: function returning list of zeros. Computes Schur complements $\mathbf{S}$ added to $\mathbf{G}$: $\mathbf{G}^{*} = (\mathbf{G}^{-1} + \mathbf{S})^{-1}$.

`need_dispersion_for_estimation`
: Default: FALSE. Logical indicator specifying whether a dispersion parameter is required for coefficient estimation (e.g. Weibull AFT).

`dispersion_function`
: Default: function returning mean squared residuals. Custom function for estimating the exponential dispersion parameter.

`K`
: Default: NULL. Integer specifying the number of knot locations. Intuitively, total partitions minus 1.

`custom_knots`
: Default: NULL. Optional matrix providing user-specified knot locations in 1-D.

`cluster_on_indicators`
: Default: FALSE. Logical flag for whether indicator variables should be used for clustering knot locations.

`make_partition_list`
: Default: NULL. Optional list allowing direct specification of custom partition assignments. The `make_partition_list` returned by one model can be supplied here to reuse knot locations.

`previously_tuned_penalties`
: Default: NULL. Optional list of pre-computed penalty components from a previous model fit.

`smoothing_spline_penalty`
: Default: NULL. Optional custom smoothing spline penalty matrix.

`opt`
: Default: TRUE. Logical switch controlling automatic penalty optimization via generalized cross-validation.

`use_custom_bfgs`
: Default: TRUE. Selects between a native damped-BFGS implementation with closed-form gradients or base R's BFGS with finite-difference gradients.

`delta`	Default: NULL. Numeric pseudocount for stabilizing optimization in non-identity link function scenarios.
`tol`	Default: `10*sqrt(.Machine$double.eps)`. Numeric convergence tolerance.
`initial_wiggle`	Default: `c(1e-10, 1e-5, 1e-1)`. Numeric vector of initial grid points for wiggle penalty optimization, on the raw (non-negative) scale.
`initial_flat`	Default: `c(0.1, 10)`. Numeric vector of initial grid points for ridge penalty optimization, on the raw scale (ratio of ridge to wiggle).
`wiggle_penalty`	Default: 2e-7. Numeric penalty on the integrated squared second derivative, governing function smoothness.
`flat_ridge_penalty`	Default: 0.5. Numeric flat ridge penalty for intercepts and linear terms only. Multiplied by `wiggle_penalty` to obtain total ridge penalty.
`unique_penalty_per_partition`	Default: TRUE. Logical flag allowing penalty magnitude to differ across partitions.
`unique_penalty_per_predictor`	Default: TRUE. Logical flag allowing penalty magnitude to differ between predictors.
`meta_penalty`	Default: 1e-8. Numeric regularization coefficient for predictor- and partition-specific penalties during tuning. On the raw scale, the implemented meta-penalty shrinks these penalty multipliers toward 1; the wiggle penalty receives only a tiny stabilizing penalty by default.
`predictor_penalties`	Default: NULL. Optional vector of custom penalties per predictor, on the raw (positive) scale.
`partition_penalties`	Default: NULL. Optional vector of custom penalties per partition, on the raw (positive) scale.
`include_quadratic_terms`	Default: TRUE. Logical switch to include squared predictor terms.
`include_cubic_terms`	Default: TRUE. Logical switch to include cubic predictor terms.
`include_quartic_terms`	Default: NULL. Includes quartic terms; when NULL, set to FALSE for single predictor and TRUE otherwise. Highly recommended for multi-predictor models to avoid over-specified constraints.
`include_2way_interactions`	Default: TRUE. Logical switch for linear two-way interactions.
`include_3way_interactions`	Default: TRUE. Logical switch for three-way interactions.
`include_quadratic_interactions`	Default: FALSE. Logical switch for linear-quadratic interaction terms.
`offset`	Default: Empty vector. Column indices/names to include as offsets. Coefficients for offset terms are automatically constrained to 1.

`just_linear_with_interactions`
: Default: NULL. Integer or character vector specifying predictors to retain as linear terms while still allowing interactions.

`just_linear_without_interactions`
: Default: NULL. Integer or character vector specifying predictors to retain only as linear terms without interactions. Eligible for blockfitting.

`exclude_interactions_for`
: Default: NULL. Integer or character vector of predictors to exclude from all interaction terms.

`exclude_these_expansions`
: Default: NULL. Character vector of basis expansions to exclude. Named columns of data, or in the form `"_1_"`, `"_2_"`, `"_1_x_2_"`, `"_2_^2"` etc.

`custom_basis_fxn`
: Default: NULL. Optional user-defined function for custom basis expansions. See `get_polynomial_expansions`.

`include_constrain_fitted`
: Default: TRUE. Logical switch to constrain fitted values at knot points.

`include_constrain_first_deriv`
: Default: TRUE. Logical switch to constrain first derivatives at knot points.

`include_constrain_second_deriv`
: Default: TRUE. Logical switch to constrain second derivatives at knot points.

`include_constrain_interactions`
: Default: TRUE. Logical switch to constrain interaction terms at knot points.

`cl`
: Default: NULL. Parallel processing cluster object (use `parallel::makeCluster()`).

`chunk_size`
: Default: NULL. Integer specifying custom chunk size for parallel processing.

`parallel_eigen`
: Default: TRUE. Logical flag for parallel eigenvalue decomposition.

`parallel_trace`
: Default: FALSE. Logical flag for parallel trace computation.

`parallel_aga`
: Default: FALSE. Logical flag for parallel $\mathbf{G}$ and $\mathbf{A}$ matrix operations.

`parallel_matmult`
: Default: FALSE. Logical flag for parallel block-diagonal matrix multiplication.

`parallel_unconstrained`
: Default: TRUE. Logical flag for parallel unconstrained MLE for non-identity-link-Gaussian models.

`parallel_find_neighbors`
: Default: FALSE. Logical flag for parallel neighbor identification.

`parallel_penalty`
: Default: FALSE. Logical flag for parallel penalty matrix construction.

`parallel_make_constraint`
: Default: FALSE. Logical flag for parallel constraint matrix generation.

`unconstrained_fit_fxn`
: Default: `unconstrained_fit_default`. Custom function for fitting unconstrained models per partition.

`keep_weighted_Lambda`
: Default: FALSE. Logical flag to retain GLM weights in penalty constraints using Tikhonov parameterization. Advised for non-canonical GLMs.

`iterate_tune`	Default: TRUE. Logical switch for iterative optimization during penalty tuning.
`iterate_final_fit`	Default: TRUE. Logical switch for iterative optimization in final model fitting.
`blockfit`	Default: TRUE. Logical switch for backfitting with mixed spline and non-interactive linear terms. Requires flat columns, `K > 0`, and no active correlation structure. Falls back to `get_B` on failure.
`qp_score_function`	Default: $\mathbf{X}^\top(\mathbf{y}-\boldsymbol{\mu})$. Score function for quadratic programming, blockfit, and GEE formulations. Accepts arguments "`X, y, mu, order_list, dispersion, VhalfInv,observation_weights, ...`".
`qp_observations`	Default: NULL. Numeric vector of observation indices at which built-in QP constraints are evaluated. Useful for reducing the size of the constrained system.
`qp_Amat`	Default: NULL. Optional pre-built QP constraint matrix. In the current pipeline its presence marks QP handling as active, but the built-in constructor does not merge it into the assembled constraint set; use `qp_Amat_fxn` for custom assembled constraints.
`qp_bvec`	Default: NULL. Optional pre-built QP right-hand side paired with `qp_Amat`. Like `qp_Amat`, it is currently treated as an advanced placeholder rather than merged into the built-in constructor.
`qp_meq`	Default: 0. Optional number of equality constraints paired with `qp_Amat`. Like `qp_Amat`, it is currently treated as an advanced placeholder rather than merged into the built-in constructor.
`qp_positive_derivative`	Default: FALSE. Constrain function to have positive first derivatives. Accepts: `FALSE` (no constraint), `TRUE` (all predictors), or a character/integer vector naming specific predictor variables to constrain. For example, `qp_positive_derivative = "Dose"` constrains only the Dose variable, while `qp_positive_derivative = c(1, 3)` constrains columns 1 and 3 of the predictor matrix.
`qp_negative_derivative`	Default: FALSE. Constrain function to have negative first derivatives. Same input types as `qp_positive_derivative`. Can be used simultaneously with `qp_positive_derivative` on different variables.
`qp_positive_2ndderivative`	Default: FALSE. Constrain function to have positive (convex) second derivatives. Same input types as `qp_positive_derivative`.
`qp_negative_2ndderivative`	Default: FALSE. Constrain function to have negative (concave) second derivatives. Same input types as `qp_positive_derivative`.
`qp_monotonic_increase`	Default: FALSE. Logical only. Constrain fitted values to be monotonically increasing in observation order.
`qp_monotonic_decrease`	Default: FALSE. Logical only. Constrain fitted values to be monotonically decreasing in observation order.
`qp_range_upper`	Default: NULL. Numeric upper bound for constrained fitted values.

`qp_range_lower`	Default: NULL. Numeric lower bound for constrained fitted values.
`qp_Amat_fxn`	Default: NULL. Custom function generating Amat.
`qp_bvec_fxn`	Default: NULL. Custom function generating bvec.
`qp_meq_fxn`	Default: NULL. Custom function generating meq.
`constraint_values`	Default: `cbind()`. Optional matrix encoding nonzero equality targets paired with `constraint_vectors`. When left empty, added equality constraints are treated as homogeneous.
`constraint_vectors`	Default: `cbind()`. Optional matrix of user-supplied equality-constraint vectors, appended to the internally generated smoothness constraints.
`return_G`	Default: TRUE. Logical switch to return the unscaled unconstrained variance-covariance matrix $\mathbf{G}$.
`return_Ghalf`	Default: TRUE. Logical switch to return $\mathbf{G}^{1/2}$.
`return_U`	Default: TRUE. Logical switch to return the constraint projection matrix $\mathbf{U}$.
`estimate_dispersion`	Default: TRUE. Logical flag to estimate dispersion after fitting.
`unbias_dispersion`	Default: NULL. Logical switch to multiply dispersion by $N/(N - \text{trace}(\mathbf{H}))$. When NULL, set to TRUE for Gaussian identity link and FALSE otherwise.
`return_varcovmat`	Default: TRUE. Logical switch to return the variance-covariance matrix of estimated coefficients. Needed for Wald inference.
`exact_varcovmat`	Default: FALSE. Logical switch to replace the default asymptotic (Bayesian posterior) variance-covariance matrix with the exact frequentist variance-covariance matrix of the constrained estimator. The asymptotic version uses the Hessian of the penalized log-likelihood: $\tilde{\sigma}^2\mathbf{U}\mathbf{G}\mathbf{U}^\top$. The exact version additionally corrects for the penalty's contribution as a shrinkage prior, giving: $$\tilde{\sigma}^2\mathbf{U}\mathbf{G}\mathbf{U}^\top - \tilde{\sigma}^2\mathbf{U}\mathbf{G}\mathbf{\Lambda}\mathbf{G}\mathbf{U}^\top$$ When a correlation structure is present (`VhalfInv` non-NULL), $\mathbf{G}_{\text{correct}}$ replaces the block-diagonal $\mathbf{G}$. For Gaussian identity link (with or without correlation structure), the result is the exact variance-covariance matrix of the constrained estimate. The returned object still stores the result in `varcovmat`. Requires `return_varcovmat = TRUE`.
`return_lagrange_multipliers`	Default: FALSE. Logical switch to return the Lagrangian multiplier vector.
`custom_penalty_mat`	Default: NULL. Optional $p \times p$ custom penalty matrix for individual partitions, replacing the default ridge on linear/intercept terms. Run with `dummy_fit = TRUE` first to inspect expansion structure.
`cluster_args`	Default: `c(custom_centers = NA, nstart = 10)`. Named vector of arguments controlling clustering. If the first argument is not `NA`, it is treated as custom cluster centers (typically an $(K + 1) \times q$ matrix). Otherwise, default k-means is used.

`dummy_dividor`	Default: 0.000000000000000000000012345672152894. Small numeric constant to prevent division by zero.
`dummy_adder`	Default: 0.00000000000000002234567210529. Small numeric constant to prevent division by zero.
`verbose`	Default: FALSE. Logical flag to print general progress messages.
`verbose_tune`	Default: FALSE. Logical flag to print detailed progress during penalty tuning.
`dummy_fit`	Default: FALSE. Runs the full pipeline but sets coefficients to zero, allowing inspection of design matrix structure, penalty matrices, and partitioning. Replaces the deprecated `expansions_only` argument.
`auto_encode_factors`	Default: TRUE. Logical switch to automatically one-hot encode factor or character variables when using the formula interface.
`observation_weights`	Default: NULL. Numeric vector of observation-specific weights for generalized least squares estimation.
`do_not_cluster_on_these`	Default: `c()`. Predictor columns to exclude from clustering. Accepts numeric column indices or character column names.
`neighbor_tolerance`	Default: `1 + 1e-8`. Numeric tolerance for determining neighboring partitions using k-means clustering. Intended for internal use.
`null_constraint`	Default: NULL. Alternative parameterization for a nonzero equality target when `constraint_vectors` is supplied and `constraint_values` is left empty.
`critical_value`	Default: `qnorm(1-0.05/2)`. Numeric critical value for Wald confidence intervals.
`data`	Default: NULL. Optional data frame for formula-based model specification.
`weights`	Default: NULL. Alias for `observation_weights`.
`no_intercept`	Default: FALSE. Logical flag to constrain intercept to 0. Formulas with `"0+"` set this to TRUE automatically.
`correlation_id`, `spacetime`	Default: NULL. N-length vector and N-row matrix of cluster ids and longitudinal/spatial variables, respectively.
`correlation_structure`	Default: NULL. Native implementations: `"exchangeable"`, `"spatial-exponential"`, `"squared-exponential"`, `"ar(1)"`, `"spherical"`, `"gaussian-cosine"`, `"gamma-cosine"`, `"matern"`, and aliases. Estimated via REML.
`VhalfInv`	Default: NULL. Fixed custom $N \times N$ square-root-inverse covariance matrix $\mathbf{V}^{-1/2}$. Triggers GLS with known covariance. Post-fit inference recomputed from whitened Gram matrices.
`Vhalf`	Default: NULL. Fixed custom $N \times N$ square-root covariance $\mathbf{V}^{1/2}$. Computed as inverse of `VhalfInv` if not supplied.
`VhalfInv_fxn`	Default: NULL. Parametric function for $\mathbf{V}^{-1/2}$; takes single numeric vector `"par"`, returns $N \times N$ matrix. Optimized via BFGS when `VhalfInv_par_init` is provided.

`Vhalf_fxn`	Default: NULL. Optional function for efficient computation of $\mathbf{V}^{1/2}$ from the same parameter vector used by `VhalfInv_fxn`. When omitted, `Vhalf` is obtained by explicit matrix inversion of `VhalfInv`.
`VhalfInv_par_init`	Default: `c()`. Initial parameter values for `VhalfInv_fxn` optimization, on unbounded transformed scale.
`REML_grad`	Default: NULL. Function for the gradient of the negative REML (or custom loss) with respect to the parameters of `VhalfInv_fxn`. Takes `"par"`, `"model_fit"`, and `"..."`.
`custom_VhalfInv_loss`	Default: NULL. Alternative to negative REML for the correlation parameter objective function. Takes `"par"`, `"model_fit"`, and `"..."`.
`VhalfInv_logdet`	Default: NULL. Function for efficient $\log\|\mathbf{V}^{-1/2}\|$ computation. Takes same `"par"` as `VhalfInv_fxn`.
`include_warnings`	Default: TRUE. Logical switch to control display of warnings.
`penalty_args`	Default: NULL. Optional named list grouping penalty-related arguments. See section "Grouped Argument Lists".
`tuning_args`	Default: NULL. Optional named list grouping tuning-related arguments.
`expansion_args`	Default: NULL. Optional named list grouping basis expansion arguments.
`constraint_args`	Default: NULL. Optional named list grouping constraint arguments.
`qp_args`	Default: NULL. Optional named list grouping quadratic programming arguments.
`parallel_args`	Default: NULL. Optional named list grouping parallel processing arguments.
`covariance_args`	Default: NULL. Optional named list grouping correlation structure arguments.
`return_args`	Default: NULL. Optional named list grouping return-control arguments.
`glm_args`	Default: NULL. Optional named list grouping GLM customization arguments.
`...`	Additional arguments passed to the unconstrained model fitting function.

Details

A flexible and interpretable implementation of smoothing splines including:

- Multiple predictors and interaction terms
- Various GLM families and link functions
- Correlation structures for longitudinal/clustered data
- Shape constraints via quadratic programming
- Parallel computation for large datasets
- Comprehensive inference tools

Value

A list of class `"lgspline"` containing model components:

y Original response vector.

ytilde Fitted/predicted values on the scale of the response.

X List of design matrices $\mathbf{X}_k$ for each partition k, containing basis expansions including intercept, linear, quadratic, cubic, and interaction terms as specified. Returned on the unstandardized scale.

