runlmc Documentation

Release 0.0

Vlad Feinberg

Feb 26, 2022

Contents

The code and open source license are available on the Github project page.

CHAPTER 1

runlmc package

1.1 Subpackages

1.1.1 runlmc.approx package

Submodules

runlmc.approx.interpolation module

runlmc.approx.interpolation.**autogrid**(*Xs*, *lo*, *hi*, *m*)

Generate a grid from *lo* to *hi* with *m* points, but with sensible defaults based on *Xs* if any of the other parameters are *None*.

In particular, the defaults are chosen such that all elements in *Xs* fall within the grid by two elements on both sides. If a user's values do not guarantee this property, they are changed.

Parameters

- **Xs** – list of 2-dimensional numpy design matrices, the inputs.
- **lo** – optional lower bound
- **hi** – optional upper bound
- **m** – optional grid size

Returns the equally-spaced grid points, as a list for each dimension

runlmc.approx.interpolation.**cubic_kernel**(*x*)

The cubic convolution kernel can be used to compute interpolation coefficients. Its definition is taken from:

Cubic Convolution Interpolation for Digital Image Processing by Robert G. Keys.

It is supported on values of absolute magnitude less than 2 and is defined as:

$$u(x) = \begin{cases} \frac{3}{2}|x|^3 - \frac{5}{2}|x|^2 + 1 & 0 \leq |x| \leq 1 \\ \frac{3}{2}|x|^3 + \frac{5}{2} - 4|x| + 2 & 1 < |x| \leq 2 \end{cases}$$

Parameters **x** (numpy.ndarray) – input array

Returns u vectorized over *x*

runlmc.approx.interpolation.**interp_bicubic**(*gridx*, *gridy*, *samples*)

Given an implicit two dimensional grid from the Cartesian product of *gridx* and *gridy*, with sizes *m1,m2*, respectively (such that each grid is an equispaced increasing sequence of values), and given *n* sample points (which should be 2D), this computes interpolation coefficients for a cubic interpolation on the grid.

An interpolation coefficient matrix *M* is then an *n* by *m1*m2* matrix that has 16 entries per row.

For a (vectorized) twice-differentiable function *f*, *M.dot(f(cartesian_product(gridx, gridy)).ravel())* approaches *f(sample)* at a rate of $O(m^{-3})$.

Returns the interpolation coefficient matrix

Raises **ValueError** – if any conditions similar to those in *interp_cubic()* are violated.

runlmc.approx.interpolation.**interp_cubic**(*grid*, *samples*)

Given a one dimensional grid *grid* of size *m* (that's sorted) and *n* sample points *samples*, compute the interpolation coefficients for a cubic interpolation on the grid.

An interpolation coefficient matrix *M* is then an *n* by *m* matrix that has 4 entries per row.

For a (vectorized) twice-differentiable function *f*, *M.dot(f(grid))* approaches *f(sample)* at a rate of $O(m^{-3})$.

samples should be contained with the range of *grid*, but this method will create a matrix capable of handling extrapolation.

Returns the interpolation coefficient matrix

Raises **ValueError** – if any of the following hold true:

1. *grid* or *samples* are not 1-dimensional
2. *grid* size less than 4
3. *grid* is not equispaced

runlmc.approx.interpolation.**multi_interpolant**(*Xs*, **inducing_grids*)

Creates a sparse CSR matrix interpolant across multiple inputs *Xs*.

Each input is mapped onto the inducing grid with a cubic interpolation, with *runlmc.approx.interpolation.interp_cubic()* or *runlmc.approx.interpolation.interp_bicubic()*, depending on the dimensionality of *Xs*.

This induces $n_i \times m$ interpolation matrices W_i for the i-th element of *Xs* onto the inducing grid, which is shared between all *Xs*. Note that m is the total size of the Cartesian product of the inducing grids for each dimension of the input. This also implies that the number of inducing grid axes passed as arguments to *multi_interpolant* must be equal to the dimension of *Xs*.

Parameters

- **Xs** – list of n_i-by-d input points, where d may be either 1 or 2. In the case $d = 1$ the input matrices can be vectors.
- **inducing_grids** – list 1-dimensional vector of grid points, one for each input dimension

Returns the rectangular block diagonal matrix of W_i.

runlmc.approx.iterative module

class runlmc.approx.iterative.**Iterative**

Bases: object

Target solve() tolerance. Only errors > tol reported.

static solve(*K*, *y*, *verbose=False*, *minres=True*, *tol=0.0001*)
Solves the linear system $K\mathbf{x} = \mathbf{y}$.

Parameters

- **K** – a `SymmetricMatrix`
- **y** – **y**
- **verbose** – whether to return number of iterations
- **minres** – uses minres if true, else lcg

Returns **x**, number of iterations and error if verbose

runlmc.approx.ski module

class runlmc.approx.ski.**SKI**(*K*, *W*, *WT*)
Bases: *runlmc.linalg.composition.Composition*

as_numpy()

Returns numpy matrix equivalent, as a 2D `numpy.ndarray`

upper_eig_bound()

Module contents

1.1.2 runlmc.kern package

Submodules

runlmc.kern.identity module

The identity kernel (no covariance).

class runlmc.kern.identity.**Identity**(*name='id'*, *active_dims=None*)
Bases: *runlmc.kern.stationary_kern.StationaryKern*

This class defines identity kernel k.

$$k(r) = 1_{r=0}$$

Parameters

- **name** –
- **active_dims** – see *runlmc.kern.stationary_kern.StationaryKern* for details.

from_dist(*dists*)

Parameters **dists** – N-size numpy array of positive (Euclidean) distances.

Returns kernel value at each of the given distances

kernel_gradient(*dists*)
Let this kernel be parameterized by some parameters $\boldsymbol{\theta} \in \mathbb{R}^p$. For every $\theta_j \in \boldsymbol{\theta}$, at any given distance d, we can compute the derivative $\partial_{\theta_j} k(d)$. For the evaluation of this partial derivative at multiple places, $\mathbf{d}$, we call the vector of partial derivatives $\partial_{\theta_j} k(\mathbf{d})$.

Parameters **dists** – a one-dimensional array of distances corresponding to $\mathbf{d}$, above.

Returns An iterable whose j-th entry is $\partial_{\theta_j} k(\mathbf{d})$.

to_gpy()

Returns GPy version of this kernel.

update_gradient(*grad*)

Parameters **grad** – a one-dimensional array, representing the gradient vector $\nabla_{\boldsymbol{\theta}} L$ for the likelihood with respect to this kernel's parameters, in the same order of parameters as the row order returned by *kernel_gradient()*.

runlmc.kern.matern32 module

class runlmc.kern.matern32.**Matern32**(*inv_lengthscale=1*, *name='matern32'*, *active_dims=None*)
Bases: *runlmc.kern.stationary_kern.StationaryKern*

This class defines the Matérn-3/2 kernel k.

$$k(r) = (1 + r\gamma\sqrt{3}) \exp\left(-\gamma r\sqrt{3}\right)$$

Parameters

- **inv_lengthscale** – γ, above.
- **name** –
- **active_dims** – see *runlmc.kern.stationary_kern.StationaryKern* for details.

from_dist(*dists*)

Parameters **dists** – N-size numpy array of positive (Euclidean) distances.

Returns kernel value at each of the given distances

kernel_gradient(*dists*)
Let this kernel be parameterized by some parameters $\boldsymbol{\theta} \in \mathbb{R}^p$. For every $\theta_j \in \boldsymbol{\theta}$, at any given distance d, we can compute the derivative $\partial_{\theta_j} k(d)$. For the evaluation of this partial derivative at multiple places, $\mathbf{d}$, we call the vector of partial derivatives $\partial_{\theta_j} k(\mathbf{d})$.

Parameters **dists** – a one-dimensional array of distances corresponding to $\mathbf{d}$, above.

Returns An iterable whose j-th entry is $\partial_{\theta_j} k(\mathbf{d})$.

to_gpy()

Returns GPy version of this kernel.

update_gradient(*grad*)

Parameters **grad** – a one-dimensional array, representing the gradient vector $\nabla_{\boldsymbol{\theta}} L$ for the likelihood with respect to this kernel's parameters, in the same order of parameters as the row order returned by *kernel_gradient()*.

runlmc.kern.rbf module

class runlmc.kern.rbf.**RBF**(*inv_lengthscale=1*, *name='rbf'*, *active_dims=None*)
Bases: *runlmc.kern.stationary_kern.StationaryKern*

This class defines the RBF kernel k.

$$k(r) = \exp \frac{-\gamma r^2}{2}$$

Parameters

- **inv_lengthscale** – γ, above.
- **name** –
- **active_dims** – see *runlmc.kern.stationary_kern.StationaryKern* for details.

from_dist(*dists*)

Parameters **dists** – N-size numpy array of positive (Euclidean) distances.

