gptools Documentation

Release 0.0

Mark Chilenski

September 24, 2013

CONTENTS

CHAPTER

ONE

OVERVIEW

`gptools` is a Python package that provides a convenient, powerful and extensible implementation of Gaussian process regression (GPR). Central to `gptool`'s implementation is support for derivatives and their variances. A number of kernels are provided to allow many types of data to be fit:

- `DiagonalNoiseKernel` implements homoscedastic noise. The noise is tied to a specific derivative order. This allows you to, for instance, have noise on your observations but have noiseless derivative constraints, or to have different noise levels for observations and derivatives. Note that you can also specify potentially heteroscedastic noise explicitly when adding data to the process.
- `SquaredExponentialKernel` implements the SE kernel which is infinitely differentiable.
- `MaternKernel` implements the entire Matern class of covariance functions, which are characterized by a hyperparameter ν. A process having the Matern kernel is only mean-square differentiable for derivative order $n < \nu$.
- `RationalQuadraticKernel` implements the rational quadratic kernel, which is a scale mixture over SE kernels.

In all cases, these kernels have been constructed in a way to allow inputs of arbitrary dimension. Each dimension has a length scale hyperparameter that can be separately optimized over or held fixed. Arbitrary derivatives with respect to each dimension can be taken, including computation of the covariance for those observations.

Other kernels can be implemented by extending the `Kernel` class. Furthermore, kernels may be added or multiplied together to yield a new, valid kernel.

CHAPTER

TWO

CONTENTS

2.1 gptools Package

2.1.1 gptools Package

gptools - Gaussian process regression with support for arbitrary derivatives

2.1.2 error_handling Module

Contains exceptions specific to the gptools package.

exception gptools.error_handling.**GPArgumentError**
Bases: exceptions.Exception

Exception class raised when an incorrect combination of keyword arguments is given.

2.1.3 gaussian_process Module

Provides the base GaussianProcess class.

class gptools.gaussian_process.**GaussianProcess**(*k*, *noise_k=None*, *X=None*, *y=None*, *err_y=0*)
Bases: object

Gaussian process.

If called with one argument, an untrained Gaussian process is constructed and training data must be added with the add_data() method. If called with the optional keywords, the values given are used as the training data. It is always possible to add additional training data with add_data().

Note that the attributes have no write protection, but you should always add data with add_data() to ensure internal consistency.

Parameters **k** : Kernel instance

Kernel instance corresponding to the desired noise-free covariance kernel of the Gaussian process. The noise is handled separately either through specification of *err_y*, or in a separate kernel. This allows noise-free predictions when needed.

noise_k : Kernel instance

Kernel instance corresponding to the noise portion of the desired covariance kernel of the Gaussian process. Note that you DO NOT need to specify this if the extent of the noise you want to represent is contained in *err_y* (or if your data are noiseless). Default value is None, which results in the `ZeroKernel` (noise specified elsewhere or not present).

NOTE :

The following are all passed to `add_data()`, refer to its docstring.

X : `Matrix` or other Array-like, (*M*, *N*), optional

M training input values of dimension *N*. Default value is None (no training data).

y : `Array` or other Array-like, (*M*,), optional

M training target values. Default value is None (no training data).

err_y : `Array` or other Array-like, (*M*,), optional

Error (given as standard deviation) in the *M* training target values. Default value is 0 (noiseless observations).

Raises **GPArgumentError** :

Gave *X* but not *y* (or vice versa).

ValueError :

Training data rejected by `add_data()`.

See Also:

`add_data` Used to process *X*, *y*, *err_y* and to add data to the process.

Attributes

k	`Kernel` instance	The non-noise portion of the covariance kernel.
noise_k	`Kernel` instance	The noise portion of the covariance kernel.
X	`Matrix`, (*M*, *N*)	The *M* training input values, each of which is of dimension *N*.
y	`Array`, (*M*,)	The *M* training target values.
err_y	`Array`, (*M*,)	The error in the *M* training input values.
n	`Matrix`, (*M*, *N*)	The orders of derivatives that each of the M training points represent, indicating the order of derivative with respect to each of the *N* dimensions.
K_up_to_date	bool	True if no data have been added since the last time the internal state was updated with a call to `compute_K_L_alpha_ll()`.
K	`Matrix`, (*M*, *M*)	Covariance matrix between all of the training inputs.
noise_K	`Matrix`, (*M*, *M*)	Noise portion of the covariance matrix between all of the training inputs. Only includes the noise from `noise_k`, not from `err_y`.
L	`Matrix`, (*M*, *M*)	Cholesky decomposition of the combined covariance matrix between all of the training inputs.
alpha	`Matrix`, (*M*, 1)	Solution to $K\alpha = y$.
ll	float	Log-likelihood of the data given the model.

num_dim

The number of dimensions of the input data.