A Constraint matrix $\mathbf{A}$ encoding smoothness constraints at knot points and any user-specified linear constraints. Only a linearly independent subset of columns is retained (via pivoted QR decomposition).

B List of fitted coefficients $\boldsymbol{\beta}_k$ for each partition k on the original, unstandardized scale of the predictors and response.

B_raw List of fitted coefficients for each partition on the predictor-and-response standardized scale.

K Number of interior knots with one predictor (number of partitions minus 1 with > 1 predictor).

p Number of basis expansions of predictors per partition.

q Number of predictor variables.

P Total number of coefficients ($p \times (K + 1)$).

N Number of observations.

penalties List containing optimized penalty matrices and components:

- Lambda: Combined penalty matrix ($\boldsymbol{\Lambda}$), includes $\mathbf{L}_{\text{predictor\_list}}$ contributions but not $\mathbf{L}_{\text{partition\_list}}$.
- L1: Smoothing spline penalty matrix ($\mathbf{L}_1$).
- L2: Ridge penalty matrix ($\mathbf{L}_2$).
- L_predictor_list: Predictor-specific penalty matrices ($\mathbf{L}_{\text{predictor\_list}}$).
- L_partition_list: Partition-specific penalty matrices ($\mathbf{L}_{\text{partition\_list}}$).

knot_scale_transf Function for transforming predictors to standardized scale used for knot placement.

knot_scale_inv_transf Function for transforming standardized predictors back to original scale.

knots Matrix of knot locations on original unstandardized predictor scale for one predictor.

partition_codes Vector assigning observations to partitions.

partition_bounds Vector or matrix specifying the boundaries between partitions.

knot_expand_function Internal function for expanding data according to partition structure.

predict Function for generating predictions on new data. For multi-predictor models, `take_first_derivatives = TRUE`, `take_second_derivatives` returns derivatives as a named list of components per predictor variable, rather than a concatenated vector. When `new_predictors` contains columns not present in the data, extraneous columns are silently dropped before prediction.

assign_partition Function for assigning new observations to partitions.

family GLM family object specifying the error distribution and link function.

estimate_dispersion Logical indicating whether dispersion parameter was estimated.

unbias_dispersion Logical indicating whether dispersion estimates should be unbiased.

backtransform_coefficients Function for converting standardized coefficients to original scale.

forwtransform_coefficients Function for converting coefficients to standardized scale.

mean_y, sd_y Mean and standard deviation of response if standardized.

og_order Original ordering of observations before partitioning.

order_list List containing observation indices for each partition.

constraint_values, constraint_vectors Matrices specifying linear equality constraints if provided.

make_partition_list List containing partition information for > 1-D cases.

expansion_scales Vector of scaling factors used for standardizing basis expansions.

take_derivative, take_interaction_2ndderivative Functions for computing derivatives of basis expansions.

get_all_derivatives_insample Function for computing all derivatives on training data.

numerics Indices of numeric predictors used in basis expansions.

power1_cols, power2_cols, power3_cols, power4_cols Column indices for linear through quartic terms.

quad_cols Column indices for all quadratic terms (including interactions).

interaction_single_cols, interaction_quad_cols Column indices for linear-linear and linear-quadratic interactions.

triplet_cols Column indices for three-way interactions.

nonspline_cols Column indices for terms excluded from spline expansion.

return_varcovmat Logical indicating whether variance-covariance matrix was computed.

raw_expansion_names Names of basis expansion terms.

std_X, unstd_X Functions for standardizing/unstandardizing design matrices.

parallel_cluster_supplied Logical indicating whether a parallel cluster was supplied.

weights Original observation weights on the data scale. When no weights were supplied, this is a vector of ones.

G List of unscaled partition-wise information inverses $\mathbf{G}_k$ if `return_G = TRUE`. These are the blockwise quantities stored on the fitting scale; correlation-aware trace, posterior, and variance calculations additionally use dense GLS analogues internally when needed.

Ghalf List of $\mathbf{G}_k^{1/2}$ matrices if `return_Ghalf = TRUE`. As with `G`, dense GLS square-root factors may also be constructed internally for correlation-aware post-fit calculations.

U Constraint projection matrix $\mathbf{U}$ if `return_U = TRUE`. For K=0 and no constraints, returns identity. Otherwise, returns $\mathbf{U} = \mathbf{I} - \mathbf{GA}(\mathbf{A}^\top\mathbf{GA})^{-1}\mathbf{A}^\top$. Used for computing the variance-covariance matrix $\sigma^2\mathbf{UG}$.

sigmasq_tilde Estimated (or fixed) dispersion parameter $\tilde{\sigma}^2$. For Gaussian identity fits without correlation, this is the weighted mean squared residual with optional bias correction. When `VhalfInv` is non-`NULL`, Gaussian-identity residuals are whitened before this calculation.

trace_XUGX Effective degrees of freedom ($\text{trace}(\mathbf{XUGX}^\top)$), where $\mathbf{XUGX}^\top$ serves as the "hat" matrix. When `VhalfInv` is non-`NULL`, computed as $\|\mathbf{V}^{-1/2}\mathbf{XUG}_{\text{correct}}^{1/2}\|_F^2$ using the full penalized GLS information.

varcovmat Variance-covariance matrix of coefficient estimates if `return_varcovmat = TRUE`. Computed as $\sigma^2(\mathbf{U}\mathbf{G}^{1/2})(\mathbf{U}\mathbf{G}^{1/2})^\top$ for numerical stability. When `VhalfInv` is non-`NULL`, uses the full $\mathbf{G}^{1/2}_{\text{correct}}$ in place of the block-diagonal $\mathbf{G}^{1/2}$.

lagrange_multipliers Vector of Lagrangian multipliers if `return_lagrange_multipliers = TRUE`. For equality-only fits these correspond to the active columns of $\mathbf{A}$; when quadratic-programming constraints are active they are taken directly from `solve.QP` and therefore refer to the combined equality/inequality constraint system. `NULL` if no constraints are active ($\mathbf{A}$ is `NULL` or `K == 0`).

VhalfInv The $\mathbf{V}^{-1/2}$ matrix used for implementing correlation structures, if specified.

VhalfInv_fxn, Vhalf_fxn, VhalfInv_logdet, REML_grad Functions for generating $\mathbf{V}^{-1/2}$, $\mathbf{V}^{1/2}$, $\log|\mathbf{V}^{-1/2}|$, and gradient of REML if provided.

VhalfInv_params_estimates Vector of estimated correlation parameters when using `VhalfInv_fxn`.

VhalfInv_params_vcov Approximate variance-covariance matrix of estimated correlation parameters from BFGS optimization.

wald_univariate Function for computing univariate Wald statistics and confidence intervals. Returns an S3 object of class `"wald_lgspline"` with dedicated `print`, `summary`, `plot`, `coef`, and `confint` methods. The `print` method uses `printCoefmat()` for standard R coefficient table formatting with significance stars.

critical_value Critical value used for confidence interval construction.

generate_posterior Function for drawing from the posterior distribution of coefficients. When `VhalfInv` is non-`NULL`, draws are from the correct joint posterior $\mathbf{U}\mathbf{G}^{1/2}_{\text{correct}}\mathbf{z}$ using the full penalized GLS information, reflecting cross-partition posterior covariance induced by off-diagonal blocks of $\mathbf{V}^{-1/2}$.

find_extremum Function for optimizing the fitted function. Accepts both numeric column indices and character column names for `vars`. When `select_vars_fl = TRUE`, L-BFGS-B bounds are correctly subsetted to the optimized variables.

plot Function for visualizing fitted curves.

quadprog_list List containing quadratic programming components if applicable.

.fit_call_args List containing the arguments passed to `lgspline`.

The returned object has class `"lgspline"` and provides comprehensive tools for model interpretation, inference, prediction, and visualization. All coefficients and predictions can be transformed between standardized and original scales using the provided transformation functions. The object includes both frequentist and Bayesian inference capabilities through Wald statistics and posterior sampling. S3 methods `logLik.lgspline` and `confint.lgspline` are available for standard log-likelihood extraction and confidence interval computation, respectively. Advanced customization options are available for analyzing arbitrarily complex study designs.

Response and Predictor Setup

These arguments control the primary data inputs and the initial standardization steps applied before knot placement and fitting.

GLM Customization

These options let you override the default GLM working-weight, dispersion, and partition-wise unconstrained fitting behavior.

Knots and Partitioning

These arguments determine how the predictor space is partitioned and how knot locations are chosen or reused.

Penalty

These arguments configure the smoothing penalty itself and the optional generalized cross-validation tuning procedure.

Basis Expansions

These arguments control which polynomial and interaction terms are included in the partition-specific design matrices.

Constraints

These arguments govern the smoothness equalities and any additional user supplied linear equality constraints.

Quadratic Programming

These arguments activate built-in or custom inequality constraints handled through quadratic programming.

Parallel Processing

These arguments control which computational subroutines may run in parallel and how work is chunked across cluster workers.

Tuning Control

These options control iterative updates during penalty tuning and the final constrained fit.

Return Control

These arguments determine which intermediate matrices and inferential quantities are retained in the returned fit object.

Correlation Structures

These arguments enable built-in or custom working-correlation structures for longitudinal, clustered, or spatially indexed responses.

Grouped Argument Lists

For convenience, related arguments can be bundled into named lists. When a grouped argument is non-NULL, its entries overwrite the corresponding individual arguments. Individual arguments remain available for backward compatibility.

penalty_args Groups: wiggle_penalty, flat_ridge_penalty, unique_penalty_per_partition, unique_penalty_per_predictor, meta_penalty, predictor_penalties, partition_penalties, custom_penalty_mat, previously_tuned_penalties, smoothing_spline_penalty.

tuning_args Groups: opt, use_custom_bfgs, delta, tol, initial_wiggle, initial_flat, iterate_tune, iterate_final_fit.

expansion_args Groups: include_quadratic_terms, include_cubic_terms, include_quartic_terms, include_2way_interactions, include_3way_interactions, include_quadratic_interactions, just_linear_with_interactions, just_linear_without_interactions, exclude_interactions_for, exclude_these_expansions, custom_basis_fxn, offset.

constraint_args Groups: include_constrain_fitted, include_constrain_first_deriv, include_constrain_seco include_constrain_interactions, constraint_values, constraint_vectors, no_intercept.

qp_args Groups all qp_* arguments.

parallel_args Groups: cl, chunk_size, and all parallel_* flags.

covariance_args Groups: correlation_id, spacetime, correlation_structure, VhalfInv, Vhalf, VhalfInv_fxn, Vhalf_fxn, VhalfInv_par_init, REML_grad, custom_VhalfInv_loss, VhalfInv_logdet.

return_args Groups: return_G, return_Ghalf, return_U, estimate_dispersion, unbias_dispersion, return_varcovmat, exact_varcovmat, return_lagrange_multipliers.

glm_args Groups: glm_weight_function, schur_correction_function, need_dispersion_for_estimation, dispersion_function, unconstrained_fit_fxn, keep_weighted_Lambda.

Miscellaneous

These remaining arguments affect inference defaults, numerical safeguards, verbosity, and developer-oriented diagnostics.

Grouped Arguments

For convenience, related arguments can be bundled into named lists. When a grouped argument is non-NULL, its entries overwrite the corresponding individual arguments. Individual arguments remain available for backward compatibility.

penalty_args List. Groups: wiggle_penalty, flat_ridge_penalty, unique_penalty_per_partition, unique_penalty_per_predictor, meta_penalty, predictor_penalties, partition_penalties, custom_penalty_mat, previously_tuned_penalties, smoothing_spline_penalty.

tuning_args List. Groups: opt, use_custom_bfgs, delta, tol, initial_wiggle, initial_flat, iterate_tune, iterate_final_fit.

expansion_args List. Groups: include_quadratic_terms, include_cubic_terms, include_quartic_terms, include_2way_interactions, include_3way_interactions, include_quadratic_interactions, just_linear_with_interactions, just_linear_without_interactions, exclude_interactions_for, exclude_these_expansions, custom_basis_fxn, offset.

constraint_args List. Groups: include_constrain_fitted, include_constrain_first_deriv, include_constrain_second_deriv, include_constrain_interactions, constraint_values, constraint_vectors, no_intercept.

qp_args List. Groups all qp_* arguments.

parallel_args List. Groups: cl, chunk_size, and all parallel_* flags.

covariance_args List. Groups: correlation_id, spacetime, correlation_structure, VhalfInv, Vhalf, VhalfInv_fxn, Vhalf_fxn, VhalfInv_par_init, REML_grad, custom_VhalfInv_loss, VhalfInv_logdet.

return_args List. Groups: return_G, return_Ghalf, return_U, estimate_dispersion, unbias_dispersion, return_varcovmat, return_lagrange_multipliers.

glm_args List. Groups: glm_weight_function, schur_correction_function, need_dispersion_for_estimation, dispersion_function, unconstrained_fit_fxn, keep_weighted_Lambda.

See Also

- lgspline.fit for the low-level fitting interface
- logLik.lgspline for log-likelihood extraction
- confint.lgspline for confidence interval extraction
- leave_one_out for leave-one-out cross-validated predictions
- blockfit_solve for the standalone backfitting solver
- solve.QP for quadratic programming optimization
- plot_ly for interactive plotting
- kmeans for k-means clustering
- optim for general purpose optimization routines