Returns kernel value at each of the given distances

kernel_gradient(*dists*)
Let this kernel be parameterized by some parameters $\boldsymbol{\theta} \in \mathbb{R}^p$. For every $\theta_j \in \boldsymbol{\theta}$, at any given distance d, we can compute the derivative $\partial_{\theta_j} k(d)$. For the evaluation of this partial derivative at multiple places, $\mathbf{d}$, we call the vector of partial derivatives $\partial_{\theta_j} k(\mathbf{d})$.

Parameters **dists** – a one-dimensional array of distances corresponding to $\mathbf{d}$, above.

Returns An iterable whose j-th entry is $\partial_{\theta_j} k(\mathbf{d})$.

to_gpy()

Returns GPy version of this kernel.

update_gradient(*grad*)

Parameters **grad** – a one-dimensional array, representing the gradient vector $\nabla_{\boldsymbol{\theta}} L$ for the likelihood with respect to this kernel's parameters, in the same order of parameters as the row order returned by *kernel_gradient()*.

runlmc.kern.scaled module

class runlmc.kern.scaled.**Scaled**(*k*)
Bases: *runlmc.kern.stationary_kern.StationaryKern*

from_dist(*dists*)

Parameters **dists** – N-size numpy array of positive (Euclidean) distances.

Returns kernel value at each of the given distances

kernel_gradient(*dists*)
Let this kernel be parameterized by some parameters $\boldsymbol{\theta} \in \mathbb{R}^p$. For every $\theta_j \in \boldsymbol{\theta}$, at any given distance d, we can compute the derivative $\partial_{\theta_j} k(d)$. For the evaluation of this partial derivative at multiple places, $\mathbf{d}$, we call the vector of partial derivatives $\partial_{\theta_j} k(\mathbf{d})$.

Parameters **dists** – a one-dimensional array of distances corresponding to $\mathbf{d}$, above.

Returns An iterable whose j-th entry is $\partial_{\theta_j} k(\mathbf{d})$.

to_gpy()

Returns GPy version of this kernel.

update_gradient(*grad*)

Parameters **grad** – a one-dimensional array, representing the gradient vector $\nabla_{\boldsymbol{\theta}} L$ for the likelihood with respect to this kernel's parameters, in the same order of parameters as the row order returned by *kernel_gradient()*.

runlmc.kern.stationary_kern module

This file contains *StationaryKern*, the kernel base class.

Note this class does not accomplish as much as the corresponding one does in *GPy*. See the class documentation for details.

Note that the corresponding GPy versions of these kernels have a scaling parameter that's avoided inthe LMC case because it would be redundant with the coregionalization constants.

class runlmc.kern.stationary_kern.**StationaryKern**(*name*, *active_dims=None*)

Bases: *runlmc.parameterization.parameterized.Parameterized*

The *StationaryKern* defines a stationary kernel.

It is a light wrapper around the mathematical definition of a kernel, which includes its parameters. A kernel object never contains any data, as a parameterized object its gradients can be changed according to whatever data it's being tuned to.

A stationary kernel is a function $k(r)$ defined on $\mathbb{R}_+$ such that a matrix K of entries $k(\|\mathbf{x}_i - \mathbf{x}_j\|)$ is positive semi-definite. Moreover, if $\{\mathbf{x}_i\}_i$ lie on a grid then K will be block-Toeplitz of Toeplitz blocks. If further $\mathbf{x}_i \in \mathbb{R}$ then K will be Toeplitz.

Parameters

- **name** –
- **active_dims** – active dimensions (from which Euclidean distances fed into the kernel as inputs are computed). I.e., if data for a problem are 3D (x, y, t) with x, y spatial coordinates and t time then the default *active_dims* setting of *None* would evaluate the kernel k between two points as $k(\|(x_1 - x_2, y_1 - y_2, t_1 - t_2)\|)$, which doesn't make much sense. In this case you might want to use *active_dims* to specify a sum kernel of two kernels, one over the (x, y) values alone and the other over the t values alone.

from_dist(*dists*)

Parameters **dists** – *N*-size numpy array of positive (Euclidean) distances.

Returns kernel value at each of the given distances

kernel_gradient(*dists*)

Let this kernel be parameterized by some parameters $\boldsymbol{\theta} \in \mathbb{R}^p$. For every $\theta_j \in \boldsymbol{\theta}$, at any given distance d, we can compute the derivative $\partial_{\theta_j} k(d)$. For the evaluation of this partial derivative at multiple places, $\mathbf{d}$, we call the vector of partial derivatives $\partial_{\theta_j} k(\mathbf{d})$.

Parameters **dists** – a one-dimensional array of distances corresponding to $\mathbf{d}$, above.

Returns An iterable whose j-th entry is $\partial_{\theta_j} k(\mathbf{d})$.

to_gpy()

Returns GPy version of this kernel.

update_gradient(*grad*)

Parameters **grad** – a one-dimensional array, representing the gradient vector $\nabla_{\boldsymbol{\theta}} L$ for the likelihood with respect to this kernel's parameters, in the same order of parameters as the row order returned by *kernel_gradient()*.

runlmc.kern.std_periodic module

class runlmc.kern.std_periodic.**StdPeriodic**(*inv_lengthscale=1*, *period=1*, *name='std_periodic'*, *active_dims=None*)

Bases: *runlmc.kern.stationary_kern.StationaryKern*

This class defines the standard periodic kernel k.

$$k(r) = \exp\left(\frac{-\gamma}{2}\sin^2\frac{\pi r}{T}\right)$$

Parameters

- **inv_lengthscale** – γ, above.
- **period** – T, above.
- **name** –
- **active_dims** – see *runlmc.kern.stationary_kern.StationaryKern* for details.

from_dist(*dists*)

Parameters **dists** – N-size numpy array of positive (Euclidean) distances.

Returns kernel value at each of the given distances

kernel_gradient(*dists*)

Let this kernel be parameterized by some parameters $\boldsymbol{\theta} \in \mathbb{R}^p$. For every $\theta_j \in \boldsymbol{\theta}$, at any given distance d, we can compute the derivative $\partial_{\theta_j} k(d)$. For the evaluation of this partial derivative at multiple places, $\mathbf{d}$, we call the vector of partial derivatives $\partial_{\theta_j} k(\mathbf{d})$.

Parameters **dists** – a one-dimensional array of distances corresponding to $\mathbf{d}$, above.

Returns An iterable whose j-th entry is $\partial_{\theta_j} k(\mathbf{d})$.

to_gpy()

Returns GPy version of this kernel.

update_gradient(*grad*)

Parameters **grad** – a one-dimensional array, representing the gradient vector $\nabla_{\boldsymbol{\theta}} L$ for the likelihood with respect to this kernel's parameters, in the same order of parameters as the row order returned by *kernel_gradient()*.

Module contents

1.1.3 runlmc.linalg package

Submodules

runlmc.linalg.block_diag module

class runlmc.linalg.block_diag.**BlockDiag**(*blocks*)
Bases: *runlmc.linalg.matrix.Matrix*

Creates a block diagonal matrix from constintuent, possibly rectangular internal ones. Note this is PSD if its constintuents are.

For constituents K_i, this matrix corresponds to the direct sum $K = \bigoplus_i K_i$.

Parameters **blocks** – blocks with which to construct the block diagonal matrix.

as_numpy()

Returns numpy matrix equivalent, as a 2D numpy.ndarray

matvec(x)
Multiply a vector **x** by this matrix, K, yielding $K\mathbf{x}$.

Parameters **x** – a one-dimensional numpy array of the same size as this matrix

Returns the matrix-vector product

runlmc.linalg.block_matrix module

class runlmc.linalg.block_matrix.**SymmSquareBlockMatrix**(*blocks*)
Bases: *runlmc.linalg.matrix.Matrix*

Creates a block matrix from a 2D array of matrices, which must all be square. The blocks array is assumed to be symmetric. :param blocks: a 2D array :raises ValueError: if the size is 0.

as_numpy()

Returns numpy matrix equivalent, as a 2D numpy.ndarray

matvec(x)
Multiply a vector **x** by this matrix, K, yielding $K\mathbf{x}$.