Examples

```
## ## ## ## Simple Examples ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ##
## Simulate some data, fit using default settings without tuning, and plot
set.seed(1234)
t <- runif(2500, -10, 10)
y <- 2*sin(t) + -0.06*t^2 + rnorm(length(t))
model_fit <- lgspline(t, y, opt = FALSE)
plot(t, y, main = 'Observed Data vs. Fitted Function, Colored by Partition',
     ylim = c(-10, 10))
plot(model_fit, add = TRUE)



## Repeat using logistic regression, with univariate inference shown
# and alternative function call
y <- rbinom(length(y), 1, 1/(1+exp(-std(y))))
df <- data.frame(t = t, y = y)
model_fit <- lgspline(y ~ spl(t),
                      df,
                      family = binomial())
plot(t, y, main = 'Observed Data vs Fitted Function with Formulas and Derivatives',
  ylim = c(-0.5, 1.05), cex.main = 0.8)
plot(model_fit,
     show_formulas = TRUE,
     text_size_formula = 0.65,
     legend_pos = 'bottomleft',
     legend_args = list(y.intersp = 1.1),
     add = TRUE)
## Notice how the coefficients match the formula, and expansions are
# homogenous across partitions without reparameterization
print(summary(model_fit))

## Overlay first and second derivatives of fitted function respectively
derivs <- predict(model_fit,
                  new_predictors = sort(t),
                  take_first_derivatives = TRUE,
                  take_second_derivatives = TRUE)
points(sort(t), derivs$first_deriv, col = 'gold', type = 'l')
points(sort(t), derivs$second_deriv, col = 'goldenrod', type = 'l')
legend('bottomright',
       col = c('gold','goldenrod'),
       lty = 1,
       legend = c('First Derivative', 'Second Derivative'))

## Simple 2D example - including a non-spline effect
z <- seq(-2, 2, length.out = length(y))
df <- data.frame(Predictor1 = t,
                 Predictor2 = z,
                 Response = sin(y)+0.1*z)
model_fit <- lgspline(Response ~ spl(Predictor1) + Predictor1*Predictor2,
                      df)

## Notice, while spline effects change over partitions,
# interactions and non-spline effects are constrained to remain the same
coefficients <- Reduce('cbind', coef(model_fit))
colnames(coefficients) <- paste0('Partition ', 1:(model_fit$K+1))
print(coefficients)

## One or two variables can be selected for plotting at a time
# even when >= 3 predictors are present
plot(model_fit,
      custom_title = 'Marginal Relationship of Predictor 1 and Response',
      vars = 'Predictor1',
      custom_response_lab = 'Response',
      show_formulas = TRUE,
      legend_pos = 'bottomright',
      digits = 4,
      text_size_formula = 0.5)

## 3D plots are implemented as well, retaining closed-formulas
my_plot <- plot(model_fit,
                show_formulas = TRUE,
                custom_response_lab = 'Response')
my_plot


## ## ## ## More Detailed 1D Example ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ##
## 1D data generating functions
t <- seq(-9, 9, length.out = 1000)
```

```
slinky <- function(x) {
  (50 * cos(x * 2) -2 * x^2 + (0.25 * x)^4 + 80)
}
coil <- function(x) {
  (100 * cos(x * 2) +-1.5 * x^2 + (0.1 * x)^4 +
  (0.05 * x^3) + (-0.01 * x^5) +
     (0.00002 * x^6) -(0.000001 * x^7) + 100)
}
exponential_log <- function(x) {
  unlist(c(sapply(x, function(xx) {
    if (xx <= 1) {
      100 * (exp(xx) - exp(1))
    } else {
      100 * (log(xx))
    }
  })))
}
scaled_abs_gamma <- function(x) {
  2*sqrt(gamma(abs(x)))
}

## Composite function
fxn <- function(x)(slinky(t) +
                   coil(t) +
                   exponential_log(t) +
                   scaled_abs_gamma(t))

## Bind together with random noise
dat <- cbind(t, fxn(t) + rnorm(length(t), 0, 50))
colnames(dat) <- c('t', 'y')
x <- dat[,'t']
y <- dat[,'y']

## Fit Model, 4 equivalent ways are shown below
model_fit <- lgspline(t, y, opt = FALSE)
model_fit <- lgspline(y ~ spl(t), as.data.frame(dat), opt = FALSE)
model_fit <- lgspline(response = y, predictors = t, opt = FALSE)
model_fit <- lgspline(data = as.data.frame(dat), formula = y ~ ., opt = FALSE)

# This is not valid: lgspline(y ~ ., t)
# This is not valid: lgspline(y, data = as.data.frame(dat))
# Do not put operations in formulas, not valid: lgspline(y ~ log(t) + spl(t))

## Basic Functionality
predict(model_fit, new_predictors = rnorm(1)) # make prediction on new data
loo_vals <- suppressWarnings(head(leave_one_out(model_fit)))
loo_vals # may contain NA when leverage is too high
coef(model_fit) # extract coefficients
summary(model_fit) # model information and Wald inference
generate_posterior(model_fit) # generate draws of parameters from posterior distribution
find_extremum(model_fit, minimize = TRUE) # find the minimum of the fitted function

## Incorporate range constraints, custom knots, keep penalization identical
```

```
# across partitions
model_fit <- lgspline(y ~ spl(t),
                      unique_penalty_per_partition = FALSE,
                      custom_knots = cbind(c(-2, -1, 0, 1, 2)),
                      data = data.frame(t = t, y = y),
                      qp_range_lower = -150,
                      qp_range_upper = 150)

## Plotting the constraints and knots
plot(model_fit,
     custom_title = 'Fitted Function Constrained to Lie Between (-150, 150)',
     cex.main = 0.75)
# knot locations
abline(v = model_fit$knots)
# lower bound from quadratic program
abline(h = -150, lty = 2)
# upper bound from quadratic program
abline(h = 150, lty = 2)
# observed data
points(t, y, cex = 0.24)

## Enforce monotonic increasing constraints on fitted values
# K = 4 => 5 partitions
t <- seq(-10, 10, length.out = 100)
y <- 5*sin(t) + t + 2*rnorm(length(t))
model_fit <- lgspline(t,
                      y,
                      K = 4,
                      qp_monotonic_increase = TRUE)
plot(t, y, main = 'Monotonic Increasing Function with Respect to Fitted Values')
plot(model_fit,
     add = TRUE,
     show_formulas = TRUE,
     legend_pos = 'bottomright',
     custom_predictor_lab = 't',
     custom_response_lab = 'y')

## Posterior draws under constraint
draw <- generate_posterior(model_fit, enforce_qp_constraints = TRUE)
pr <- predict(model_fit, B_predict = draw$post_draw_coefficients)
points(t, pr, col = 'grey')

## ## ## ## 2D Example using Volcano Dataset ## ## ## ## ## ## ## ## ## ## ## ## ## ## ##
## Prep
data('volcano')
volcano_long <-
  Reduce('rbind', lapply(1:nrow(volcano), function(i){
    t(sapply(1:ncol(volcano), function(j){
      c(i, j, volcano[i,j])
    }))
  }))
colnames(volcano_long) <- c('Length', 'Width', 'Height')
```

```
## Fit, with 50 partitions
# When fitting with > 1 predictor and large K, including quartic terms
# is highly recommended, and/or dropping the second-derivative constraint.
# Otherwise, the constraints can impose all partitions to be equal, with one
# cubic function fit for all (there is not enough degrees of freedom to fit
# unique cubic functions due to the massive amount of constraints).
# Below, quartic terms are included and the constraint of second-derivative
# smoothness at knots is ignored.
model_fit <- lgspline(volcano_long[,c(1, 2)],
                      volcano_long[,3],
                      include_quadratic_interactions = TRUE,
                      K = 49,
                      opt = FALSE,
                      return_U = FALSE,
                      return_varcovmat = FALSE,
                      estimate_dispersion = TRUE,
                      return_Ghalf = FALSE,
                      return_G = FALSE,
                      include_constrain_second_deriv = FALSE,
                      unique_penalty_per_predictor = FALSE,
                      unique_penalty_per_partition = FALSE,
                      wiggle_penalty = 1e-10, # the fixed wiggle penalty
                      flat_ridge_penalty = 1e-2) # the ridge penalty / wiggle penalty

## Plotting on new data with interactive visual + formulas
new_input <- expand.grid(seq(min(volcano_long[,1]),
                             max(volcano_long[,1]),
                             length.out = 250),
                         seq(min(volcano_long[,2]),
                             max(volcano_long[,2]),
                             length.out = 250))
plot(model_fit,
     new_predictors = new_input,
     show_formulas = TRUE,
     custom_response_lab = "Height",
     custom_title = 'Volcano 3-D Map',
     digits = 2)

## Get AUC
area_under_volcano <- integrate(model_fit,
                               lower = apply(volcano_long, 2, min)[1:2],
                               upper = apply(volcano_long, 2, max)[1:2])

## ## ## ## Advanced Techniques using Trees Dataset ## ## ## ## ## ## ## ## ## ## ## ## ##
## Goal here is to introduce how lgspline works with non-canonical GLMs and
# demonstrate some custom features
data('trees')

## L1-regularization constraint function on standardized coefficients
# Bound all coefficients to be less than a certain value (l1_bound) in absolute
# magnitude such that | B^{(j)}_k | < lambda for all j = 1....p coefficients,
# and k = 1...K+1 partitions.
l1_constraint_matrix <- function(p, K) {
  ## Total number of coefficients
  P <- p * (K + 1)

  ## Create diagonal matrices for L1 constraint
  # First matrix: lamdba > -bound
  # Second matrix: -lambda > -bound
  first_diag <- diag(P)
  second_diag <- -diag(P)

  ## Combine matrices
  l1_Amat <- cbind(first_diag, second_diag)

  return(l1_Amat)
}

## Bounds absolute value of coefficients to be < l1_bound
l1_bound_vector <- function(qp_Amat,
                            scales,
                            l1_bound) {

  ## Combine matrices
  l1_bvec <- rep(-l1_bound, ncol(qp_Amat)) * c(1, scales)

  return(l1_bvec)
}

## Fit model, using predictor-response formulation, assuming
# Gamma-distributed response, and custom quadratic-programming constraints,
# with qp_score_function/glm_weight_function updated for non-canonical GLMs
# as well as quartic terms, keeping the effect of height constant across
# partitions, and 3 partitions in total. Hence, this is an advanced-usage
# case.
# You can modify this code for performing l1-regularization in general.
# For canonical GLMs, the default qp_score_function/glm_weight_function are
# correct and do not need to be changed
# (custom functionality is not needed for canonical GLMs).
model_fit <- lgspline(
  Volume ~ spl(Girth) + Height*Girth,
  data = with(trees, cbind(Girth, Height, Volume)),
  family = Gamma(link = 'log'),
  keep_weighted_Lambda = TRUE,
  glm_weight_function = function(
    mu,
    y,
    order_indices,
    family,
    dispersion,
    observation_weights,
   ...){
     rep(1/dispersion, length(y))
   },
   dispersion_function = function(
     mu,
     y,
     order_indices,
     family,
     observation_weights,
     VhalfInv,
     ...){
    mean(
      mu^2/((y-mu)^2)
    )
  }, # = biased estimate of 1/shape parameter
  need_dispersion_for_estimation = TRUE,
  unbias_dispersion = TRUE, # multiply dispersion by N/(N-trace(XUGX^{T}))
  K = 2, # 3 partitions
  opt = FALSE, # keep penalties fixed
  unique_penalty_per_partition = FALSE,
  unique_penalty_per_predictor = FALSE,
  flat_ridge_penalty = 1e-64,
  wiggle_penalty = 1e-64,
  qp_score_function = function(X, y, mu, order_list, dispersion, VhalfInv,
    observation_weights, ...){
   t(X) %*% diag(c(1/mu * 1/dispersion)) %*% cbind(y - mu)
  }, # updated score for gamma regression with log link
  qp_Amat_fxn = function(N, p, K, X, colnm, scales, deriv_fxn, ...) {
    l1_constraint_matrix(p, K)
  },
  qp_bvec_fxn = function(qp_Amat, N, p, K, X, colnm, scales, deriv_fxn, ...) {
    l1_bound_vector(qp_Amat, scales, 25)
  },
  qp_meq_fxn = function(qp_Amat, N, p, K, X, colnm, scales, deriv_fxn, ...) 0
)

## Notice, interaction effect is constant across partitions as is the effect
# of Height alone
Reduce('cbind', coef(model_fit))

## Many constraints, many coefficients, and small sample size makes inference
#  using asymptotic variance-covariance matrix untrustworthy.
#  Confidence intervals are often too wide or narrow, even for "good" fit.
#  Consider bootstrapping or alternative.
print(summary(model_fit))

## Plot results
plot(model_fit, custom_predictor_lab1 = 'Girth',
     custom_predictor_lab2 = 'Height',
     custom_response_lab = 'Volume',
     custom_title = 'Girth and Height Predicting Volume of Trees',
     show_formulas = TRUE)

## Verify magnitude of unstandardized coefficients does not exceed bound (25)
print(max(abs(unlist(model_fit$B))))

## Find height and girth where tree volume is closest to 42
# Uses custom objective that minimizes MSE discrepancy between predicted
# value and 42.
# The vanilla find_extremum function can be thought of as
# using "function(mu)mu" aka the identity function as the
# objective, where mu = "f(t)", our estimated function. The derivative is then
# d_mu = "df/dt" with respect to predictors t.
# But with more creative objectives, and since we have machinery for
# df/dt already available, we can compute gradients for (and optimize)
# arbitrary differentiable functions of our predictors too.
# For any objective, differentiate w.r.t. to mu, then multiply by d_mu to
# satisfy chain rule.
# Here, we have objective function: 0.5*(mu-42)^2
# and gradient                    : (mu-42)*d_mu
# and L-BFGS-B will be used to find the height and girth that most closely
# yields a prediction of 42 within the bounds of the observed data.
# The d_mu also takes into account link function transforms automatically
# for most common link functions, and will return warning + instructions
# on how to program the link-function derivatives otherwise.

## Custom acquisition functions for Bayesian optimization could be coded here.
find_extremum(
  model_fit,
  minimize = TRUE,
  custom_objective_function = function(mu, sigma, ybest, ...){
    0.5*(mu - 42)^2
  },
  custom_objective_derivative = function(mu, sigma, ybest, d_mu, ...){
    (mu - 42) * d_mu
  }
)

## ## ## ## How to Use Formulas in lgspline ## ## ## ## ## ## ## ## ## ## ## ## ## ##
## Demonstrates splines with multiple mixed predictors and interactions

## Generate data
n <- 2500
x <- rnorm(n)
y <- rnorm(n)
z <- sin(x)*mean(abs(y))/2

## Categorical predictors
cat1 <- rbinom(n, 1, 0.5)
cat2 <- rbinom(n, 1, 0.5)
cat3 <- rbinom(n, 1, 0.5)

## Response with mix of effects
response <- y + z + 0.1*(2*cat1 - 1)

## Continuous predictors re-named
continuous1 <- x
continuous2 <- z

## Combine data
dat <- data.frame(
  response = response,
  continuous1 = continuous1,
  continuous2 = continuous2,
  cat1 = cat1,
  cat2 = cat2,
  cat3 = cat3
)

## Example 1: Basic Model with Default Terms, No Intercept
# standardize_response = FALSE often needed when constraining intercepts to 0
fit1 <- lgspline(
  formula = response ~ 0 + spl(continuous1, continuous2) +
    cat1*cat2*continuous1 + cat3,
  K = 2,
  standardize_response = FALSE,
  data = dat
)
## Examine coefficients included
rownames(fit1$B$partition1)
## Verify intercept term is near 0 up to some numeric tolerance
abs(fit1$B[[1]][1]) < 1e-8

## Example 2: Similar Model with Intercept, Other Terms Excluded
fit2 <- lgspline(
  formula = response ~ spl(continuous1, continuous2) +
    cat1*cat2*continuous1 + cat3,
  K = 1,
  standardize_response = FALSE,
  include_cubic_terms = FALSE,
  exclude_these_expansions = c( # Not all need to actually be present
    '_batman_x_robin_',
    '_3_x_4_', # no cat1 x cat2 interaction, coded using column indices
    'continuous1xcontinuous2', # no continuous1 x continuous2 interaction
    'thejoker'
  ),
  data = dat
)
## Examine coefficients included
rownames(Reduce('cbind',coef(fit2)))
# Intercept will probably be present and non-0 now
abs(fit2$B[[1]][1]) < 1e-8

## ## ## ## Compare Inference to survreg for Weibull AFT Model Validation ##
# Only linear predictors, no knots, no penalties, using Weibull AFT Model
# The goal here is to ensure that for the special case of no spline effects
# and no knots, this implementation will be consistent with other model
# implementations.
# Also note, that when using models (like Weibull AFT) where dispersion is
# being estimated and is required for estimating beta coefficients,
# we use a schur complement correction function to adjust (or "correct") our
# variance-covariance matrix for both estimation and inference to account for
# uncertainty in estimating the dispersion.
# Typically the schur_correction_function would return a negative-definite
# matrix, as its output is elementwise added to the information matrix prior
# to inversion.
if (requireNamespace("survival", quietly = TRUE)) {
  data("pbc", package = "survival")
  df <- data.frame(na.omit(
    pbc[, c("time", "trt", "stage", "hepato", "bili", "age", "status")]
  ))

  ## Weibull AFT using lgspline, showing how some custom options can be used to
  # fit more complicated models
  model_fit <- lgspline(time ~ trt + stage + hepato + bili + age,
                        df,
                        family = weibull_family(),
                        need_dispersion_for_estimation = TRUE,
                        dispersion_function = weibull_dispersion_function,
                        glm_weight_function = weibull_glm_weight_function,
                        schur_correction_function = weibull_schur_correction,
                        unconstrained_fit_fxn = unconstrained_fit_weibull,
                        opt = FALSE,
                        wiggle_penalty = 0,
                        flat_ridge_penalty = 0,
                        K = 0,
                        status = df$status != 0)
  print(summary(model_fit))

  ## Survreg results match closely on estimates and inference for coefficients
  survreg_fit <- survival::survreg(
    survival::Surv(time, status != 0) ~ trt + stage + hepato + bili + age,
    df
  )
  print(summary(survreg_fit))

  ## sigmasq_tilde = scale^2 of survreg
  print(c(sqrt(model_fit$sigmasq_tilde), survreg_fit$scale))
}

## ## ## ## Modelling Correlation Structures ## ## ## ## ## ## ## ## ## ## ## ## ## ## ##
## Setup
n_blocks <- 200 # Number of correlation_ids (subjects)
block_size <- 5 # Size of each correlation_ids (number of repeated measures per subj.)
N <- n_blocks * block_size # total sample size (balanced here)
rho_true <- 0.25  # True correlation

## Generate predictors and mean structure
t <- seq(-9, 9, length.out = N)
true_mean <- sin(t)

## Create block compound symmetric errors = I(1-p) + Jp
errors <- Reduce('rbind',
                 lapply(1:n_blocks,
                        function(i){
                          sigma <- diag(block_size) + rho_true *
                            (matrix(1, block_size, block_size) -
                                  diag(block_size))
                             matsqrt(sigma) %*% rnorm(block_size)
                           }))

## Generate response with correlated errors
y <- true_mean + errors * 0.5

## Fit model with correlation structure
# include_warnings = FALSE is a good idea here, since many proposed
# correlations will not work
model_fit <- lgspline(t,
                      y,
                      K = 4,
                      correlation_id = rep(1:n_blocks, each = block_size),
                      correlation_structure = 'exchangeable',
                      include_warnings = FALSE
)

## Assess overall fit
plot(t, y, main = 'Sinosudial Fit Under Correlation Structure')
plot(model_fit, add = TRUE, show_formulas = TRUE, custom_predictor_lab = 't')

## Compare estimated vs true correlation
# Built-in exchangeable uses rho = exp(-exp(par)), so par in (-Inf, Inf)
# maps to rho in (0, 1). Only positive correlation is supported.
rho_est <- exp(-exp(model_fit$VhalfInv_params_estimates))
print(c("True correlation:" = rho_true,
        "Estimated correlation:" = rho_est))

## Quantify uncertainty in correlation estimate with 95% confidence interval
#  CI is constructed on the working scale and back-transformed
ci_transformed <- confint(model_fit)['Correlation parameter 1',]
ci_natural <- sort(exp(-exp(ci_transformed)))
print("95% CI for correlation:")
print(ci_natural)

## Also check SD (should be close to 0.5)
print(sqrt(model_fit$sigmasq_tilde))

## Toeplitz Simulation Setup, with demonstration of custom functions
# and boilerplate. Toep is not implemented by default, because it makes
# strong assumptions on the study design and missingness that are rarely met,
# with non-obvious workarounds.
# If a GLM was to-be-fit, you would also submit a function "Vhalf_fxn" analogous
# to VhalfInv_fxn with same argument (par) and an output of an N x N matrix
# that yields the inverse of VhalfInv_fxn output.
n_blocks <- 250   # Number of correlation_ids
block_size <- 8   # Observations per correlation_id
N <- n_blocks * block_size # total sample size
sigma_true <- 2   # Marginal standard deviation

## True Toeplitz components
# This example uses a convex combination of two geometric lag kernels:
```

```
# corr(h) = mix * rho_fast^h + (1 - mix) * rho_slow^h
# which is Toeplitz and positive definite for mix in (0, 1) and
# rho_fast, rho_slow in (0, 1).
rho_fast_true <- 0.25
rho_slow_true <- 0.75
mix_true <- 0.40

## Create time and correlation_id variables
time_var <- rep(1:block_size, n_blocks)
correlation_id_var <- rep(1:n_blocks, each = block_size)

## Create nonlinear predictor-response relationship
# Not sinusoidal and not polynomial.
t_base <- seq(-2, 2, length.out = block_size)
t <- rep(t_base, n_blocks) + rnorm(N, sd = 0.10)
f_true <- function(t) {
  1.4 + 0.9 * atan(1.8 * t) + 0.8 * exp(-1.2 * (t - 0.4)^2)
}

## Generate mean structure
mu_true <- f_true(t)

## Toeplitz correlation helper
corr_from_components <- function(rho_fast, rho_slow, mix) {
  corr <- matrix(0, block_size, block_size)
  for(i in 1:block_size) {
    for(j in 1:block_size) {
      lag <- abs(i - j)
      if(lag == 0) {
        corr[i, j] <- 1
      } else {
        corr[i, j] <- mix * rho_fast^lag + (1 - mix) * rho_slow^lag
      }
    }
  }
  corr
}

## Toeplitz correlation function
# Custom functions can use any parameterization. Here we map:
#   par[1] -> rho_fast = exp(-exp(par[1]))
#   par[2] -> rho_slow = exp(-exp(par[2]))
#   par[3] -> mix      = plogis(par[3])
# so the parameter space is unconstrained, while the resulting Toeplitz
# correlation matrix remains valid.
corr_from_par <- function(par) {
  rho_fast <- exp(-exp(par[1]))
  rho_slow <- exp(-exp(par[2]))
  mix <- plogis(par[3])
  corr_from_components(rho_fast, rho_slow, mix)
}

## Create block Toeplitz errors from the same family we will fit
```

```
corr_true <- corr_from_components(rho_fast_true, rho_slow_true, mix_true)
errors <- Reduce('c',
                 lapply(1:n_blocks, function(i) {
                   c(matsqrt(corr_true) %*% rnorm(block_size))
                 }))

## Generate response with correlated errors and nonlinear covariate effect
y <- mu_true + sigma_true * errors

VhalfInv_fxn <- function(par) {
  corr <- corr_from_par(par)
  kronecker(diag(n_blocks), matinvsqrt(corr))
}

Vhalf_fxn <- function(par) {
  corr <- corr_from_par(par)
  kronecker(diag(n_blocks), matsqrt(corr))
}

## Determinant function (for efficiency)
# This avoids taking determinant of N by N matrix
VhalfInv_logdet <- function(par) {
  corr <- corr_from_par(par)
  log_det_invsqrt_corr <- -0.5 * determinant(corr, logarithm = TRUE)$modulus[1]
  n_blocks * log_det_invsqrt_corr
}

## GLM weights for REML gradient helper
# For Gaussian identity, these are all 1.
glm_weight_function <- function(mu, y, order_indices, family,
                                dispersion, observation_weights, ...) {
  rep(1, length(mu))
}

## REML gradient function
# The helper reml_grad_from_dV computes the three REML terms once dV / dpar
# is supplied. For this parameterization, dV / dpar has closed form.
REML_grad <- function(par, model_fit, ...) {
  rho_fast <- exp(-exp(par[1]))
  rho_slow <- exp(-exp(par[2]))
  mix <- plogis(par[3])

  dV1_block <- matrix(0, block_size, block_size)
  dV2_block <- matrix(0, block_size, block_size)
  dV3_block <- matrix(0, block_size, block_size)

  for(i in 1:block_size) {
    for(j in 1:block_size) {
      lag <- abs(i - j)
      if(lag > 0) {
        ## d/dpar[1] through rho_fast = exp(-exp(par[1]))
        dV1_block[i, j] <- -mix * lag * exp(par[1]) * rho_fast^lag
        ## d/dpar[2] through rho_slow = exp(-exp(par[2]))
        dV2_block[i, j] <- -(1 - mix) * lag * exp(par[2]) * rho_slow^lag
        ## d/dpar[3] through mix = plogis(par[3])
        dV3_block[i, j] <- mix * (1 - mix) * (rho_fast^lag - rho_slow^lag)
      }
    }
  }

  dV1 <- kronecker(diag(n_blocks), dV1_block)
  dV2 <- kronecker(diag(n_blocks), dV2_block)
  dV3 <- kronecker(diag(n_blocks), dV3_block)

  gradient <- numeric(3)
  gradient[1] <- lgspline::reml_grad_from_dV(dV1, model_fit,
                                  glm_weight_function, ...)
  gradient[2] <- reml_grad_from_dV(dV2, model_fit,
                                  glm_weight_function, ...)
  gradient[3] <- reml_grad_from_dV(dV3, model_fit,
                                  glm_weight_function, ...)
  gradient
}

## Visualization
plot(t, y, col = correlation_id_var,
     main = 'Simulated Data with Toeplitz Correlation')

## Fit model with custom Toeplitz
model_fit <- lgspline(
  response = y,
  predictors = t,
  K = 4,
  standardize_response = FALSE,
  VhalfInv_fxn = VhalfInv_fxn,
  Vhalf_fxn = Vhalf_fxn,
  VhalfInv_logdet = VhalfInv_logdet,
  REML_grad = REML_grad,
  VhalfInv_par_init = c(0, -1, 0),
  include_warnings = FALSE
)

## Print comparison of true and estimated correlations
lag_values <- 1:(block_size - 1)
corr_true_by_lag <- sapply(lag_values, function(h) {
  mix_true * rho_fast_true^h + (1 - mix_true) * rho_slow_true^h
})
rho_fast_est <- exp(-exp(model_fit$VhalfInv_params_estimates[1]))
rho_slow_est <- exp(-exp(model_fit$VhalfInv_params_estimates[2]))
mix_est <- plogis(model_fit$VhalfInv_params_estimates[3])
corr_est_by_lag <- sapply(lag_values, function(h) {
  mix_est * rho_fast_est^h + (1 - mix_est) * rho_slow_est^h
})
cat('Toeplitz Correlation Estimates by Lag:\n')
print(data.frame(
  Lag = lag_values,
  True.Correlation = round(corr_true_by_lag, 4),
  Estimated.Correlation = round(corr_est_by_lag, 4)
))

## Should be ~ 2
print(sqrt(model_fit$sigmasq_tilde))

## ## ## ## Parallelism ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ##
if (requireNamespace("parallel", quietly = TRUE)) {
  ## Data generating function
  a <- runif(500000, -9, 9)
  b <- runif(500000, -9, 9)
  c <- rnorm(500000)
  d <- rpois(500000, 1)
  y <- sin(a) + cos(b) - 0.2*sqrt(a^2 + b^2) +
    abs(a) + b +
    0.5*(a^2 + b^2) +
    (1/6)*(a^3 + b^3) +
    a*b*c -
    c +
    d +
    rnorm(500000, 0, 5)

  ## Set up cores
  cl <- parallel::makeCluster(1)
  on.exit(try(parallel::stopCluster(cl), silent = TRUE), add = TRUE)

  ## This example shows some options for what operations can be parallelized
  # By default, only parallel_eigen and parallel_unconstrained are TRUE
  # parallel_unconstrained is only for GLMs, for identity link Gaussian
  # response, use parallel_matmult=TRUE to ensure parallel fitting across
  # partitions.
  # G, G^{-1/2}, and G^{1/2} are computed in parallel across each of the
  # K+1 partitions.
  # However, parallel_unconstrained only affects GLMs without corr. components
  # - it does not affect fitting here
  system.time({
    parfit <- lgspline(y ~ spl(a, b) + a*b*c + d,
                       data = data.frame(y = y,
                                         a = a,
                                         b = b,
                                         c = c,
                                         d = d),
                       cl = cl,
                       K = 1,
                       parallel_eigen = TRUE,
                       parallel_unconstrained = TRUE,
                       parallel_aga = FALSE,
                       parallel_find_neighbors = FALSE,
                       parallel_trace = FALSE,
                       parallel_matmult = TRUE,
                       parallel_make_constraint = FALSE,
                       parallel_penalty = FALSE)
  })
  print(summary(parfit))
}
```

`lgspline_cox` *Fit Cox Proportional Hazards Model via lgspline*

Description

Convenience wrapper that calls `lgspline` with the correct family, weight, dispersion, score, and unconstrained-fit functions for Cox proportional hazards regression. All standard lgspline arguments (knots, penalties, constraints, parallel, etc.) are passed through.

Usage

```
lgspline_cox(formula, data, status, ...)
```

Arguments

`formula`	Formula specifying the model. The response should be survival time; `status` is passed separately.
`data`	Data frame.
`status`	Integer vector of event indicators (1 = event, 0 = censored), same length as the number of rows in `data`.
`...`	Additional arguments passed to `lgspline`.

Details

Internally sets:

- `family` = `cox_family()`
- `unconstrained_fit_fxn` = `unconstrained_fit_cox`
- `glm_weight_function` = `cox_glm_weight_function`
- `qp_score_function` = `cox_qp_score_function`
- `dispersion_function` = `cox_dispersion_function`
- `schur_correction_function` = `cox_schur_correction`
- `need_dispersion_for_estimation` = `FALSE`
- `estimate_dispersion` = `FALSE`
- `standardize_response` = `FALSE`

A formula interface is needed, e.g. `lgspline_cox(t, y, ...)` won't work, unlike for ordinary `lgspline`.

Value

An object of class "lgspline".

Examples

```
## Cox PH with a nonlinear age effect on lung cancer survival
if(requireNamespace("survival", quietly = TRUE)) {
  library(survival)
  set.seed(1234)
  lung <- na.omit(lung[, c("time", "status", "age")])
  lung$age_std <- std(lung$age)

  ## survival codes status as 1 = censored, 2 = dead
  event <- as.integer(lung$status == 2)

  ## Spline on age
  fit <- lgspline_cox(
    time ~ spl(age_std),
    data = lung,
    status = event,
    K = 1
  )
  print(summary(fit))
  plot(fit,
       show_formulas = TRUE,
       custom_response_lab = 'HR',
       custom_predictor_lab = 'Standardized Age',
       ylim = c(0, 5))

}
```

`lgspline_negbin` *Fit Negative Binomial Model via lgspline*

Description

Convenience wrapper that calls `lgspline` with the correct family, weight, dispersion, score, and unconstrained-fit functions for NB2 regression. All standard lgspline arguments (knots, penalties, constraints, parallel, correlation structures, etc.) are passed through.

Usage

```
lgspline_negbin(formula, data, ...)
```

Arguments

formula	Formula specifying the model. The response should be non-negative integer counts.
data	Data frame.
...	Additional arguments passed to lgspline, including Vhalf and VhalfInv for correlation structures.

Details

Internally sets:

- family = negbin_family()
- unconstrained_fit_fxn = unconstrained_fit_negbin
- glm_weight_function = negbin_glm_weight_function
- qp_score_function = negbin_qp_score_function
- dispersion_function = negbin_dispersion_function
- schur_correction_function = negbin_schur_correction
- need_dispersion_for_estimation = TRUE
- estimate_dispersion = TRUE
- standardize_response = FALSE

A formula interface is needed, e.g. lgspline_negbin(t, y) won't work, unlike for ordinary lgspline.

When a correlation structure is supplied via Vhalf/VhalfInv, the model is fitted through the GEE Path 1b machinery in get_B. The dispersion function uses VhalfInv to whiten Pearson residuals for a better moment-based initialization of θ, which stabilizes the profile MLE under moderate to strong correlation. The score function handles the whitened design consistently with the Weibull AFT GEE convention.

Value

An object of class "lgspline".

See Also

lgspline_cox for Cox PH, lgspline_weibull for Weibull AFT, negbin_family, unconstrained_fit_negbin

Examples

```
set.seed(1234)
N <- 300
t <- rnorm(N)
mu <- exp(1 + 0.5 * sin(2 * t))
y <- rnbinom(N, size = 3, mu = mu)
df <- data.frame(response = y, predictor = t)

fit <- lgspline_negbin(
  response ~ spl(predictor),
  data = df,
  K = 2,
  opt = FALSE,
  wiggle_penalty = 1e-2
)
print(summary(fit))
plot(fit, show_formulas = TRUE,
     custom_response_lab = 'Count')
points(t, mu, col = 'grey', cex=0.67)
```

`lgspline_weibull` *Fit Weibull Accelerated Failure Time Model via lgspline*

Description

Convenience wrapper that calls `lgspline` with the correct family, weight, dispersion, score, and unconstrained-fit functions for Weibull accelerated failure time regression. All standard lgspline arguments (knots, penalties, constraints, parallel, etc.) are passed through.

Usage

```
lgspline_weibull(formula, data, status, ...)
```

Arguments

`formula`	Formula specifying the model. The response should be survival time; `status` is passed separately.
`data`	Data frame.
`status`	Integer vector of event indicators (1 = event, 0 = censored), same length as the number of rows in `data`.
`...`	Additional arguments passed to `lgspline`.

Details

Internally sets:

- `family` = `weibull_family()`
- `unconstrained_fit_fxn` = `unconstrained_fit_weibull`
- `glm_weight_function` = `weibull_glm_weight_function`
- `qp_score_function` = `weibull_qp_score_function`
- `dispersion_function` = `weibull_dispersion_function`
- `schur_correction_function` = `weibull_schur_correction`
- `need_dispersion_for_estimation` = `TRUE`

- estimate_dispersion = TRUE
- standardize_response = FALSE

A formula interface is needed, e.g. `lgspline_weibull(t, y, ...)` won't work, unlike for ordinary `lgspline`.

Value

An object of class "lgspline".

See Also

`lgspline_cox` for Cox proportional hazards, `weibull_family`, `unconstrained_fit_weibull`

Examples

```
## Weibull AFT with a nonlinear age effect on lung cancer survival
if(requireNamespace("survival", quietly = TRUE)) {
  library(survival)
  set.seed(1234)
  lung <- na.omit(lung[, c("time", "status", "age")])
  lung$age_std <- std(lung$age)

  ## survival codes status as 1 = censored, 2 = dead
  event <- as.integer(lung$status == 2)

  ## Spline on age
  fit <- lgspline_weibull(
    time ~ spl(age_std),
    data = lung,
    status = event,
    K = 1,
    opt = FALSE,
    wiggle_penalty = 1e-4,
    flat_ridge_penalty = 1
  )
  print(summary(fit))
  plot(fit,
       show_formulas = TRUE,
       custom_response_lab = 'Survival Time',
       custom_predictor_lab = 'Standardized Age')
}
```

---

logLik.lgspline *Extract Log-Likelihood from a Fitted lgspline*

Description

Returns the log-likelihood as a "logLik" object for use with `AIC`, `BIC`, and other model comparison tools.

Usage

```
## S3 method for class 'lgspline'
logLik(object, include_prior = TRUE, new_weights = NULL, ...)
```

Arguments

`object`	A fitted lgspline model object.
`include_prior`	Logical; add the log-prior penalty term. Default TRUE.
`new_weights`	Numeric scalar or N-vector; optional observation weights overriding `object$weights`.
`...`	Not used.

Details

Gaussian identity, no correlation.

$$\ell = -\frac{N}{2}\log(2\pi\tilde{\sigma}^2) - \frac{1}{2\tilde{\sigma}^2}\sum_i (y_i - \hat{y}_i)^2$$

Gaussian identity, with correlation. GLS log-likelihood:

$$\ell = -\frac{N}{2}\log(2\pi\tilde{\sigma}^2) + \log|\mathbf{V}^{-1/2}| - \frac{1}{2\tilde{\sigma}^2}\|\mathbf{V}^{-1/2}(\mathbf{y} - \hat{\mathbf{y}})\|^2$$

$\log|\mathbf{V}^{-1/2}|$ is obtained from `VhalfInv_logdet` when available, or computed directly from `VhalfInv`.

Prior contribution. When `include_prior = TRUE` (default), the log-prior

$$-\frac{1}{2\tilde{\sigma}^2}\sum_k \boldsymbol{\beta}_k^\top \boldsymbol{\Lambda}_k \boldsymbol{\beta}_k$$

is added, giving the penalised MAP log-likelihood coherent with the smoothing spline objective. Set `include_prior = FALSE` for the unpenalised marginal likelihood, which is more appropriate when comparing models with different penalty structures or numbers of knots.