Parameters **x** – a one-dimensional numpy array of the same size as this matrix

Returns the matrix-vector product

upper_eig_bound()

runlmc.linalg.bttb module

class runlmc.linalg.bttb.**BTTB**(*top*, *sizes*)
Bases: *runlmc.linalg.matrix.Matrix*

Creates a class with a parsimonious representation of a symmetric, square block-Toeplitz matrix of Toeplitz blocks (BTTB).

The Toeplitz blocks requirement implies each block of this matrix is a Toeplitz matrix. See *runlmc.linalg.toeplitz.Toeplitz* for a description of how the top row of a Toeplitz matrix specifies the rest of it.

The block requirement specifies that each sub-block is further replicated in a Toeplitz manner. These matrices typically form when we have a stationary kernel applied to a multidimentional grid.

For instance, suppose we have a two dimensional grid of size $x \times y$.

For a fixed x_i, x_j, we have a one-dimensional square Toeplitz matrix of order y for pairs $(x_i, y_k), (x_j, y_m)$ and variable $k, m \in [y]$. It is easy to see how this matrix is identical for x_{i+1}, x_{j+1} if the x values are on a grid.

For higher dimensions, e.g., three, we can correspondingly construct block-Toeplitz matrices of BTTB blocks. Let's call these third order BTTB matrices (with first order BTTB being Toeplitz). This class generalizes all BTTB orders.

Fix a P-dimensional grid of points U with points $\{\mathbf{z}_{i_1 i_2 \cdots i_P}\}$ where the multidimensional index $i_1 i_2 \cdots i_P$ is flattened with P the fastest-changing dimension. Since we are on a grid, any point has an explicit form for fixed, positive Δ_i and standard basis vectors $\mathbf{e}_i$:

$$\mathbf{z}_{i_1 i_2 \cdots i_P} = \mathbf{z_0} + \sum_{p=1}^{P} \Delta_p i_p \mathbf{e}_p$$

Assuming we have grid size n_p along the p-th dimension, with $N = \prod_p n_p$ the total grid size, we thus assume to have a stationary kernel k evaluated at distances $k(\mathbf{z}'_j - \mathbf{z}'_0)$ which make up element j of the parameter *top*, with the straightforward index-flattening conversion:

$$\mathbf{z}'_j = \mathbf{z}_{i_1 i_2 \cdots i_P}\big|_{i_p = j \bmod n'_p}; \qquad n'_p = \prod_{p'=p}^{P} n_{p'}$$

We can still meaningfully define the BTTB without the context of the kernel, with *top* the flattened version of the nested *len(sizes)*-order BTTB matrix first row. In other words, we have the symmetric Toeplitz extension of N/n_P Toeplitz matrices.

For details, see Fast multiplication of a recursive block Toeplitz matrix by a vector and its application by David Lee 1986.

Parameters

- **top** – 1-dimensional numpy array, used as the underlying storage, which represents the first row t_{1j}. Should be castable to a float64.
- **sizes** – array of n_p.

Raises **ValueError** – if *top* or *shape* aren't of the right shape or are empty.

as_numpy()

Returns numpy matrix equivalent, as a 2D numpy.ndarray

matvec(x)

Multiply a vector **x** by this matrix, K, yielding $K\mathbf{x}$.

Parameters **x** – a one-dimensional numpy array of the same size as this matrix

Returns the matrix-vector product

runlmc.linalg.composition module

class runlmc.linalg.composition.**Composition**(*mats*)

Bases: *runlmc.linalg.matrix.Matrix*

matmat(x)

Multiply a matrix X by this matrix, K, yielding KX. By default, this just repeatedly calls *matvec()*.

Parameters **X** – a (possibly rectangular) dense matrix.

Returns the matrix-matrix product

matvec(x)

Multiply a vector **x** by this matrix, K, yielding $K\mathbf{x}$.

Parameters **x** – a one-dimensional numpy array of the same size as this matrix

Returns the matrix-vector product

runlmc.linalg.diag module

class runlmc.linalg.diag.**Diag**(*v*)
Bases: *runlmc.linalg.matrix.Matrix*

Creates a diagonal matrix. :param v: main diagaonal :raises ValueError: if v is not a non-empty vector

as_numpy()

Returns numpy matrix equivalent, as a 2D numpy.ndarray

matmat(*x*)
Multiply a matrix X by this matrix, K, yielding KX. By default, this just repeatedly calls *matvec()*.

Parameters **X** – a (possibly rectangular) dense matrix.

Returns the matrix-matrix product

matvec(*x*)
Multiply a vector **x** by this matrix, K, yielding $K\mathbf{x}$.

Parameters **x** – a one-dimensional numpy array of the same size as this matrix

Returns the matrix-vector product

upper_eig_bound()

runlmc.linalg.identity module

class runlmc.linalg.identity.**Identity**(*n*)
Bases: *runlmc.linalg.matrix.Matrix*

as_numpy()

Returns numpy matrix equivalent, as a 2D numpy.ndarray

matmat(*x*)
Multiply a matrix X by this matrix, K, yielding KX. By default, this just repeatedly calls *matvec()*.

Parameters **X** – a (possibly rectangular) dense matrix.

Returns the matrix-matrix product

matvec(*x*)
Multiply a vector **x** by this matrix, K, yielding $K\mathbf{x}$.

Parameters **x** – a one-dimensional numpy array of the same size as this matrix

Returns the matrix-vector product

upper_eig_bound()

runlmc.linalg.kronecker module

class runlmc.linalg.kronecker.**Kronecker**(*A*, *B*)
Bases: *runlmc.linalg.matrix.Matrix*

Creates a class with a parsimonious representation of a Kronecker product of two *runlmc.linalg.matrix.Matrix* instances. For the Kronecker matrix $K = A \otimes B$, the ij-th block entry is $a_{ij}B$.

K is PSD if A, B are.

The implementation is based off of Gilboa, Saatçi, and Cunningham (2015).

Creates a *Kronecker* matrix.

Parameters

- **A** – the first matrix
- **B** – the second matrix

as_numpy()

Returns numpy matrix equivalent, as a 2D numpy.ndarray

matvec(x)

Multiply a vector **x** by this matrix, K, yielding $K\mathbf{x}$.

Parameters **x** – a one-dimensional numpy array of the same size as this matrix

Returns the matrix-vector product

upper_eig_bound()

runlmc.linalg.matrix module

class runlmc.linalg.matrix.**Matrix**(n, m)

Bases: object

An abstract class defining the interface for the necessary sparse matrix operations.

All matrices are assumed real.

Parameters

- **n** – number of rows in this matrix
- **m** – number of columns in this matrix

Raises **ValueError** – if *n < 1 or m < 1*

as_linear_operator()

Returns this matrix as a scipy.sparse.linalg.LinearOperator

as_numpy()

Returns numpy matrix equivalent, as a 2D numpy.ndarray

is_square()

matmat(X)

Multiply a matrix X by this matrix, K, yielding KX. By default, this just repeatedly calls *matvec()*.

Parameters **X** – a (possibly rectangular) dense matrix.

Returns the matrix-matrix product

matvec(x)

Multiply a vector **x** by this matrix, K, yielding $K\mathbf{x}$.

Parameters **x** – a one-dimensional numpy array of the same size as this matrix

Returns the matrix-vector product

static wrap(*shape*, *mvm*)

runlmc.linalg.numpy_matrix module

class runlmc.linalg.numpy_matrix.**NumpyMatrix**(*nparr*)
Bases: *runlmc.linalg.matrix.Matrix*

Adapter to `Matrix` with `numpy` arrays.

Creates a *NumpyMatrix* matrix.

Parameters **nparr** – 2-dimensional `numpy` array

Raises **ValueError** – if *nparr* isn't 2D

as_numpy()

Returns numpy matrix equivalent, as a 2D `numpy.ndarray`

matmat(*x*)
Multiply a matrix X by this matrix, K, yielding KX. By default, this just repeatedly calls *matvec()*.

Parameters **X** – a (possibly rectangular) dense matrix.

Returns the matrix-matrix product

matvec(*x*)
Multiply a vector **x** by this matrix, K, yielding $K\mathbf{x}$.