Other GLM families. Uses `family$aic()` when available. For correlated models the whitened residuals and fitted values are passed. When `family$aic()` is unavailable, a deviance-based approximation is used (valid for relative comparisons; a warning is emitted).

This function returns the marginal (full) GLS log-likelihood, not the REML log-likelihood. This is consistent with `REML = FALSE` in `lme` and `gls`, and is the conventional choice for AIC/BIC comparisons of fixed-effects structure.

The `df` attribute is set to $N - \mathrm{trace}(\mathbf{XUGX}^\top)$.

Value

A `"logLik"` object with attributes `df` (effective degrees of freedom) and `nobs` (number of observations).

See Also

`lgspline`, `prior_loglik`, `logLik`, `AIC`, `BIC`

Examples

```
set.seed(1234)
t <- runif(1000, -10, 10)
y <- 2*sin(t) + -0.06*t^2 + rnorm(length(t))
model_fit <- lgspline(t, y)

logLik(model_fit)
logLik(model_fit, include_prior = FALSE)

AIC(model_fit)
BIC(model_fit)

## Compare models with different K using unpenalized likelihood
fit_k3 <- lgspline(t, y, K = 3)
fit_k7 <- lgspline(t, y, K = 7)
AIC(fit_k3, fit_k7)
```

`loglik_cox` *Compute Cox Partial Log-Likelihood*

Description

Evaluates the Cox partial log-likelihood for a given coefficient vector, using the Breslow approximation for tied event times.

Observations must be sorted in ascending order of survival time before calling this function. The internal helpers handle sorting automatically; this function is exposed for diagnostics and testing.

Usage

```
loglik_cox(eta, status, y = NULL, weights = 1)
```

Arguments

`eta`	Numeric vector of linear predictors $\mathbf{X}\boldsymbol{\beta}$, length N, sorted by ascending event time.
`status`	Integer/logical vector of event indicators (1 = event, 0 = censored), same length and order as `eta`.
`y`	Optional numeric vector of observed event/censor times, same length and order as `eta`. When supplied, tied event times are handled by the Breslow approximation using a common risk-set denominator within each tied event-time block. When omitted, the function assumes there are no ties (or that ties have already been expanded appropriately).
`weights`	Optional numeric vector of observation weights (default 1).

Details

The partial log-likelihood (Breslow) is

$$\ell(\boldsymbol{\beta}) = \sum_g \Big[\sum_{i \in D_g} w_i \eta_i - d_g^{(w)} \log\Big(\sum_{j \in R_g} w_j \exp(\eta_j) \Big) \Big]$$

where D_g is the event set at tied event time t_g, $R_g = \{j : t_j \geq t_g\}$ is the corresponding risk set, and $d_g^{(w)} = \sum_{i \in D_g} w_i$.

Value

Scalar partial log-likelihood value.

Examples

```
set.seed(1234)
eta <- rnorm(50)
status <- rbinom(50, 1, 0.6)
y <- rexp(50)
loglik_cox(eta, status, y)
```

`loglik_negbin` *Compute Negative Binomial Log-Likelihood*

Description

Evaluates the NB2 log-likelihood for given mean vector and shape parameter.

Usage

```
loglik_negbin(y, mu, theta, weights = 1)
```

Arguments

`y`	Non-negative integer response vector.
`mu`	Positive mean vector, same length as `y`.
`theta`	Positive scalar shape parameter.
`weights`	Optional observation weights (default 1).

Details

The log-likelihood is

$$\ell(\mu, \theta) = \sum_i w_i \big[\log\Gamma(y_i + \theta) - \log\Gamma(\theta) - \log\Gamma(y_i + 1) + \theta \log\theta - \theta \log(\mu_i + \theta) + y_i \log\mu_i - y_i \log(\mu_i + \theta) \big]$$

Value

Scalar log-likelihood value.

Examples

```
set.seed(1234)
mu <- exp(rnorm(50))
y <- rpois(50, mu)
loglik_negbin(y, mu, theta = 5)
```

`loglik_weibull` *Compute Log-Likelihood for Weibull Accelerated Failure Time Model*

Description

Calculates the log-likelihood for a Weibull accelerated failure time (AFT) survival model, supporting right-censored survival data.

Usage

```
loglik_weibull(log_y, log_mu, status, scale, weights = 1)
```

Arguments

`log_y`	Numeric vector of logarithmic response/survival times
`log_mu`	Numeric vector of logarithmic predicted survival times
`status`	Numeric vector of censoring indicators (1 = event, 0 = censored) Indicates whether an event of interest occurred (1) or the observation was right-censored (0). In survival analysis, right-censoring occurs when the full survival time is unknown, typically because the study ended or the subject was lost to follow-up before the event of interest occurred.
`scale`	Numeric scalar representing the Weibull scale parameter (sigma), equivalent to `survreg$scale`. This is the square root of the dispersion stored in `lgspline$sigmasq_tilde`.
`weights`	Optional numeric vector of observation weights (default = 1)

Details

The function computes log-likelihood contributions for a Weibull AFT model, explicitly accounting for right-censored observations. It supports optional observation weighting to accommodate complex sampling designs.

This both provides a tool for actually fitting Weibull AFT models, and boilerplate code for users who wish to incorporate Lagrangian multiplier smoothing splines into their own custom models.

Value

A numeric scalar representing the weighted total log-likelihood of the model

Examples

```
## Minimal example of fitting a Weibull Accelerated Failure Time model
# Simulating survival data with right-censoring
set.seed(1234)
x1 <- rnorm(1000)
x2 <- rbinom(1000, 1, 0.5)
yraw <- rexp(exp(0.01*x1 + 0.01*x2))
# status: 1 = event occurred, 0 = right-censored
status <- rbinom(1000, 1, 0.25)
yobs <- ifelse(status, runif(length(yraw), 0, yraw), yraw)
df <- data.frame(
  y = yobs,
  x1 = x1,
  x2 = x2
)

## Fit model using lgspline with Weibull AFT specifics
model_fit <- lgspline(y ~ spl(x1) + x2,
                      df,
                      unconstrained_fit_fxn = unconstrained_fit_weibull,
                      family = weibull_family(),
                      need_dispersion_for_estimation = TRUE,
                      dispersion_function = weibull_dispersion_function,
                      glm_weight_function = weibull_glm_weight_function,
                      schur_correction_function = weibull_schur_correction,
                      status = status,
                      opt = FALSE,
                      K = 1)

loglik_weibull(log(model_fit$y), log(model_fit$ytilde), status,
  sqrt(model_fit$sigmasq_tilde))
```

`matinvsqrt` *Calculate Matrix Inverse Square Root for Symmetric Matrices*

Description

Calculate Matrix Inverse Square Root for Symmetric Matrices

Usage

```
matinvsqrt(mat)
```

Arguments

`mat`	A symmetric matrix $\mathbf{M}$

Details

Uses an eigenvalue-decomposition-based approach.

Non-positive eigenvalues are set to 0 before taking inverse fourth roots.

This implementation is particularly useful for whitening procedures in GLMs with correlation structures and for computing variance-covariance matrices under constraints.

You can use this to help construct a custom `VhalfInv_fxn` for `lgspline`. When only `VhalfInv` is supplied there, the corresponding `Vhalf` is reconstructed internally by inversion for the GEE code paths.

Value

A matrix **B** such that **BB** equals the Moore-Penrose-style inverse on the positive-eigenvalue subspace, with non-positive components truncated to 0.

Examples

```
## Identity matrix
m1 <- diag(2)
matinvsqrt(m1)  # Returns identity matrix

## Compound symmetry correlation matrix
rho <- 0.5
m2 <- matrix(rho, 3, 3) + diag(1-rho, 3)
B <- matinvsqrt(m2)
# Verify: B %**% B approximately equals solve(m2)
all.equal(B %**% B, solve(m2))

## Example for GLM correlation structure
n_blocks <- 2  # Number of subjects
block_size <- 3  # Measurements per subject
rho <- 0.7  # Within-subject correlation
# Correlation matrix for one subject
R <- matrix(rho, block_size, block_size) +
     diag(1-rho, block_size)
## Full correlation matrix for all subjects
V <- kronecker(diag(n_blocks), R)
## Create whitening matrix
VhalfInv <- matinvsqrt(V)

# Example construction of VhalfInv_fxn for lgspline
VhalfInv_fxn <- function(par) {
  rho <- tanh(par)  # Transform parameter to (-1, 1)
  R <- matrix(rho, block_size, block_size) +
       diag(1-rho, block_size)
  kronecker(diag(n_blocks), matinvsqrt(R))
}
```

matsqrt *Calculate Matrix Square Root for Symmetric Matrices*

Description

Calculate Matrix Square Root for Symmetric Matrices

Usage

```
matsqrt(mat)
```

Arguments

`mat`	A symmetric matrix $\mathbf{M}$

Details

Uses an eigenvalue-decomposition-based approach.

Non-positive eigenvalues are set to 0 before taking fourth roots.

This implementation is particularly useful for whitening procedures in GLMs with correlation structures and for computing variance-covariance matrices under constraints.

You can use this to help construct a custom `Vhalf_fxn`, or more directly to build the $\mathbf{V}^{1/2}$ input supplied to `lgspline` for correlation-aware fits.

Value

A matrix $\mathbf{B}$ such that $\mathbf{BB}$ equals $\mathbf{M}$ on the positive-eigenvalue subspace, with non-positive components truncated to 0.

Examples

```
## Identity matrix
m1 <- diag(2)
matsqrt(m1)  # Returns identity matrix

## Compound symmetry correlation matrix
rho <- 0.5
m2 <- matrix(rho, 3, 3) + diag(1-rho, 3)
B <- matsqrt(m2)
# Verify: B %**% B approximately equals m2
all.equal(B %**% B, m2)

## Example for correlation structure
n_blocks <- 2  # Number of subjects
block_size <- 3  # Measurements per subject
rho <- 0.7  # Within-subject correlation
# Correlation matrix for one subject
R <- matrix(rho, block_size, block_size) +
    diag(1-rho, block_size)
# Full correlation matrix for all subjects
V <- kronecker(diag(n_blocks), R)
Vhalf <- matsqrt(V)
```

`negbin_dispersion_function`
NB Dispersion Function

Description

Estimates the shape parameter θ from current fitted values. When a correlation structure is present (`VhalfInv` is non-NULL), the Pearson residuals are whitened before computing the moment-based initial value, giving a better starting point for the profile MLE under correlated data. The final estimate is always the profile MLE over θ.

Usage

```
negbin_dispersion_function(
  mu,
  y,
  order_indices,
  family,
  observation_weights,
  VhalfInv
)
```

Arguments

`mu`	Predicted values.
`y`	Observed counts.
`order_indices`	Observation indices.
`family`	NB family object.
`observation_weights`	Observation weights.
`VhalfInv`	Inverse square root of the correlation matrix, or NULL for independent observations. When non-NULL, used to whiten residuals for the moment-based initialization of θ.

Details

The profile MLE maximizes $\ell(\theta \mid \mu)$ via Brent's method. When `VhalfInv` is provided, the Pearson residuals $r_i = (y_i - \mu_i)/\sqrt{V(\mu_i)}$ are pre-whitened as $\tilde{r} = V^{-1/2}r$ before computing the moment estimator used for initialization. This accounts for the correlation structure in the variance decomposition and produces a more stable starting point for the optimizer, particularly when the correlation inflates the marginal variance beyond what the NB model alone would predict.

The profile MLE itself does not use `VhalfInv` because the NB log-likelihood is a marginal quantity; the correlation structure affects estimation only through the mean model (handled by the GEE paths in `get_B`).

Value

Scalar θ estimate (stored as `sigmasq_tilde`).

`negbin_family` *Negative Binomial Family for lgspline*

Description

Creates a family-like object for NB2 regression. The link is log, the variance function is $V(\mu) = \mu + \mu^2/\theta$, and the dispersion stored by lgspline (`sigmasq_tilde`) is the shape parameter θ.

Usage

```
negbin_family()
```

Details

The NB2 model has a nuisance shape parameter θ analogous to the Weibull scale parameter. It is estimated jointly with $\boldsymbol{\beta}$ and its uncertainty is propagated via the Schur complement correction.

Value

A list with family components used by `lgspline`.

Examples

```
fam <- negbin_family()
fam$family
fam$link
```

`negbin_glm_weight_function` *NB GLM Weight Function*

Description

Computes working weights for the NB2 information matrix used by lgspline when updating $\mathbf{G}$ after obtaining constrained estimates.

Usage

```
negbin_glm_weight_function(
  mu,
  y,
  order_indices,
  family,
  dispersion,
  observation_weights
)
```

Arguments

`mu`	Predicted values $\exp(\eta)$.
`y`	Observed counts.
`order_indices`	Observation indices in partition order.
`family`	NB family object (unused, for interface compatibility).
`dispersion`	Shape parameter θ.
`observation_weights`	Observation weights.

Details

The IRLS weight for NB2 with log link is

$$W_i = w_i \mu_i \theta / (\theta + \mu_i)$$

Falls back to observation weights when natural weights are degenerate.

Value

Numeric vector of working weights, length N.

negbin_helpers *Negative Binomial Regression Helpers for lgspline*

Description

Functions for fitting negative binomial (NB2) regression models within the lgspline framework. Analogous to the Weibull AFT and Cox PH helpers, these provide the log-likelihood, score, information, and all interface functions needed by lgspline's unconstrained fitting, penalty tuning, and inference machinery.

Details

The parameterization follows NB2: $Y \sim \mathrm{NB}(\mu, \theta)$ with $\mathrm{Var}(Y) = \mu + \mu^2/\theta$, where $\theta > 0$ is the shape (size) parameter. The canonical link is log: $\eta = \log \mu$. The dispersion stored in `lgspline$sigmasq_tilde` is θ itself, not $1/\theta$.

negbin_qp_score_function *NB Score Function for Quadratic Programming and Blockfit*

Description

Computes the score (gradient of NB log-likelihood) in the format expected by lgspline's `qp_score_function` interface.

Usage

```
negbin_qp_score_function(
  X,
  y,
  mu,
  order_list,
  dispersion,
  VhalfInv,
  observation_weights
)
```

Arguments

`X`	Block-diagonal design matrix (N x P).
`y`	Response vector (N x 1).
`mu`	Predicted values (N x 1), same order as X and y.
`order_list`	List of observation indices per partition.
`dispersion`	Shape parameter θ.

`VhalfInv`	Inverse square root of correlation matrix; when non-NULL the score is computed on the whitened scale as $\tilde{X}^\top \tilde{r}$ where $\tilde{X} = V^{-1/2}X$ and the residual vector accounts for the correlation.
`observation_weights`	Observation weights.

Details

Without correlation (`VhalfInv = NULL`), the score is $\mathbf{X}^\top \mathbf{w} \odot (y-\mu)\theta/(\theta+\mu)$.

With correlation, the GEE score is $\tilde{\mathbf{X}}^\top \mathrm{diag}(\mathbf{W})\mathbf{V}^{-1}(\mathbf{y}-\boldsymbol{\mu})$ where $\mathbf{W}$ contains the NB working weights. The whitening is absorbed by pre-multiplying both $\mathbf{X}$ and the residual by $\mathbf{V}^{-1/2}$.

Value

Numeric column vector of length P.

`negbin_schur_correction` *NB Schur Correction*

Description

Computes the Schur complement correction to account for uncertainty in estimating θ. Structure is identical to `weibull_schur_correction`: the joint information is partitioned into $(\boldsymbol{\beta}, \theta)$ blocks and the correction is $-\mathbf{I}_{\beta\theta} I_{\theta\theta}^{-1} \mathbf{I}_{\beta\theta}^\top$.

Usage

```
negbin_schur_correction(
  X,
  y,
  B,
  dispersion,
  order_list,
  K,
  family,
  observation_weights
)
```

Arguments

`X`	List of partition design matrices.
`y`	List of partition response vectors.
`B`	List of partition coefficient vectors.
`dispersion`	Scalar shape parameter θ.
`order_list`	List of observation indices per partition.

`K`	Number of knots.
`family`	Family object.
`observation_weights`	Observation weights.

Details

The cross-derivative (score of η_i w.r.t. θ) is

$$\frac{\partial^2 \ell}{\partial \eta_i \partial \theta} = \frac{(y_i - \mu_i)\mu_i}{(\theta + \mu_i)^2}$$

The second derivative of the log-likelihood w.r.t. θ is

$$I_{\theta\theta} = -\sum_i w_i \big[\psi'(y_i + \theta) - \psi'(\theta) + \frac{1}{\theta} - \frac{2}{\mu_i + \theta} + \frac{y_i + \theta}{(\mu_i + \theta)^2}\big]$$

where ψ' is the trigamma function.

Value

List of K+1 correction matrices, with `0` for empty partitions.

`negbin_theta` *Estimate Negative Binomial Shape Parameter*

Description

Computes the profile MLE of the shape parameter θ given current mean estimates μ.

Usage

```
negbin_theta(y, mu, weights = 1, init = NULL)
```

Arguments

`y`	Response vector.
`mu`	Mean vector.
`weights`	Observation weights (default 1).
`init`	Optional initial value for θ. If NULL, uses a moment-based estimate.