Parameters **x** – a one-dimensional numpy array of the same size as this matrix

Returns the matrix-vector product

runlmc.linalg.shur module

runlmc.linalg.shur.**shur**(*top*)

runlmc.linalg.sum_matrix module

class runlmc.linalg.sum_matrix.**SumMatrix**(*Ks*)
Bases: *runlmc.linalg.matrix.Matrix*

The sum matrix represents a sum of other possibly sparse `runlmc.linalg.SymmetricMatrix` instances A_i, taking on the meaning $\sum_i A_i$.

Parameters **Ks** – decomposable matrices to sum

Raises **ValueError** – If *Ks* is empty

as_numpy()

Returns numpy matrix equivalent, as a 2D `numpy.ndarray`

matvec(*x*)
Multiply a vector **x** by this matrix, K, yielding $K\mathbf{x}$.

Parameters **x** – a one-dimensional numpy array of the same size as this matrix

Returns the matrix-vector product

upper_eig_bound()

runlmc.linalg.toeplitz module

class runlmc.linalg.toeplitz.**Toeplitz**(*top*)
Bases: *runlmc.linalg.matrix.Matrix*

Creates a class with a parsimonious representation of a symmetric, square Toeplitz matrix; that is, a matrix T with entries T_{ij} which for all i, j and $i' = i + 1, j' = j + 1$ satisfy:

$$t_{ij} = t_{i'j'}$$

Parameters **top** – 1-dimensional numpy array, used as the underlying storage, which represents the first row t_{1j}. Should be castable to a float64.

Raises **ValueError** – if *top* isn't of the right shape or is empty.

as_numpy()

Returns numpy matrix equivalent, as a 2D numpy.ndarray

matvec(*x*)
Multiply a vector **x** by this matrix, K, yielding $K\mathbf{x}$.

Parameters **x** – a one-dimensional numpy array of the same size as this matrix

Returns the matrix-vector product

upper_eig_bound()

Module contents

1.1.4 runlmc.lmc package

Submodules

runlmc.lmc.derivative module

class runlmc.lmc.derivative.**Derivative**
Bases: object

d_logdet_K(*dKdt*)

d_normal_quadratic(*dKdt*)

derivative(*dKdt*)

runlmc.lmc.exact_deriv module

class runlmc.lmc.exact_deriv.**ExactDeriv**(*L*, *y*)
Bases: *runlmc.lmc.derivative.Derivative*

d_logdet_K(*dKdt*)

d_normal_quadratic(*dKdt*)

runlmc.lmc.functional_kernel module

class runlmc.lmc.functional_kernel.**FunctionalKernel**(*D=None*, *lmc_kernels=None*, *lmc_ranks=None*, *slfm_kernels=None*, *indep_gp=None*, *indep_gp_index=None*, *name='kern'*)

Bases: *runlmc.parameterization.parameterized.Parameterized*

An LMC kernel can be specified by the number of latent GP kernels it contains. Recall a full LMC kernel defines the similarity between two inputs $\mathbf{x}_i, \mathbf{x}_j$ belonging to two outputs a, b, respectively, as follows (noise not included)

$$K((\mathbf{x}_i, a), (\mathbf{x}_j, b)) = \sum_{q=1}^{Q} B_{ab}^{(q)} k_q(\mathbf{x}_i, \mathbf{x}_j)$$

If we enumerate all inputs across all our D outputs $\{z_j\}_j = \{(\mathbf{x}_i, a) | a \in [D]\}$, then the complete LMC kernel evaluated as single matrix over an entire multi-output dataset X gives $K_{X,X} \in \mathbb{R}^{n \times n}$, with mn-th entry $(K_{X,X})_{mn} = K(z_m, z_n)$.

Since we can perform certain optimizations if $B^{(q)}$ contains a single nonzero diagonal entry or is of single rank. We refer to this as the coregionalization matrix for stationary subkernel k_q.

FunctionalKernel provides a convenient wrapper for specifying such kernels, which should all be instances of `runlmc.kern.stationary_kernel.StationaryKern`. This class is not tied to any data X but represents the K function, not matrix, above. This class is, however, tied to parameters. Especially important is the dichotomy with *runlmc.lmc.likelihood.LMCLikelihood*, which is a fixed evaluation of a *FunctionalKernel* with a fixed set of parameters on fixed data.

After a successful initialization, we have *Q == len(lmc_kernels) + len(slfm_kernels) + len(indep_gp)* and *len(indep_gp) == D*. Each $A_q, \boldsymbol{\kappa}_q$, becomes this model, with name *a<q>*, where *<q>* is replaced with a specific number.

Before use, input dimension should be specified with *set_input_dim()*. This is usually done automatically by the model, such as *runlmc.models.interpolated_llgp.InterpolatedLLGP*.

Parameters

- **D** – number of outputs
- **lmc_kernels** – a list of kernels for which the corresponding coregionalization matrix has full rank
- **lmc_ranks** – a list of integers of the same length as *lmc_kernels* each with value r_q at least 1 which specify that the coregionalization matrix for the corresponding kernel k_q in the *lmc_kernels* list can be decomposed as $B^{(q)} = A_q A_q^\top + \text{diag}\, \boldsymbol{\kappa}_q$, with A_q of rank r_q.
- **slfm_kernels** – an SLFM kernel restricts its coregionalization matrix to a single rank $A_q A_q^\top$
- **indep_gp** – indpedent GPs for each output i, with associated coregionalization matrices $\mathbf{e}_i \mathbf{e}_i^\top$.
- **indep_gp_index** – should be the same length as *indep_gp*, and specifies which output the kernel in the *indep_gp* list in the same place as an index is associated with. Defaults to *range(len(indep_gp))*.
- **name** – `paramz` name for this kernel

Raises **ValueError** – if any of the parameters don't meet the above requirements, or *D,Q* are unspecified, 0, or inconsistent.

Variables

- **Q** – *Q*, subkernel count including SLFM and indpendent kernels
- **D** – *D*, output dimension
- **num_lmc** – number of LMC kernels (a dictionary, where the key is the active dimensions and the value is the number of LMC kernels for that set of active dimensions)
- **num_slfm** – number of SLFM kernels, as *num_lmc*
- **num_indep** – number of independent GP kernels, as *num_lmc*
- **active_dims** – a dictionary whose keys are the subsets of the full input dimension set *{1, . . . , P}*. Only defined after *set_input_dim()* has been called.

Q

coreg_diags

coreg_mats(*active_dim=None*)

coreg_vecs

eval_kernel_gradients(*dists*)
Computes the list of grad k_q applied to each distance in *dists*, where *dists* should be a dict of *active_dim*-keyed distances.

eval_kernels(*dists*)
Computes the array of k_q applied to each distance in *dists*, where *dists* should be a dict of *active_dim*-keyed distances.

eval_kernels_fixed_dim(*dists*, *active_dim*)
Computes the array of k_q applied to each distance in *dists*, where only kernels with the passed-in active dimensions are evaluated.

filter_non_indep_idxs(*idxs*)
Return only the kernel indices associated with coregionalized (non-independent) kernels

get_active_dims(*q*)
Returns the active dimensions for the kernel with index *q*

noise

set_input_dim(*P*)
Set the input dimension for the kernel.

total_rank(*active_dim*)
Total (added) coregionalization rank for all B_q matrices

update_gradient(*grads*)
Update the gradients of parameters in the functional kernel with respect to those calculated by a concrete *LMCLikelihood* given data.

runlmc.lmc.grid_kernel module

class runlmc.lmc.grid_kernel.**GridKernel**(*functional_kernel*, *grid_dists*, *interpolant*, *interpolantT*, *ktype*, *active_dim*)

Bases: *runlmc.linalg.matrix.Matrix*

matvec(*x*)
Multiply a vector **x** by this matrix, K, yielding $K\mathbf{x}$.