Details

Maximizes the profile log-likelihood over θ via Brent's method on $[10^{-4}, 10^4]$.

Value

Scalar MLE of θ.

Examples

```
set.seed(1234)
mu <- rep(5, 200)
y <- rnbinom(200, size = 3, mu = 5)
negbin_theta(y, mu)
```

`plot.lgspline` *Plot Method for lgspline Objects*

Description

Wrapper for the internal lgspline plot function. Produces a 1D line plot (base R) or interactive 3D surface plot (plotly) depending on the number of predictors, with optional formula overlays per partition. When plotting a subset of variables via `vars`, non-plotted predictors are automatically set to zero (or a user-specified value via `fixed_values`).

Usage

```
## S3 method for class 'lgspline'
plot(
  x,
  show_formulas = FALSE,
  digits = 4,
  legend_pos = "topright",
  custom_response_lab = "y",
  custom_predictor_lab = NULL,
  custom_predictor_lab1 = NULL,
  custom_predictor_lab2 = NULL,
  custom_formula_lab = NULL,
  custom_title = "Fitted Function",
  text_size_formula = NULL,
  legend_args = list(),
  new_predictors = NULL,
  xlim = NULL,
  ylim = NULL,
  color_function = NULL,
  add = FALSE,
  vars = c(),
  legend_order = NULL,
  se.fit = FALSE,
  cv = 1,
  band_col = "grey80",
  band_border = NA,
  fixed_values = NULL,
  n_grid = 200,
  ...
)
```

Arguments

`x`	A fitted lgspline model object.
`show_formulas`	Logical; display partition-level polynomial formulas. Default FALSE.
`digits`	Integer; decimal places for formula coefficients. Default 4.
`legend_pos`	Character; legend position for 1D plots. Default `"topright"`.
`custom_response_lab`	Character; response axis label. Default `"y"`.
`custom_predictor_lab`	Character; predictor axis label (1D). Default NULL uses the column name.
`custom_predictor_lab1`	Character; first predictor axis label (2D). Default NULL.
`custom_predictor_lab2`	Character; second predictor axis label (2D). Default NULL.
`custom_formula_lab`	Character; fitted response label on the link scale. Default NULL.
`custom_title`	Character; plot title. Default `"Fitted Function"`.
`text_size_formula`	Numeric; formula text size. Passed to `cex` (1D) or hover font size (2D). Default NULL (0.8 for 1D, 8 for 2D).
`legend_args`	List; additional arguments passed to `legend()` (1D only).
`new_predictors`	Matrix; optional predictor values for prediction. Default NULL. When `vars` is specified and `new_predictors` is NULL, a grid is automatically generated with non-plotted variables set to zero (or values from `fixed_values`).
`xlim`	Numeric vector; x-axis limits (1D only). Default NULL.
`ylim`	Numeric vector; y-axis limits (1D only). Default NULL.
`color_function`	Function; returns K+1 colors, one per partition. Default NULL uses `grDevices::rainbow(K+1)` for 1D and a Spectral palette for 2D.
`add`	Logical; add to an existing plot (1D only). Default FALSE.
`vars`	Numeric or character vector; predictor indices or names to plot. Default `c()` plots all.
`legend_order`	Numeric; re-ordered partition indices for the legend.
`se.fit`	Logical; if TRUE, plot pointwise confidence bands. Default FALSE.
`cv`	Numeric; critical value for confidence bands. Default 1.
`band_col`	Character; color for confidence band fill. Default `"grey80"`.
`band_border`	Character or NA; border color for confidence band polygon. Default NA (no border).
`fixed_values`	Named list; fixed values for non-plotted predictors when `vars` is specified. Names should match predictor names. Default NULL sets non-plotted predictors to zero.
`n_grid`	Integer; number of grid points for automatic grid generation when `vars` is specified and `new_predictors` is NULL. Default 200.
`...`	Additional arguments passed to `plot` (1D) or `plot_ly` (2D).

Details

Partition boundaries are indicated by color changes. For 1D models, observation points can be overlaid. For 2D models, plotly is used.

When using `vars` to plot a subset of predictors, the non-plotted predictors are automatically set to zero. This can be overridden by passing a named list to `fixed_values` (e.g., `fixed_values = list(Height = 75)`). The automatic zeroing replaces the previous behavior where the user had to manually construct `new_predictors` with non-plotted variables set to fixed values.

When `se.fit = TRUE`, pointwise confidence bands are drawn around the fitted function. These are Wald-type intervals constructed on the link scale and back-transformed to the response scale, using `cv` as the critical value to multiply se.fit by (default 1 for actual se).

The function extracts predictor positions from linear expansion terms. If linear terms are excluded (e.g., via `exclude_these_expansions`), plotting will fail. As a workaround, constrain those terms to zero via `constraint_vectors` / `constraint_values` so they remain in the expansion but are zeroed out.

Value

For 1D models: invisibly returns NULL (base R plot drawn to device). For 2D models: returns a plotly object.

See Also

`lgspline`, `plot`, `plot_ly`

Examples

```
set.seed(1234)
t_data <- runif(1000, -10, 10)
y_data <- 2*sin(t_data) + -0.06*t_data^2 + rnorm(length(t_data))
model_fit <- lgspline(t_data, y_data, K = 9)

## Basic plot
plot(model_fit)

## Plot with confidence bands
plot(model_fit,
     se.fit = TRUE,
     cv = 1.96,
     custom_title = 'Fitted Function with 95% CI')

## Multi-predictor: automatically zeros non-plotted variables
# plot(model_fit_2d, vars = 'x1', se.fit = TRUE)
```

plot.wald_lgspline *Plot Method for wald_lgspline Objects*

Description

Forest-style plot of coefficient estimates with confidence intervals.

Usage

```
## S3 method for class 'wald_lgspline'
plot(
  x,
  parm = NULL,
  which = NULL,
  main = "Coefficient Estimates and CIs",
  xlab = "Estimate",
  ...
)
```

Arguments

`x`	A `"wald_lgspline"` object.
`parm`	Integer vector of coefficient indices or character vector of names to plot. Default NULL plots all.
`which`	Integer vector of coefficient indices to plot (alternative to `parm`). Default NULL.
`main`	Plot title. Default `"Coefficient Estimates and CIs"`.
`xlab`	x-axis label. Default `"Estimate"`.
`...`	Additional arguments passed to `plot`.

Value

Invisibly returns NULL.

See Also

`wald_univariate`, `confint.lgspline`

`predict.lgspline` *Predict Method for lgspline Objects*

Description

Generates predictions, derivatives, and basis expansions from a fitted lgspline model. Wrapper for the internal predict closure stored in the object.

Usage

```
## S3 method for class 'lgspline'
predict(
  object,
  newdata = NULL,
  parallel = FALSE,
  cl = NULL,
  chunk_size = NULL,
  num_chunks = NULL,
  rem_chunks = NULL,
  B_predict = NULL,
  take_first_derivatives = FALSE,
  take_second_derivatives = FALSE,
  expansions_only = FALSE,
  new_predictors = NULL,
  ...
)
```

Arguments

`object`	A fitted lgspline model object.
`newdata`	Matrix or data.frame; new predictor values. Default NULL.
`parallel`	Logical; use parallel processing (experimental). Default FALSE.
`cl`	Cluster object for parallel processing. Default NULL.
`chunk_size`	Integer; chunk size for parallel processing. Default NULL.
`num_chunks`	Integer; number of chunks. Default NULL.
`rem_chunks`	Integer; remainder chunks. Default NULL.
`B_predict`	List; per-partition coefficient list for prediction, e.g. from `generate_posterior`. Default NULL uses `object$B`.
`take_first_derivatives`	Logical; compute first derivatives. Default FALSE.
`take_second_derivatives`	Logical; compute second derivatives. Default FALSE.
`expansions_only`	Logical; return basis expansion matrix only. Default FALSE.
`new_predictors`	Matrix or data.frame; overrides `newdata`.
`...`	Additional arguments passed to the internal predict method.

Details

`new_predictors` takes priority over `newdata` when both are supplied. When both are NULL, the training data is used.

Fitted values are also accessible directly as `model_fit$ytilde` or via `model_fit$predict()`.

The parallel processing feature is experimental.

Additional arguments passed through `...` include `se.fit` and `cv` for pointwise interval summaries.

Predictor input should use the original predictor columns. Named extra columns are dropped when they can be identified as irrelevant to the fitted expansions.

Value

A numeric vector of predictions, or a list when derivatives or interval summaries are requested:

preds Numeric vector of predictions when derivatives are requested.

fit Numeric vector of predictions when `se.fit` = `TRUE` and no derivatives are requested.

first_deriv Numeric vector or named list of first derivatives (if requested).

second_deriv Numeric vector or named list of second derivatives (if requested).

se.fit Pointwise standard errors on the link scale (if requested).

lower Pointwise lower interval bound (if requested).

upper Pointwise upper interval bound (if requested).

cv Critical value returned when `se.fit` = `TRUE` without derivative requests.

If `expansions_only` = `TRUE`, returns a list of basis expansions.

See Also

`lgspline`, `plot.lgspline`

Examples

```
set.seed(1234)
t <- runif(1000, -10, 10)
y <- 2*sin(t) + -0.06*t^2 + rnorm(length(t))
model_fit <- lgspline(t, y)

newdata <- matrix(sort(rnorm(10000)), ncol = 1)
preds <- predict(model_fit, newdata)

deriv1_res <- predict(model_fit, newdata, take_first_derivatives = TRUE)
deriv2_res <- predict(model_fit, newdata, take_second_derivatives = TRUE)

oldpar <- par(no.readonly = TRUE)
layout(matrix(c(1,1,2,2,3,3), byrow = TRUE, ncol = 2))

plot(newdata[,1], preds, main = 'Fitted Function',
     xlab = 't', ylab = "f(t)", type = 'l')
plot(newdata[,1], deriv1_res$first_deriv, main = 'First Derivative',
     xlab = 't', ylab = "f'(t)", type = 'l')
plot(newdata[,1], deriv2_res$second_deriv, main = 'Second Derivative',
     xlab = 't', ylab = "f''(t)", type = 'l')

par(oldpar)
```

print.lgspline *Print Method for lgspline Objects*

Description

Prints a concise summary of the fitted model to the console.

Usage

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

Arguments

x	An lgspline model object.
...	Not used.

Value

Invisibly returns x.

print.summary.lgspline *Print Method for lgspline Summaries*

Description

Displays a formatted model summary using printCoefmat for the coefficient table.

Usage

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

Arguments

x	A summary.lgspline object.
...	Not used.

Value

Invisibly returns x.

See Also

printCoefmat

---

print.wald_lgspline *Print Method for wald_lgspline Objects*

Description

Prints the coefficient table using printCoefmat with significance stars.

Usage

```
## S3 method for class 'wald_lgspline'
print(
  x,
  digits = max(3, getOption("digits") - 3),
  signif.stars = getOption("show.signif.stars"),
  ...
)
```

Arguments

x	A "wald_lgspline" object from wald_univariate.
digits	Number of significant digits.
signif.stars	Logical; show significance stars.
...	Additional arguments passed to printCoefmat.

Value

Invisibly returns x.

See Also

wald_univariate, printCoefmat

`prior_loglik` *Log-Prior Distribution Evaluation for lgspline Models*

Description

Evaluates the log-prior on the spline coefficients conditional on the dispersion and penalty matrices.

Usage

```
prior_loglik(model_fit, sigmasq = NULL, include_constant = TRUE)
```

Arguments

`model_fit`	An lgspline model object.
`sigmasq`	Numeric scalar dispersion parameter. If NULL, `model_fit$sigmasq_tilde` is used.
`include_constant`	Logical; if TRUE (default), include the multivariate normal normalizing constant.

Details

Returns the quadratic form of $\beta^T \Lambda \beta$ evaluated at the tuned or fixed penalties, scaled by negative one-half inverse dispersion.

Assuming fixed penalties, the prior on β is taken to be

$$\beta|\sigma^2 \sim \mathcal{N}(\mathbf{0}, \sigma^2\Lambda^{-1})$$

so that, up to a normalizing constant C with respect to β,

$$\implies \log P(\beta|\sigma^2) = C - \frac{1}{2\sigma^2}\beta^T \Lambda \beta$$

The value of C is included when `include_constant = TRUE`, and omitted when `FALSE`.

This is useful for computing joint penalized log-likelihoods and related MAP-style diagnostics for a fitted `lgspline` object.

Value

A numeric scalar representing the prior log-likelihood.

See Also

`lgspline`

Examples

```
## Data
t <- sort(runif(100, -5, 5))
y <- sin(t) - 0.1*t^2 + rnorm(100)

## Model keeping penalties fixed
model_fit <- lgspline(t, y, opt = FALSE)

## Full joint log-likelihood, conditional upon known sigma^2 = 1
jntloglik <- sum(dnorm(model_fit$y,
                    model_fit$ytilde,
                    1,
                    log = TRUE)) +
          prior_loglik(model_fit, sigmasq = 1)
print(jntloglik)
```

---

process_qp *Prepare Quadratic Programming Constraints for lgspline*

Description

Constructs `qp_Amat`, `qp_bvec`, and `qp_meq` from the built-in quadratic programming constraints handled here: range bounds, derivative sign constraints (first and second), monotonicity constraints, and user-supplied custom constraint functions.

This function was refactored from the inline QP setup block in `lgspline.fit` to improve readability and testability. The interface is fully backward-compatible: existing calls that pass `TRUE`/`FALSE` for derivative flags continue to work.

Usage

```
process_qp(
  X,
  K,
  p_expansions,
  order_list,
  colnm_expansions,
  expansion_scales,
  power1_cols,
  power2_cols,
  nonspline_cols,
  interaction_single_cols,
  interaction_quad_cols,
  triplet_cols,
  include_2way_interactions,
  include_3way_interactions,
  include_quadratic_interactions,
  family,
  mean_y,
  sd_y,
  N_obs,
  qp_observations = NULL,
  qp_positive_derivative = FALSE,
  qp_negative_derivative = FALSE,
  qp_positive_2ndderivative = FALSE,
  qp_negative_2ndderivative = FALSE,
  qp_monotonic_increase = FALSE,
  qp_monotonic_decrease = FALSE,
  qp_range_upper = NULL,
  qp_range_lower = NULL,
  qp_Amat_fxn = NULL,
  qp_bvec_fxn = NULL,
  qp_meq_fxn = NULL,
  qp_Amat = NULL,
  qp_bvec = NULL,
  qp_meq = 0,
  all_derivatives_fxn = NULL,
  og_cols = NULL,
  include_warnings = TRUE,
  ...
)
```

Arguments

`X`	List of per-partition design matrices.
`K`	Integer. Number of interior knots.
`p_expansions`	Integer. Number of basis expansions per partition.
`order_list`	List of per-partition observation index vectors.
`colnm_expansions`	Character vector of expansion column names.
`expansion_scales`	Numeric vector of expansion standardization scales.
`power1_cols`	Integer vector of linear-term column indices.
`power2_cols`	Integer vector of quadratic-term column indices.
`nonspline_cols`	Integer vector of non-spline linear column indices.
`interaction_single_cols`	Integer vector of linear-by-linear interaction column indices.
`interaction_quad_cols`	Integer vector of linear-by-quadratic interaction column indices.
`triplet_cols`	Integer vector of three-way interaction column indices.
`include_2way_interactions`	Logical switch controlling whether two-way interactions are included in derivative construction.

`include_3way_interactions`
: Logical switch controlling whether three-way interactions are included in derivative construction.

`include_quadratic_interactions`
: Logical switch controlling whether quadratic interactions are included in derivative construction.

`family`
: GLM family object.

`mean_y`, `sd_y`
: Numeric scalars for response standardization.

`N_obs`
: Integer. Total sample size.

`qp_observations`
: Optional integer vector of observation indices.

`qp_positive_derivative`, `qp_negative_derivative`
: Logical scalar, character vector, or integer vector. See section *Per-Variable Derivative Constraints*.

`qp_positive_2ndderivative`, `qp_negative_2ndderivative`
: Same as above but for second derivatives.

`qp_monotonic_increase`, `qp_monotonic_decrease`
: Logical. Constrain fitted values to be monotonic in observation order.

`qp_range_upper`, `qp_range_lower`
: Optional numeric upper/lower bounds for fitted values.

`qp_Amat_fxn`, `qp_bvec_fxn`, `qp_meq_fxn`
: Optional user-supplied constraint-generating functions.

`qp_Amat`, `qp_bvec`, `qp_meq`
: Optional pre-built QP objects. Their presence marks QP handling as active, but this helper does not append them to the built-in constraints it constructs; they are expected to be handled outside this constructor.

`all_derivatives_fxn`
: Function to compute derivatives from expansion matrices (the `all_derivatives` closure from `lgspline.fit`).

`og_cols`
: Optional character vector of original predictor column names.

`include_warnings`
: Logical. Whether to issue warnings.

`...`
: Additional arguments forwarded to custom constraint functions.

Value

A list with components:

qp_Amat $P \times M$ combined constraint matrix.

qp_bvec Numeric vector of length M.

qp_meq Integer. Number of leading equality constraints.

quadprog Logical. `TRUE` if any QP constraints are active.

Per-Variable Derivative Constraints

The arguments `qp_positive_derivative`, `qp_negative_derivative`, `qp_positive_2ndderivative`, and `qp_negative_2ndderivative` now accept three forms:

`FALSE` No constraint (default).

`TRUE` Constrain **all** predictor variables (backward compatible with previous behavior).

Character or integer vector Constrain only the specified predictor variables. Character entries are resolved via `og_cols`; integer entries refer to column positions in the predictor matrix.

This allows, for example, enforcing a positive first derivative for variable `"Dose"` and a negative first derivative for variable `"Time"` simultaneously:

```
  lgspline(...,
           qp_positive_derivative = "Dose",
           qp_negative_derivative = "Time")
```

The arguments `qp_monotonic_increase` and `qp_monotonic_decrease` remain `TRUE`/`FALSE` only, because they constrain fitted values in observation order (not per-variable).

Examples

```
## Not run:
## Standalone verification: simple 1-D monotonic increase
set.seed(42)
t <- seq(-5, 5, length.out = 200)
y <- 3 * sin(t) + t + rnorm(200, 0, 0.5)

## Fit with positive first-derivative constraint on all variables
fit1 <- lgspline(t, y, K = 3,
                 qp_positive_derivative = TRUE)

## Verify: first derivative should be >= 0 everywhere
derivs1 <- predict(fit1, new_predictors = sort(t),
                   take_first_derivatives = TRUE)
stopifnot(all(derivs1$first_deriv >= -1e-8))

## Fit with monotonic increase (observation-order)
fit2 <- lgspline(t, y, K = 3,
                 qp_monotonic_increase = TRUE)
preds2 <- predict(fit2, new_predictors = sort(t))
stopifnot(all(diff(preds2) >= -1e-8))

## Per-variable constraints: 2-D example
x1 <- runif(500, -5, 5)
x2 <- runif(500, -5, 5)
y2 <- x1 + sin(x2) + rnorm(500, 0, 0.5)
dat2 <- data.frame(x1 = x1, x2 = x2, y = y2)

## Constrain x1 to have positive derivative, x2 to have negative
fit3 <- lgspline(y ~ spl(x1, x2), data = dat2, K = 2,
                 qp_positive_derivative = "x1",
                 qp_negative_derivative = "x2")

## Verify per-variable derivatives
newdat <- expand.grid(x1 = seq(-4, 4, length.out = 50),
                      x2 = seq(-4, 4, length.out = 50))
d3 <- predict(fit3, new_predictors = newdat,
              take_first_derivatives = TRUE)

## x1 derivative should be >= 0
stopifnot(all(unlist(d3$first_deriv[["_1_"]]) >= -1e-6))
## x2 derivative should be <= 0
stopifnot(all(unlist(d3$first_deriv[["_2_"]]) <= 1e-6))

## Numeric column indices work identically
fit4 <- lgspline(y ~ spl(x1, x2), data = dat2, K = 2,
                 qp_positive_derivative = 1,
                 qp_negative_derivative = 2)

## Range + derivative constraints simultaneously
fit5 <- lgspline(t, y, K = 3,
                 qp_positive_derivative = TRUE,
                 qp_range_lower = -5,
                 qp_range_upper = 15)
preds5 <- predict(fit5)
stopifnot(all(preds5 >= -5 - 1e-6))
stopifnot(all(preds5 <= 15 + 1e-6))

## End(Not run)
```

`reml_grad_from_dV` *Evaluate the REML gradient with respect to a single correlation parameter*

Description

Computes the derivative of the negative REML objective with respect to a scalar correlation parameter, given the matrix derivative $\partial \mathbf{V}/\partial\rho$.

Usage

```
reml_grad_from_dV(dV, model_fit, glm_weight_function, ...)
```

Arguments

`dV`	$N \times N$ numeric matrix giving $\partial \mathbf{V}/\partial\rho$ for one scalar parameter ρ, with the chain rule through any reparameterization already applied.
`model_fit`	A fitted `lgspline` object; see `lgspline`.

`glm_weight_function`
: The GLM weight function used during model fitting; see the `glm_weight_function` argument of `lgspline`.

`...`
: Additional arguments forwarded to `glm_weight_function`.

Details

Notation: $\mathbf{D} = \mathrm{diag}(d_1, \ldots, d_N)$ is the diagonal matrix of observation weights (`observation_weights`), and $\mathbf{W} = \mathrm{diag}(w_1, \ldots, w_N)$ is the diagonal matrix of GLM working weights evaluated at the current fitted values via `glm_weight_function`, with the observation-weight contribution carried separately by $\mathbf{D}$. For canonical GLM families, w_i is the usual IRWLS/Fisher-scoring weight on the mean–variance scale; for example, logistic regression gives $w_i = \mu_i(1 - \mu_i)$. The combined weighting entering the information matrix is $\mathbf{WD}$. In Gaussian identity models both reduce to scalar multiples of the identity.

$\mathbf{V}$ is the $N \times N$ correlation matrix implied by `VhalfInv`, with $\mathbf{V}^{-1} = (\mathbf{V}^{-1/2})^\top \mathbf{V}^{-1/2}$.

The penalized observed information at the current iterate is

$$\mathbf{M} = (\mathbf{X}^*)^\top \mathbf{V}^{-1} \mathbf{WD}\, \mathbf{X}^* + \mathbf{U}^\top \boldsymbol{\Lambda} \mathbf{U},$$

where $\mathbf{X}^* = \mathbf{XU}$ is the constrained design ($N \times P$) and the first term is the quadratic approximation to the penalized log-likelihood Hessian at the current $\boldsymbol{\mu}$. For non-Gaussian families this is a local approximation (the IRWLS/Fisher scoring Hessian), not an exact GLS information matrix.

The constraint projection is $\mathbf{U} = \mathbf{I} - \mathbf{GA}(\mathbf{A}^\top \mathbf{GA})^{-1}\mathbf{A}^\top$ with $\mathbf{G} = \mathbf{M}^{-1}$. $\mathbf{U}$ is idempotent ($\mathbf{U}^2 = \mathbf{U}$) but *not* symmetric, so $\mathbf{U}^\top \boldsymbol{\Lambda} \mathbf{U} \neq \mathbf{U} \boldsymbol{\Lambda} \mathbf{U}^\top$.

The `U` stored in `model_fit$U` is on the expansion- and response-standardised scale. The rescaled version used here is

$$\tilde{\mathbf{U}} = \mathbf{U} \cdot \mathrm{diag}(1, s_1, \ldots, s_{p-1}, 1, s_1, \ldots)/\hat{\sigma}_y,$$

where s_j are the expansion scales and $\hat{\sigma}_y$ standardises the response. All quantities below use $\tilde{\mathbf{U}}$ in place of $\mathbf{U}$.

REML objective and its gradient: The REML objective is constructed by integrating out the fixed effects from the penalized log-likelihood, using a Laplace approximation to the marginal likelihood for non-Gaussian families. This approximation is exact for Gaussian identity models and is the standard extension used in restricted maximum likelihood estimation for GLMMs.

The REML correction term is $-\frac{1}{2}\log|\mathbf{M}|$, where $\mathbf{M}$ is the penalized observed information defined above. Differentiating with respect to ρ (noting only $\mathbf{V}$ depends on ρ) gives the REML correction gradient

$$-\frac{1}{2}\mathrm{tr}\Big(\mathbf{M}^{-1}(\mathbf{X}^*)^\top \mathbf{V}^{-1}\frac{\partial \mathbf{V}}{\partial \rho}\mathbf{V}^{-1}\mathbf{WD}\,\mathbf{X}^*\Big).$$

Full gradient:

$$\frac{\partial(-\ell_R)}{\partial \rho} = \frac{1}{N}\left[\underbrace{\frac{1}{2}\mathrm{tr}\Big(\mathbf{V}^{-1}\frac{\partial \mathbf{V}}{\partial \rho}\Big)}_{\text{(i) log-det of } \mathbf{V}} \underbrace{-\frac{1}{2\tilde{\sigma}^2}\mathbf{r}^\top \frac{\partial \mathbf{V}}{\partial \rho}\mathbf{r}}_{\text{(ii) residual quadratic form}} \underbrace{-\frac{1}{2}\mathrm{tr}\Big(\mathbf{M}^{-1}(\mathbf{X}^*)^\top \mathbf{V}^{-1}\frac{\partial \mathbf{V}}{\partial \rho}\mathbf{V}^{-1}\mathbf{WD}\,\mathbf{X}^*\Big)}_{\text{(iii) REML correction}}\right],$$

where the whitened residual is

$$\mathbf{r} = \mathrm{diag}\Big(\sqrt{d_i}/\sqrt{w_i}\Big)\,\mathbf{V}^{-1/2}(\mathbf{y} - \boldsymbol{\mu}),$$

and $r_i = [\mathbf{V}^{-1/2}(\mathbf{y} - \boldsymbol{\mu})]_i \sqrt{d_i}/\sqrt{w_i}$.

Term (i) does not involve $\mathbf{D}$ or $\mathbf{W}$; the log-determinant of $\mathbf{V}$ depends only on the correlation structure. For non-Gaussian families terms (ii) and (iii) are evaluated at the current IRWLS iterate and constitute a local approximation.

Value

A scalar: the gradient of the negative REML objective with respect to ρ, divided by N.

See Also

lgspline, lgspline.fit

---

std *Standardize Vector to Z-Scores*

Description

Centers a vector by its sample mean, then scales it by its sample standard deviation $(\mathrm{x}-\mathrm{mean}(\mathrm{x}))/\mathrm{sd}(\mathrm{x})$.

Usage

```
std(x)
```

Arguments

x	Numeric vector to standardize

Value

Standardized vector with sample mean 0 and standard deviation 1

Examples

```
x <- c(1, 2, 3, 4, 5)
std(x)
print(mean(x))
print(sd(x))
```

`summary.lgspline` *Summary Method for lgspline Objects*

Description

Summary Method for lgspline Objects

Usage

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

Arguments

`object`	An lgspline model object.
`...`	Not used.

Value

An object of class `summary.lgspline`, a list containing:

model_family The `family` object.

observations Number of observations N.

predictors Number of predictor variables q.

knots Number of knots K.

basis_functions Basis functions per partition p.

estimate_dispersion "Yes" or "No".

cv Critical value used for confidence intervals.

coefficients Coefficient matrix from `wald_univariate`, or a single-column estimate matrix if `return_varcovmat = FALSE`.

sigmasq_tilde Estimated dispersion $\tilde{\sigma}^2$.

trace_XUGX Trace of the hat matrix $\mathrm{trace}(\mathbf{XUGX}^\top)$.

N Number of observations.

summary.wald_lgspline *Summary Method for wald_lgspline Objects*

Description

Prints a header with model info then delegates to `print.wald_lgspline`.

Usage

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

Arguments

`object`	A `"wald_lgspline"` object.
`...`	Passed to `print.wald_lgspline`.

Value

Invisibly returns `object`.

See Also

`wald_univariate`, `print.wald_lgspline`

wald_univariate *Univariate Wald Tests and Confidence Intervals for lgspline Coefficients*

Description

Computes per-coefficient Wald tests and confidence intervals from a fitted lgspline. For Gaussian identity-link models, t-statistics and t-intervals are used; otherwise z-statistics.

Usage

```
wald_univariate(object, scale_vcovmat_by = 1, cv, ...)
```

Arguments

`object`	A fitted lgspline object. Must have been fit with `return_varcovmat = TRUE`.
`scale_vcovmat_by`	Numeric; scaling factor for the variance-covariance matrix. Default 1.
`cv`	Numeric; critical value for confidence intervals. If missing, defaults to `object$critical_value` or `qnorm(0.975)`.
`...`	Additional arguments passed to the internal `wald_univariate` method.

Value

An object of class "wald_lgspline", a list with:

coefficients Matrix with columns: Estimate, Std. Error, t value or z value, Pr(>|t|) or Pr(>|z|), CI LB, CI UB.

critical_value Critical value used.

family GLM family from the fitted model.

N Number of observations.

trace_XUGX Effective df trace term.

statistic_name "t value" or "z value".

p_value_name "Pr(>|t|)" or "Pr(>|z|)".

df.residual Residual degrees of freedom when supplied by the internal Wald method.

Print, summary, and plot methods are available; see `print.wald_lgspline`, `summary.wald_lgspline`, `plot.wald_lgspline`.

See Also

`lgspline`, `confint.lgspline`, `print.wald_lgspline`, `summary.wald_lgspline`, `plot.wald_lgspline`

Examples

```
set.seed(1234)
t <- runif(1000, -10, 10)
y <- 2*sin(t) + -0.06*t^2 + rnorm(length(t))
model_fit <- lgspline(t, y, return_varcovmat = TRUE)

wald_default <- wald_univariate(model_fit)
print(wald_default)

## t-distribution critical value
eff_df <- model_fit$N - model_fit$trace_XUGX
wald_t <- wald_univariate(model_fit, cv = qt(0.975, eff_df))
print(wald_t)

coef_table <- wald_default$coefficients
plot(wald_default)
```

`weibull_dispersion_function` *Estimate Weibull Dispersion for Accelerated Failure Time Model*

Description

Computes the dispersion parameter (sigma^2 = scale^2) for a Weibull accelerated failure time (AFT) model, supporting right-censored survival data. The returned value is sigma^2, where sigma is the Weibull scale parameter matching `survreg$scale`.

This both provides a tool for actually fitting Weibull AFT Models, and boilerplate code for users who wish to incorporate Lagrangian multiplier smoothing splines into their own custom models.

Usage

```
weibull_dispersion_function(
  mu,
  y,
  order_indices,
  family,
  observation_weights,
  VhalfInv,
  status
)
```

Arguments

`mu`	Predicted survival times
`y`	Observed response/survival times
`order_indices`	Indices to align status with response
`family`	Weibull AFT model family specification; unused here and retained for interface compatibility.
`observation_weights`	Optional observation weights
`VhalfInv`	Inverse square root of the correlation matrix; unused here and retained for interface compatibility.
`status`	Censoring indicator (1 = event, 0 = censored) Indicates whether an event of interest occurred (1) or the observation was right-censored (0). In survival analysis, right-censoring occurs when the full survival time is unknown, typically because the study ended or the subject was lost to follow-up before the event of interest occurred.

Value

Dispersion estimate (sigma^2) for the Weibull AFT model, i.e., the squared scale parameter. The Weibull scale (sigma) matching `survreg$scale` is `sqrt()` of this value.

See Also

`weibull_scale` for the underlying scale estimation function

Examples

```
## Simulate survival data with covariates
set.seed(1234)
n <- 1000
t1 <- rnorm(n)
t2 <- rbinom(n, 1, 0.5)

## Generate survival times with Weibull-like structure
lambda <- exp(0.5 * t1 + 0.3 * t2)
yraw <- rexp(n, rate = 1/lambda)

## Introduce right-censoring
status <- rbinom(n, 1, 0.75)
y <- ifelse(status, yraw, runif(length(yraw), 0, yraw))

## Example of using dispersion function
mu <- mean(y)
order_indices <- seq_along(y)
weights <- rep(1, n)

## Estimate dispersion (= scale^2 = sigma^2)
dispersion_est <- weibull_dispersion_function(
  mu = mu,
  y = y,
  order_indices = order_indices,
  family = weibull_family(),
  observation_weights = weights,
  VhalfInv = NULL,
  status = status
)

print(dispersion_est)          # sigma^2
print(sqrt(dispersion_est))    # sigma (comparable to survreg$scale)
```

---

weibull_family   *Weibull Family for Survival Model Specification*

Description

Creates a family-like object for Weibull accelerated failure time (AFT) models, including custom log-likelihood, AIC, and deviance helpers.

This both provides a tool for actually fitting Weibull AFT Models, and boilerplate code for users who wish to incorporate Lagrangian multiplier smoothing splines into their own custom models.

Usage

```
weibull_family()
```

Details

Provides a comprehensive family specification for Weibull AFT models, including family name, link function, inverse link function, custom loss function for model tuning, and methods for AIC and log-likelihood computation compatible with `logLik.lgspline`.

Supports right-censored survival data with flexible parameter estimation.

Note on scale vs. dispersion: throughout this package, the lgspline object stores `sigmasq_tilde` which equals sigma^2 (dispersion), where sigma is the Weibull scale parameter matching `survreg$scale`. Functions that accept a `dispersion` argument receive sigma^2; functions that accept a `scale` argument receive sigma.

Value

A family-like list containing the link functions and Weibull-specific methods used by `lgspline`.