Parameters **x** – a one-dimensional numpy array of the same size as this matrix

Returns the matrix-vector product

runlmc.lmc.grid_kernel.**gen_grid_kernel**(*fk*, *grid_dists*, *interpolants*, *lens_per_output*)

runlmc.lmc.likelihood module

class runlmc.lmc.likelihood.**ApproxLMCLikelihood**(*functional_kernel*, *grid_kern*, *grid_dists*, *interpolants*, *Ys*, *deriv*)
Bases: *runlmc.lmc.likelihood.LMCLikelihood*

alpha()

class runlmc.lmc.likelihood.**ExactLMCLikelihood**(*functional_kernel*, *Xs*, *Ys*)
Bases: *runlmc.lmc.likelihood.LMCLikelihood*

alpha()

static kernel_from_indices(*Xs*, *Zs*, *functional_kernel*)
Computes the dense, exact kernel matrix for an LMC kernel specified by *functional_kernel*. The kernel matrix that is computed is relative to the kernel application to pairs from the Cartesian product *Xs* and *Zs*.

class runlmc.lmc.likelihood.**LMCLikelihood**(*functional_kernel*, *Ys*)
Bases: object

Separate hyperparameter-based likelihood differentiation from the model class for separation of concerns. Different sub-classes may implement the below methods differently, with different asymptotic performance properties.

alpha()

coreg_diags_gradients()

coreg_vec_gradients()

kernel_gradients()

noise_gradient()

runlmc.lmc.metrics module

class runlmc.lmc.metrics.**Metrics**
Bases: object

runlmc.lmc.stochastic_deriv module

class runlmc.lmc.stochastic_deriv.**StochasticDeriv**(*alpha*, *rs*, *inv_rs*, *n_it*)
Bases: *runlmc.lmc.derivative.Derivative*

Given the inverse of random binary vectors *inv_rs* with respect to some kernel K and the similar inverse *alpha* of observations $K^{-1}y$, this class produces the derivatives of K with respect to its hyperparameters.

d_logdet_K(*dKdt*)

d_normal_quadratic(*dKdt*)

class runlmc.lmc.stochastic_deriv.**StochasticDerivService**(*metrics*, *pool*, *n_it*, *tol*)
Bases: object

This service generates runlmc.lmc.Derivative instances with pre-specified configurations for recording metrics or using multiprocessing, which enables decoupling of the math from the systems in the GP logic.

Parameters

- **metrics** – a *runlmc.lmc.metrics.Metrics* instance or *None* (if no metrics are to be recorded)
- **pool** – pool for parallel processing
- **n_it** – iterations to use in stochastic trace approximation
- **tol** – tolerance in inversion routine

Variables ***metrics*** – the metrics instance used by this class

generate(*K*, *y*)

Module contents

1.1.5 runlmc.mean package

Submodules

runlmc.mean.constant module

class runlmc.mean.constant.**Constant**(*input_dim*, *output_dim*, *c0=None*, *name='constant'*)
Bases: *runlmc.mean.mean_function.MeanFunction*

The constant mapping (constant for each output).

This mean function is not useful with normalization activated.

It may be useful if you would like to have no normalization *and* want to impose priors on the mean adjustment.

Parameters

- **input_dim** –
- **output_dim** –
- **c0** – optional vector of length output_dim for the initial offsets that the constant takes on.

f(*Xs*)
Evaluation of the mean function f.

Returns

mean_gradient(*Xs*)
Let this mean be parameterized by some parameters $\boldsymbol{\theta} \in \mathbb{R}^p$. For every $\theta_j \in \boldsymbol{\theta}$, at each input point $\mathbf{x}^{(i)}$ (for a certain output index i), we can compute the derivative $\partial_{\theta_j} f(\mathbf{x}^{(i)})$. For the evaluation of this partial derivative at multiple places, $\mathbf{X}$, we call the list of vectors of partial derivatives $\partial_{\theta_j} f(\mathbf{X})$ (a list with one vector per output index).

Parameters **Xs** – inputs to evaluate at

Returns A list of parameter gradients; the j-th entry of this list is $\partial_{\theta_j} f(\mathbf{X})$. Each $\partial_{\theta_j} f(\mathbf{X})$ in turn is another list with one entry per output i; the i-th entry is a one-dimensional numpy

array with k-th entry the derivative of the j-th mean parameter at the k-th input for the i-th output, $\partial_{\theta_j} f(\mathbf{x}_k^{(i)})$.

update_gradient(*grad*)

Parameters **grad** – a one-dimensional array, representing the gradient vector $\nabla_{\boldsymbol{\theta}} L$ for the likelihood with respect to this mean's parameters, in the same order of parameters as the row order returned by *mean_gradient()*.

runlmc.mean.mean_function module

class runlmc.mean.mean_function.**MeanFunction**(*input_dim*, *output_dim*, *name='mapping'*)

Bases: *runlmc.parameterization.parameterized.Parameterized*

Base class for mean functions in multi-output regressions.

f(*Xs*)

Evaluation of the mean function f.

Returns

mean_gradient(*Xs*)

Let this mean be parameterized by some parameters $\boldsymbol{\theta} \in \mathbb{R}^p$. For every $\theta_j \in \boldsymbol{\theta}$, at each input point $\mathbf{x}^{(i)}$ (for a certain output index i), we can compute the derivative $\partial_{\theta_j} f(\mathbf{x}^{(i)})$. For the evaluation of this partial derivative at multiple places, $\mathbf{X}$, we call the list of vectors of partial derivatives $\partial_{\theta_j} f(\mathbf{X})$ (a list with one vector per output index).

Parameters **Xs** – inputs to evaluate at

Returns A list of parameter gradients; the j-th entry of this list is $\partial_{\theta_j} f(\mathbf{X})$. Each $\partial_{\theta_j} f(\mathbf{X})$ in turn is another list with one entry per output i; the i-th entry is a one-dimensional numpy array with k-th entry the derivative of the j-th mean parameter at the k-th input for the i-th output, $\partial_{\theta_j} f(\mathbf{x}_k^{(i)})$.

update_gradient(*grad*)

Parameters **grad** – a one-dimensional array, representing the gradient vector $\nabla_{\boldsymbol{\theta}} L$ for the likelihood with respect to this mean's parameters, in the same order of parameters as the row order returned by *mean_gradient()*.

runlmc.mean.zero module

class runlmc.mean.zero.**Zero**(*input_dim*, *output_dim*, *name='zero'*)

Bases: *runlmc.mean.mean_function.MeanFunction*

The zero mapping. Note that leaving the *mean_function* parameter as none in all of the models does the same job.

f(*Xs*)

Evaluation of the mean function f.

Returns

mean_gradient(*Xs*)

Let this mean be parameterized by some parameters $\boldsymbol{\theta} \in \mathbb{R}^p$. For every $\theta_j \in \boldsymbol{\theta}$, at each input point $\mathbf{x}^{(i)}$ (for a certain output index i), we can compute the derivative $\partial_{\theta_j} f(\mathbf{x}^{(i)})$. For the evaluation of this partial derivative at multiple places, $\mathbf{X}$, we call the list of vectors of partial derivatives $\partial_{\theta_j} f(\mathbf{X})$ (a list with one vector per output index).

Parameters **Xs** – inputs to evaluate at

Returns A list of parameter gradients; the j-th entry of this list is $\partial_{\theta_j} f(\mathbf{X})$. Each $\partial_{\theta_j} f(\mathbf{X})$ in turn is another list with one entry per output i; the i-th entry is a one-dimensional numpy array with k-th entry the derivative of the j-th mean parameter at the k-th input for the i-th output, $\partial_{\theta_j} f(\mathbf{x}_k^{(i)})$.

update_gradient(*grad*)

Parameters **grad** – a one-dimensional array, representing the gradient vector $\nabla_{\boldsymbol{\theta}} L$ for the likelihood with respect to this mean's parameters, in the same order of parameters as the row order returned by *mean_gradient()*.

Module contents

1.1.6 runlmc.models package

Submodules

runlmc.models.gpy_lmc module

class runlmc.models.gpy_lmc.**GPyLMC**(*Xs*, *Ys*, *kernels*, *ranks*, *name='GPyLMC'*, *sparse=0*)

Bases: *runlmc.models.multigp.MultiGP*

This wraps GPy for the Gaussian Process model for multioutput regression under a Linear Model of Coregionalization.

This performs the inversion-based cubic-time algorithm.

Uses the Gaussian likelihood. See *runlmc.lmc.functional_kernel* for the explicit LMC formula.

The DTCVAR algorithm (the *sparse* parameter) is based on Efficient Multioutput Gaussian Processes through Variational Inducing Kernels by Álvarez et al. 2010.

Parameters

- **Xs** – input observations, should be a list of numpy arrays, where each numpy array is a design matrix for the inputs to output i. If the i-th input has n_i data points, then this matrix can be n_i or $n_i \times P$ shape for input dimension P, with the former re-interpreted as $P = 1$.
- **Ys** – output observations, this must be a list of one-dimensional numpy arrays, matching up with the number of rows in *Xs*.
- **kernels** – a list of (stationary) kernels which constitute the terms of the LMC sums prior to coregionalization.
- **ranks** (*list of integer*) – list of ranks for coregionalization factors
- **name** (*string*) – model name
- **sparse** – an integer. If 0, uses `GPy.models.GPCoregionalizedRegression`, the typical cholesky algorithm. If >0, then this determines the number of inducing points used by the DTCVAR algorithm in use `GPy.models.SparseGPCoregionalizedRegression`

log_likelihood()

The log marginal likelihood of the model, $p(\mathbf{y})$, this is the objective function of the model being optimised

optimize(***kwargs*)
Optimize the model using *log_likelihood()* with a gradient descent method that involves the priors.

kwargs are passed to the optimizer. See parameters for handled keywords.

Parameters **optimizer** – A `paramz.optimization.Optimizer`. Pre-built ones available in *runlmc.models.optimization*.

parameters_changed()
This method is called automatically when linked parameters change, which the may during the optimization process.

Classes should update their posterior information, log likelihood, and gradients when this happens, such that `_raw_predict()`, *log_likelihood()*, and `gradient()` are consistent with the new parameters.

predict(*Xs*)
Predict the functions at the new points *Xs*.

Parameters **Xs** – The points at which to make a prediction for each output. Should be empty if no output desired for a certain index. This is a `list` of `numpy.ndarray` for each output *d*, one-dimensional of size *n_d* each. Length of *Xs* should be equal to the number of outputs `output_dim`.

Returns

(mean, var):

***mean*: posterior mean, a list of length `output_dim` with** one-dimensional numpy arrays of length *n_d* at index *d*.

var: posterior variance, corresponding to each mean entry.

predict_quantiles(*Xs*, *quantiles=(2.5, 97.5)*)
Identical to predict, but returns quantiles instead of mean and variance according to the Gaussian likelihood.

Parameters **quantiles** (*tuple of doubles*) – tuple of quantiles, default is (2.5, 97.5), which is the 95% interval; shouldn't be 0 or 100

Returns list of quantiles for each output's input, as a numpy array, 2-D, the first axis corresponding to the input index and the second to the quantile index.

runlmc.models.interpolated_llgp module

class runlmc.models.interpolated_llgp.**InterpolatedLLGP**(*Xs*, *Ys*, *normalize=True*, *lo=None*, *hi=None*, *m=None*, *name='lmc'*, *metrics=False*, *prediction='on-the-fly'*, *max_procs=None*, *trace_iterations=15*, *tolerance=0.0001*, *functional_kernel=None*)
Bases: *runlmc.models.multigp.MultiGP*

The main class of this package, *InterpolatedLLGP* implements linearithmic Gaussian Process learning in the multi-output case. See the paper on arxiv.

Upon construction, this class assumes ownership of its parameters and does not account for changes in their values.

For a dataset of inputs *Xs* across multiple outputs *Ys*, let X refer to the concatenation of *Xs*. According to the functional specification of the LMC kernel by *functional_kernel* (see documentation in `runlmc.lmc.functional_kernel.FunctionalKernel`), we can create the covariance matrix for a multi-output GP model applied to all pairs of X, resulting in $K_{X,X}$.

The point of this class is to vary hyperparameters of K, the *FunctionalKernel* given by *functional_kernel*, until the model log likelihood is as large as possible.

This class uses the SKI approximation to do this efficiently, which shares a single grid U as the input array for all the outputs. Then, $K_{X,X}$ is interpolated from the approximation kernel K_{SKI}, as directed in *Thoughts on Massively Scalable Gaussian Processes* by Wilson, Dann, and Nickisch. This is done with sparse interpolation matrices W.

$$K_{X,X} \approx K_{\text{SKI}} = W K_{U,U} W^\top + \boldsymbol{\epsilon} I$$

Above, $K_{U,U}$ is a structured kernel over a grid U. This grid is specified by *lo,hi,m*.

The functionality for the various prediction modes is summarized below.

- *'on-the-fly'* - Use matrix-free inversion to compute the covariance for the entire set of points on which we're predicting. This means that variance prediction take $O(n \log n)$ time per test point, where *Xs* has n datapoints total. This should be preferred for small test sets.
- *'precompute'* - Compute an auxiliary predictive variance matrix for the grid points, but then cheaply re-use that work for prediction. This is an up-front $O(n^2 \log n)$ payment for $O(1)$ predictive variance afterwards per test point. This is not available if using split kernels (i.e., different active dimensions for different kernels).
- *'exact'* - Use the exact cholesky-based algorithm (not matrix free), $O(n^3)$ runtime up-front and then $O(n^2)$ per query.

Note *'on-the-fly', 'precompute'* can be parallelized by the number of test points and training points, respectively.

Parameters

- **Xs** – input observations, should be a list of numpy arrays, where each numpy array is a design matrix for the inputs to output i. If the i-th input has n_i data points, then this matrix can be n_i or $n_i \times P$ shape for input dimension P, with the former re-interpreted as $P = 1$.
- **Ys** – output observations, this must be a list of one-dimensional numpy arrays, matching up with the number of rows in *Xs*.
- **normalize** – optional normalization for outputs *Ys*. Prediction will be un-normalized.
- **lo** – lexicographically smallest point in inducing point grid used (by default, a bit less than the minimum of input). For multidimensional inputs this should be a vector.
- **hi** – lexicographically largest point in inducing point grid used (by default, a bit more than the maximum of input). For multidimensional inputs this should be a vector.
- **m** – number of inducing points to use. For multidimensional inputs this should be a vector indicating how many grid points there should be along each dimension. The total number of points used is then *np.prod(m)*. By default, *m* is a constant array of dimension P, the input dimension, of size equal to the average input sequence length.
- **name** (*str*) –
- **metrics** – whether to record optimization metrics during optimization (runs exact solution alongside this one, may be slow).
- **prediction** – one of *'matrix-free'*, *'on-the-fly'*, *'precompute'*, *'exact'*, *'sample'*.
- **max_procs** – maximum number of processes to use for parallelism, defaults to cpu count.

- **functional_kernel** – a *runlmc.lmc.functional_kernel.FunctionalKernel* determining K.
- **trace_iterations** – number of iterations to be used in approximate trace algorithm.

Raises `ValueError` if *Xs* and *Ys* lengths do not match.

Raises `ValueError` if normalization if any *Ys* have no variance or values in *Xs* have multiple identical values.

Variables ***metrics*** – the *runlmc.lmc.metrics.Metrics* instance associated with the model

EVAL_NORM = inf

K()

Warning: This generates the entire kernel, a quadratic operation in memory and time.

Returns K_{SKI}, the approximation of the exact kernel.

log_det_K()

Returns an upper bound of the approximate log determinant, uses K_{SKI} to find an approximate upper bound for $\log\det K_{extexact}$

log_likelihood()
The log marginal likelihood of the model, $p(\mathbf{y})$, this is the objective function of the model being optimised

normal_quadratic()
If the flattened (Stacked)outputs are written as $\mathbf{y}$, this returns $\mathbf{y}^\top K_{\text{SKI}}^{-1}\mathbf{y}$.

Returns the normal quadratic term for the current outputs *Ys*.

optimize(***kwargs*)
Optimize the model using *log_likelihood()* with a gradient descent method that involves the priors.

kwargs are passed to the optimizer. See parameters for handled keywords.

Parameters **optimizer** – A `paramz.optimization.Optimizer`. Pre-built ones available in *runlmc.models.optimization*.

parameters_changed()
This method is called automatically when linked parameters change, which the may during the optimization process.

Classes should update their posterior information, log likelihood, and gradients when this happens, such that `_raw_predict()`, *log_likelihood()*, and `gradient()` are consistent with the new parameters.

runlmc.models.multigp module

This module houses the base model that the package centers around: `runlmc.model.MultiGP`, the parent class for all multi-output GP models in this package.

class runlmc.models.multigp.**MultiGP**(*Xs*, *Ys*, *normalize=True*, *name='multigp'*)
Bases: *runlmc.parameterization.model.Model*

The generic GP model for multi-output regression. This handles common functionality between all models regarding input validation and high level parameter optimization routines.

This model assumes Gaussian noise.

This class shouldn't be instantiated directly.