Examples

```
## Simulate survival data with covariates
set.seed(1234)
n <- 1000
t1 <- rnorm(n)
t2 <- rbinom(n, 1, 0.5)

## Generate survival times with Weibull-like structure
lambda <- exp(0.5 * t1 + 0.3 * t2)
yraw <- rexp(n, rate = 1/lambda)

## Introduce right-censoring
status <- rbinom(n, 1, 0.75)
y <- ifelse(status, yraw, runif(length(yraw), 0, yraw))

## Prepare data
df <- data.frame(y = y, t1 = t1, t2 = t2, status = status)

## Fit model using custom Weibull family
model_fit <- lgspline(y ~ spl(t1) + t2,
                      df,
                      unconstrained_fit_fxn = unconstrained_fit_weibull,
                      family = weibull_family(),
                      need_dispersion_for_estimation = TRUE,
                      dispersion_function = weibull_dispersion_function,
                      glm_weight_function = weibull_glm_weight_function,
                      schur_correction_function = weibull_schur_correction,
                      status = status,
                      opt = FALSE,
                      K = 1)

summary(model_fit)

## Log-likelihood now works via logLik.lgspline:
# logLik(model_fit)
```

weibull_glm_weight_function

Weibull GLM Weight Function for Constructing Information Matrix

Description

Computes the working weights used in the Weibull AFT information matrix, including the observation-weight contribution returned on the vector scale.

Usage

```
weibull_glm_weight_function(
  mu,
  y,
  order_indices,
  family,
  dispersion,
  observation_weights,
  status
)
```

Arguments

`mu`	Predicted survival times
`y`	Observed response/survival times
`order_indices`	Order of observations when partitioned to match "status" to "response"
`family`	Weibull AFT family; unused here and retained for interface compatibility.
`dispersion`	Estimated dispersion parameter (sigma^2 = scale^2). The lgspline framework stores and passes dispersion (sigma^2); the Weibull scale (sigma) matching `survreg$scale` is `sqrt(dispersion)`.
`observation_weights`	Weights of observations submitted to function
`status`	Censoring indicator (1 = event, 0 = censored) Indicates whether an event of interest occurred (1) or the observation was right-censored (0). In survival analysis, right-censoring occurs when the full survival time is unknown, typically because the study ended or the subject was lost to follow-up before the event of interest occurred.

Details

This function generates weights used in constructing the information matrix after unconstrained estimates have been found. Specifically, it is used in the construction of the **U** and **G** matrices following initial unconstrained parameter estimation.

These weights are analogous to the variance terms in generalized linear models (GLMs). Like logistic regression uses $\mu(1-\mu)$, Poisson regression uses e^{μ}, and Linear regression uses constant weights, Weibull AFT models use $\exp((\log y - \log \mu)/\sigma)$ where $\sigma = \sqrt{\text{dispersion}}$ is the scale parameter.

Value

Numeric vector of working weights for the information matrix, including observation weights when finite and a fallback of 1s when the natural Weibull weights are non-finite.

Examples

```
## Demonstration of glm weight function in constrained model estimation
set.seed(1234)
n <- 1000
t1 <- rnorm(n)
t2 <- rbinom(n, 1, 0.5)

## Generate survival times
lambda <- exp(0.5 * t1 + 0.3 * t2)
yraw <- rexp(n, rate = 1/lambda)

## Introduce right-censoring
status <- rbinom(n, 1, 0.75)
y <- ifelse(status, yraw, runif(length(yraw), 0, yraw))

## Fit model demonstrating use of custom glm weight function
model_fit <- lgspline(y ~ spl(t1) + t2,
                      data.frame(y = y, t1 = t1, t2 = t2),
                      unconstrained_fit_fxn = unconstrained_fit_weibull,
                      family = weibull_family(),
                      need_dispersion_for_estimation = TRUE,
                      dispersion_function = weibull_dispersion_function,
                      glm_weight_function = weibull_glm_weight_function,
                      schur_correction_function = weibull_schur_correction,
                      status = status,
                      opt = FALSE,
                      K = 1)

print(summary(model_fit))
```

`weibull_qp_score_function` *Compute Gradient of Log-Likelihood of Weibull Accelerated Failure Model*

Description

Calculates the gradient of log-likelihood for a Weibull accelerated failure time (AFT) survival model, supporting right-censored survival data.

Usage

```
weibull_qp_score_function(
  X,
  y,
  mu,
  order_list,
  dispersion,
  VhalfInv,
  observation_weights,
  status
)
```

Arguments

`X`	Design matrix
`y`	Response vector
`mu`	Predicted mean vector
`order_list`	List of observation indices per partition
`dispersion`	Dispersion parameter (sigma^2 = scale^2). The lgspline framework stores and passes dispersion (sigma^2); the Weibull scale (sigma) matching `survreg$scale` is `sqrt(dispersion)`.
`VhalfInv`	Inverse square root of correlation matrix; unused here and retained for interface compatibility.
`observation_weights`	Observation weights
`status`	Censoring indicator (1 = event, 0 = censored)

Details

Needed if using "blockfit", correlation structures, or quadratic programming with Weibull AFT models.

The gradient is computed on a scale that omits the 1/sigma prefactor. Specifically, the true score is (1/sigma) * X^T diag(w) (exp(z) - status), but both this function and the corresponding information matrix used internally omit 1/sigma and 1/sigma^2 respectively, so the Newton-Raphson step G*u remains correct. This matches the convention in `unconstrained_fit_weibull`.

Value

Numeric column vector representing the gradient with respect to coefficients.

Examples

```
set.seed(1234)
t1 <- rnorm(1000)
t2 <- rbinom(1000, 1, 0.5)
yraw <- rexp(exp(0.01*t1 + 0.01*t2))
status <- rbinom(1000, 1, 0.25)
```

```
yobs <- ifelse(status, runif(length(yraw), 0, yraw), yraw)
df <- data.frame(
  y = yobs,
  t1 = t1,
  t2 = t2
)

## Example using blockfit for t2 as a linear term - output does not look
# different, but internal methods used for fitting change
model_fit <- lgspline(y ~ spl(t1) + t2,
                      df,
                      unconstrained_fit_fxn = unconstrained_fit_weibull,
                      family = weibull_family(),
                      need_dispersion_for_estimation = TRUE,
                      qp_score_function = weibull_qp_score_function,
                      dispersion_function = weibull_dispersion_function,
                      glm_weight_function = weibull_glm_weight_function,
                      schur_correction_function = weibull_schur_correction,
                      K = 1,
                      blockfit = TRUE,
                      opt = FALSE,
                      status = status,
                      verbose = TRUE)

print(summary(model_fit))
```

`weibull_scale` *Estimate Scale for Weibull Accelerated Failure Time Model*

Description

Computes maximum log-likelihood scale estimate (sigma) for a Weibull accelerated failure time (AFT) survival model.

This both provides a tool for actually fitting Weibull AFT Models, and boilerplate code for users who wish to incorporate Lagrangian multiplier smoothing splines into their own custom models.

Usage

```
weibull_scale(log_y, log_mu, status, weights = 1)
```

Arguments

`log_y`	Logarithm of response/survival times
`log_mu`	Logarithm of predicted survival times
`status`	Censoring indicator (1 = event, 0 = censored) Indicates whether an event of interest occurred (1) or the observation was right-censored (0). In survival analysis, right-censoring occurs when the full survival time is unknown, typically because the study ended or the subject was lost to follow-up before the event of interest occurred.

`weights`	Optional observation weights (default = 1)

Details

Calculates maximum log-likelihood estimate of scale (sigma) for Weibull AFT model accounting for right-censored observations using Brent's method for optimization.

Value

Scalar representing the estimated Weibull scale (sigma), equivalent to `survreg$scale`. The dispersion (as stored in `lgspline$sigmasq_tilde`) is sigma^2.

Examples

```
## Simulate exponential data with censoring
set.seed(1234)
mu <- 2  # mean of exponential distribution
n <- 500
y <- rexp(n, rate = 1/mu)

## Introduce censoring (25% of observations)
status <- rbinom(n, 1, 0.75)
y_obs <- ifelse(status, y, NA)

## Compute scale estimate
scale_est <- weibull_scale(
  log_y = log(y_obs[!is.na(y_obs)]),
  log_mu = log(mu),
  status = status[!is.na(y_obs)]
)

print(scale_est)
```

`weibull_schur_correction` *Correction for the Variance-Covariance Matrix for Uncertainty in Scale*

Description

Computes the Schur complement **S** such that $\mathbf{G}^* = (\mathbf{G}^{-1} + \mathbf{S})^{-1}$ properly accounts for uncertainty in estimating the Weibull scale parameter when estimating the variance-covariance matrix. Otherwise, the variance-covariance matrix is optimistic and assumes the scale is known, when it was in fact estimated. Note that the parameterization adds the output of this function elementwise (not subtract) so for most cases, the output of this function will be negative or a negative definite/semi-definite matrix.

Usage

```
weibull_schur_correction(
  X,
  y,
  B,
  dispersion,
  order_list,
  K,
  family,
  observation_weights,
  status
)

weibull_shur_correction(
  X,
  y,
  B,
  dispersion,
  order_list,
  K,
  family,
  observation_weights,
  status
)
```

Arguments

`X`	Block-diagonal matrices of spline expansions
`y`	Block-vector of response
`B`	Block-vector of coefficient estimates
`dispersion`	Scalar, estimate of dispersion (sigma^2 = scale^2). The lgspline framework stores and passes dispersion (sigma^2); the Weibull scale (sigma) matching `survreg$scale` is `sqrt(dispersion)`.
`order_list`	List of partition orders
`K`	Number of partitions minus 1 (K)
`family`	Distribution family
`observation_weights`	Optional observation weights (default = 1)
`status`	Censoring indicator (1 = event, 0 = censored) Indicates whether an event of interest occurred (1) or the observation was right-censored (0). In survival analysis, right-censoring occurs when the full survival time is unknown, typically because the study ended or the subject was lost to follow-up before the event of interest occurred.

Details

Adjusts the variance-covariance matrix unscaled for coefficients to account for uncertainty in estimating the Weibull scale parameter, that otherwise would be lost if simply using $\mathbf{G} = (\mathbf{X}^T\mathbf{W}\mathbf{X} + \mathbf{L})^{-1}$. This is accomplished using a correction based on the Schur complement so we avoid having to construct the entire variance-covariance matrix, or modifying the procedure for `lgspline` substantially. For any model with nuisance parameters that must have uncertainty accounted for, this tool will be helpful.

This both provides a tool for actually fitting Weibull accelerated failure time (AFT) models, and boilerplate code for users who wish to incorporate Lagrangian multiplier smoothing splines into their own custom models.

Value

List of blockwise Schur-complement corrections $\mathbf{S}_k$ to be elementwise added to each block of the information matrix before inversion, with `0` returned for empty partitions.

Examples

```
## Minimal example of fitting a Weibull Accelerated Failure Time model
# Simulating survival data with right-censoring
set.seed(1234)
t1 <- rnorm(1000)
t2 <- rbinom(1000, 1, 0.5)
yraw <- rexp(exp(0.01*t1 + 0.01*t2))
# status: 1 = event occurred, 0 = right-censored
status <- rbinom(1000, 1, 0.25)
yobs <- ifelse(status, runif(length(yraw), 0, yraw), yraw)
df <- data.frame(
  y = yobs,
  t1 = t1,
  t2 = t2
)

## Fit model using lgspline with Weibull Schur correction
model_fit <- lgspline(y ~ spl(t1) + t2,
                      df,
                      unconstrained_fit_fxn = unconstrained_fit_weibull,
                      family = weibull_family(),
                      need_dispersion_for_estimation = TRUE,
                      dispersion_function = weibull_dispersion_function,
                      glm_weight_function = weibull_glm_weight_function,
                      schur_correction_function = weibull_schur_correction,
                      status = status,
                      opt = FALSE,
                      K = 1)

print(summary(model_fit))

## Fit model using lgspline without Weibull Schur correction
naive_fit <- lgspline(y ~ spl(t1) + t2,
                      df,
                      unconstrained_fit_fxn = unconstrained_fit_weibull,
                      family = weibull_family(),
                      need_dispersion_for_estimation = TRUE,
                      dispersion_function = weibull_dispersion_function,
                      glm_weight_function = weibull_glm_weight_function,
                      status = status,
                      opt = FALSE,
                      K = 1)

print(summary(naive_fit))
```

Index

.active_set_refine, *24*, *27*
.bf_case_gauss_gee, *26*
.bf_case_gauss_no_corr, *26*
.bf_case_glm_gee, *26*
.bf_case_glm_no_corr, *26*
.bf_constrained_flat_update, *26*
.bf_inner_weighted, *26*
.bf_lagrangian_project, *26*
.bf_split_components, *26*
.bf_sqp_loop, *26, 27*
.check_kkt_partitionwise, *24*
.detect_qp_global, *24*
.get_B_gaussian_nocorr, *17*
.get_B_gee_gaussian, *17*, *19*
.get_B_gee_glm, *17*, *19*, *26*
.get_B_gee_glm_woodbury, *18, 19*
.get_B_gee_woodbury, *18, 19*
.get_B_glm_nocorr, *17*
.qp_refine, *24*
.woodbury_decompose_V, *16, 17*
.woodbury_halfsqrt_components, *17*
.woodbury_redecompose_weighted, *18*

AIC, *87, 88*
approx_grad, *19*, *31*

BIC, *87, 88*
blockfit_solve, *18*, *21*, *26, 27*, *68*

coef.lgspline, 3, *33*, *37*
coef.wald_lgspline, 4
compute_dG_dlambda, *30*
compute_dG_u_dlambda_xy, *30*
compute_dGhalf, *30*
compute_dW_dlambda_wrapper, *30*
compute_G_eigen, *15*, *23*
compute_Lambda, *14*, *29*
compute_trace_H, *22*
compute_trace_UGXX_wrapper, *30*
confint.lgspline, 5, *21–23*, *33*, *65*, *68*, *104*, *119*
confint.wald_lgspline, 5
cox_dispersion_function, 6
cox_family, 7
cox_glm_weight_function, 7
cox_helpers, 9
cox_qp_score_function, 9
cox_schur_correction, 10
create_onehot, 11, *13*

damped_newton_r, *16*, *18*
Details, 12

efficient_bfgs, *19*, *31*
equation, *33*, 34

family, *117*
find_extremum, *33*, 38

generate_posterior, *20*, *23*, *33*, *38*, *39*, 40, *43*, *46*, *105*
generate_posterior_correlation, *24*, *33*, *41, 42*, 43
get.knnx, *27*
get_2ndDerivPenalty, *29*
get_2ndDerivPenalty_wrapper, *29*
get_B, *18*, *21*, *26, 27*
get_B_verification_examples, 46
get_polynomial_expansions, *14*, *58*
get_U, *16*

integrate, *48*, 48
integrate.lgspline, *32, 33*, 48
invert, *18*

kmeans, *68*

leave_one_out, 50, *68*
lgspline, *4*, *14*, *16*, *19*, *21*, *25*, *27*, *30*, *37*, *39*, *42*, *46*, 51, *65*, *83*, *85*, *87*, *88*, *93*, *94*, *103*, *106*, *109*, *114–116*, *119*, *129*

lgspline.fit, *13, 14*, *16*, *18*, *46*, *68*, *110*, *116*
lgspline_cox, 83, *85*, *87*
lgspline_negbin, 84
lgspline_weibull, *85*, 86
logLik, *88*
logLik.lgspline, *7*, *33*, *65*, *68*, 87
loglik_cox, 89
loglik_negbin, 90
loglik_weibull, 91

make_constraint_matrix, *15*
make_derivative_matrix, *24, 25*
make_partitions, *14*, *27*, *56*
matinvsqrt, *18*, *23*, 92
matsqrt, *18*, *23*, 94

negbin_dispersion_function, 95
negbin_family, *85*, 96
negbin_glm_weight_function, 97
negbin_helpers, 98
negbin_qp_score_function, 98
negbin_schur_correction, 99
negbin_theta, 100
nr_iterate, *16*

optim, *31*, *68*

plot, *102–104*
plot.lgspline, *33*, *37*, 101, *106*
plot.wald_lgspline, 104, *119*
plot_ly, *68*, *102, 103*
predict.lgspline, *22, 23*, *32, 33*, 105
print.equation (equation), 34
print.lgspline, *33*, 107
print.summary.lgspline, *33*, 107
print.wald_lgspline, 108, *118, 119*
printCoefmat, *107, 108*
prior_loglik, *88*, 109
process_input, *13*
process_qp, *24, 25*, 110

reml_grad_from_dV, *19*, 114

solve.QP, *16, 17*, *24*, *33*, *68*
std, 116
summary.lgspline, *33*, 117
summary.wald_lgspline, 118, *119*

tune_Lambda, *30, 31*

unconstrained_fit_default, *16*, *18*, *58*
unconstrained_fit_negbin, *85*
unconstrained_fit_weibull, *87*, *125*

wald_univariate, *5*, *22, 23*, *33*, *42*, *104*, *108*, *117, 118*, 118
weibull_dispersion_function, *22*, 119
weibull_family, *22*, *87*, 121
weibull_glm_weight_function, *22*, 123
weibull_qp_score_function, 124
weibull_scale, *120*, 126
weibull_schur_correction, *22*, *99*, 127
weibull_shur_correction (weibull_schur_correction), 127