Upon construction, this class assumes ownership of its parameters and does not account for changes in their values.

Parameters

- **Xs** – input observations, should be a list of numpy arrays, where each numpy array is a design matrix for the inputs to output i. If the i-th input has n_i data points, then this matrix can be n_i or $n_i \times P$ shape for input dimension P, with the former re-interpreted as $P = 1$.
- **Ys** – output observations, this must be a list of one-dimensional numpy arrays, matching up with the number of rows in *Xs*.
- **normalize** – optional normalization for outputs *Ys*. Prediction will be un-normalized.
- **name** (*str*) –

Raises `ValueError` if *Xs* and *Ys* lengths do not match.

Raises `ValueError` if normalization if any *Ys* have no variance or values in *Xs* have multiple identical values.

log_likelihood()
The log marginal likelihood of the model, $p(\mathbf{y})$, this is the objective function of the model being optimised

optimize(***kwargs*)
Optimize the model using *log_likelihood()* with a gradient descent method that involves the priors.

kwargs are passed to the optimizer. See parameters for handled keywords.

Parameters **optimizer** – A `paramz.optimization.Optimizer`. Pre-built ones available in *runlmc.models.optimization*.

parameters_changed()
This method is called automatically when linked parameters change, which the may during the optimization process.

Classes should update their posterior information, log likelihood, and gradients when this happens, such that `_raw_predict()`, *log_likelihood()*, and `gradient()` are consistent with the new parameters.

predict(*Xs*)
Predict the functions at the new points *Xs*.

Parameters **Xs** – The points at which to make a prediction for each output. Should be empty if no output desired for a certain index. This is a `list` of `numpy.ndarray` for each output *d*, one-dimensional of size *n_d* each. Length of *Xs* should be equal to the number of outputs `output_dim`.

Returns

(mean, var):

***mean*: posterior mean, a list of length output_dim with** one-dimensional numpy arrays of length *n_d* at index *d*.

var: posterior variance, corresponding to each mean entry.

predict_quantiles(*Xs*, *quantiles=(2.5, 97.5)*)
Identical to predict, but returns quantiles instead of mean and variance according to the Gaussian likelihood.

Parameters **quantiles** (*tuple of doubles*) – tuple of quantiles, default is (2.5, 97.5), which is the 95% interval; shouldn't be 0 or 100

Returns list of quantiles for each output's input, as a numpy array, 2-D, the first axis corresponding to the input index and the second to the quantile index.

runlmc.models.optimization module

class runlmc.models.optimization.**AdaDelta**(***kwargs*)
Bases: paramz.optimization.optimization.Optimizer

static noop()

opt(*x*, *f_fp=None*, *f=None*, *fp=None*)

Module contents

1.1.7 runlmc.parameterization package

Submodules

runlmc.parameterization.model module

This module defines a generic internal *Model* class, which handles the interface between this class and the paramz optimization layer.

class runlmc.parameterization.model.**Model**(*name*)
Bases: paramz.model.Model, runlmc.parameterization.priorizable._PriorizableNode

A *Model* provides a graphical model dependent on latent parameters, which it contains either explicitly (as attributes in derived classes of *Model*) or implicitly (as parameters implicitly linked to a model's explicit parameters).

Access to any parameter in this tree can be done by the name of those parameters. See the Parameterized documentation for details.

The parameters can be either Param`s or :class:`Parameterized. In fact, the tree of references from objects to their attributes, as represented in the Python garbage collector, matches identically with the graphical model that this *Model* represents (though the direction of the links is reversed).

The *Model* binds together likelihood values computed from the model without the priors (which is implemented by derived classes) with the priors. In other words, for observations $\mathbf{y}$, parameters θ dependent on priors ϕ, the user supplies $\log p(\mathbf{y}|\theta, \phi)$ as well as its derivative with respect to θ. This class automatically adds in the missing $\log p(\theta|\phi)$ term and its derivative.

log_likelihood()

Returns the log likelihood of the current model with respect to its current inputs and outputs and the current prior. This should NOT include the likelihood of the parameters given their priors. In other words, this value should be $\log p(\mathbf{y}|\theta, \phi)$

log_likelihood_with_prior()
Let the observations be $\mathbf{y}$, parameters be θ, and the prior ϕ.

$$\log p(\mathbf{y}|\phi) = \log p(\mathbf{y}|\theta, \phi) + \log p(\mathbf{y}|\theta, \phi)$$

Returns the overall log likelihood shown above.

log_prior()

Returns the log prior $\log p(\theta|\phi)$

objective_function()
The objective function for the given algorithm.

This function is the true objective, which wants to be minimized. Note that all parameters are already set and in place, so you just need to return the objective function here.

For probabilistic models this is the negative log_likelihood (including the MAP prior), so we return it here. If your model is not probabilistic, just return your objective to minimize here!

objective_function_gradients()
The gradients for the objective function for the given algorithm. The gradients are w.r.t. the *negative* objective function, as this framework works with *negative* log-likelihoods as a default.

You can find the gradient for the parameters in self.gradient at all times. This is the place, where gradients get stored for parameters.

This function is the true objective, which wants to be minimized. Note that all parameters are already set and in place, so you just need to return the gradient here.

For probabilistic models this is the gradient of the negative log_likelihood (including the MAP prior), so we return it here. If your model is not probabilistic, just return your *negative* gradient here!

runlmc.parameterization.param module

This module contains the *Param* class, used to keep track of optimization parameters.

Developers familiar with `paramz` can add their own parameters for custom kernels and priors.

This class differs from the corresponding `paramz` one because it is priorizable.

class runlmc.parameterization.param.**Param**(*name*, *input_array*, *default_constraint=None*, **a*, ***kw*)
Bases: `paramz.param.Param`, *runlmc.parameterization.priorizable.PriorizableLeaf*

A *Param* should be initialized and used just like a `paramz.param.Param`. It contains additional functionality for adding priors.

runlmc.parameterization.parameterized module

This module contains the *Parameterized* class.

It only exists for convenience, documentation, and to indicate which methods of the corresponding `paramz` class should be used.

class runlmc.parameterization.parameterized.**Parameterized**(*name=None*, *parameters=[]*)
Bases: `paramz.parameterized.Parameterized`, `runlmc.parameterization.priorizable._PriorizableNode`

A *Parameterized* class is responsible for keeping track of which parameters it is dependent on.

It does NOT manage updates for those parameters if they change during some optimization process. The runlmc.parametrization.Model should take care of that.

Say *m* is a handle to a parameterized class.

Printing parameters:

- *print(m)* prints a nice summary over all parameters
- *print(name)* prints details for param with name *name*
- *print(m[regexp])* prints details for all the parameters which match regexp
- *print(m['*'])* prints details for all parameters

Getting and setting parameters:

```
m.subparameterized1.subparameterized2.param[:] = 1
```

Handling of constraining, fixing and tieing parameters:

You can constrain parameters by calling the constrain on the param itself, e.g:

```
m.name[:,1].constrain_positive()
m.name[0].tie_to(m.name[1])
```

Fixing parameters will fix them to the value they are right now. If you change the parameters value, the param will be fixed to the new value!

If you want to operate on all parameters use *m['*']* to wildcard select all parameters and concatenate them. Printing *m['*']* will result in printing of all parameters in detail.

link_parameter(*param*, *index=None*)

Indicate that a class depends on a certain parameter, and therefore should include it in gradient computations, and change it during, e.g, model optimization.

Note: Unless you're creating your own parameters, you shouldn't need to call this.

Parameters

- **param** – the parameter to add.
- **[index]** – index of where to put parameters

Raises **TypeError** – if *param* is not the *runlmc* (priorizable) parameter.

runlmc.parameterization.priorizable module

This module contains internal classes having to do with adding priors and containing parameters that have priors.

Both _PriorizableNode and *PriorizableLeaf* shouldn't be used externally.

class runlmc.parameterization.priorizable.**PriorizableLeaf**(*name*, *a, **kw*)

Bases: runlmc.parameterization.priorizable._PriorizableNode

A *PriorizableLeaf* contains a prior, and, by virtue of being a _PriorizableNode, will automatically notify parents of a new prior being set.

set_prior(*prior*)
Set the prior for this object to prior.

Parameters **prior** (runlmc.parameterization.Prior) – prior set for this parameter

unset_prior()
Unset prior, if present.

runlmc.parameterization.priors module

This modules contains *Prior*, the base type for all priors available.

class runlmc.parameterization.priors.**Gamma**(*a*, *b*)
Bases: *runlmc.parameterization.priors.Prior*

domain = 'positive'

static from_EV(*E*, *V*)
Creates an instance of a Gamma Prior with prescribed statistics

Parameters

- **E** – expected value
- **V** – variance

lnpdf(*x*)

Parameters **x** – query float or numpy array (for multiple parameters with this same prior)

Returns the log density of the prior at (each) *x*

lnpdf_grad(*x*)

Parameters **x** – query float or numpy array (for multiple parameters with this same prior)

Returns the gradient of the log density of the prior at (each) *x*

class runlmc.parameterization.priors.**Gaussian**(*mu*, *var*)
Bases: *runlmc.parameterization.priors.Prior*

domain = 'real'

lnpdf(*x*)

Parameters **x** – query float or numpy array (for multiple parameters with this same prior)

Returns the log density of the prior at (each) *x*

lnpdf_grad(*x*)

Parameters **x** – query float or numpy array (for multiple parameters with this same prior)

Returns the gradient of the log density of the prior at (each) *x*

class runlmc.parameterization.priors.**HalfLaplace**(*b*)
Bases: object

domain = 'positive'

lnpdf(*x*)

lnpdf_grad(*_*)

class runlmc.parameterization.priors.**InverseGamma**(*a*, *b*)
Bases: *runlmc.parameterization.priors.Prior*

domain = 'positive'

lnpdf(*x*)

Parameters **x** – query float or numpy array (for multiple parameters with this same prior)

Returns the log density of the prior at (each) *x*

lnpdf_grad(*x*)

Parameters **x** – query float or numpy array (for multiple parameters with this same prior)

Returns the gradient of the log density of the prior at (each) *x*

class runlmc.parameterization.priors.**Prior**

Bases: object

Prior allows for incorporating a Bayesian prior in the first-order gradient-based optimization performed on the GP models.

Priors are placed over scalar values.

Prior objects are immutable.

Methods are intended to be vectorized over parameters with the same priors. In other words, mapping *lnpdf()* and *lnpdf_grad()* over each point individually should produce the same result as passing in a list of those points.

domain = None

Attribute domain Domain on which the prior is defined

lnpdf(*x*)

Parameters **x** – query float or numpy array (for multiple parameters with this same prior)

Returns the log density of the prior at (each) *x*

lnpdf_grad(*x*)

Parameters **x** – query float or numpy array (for multiple parameters with this same prior)

Returns the gradient of the log density of the prior at (each) *x*

Module contents

1.1.8 runlmc.util package

Submodules

runlmc.util.docs module

This module contains a documentation-inheritance utility decorator, *inherit_doc()*.

runlmc.util.docs.**inherit_doc**(*cls*)

From StackOverflow.

This will copy all the missing documentation for methods from the parent classes.

Parameters **cls** (*type*) – class to fix up.

Return type the fixed class.

runlmc.util.inline_pool module

class runlmc.util.inline_pool.**InlinePool**(*pool*)
Bases: object

Basic extension to a pool which supports no parallelism transparently. Takes ownership of the parameter pool.

Parameters **pool** – a multiprocessing.Pool or *None*

starmap(*f*, *ls*)

runlmc.util.normalizer module

class runlmc.util.normalizer.**Norm**
Bases: object

inverse_mean(*X*)
Project the normalized object X into space of Y

inverse_variance(*var*)

normalize(*Y*)
Project Y into normalized space

scale_by(*Y*)
Use data matrix Y as normalization space to work in.

scaled()
Whether this Norm object has been initialized.

runlmc.util.numpy_convenience module

Convenience functions for working with numpy arrays.

runlmc.util.numpy_convenience.**begin_end_indices**(*lens*)

runlmc.util.numpy_convenience.**cartesian_product**(**arrays*)

runlmc.util.numpy_convenience.**chunks**(*l*, *n*)
Yield successive n-sized chunks from l.

runlmc.util.numpy_convenience.**map_entries**(*f*, *nparr*)
Map a function over a numpy array.

Parameters

- **f** – single-parameter function over the same types
- **nparr** (*np.ndarray*) – arbitrary numpy array

Returns A numpy array with *f* evaluated on each element of the same shape.

runlmc.util.numpy_convenience.**search_descending**(*x*, *xs*, *inclusive*)

Parameters

- **x** – threshold
- **xs** – descending-ordered array to search
- **inclusive** – whether to include values of *x* in *xs*

Returns the largest index index *i* such that *xs[:i]* >= *x* if *inclusive* else *xs[:i]* > *x*.

Raises **ValueError** – if array is not weakly decreasing

runlmc.util.numpy_convenience.**smallest_eig**(*top*)

Parameters **top** – top row of Toeplitz matrix to get eigenvalues for

Returns the smallest eigenvalue

runlmc.util.numpy_convenience.**symm_2d_list_map**(*f*, *arr*, *D*, **args*, *dtype='object'*)
Symmetric map construction

runlmc.util.numpy_convenience.**tesselate**(*nparr*, *lenit*)
Create a ragged array by splitting *nparr* into contiguous segments of size determined by the length list *lenit*

Parameters

- **nparr** – array to split along axis 0.
- **lenit** – iterator of lengths to split into.

Returns A list of size equal to *lenit*'s iteration with *nparr*'s segments split into corresponding size chunks.

Raises **ValueError** – if the sum of lengths doesn't correspond to the array size.

runlmc.util.testing_utils module

The following methods are useful for generating various matrices for testing.

"PSDT" stands for positive semi-definite Toeplitz (and, implicitly, symmetric).

class runlmc.util.testing_utils.**BasicModel**(*dists*, *Y*, *kern*)
Bases: *runlmc.parameterization.model.Model*

log_likelihood()

Returns the log likelihood of the current model with respect to its

current inputs and outputs and the current prior. This should NOT include the likelihood of the parameters given their priors. In other words, this value should be $\log p(\mathbf{y}|\theta, \phi)$

parameters_changed()
This method gets called when parameters have changed. Another way of listening to param changes is to add self as a listener to the param, such that updates get passed through. See :py:function:paramz.param.Observable.add_observer

class runlmc.util.testing_utils.**RandomTest**(*methodName='runTest'*)
Bases: unittest.case.TestCase

This test case sets the random seed to be based on the time that the test is run.

If there is a *SEED* variable in the enviornment, then this is used as the seed.

Sets both random and numpy.random. Prints the seed to stdout before running each test case.

setUp()
Hook method for setting up the test fixture before exercising it.

class runlmc.util.testing_utils.**SingleGradOptimizer**(*lipschitz=1*)
Bases: paramz.optimization.optimization.Optimizer

opt(*x_init*, *f_fp=None*, *f=None*, *fp=None*)

runlmc.util.testing_utils.**check_np_lists**(*a*, *b*, *atol=1e-07*, *rtol=1e-07*)
Verifies that two lists of numpy arrays are all close. :param a: :param b:

runlmc.util.testing_utils.**error_context**(*s*)

runlmc.util.testing_utils.**exp_decr_toep**(*n*)

Returns top row of a PSDT matrix of size *n* with terms exponentially decreasing in distance from the main diagonal; the rate of which is randomly generated but at least e.

runlmc.util.testing_utils.**poor_cond_toep**(*n*)

Parameters **n** – size of output

Returns the top row of a randomly scaled PSDT matrix whose L^2 condition number scales exponentially with *n*

runlmc.util.testing_utils.**rand_pd**(*n*)

Returns a random *n* by *n* symmetric PSD matrix with positive entries

runlmc.util.testing_utils.**random_toep**(*n*)

Returns top row of a random PSDT matrix of size *n*.

runlmc.util.testing_utils.**run_main**(*f*, *help_str*)

A helper function which is re-used in setting up benchmarks for kernels that are shaped in a specific manner; namely, sums of Kronecker products of small dense and large Toeplitz matrices.

This function reads in command-line arguments and executes a simple benchmarking script.

Parameters

- **f** – benchmarking function to pass generated inputs with user-specified parameters to.
- **help_str** – a help-string to be printed when the user does not call a correct invocation of the program.

runlmc.util.testing_utils.**vectorize_inputs**(*f*)

Convert a multi-input vector function to accept matrices with each column corresponding to an input

Module contents

1.2 Module contents

CHAPTER 2

Index

- genindex

Python Module Index

r

Index

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z