Returns **num_dim: int** :

The number of dimensions of the input data as defined in the kernel.

add_data (*X*, *y*, *err_y=0*, *n=0*)

Add data to the training data set of the GaussianProcess instance.

Parameters **X** : `Matrix` or other Array-like, (*M*, *N*)

M training input values of dimension *N*.

y : `Array` or other Array-like, (*M*,)

M training target values.

err_y : `Array` or other Array-like (*M*,) or scalar float, optional

Non-negative values only. Error given as standard deviation) in the *M* training target values. If *err_y* is a scalar, the data set is taken to be homoscedastic (constant error). Otherwise, the length of *err_y* must equal the length of *y*. Default value is 0 (noiseless observations).

n : `Matrix` or other Array-like (*M*, *N*) or scalar float, optional

Non-negative integer values only. Degree of derivative for each training target. If *n* is a scalar it is taken to be the value for all points in *y*. Otherwise, the length of n must equal the length of *y*. Default value is 0 (observation of target value). If non-integer values are passed, they will be silently rounded.

Raises **ValueError** :

Bad shapes for any of the inputs, negative values for *err_y* or *n*.

compute_Kij(*Xi*, *Xj*, *ni*, *nj*, *noise=False*, *hyper_deriv=None*)

Compute covariance matrix between datasets *Xi* and *Xj*.

Specify the orders of derivatives at each location with the *ni*, *nj* arrays. The *include_noise* flag is passed to the covariance kernel to indicate whether noise is to be included (i.e., for evaluation of $K + \sigma I$ versus K_*).

If *Xj* is None, the symmetric matrix $K(X, X)$ is formed.

Note that type and dimension checking is NOT performed, as it is assumed the data are from inside the instance and have hence been sanitized by `add_data()`.

Parameters **Xi** : `Matrix`, (*M*, *N*)

M input values of dimension *N*.

Xj : `Matrix`, (*P*, *N*)

P input values of dimension *N*.

ni : `Array`, (*M*,), non-negative integers

M derivative orders with respect to the *Xi* coordinates.

nj : `Array`, (*P*,), non-negative integers

P derivative orders with respect to the *Xj* coordinates.

noise : bool, optional

If True, uses the noise kernel, otherwise uses the regular kernel. Default is False (use regular kernel).

hyper_deriv : None or non-negative int

Index of the hyperparameter to compute the first derivative with respect to. If None, no derivatives are taken. Default is None (no hyperparameter derivatives).

Returns **Kij** : `Matrix`, (*M*, *P*)

Covariance matrix between *Xi* and *Xj*.

compute_K_L_alpha_ll(*diag_factor=100.0*)

Compute *K*, *L*, *alpha* and log-likelihood according to the first part of Algorithm 2.1 in R&W.

Computes *K* and the noise portion of *K* using `compute_Kij()`, computes *L* using `scipy.linalg.cholesky()`, then computes *alpha* as *L.T\(L\y)*.

Only does the computation if `K_up_to_date` is False – otherwise leaves the existing values.

Parameters **diag_factor** : float, optional

Factor of `sys.float_info.epsilon` which is added to the diagonal of the total *K* matrix to improve the stability of the Cholesky decomposition. If you are having issues, try increasing this by a factor of 10 at a time. Default is 1e2.

update_hyperparameters(*new_params*, *return_jacobian=False*)

Update the kernel's hyperparameters to the new parameters.

This will call `compute_K_L_alpha_ll()` to update the state accordingly.

Parameters **new_params** : `Array` or other Array-like, length dictated by kernel

New parameters to use.

return_jacobian : bool, optional

If True, the return is (*ll*, *jac*). Otherwise, return is *ll* only and the execution is faster. Default is False (do not compute Jacobian).

Returns **-1*ll** : float

The updated log likelihood.

-1*jac : `Array`, length equal to the number of parameters

The derivative of *ll* with respect to each of the parameters, in order. Only computed and returned if *return_jacobian* is True.

optimize_hyperparameters(*method='SLSQP'*, *opt_kwargs={}*, *verbose=False*)

Optimize the hyperparameters by maximizing the log likelihood.

Leaves the `GaussianProcess` instance in the optimized state.

If `scipy.optimize.minimize()` is not available (i.e., if your `scipy` version is older than 0.11.0) then `fmin_slsqp()` is used independent of what you set for the *method* keyword.

Parameters **method** : str, optional

The method to pass to `scipy.optimize.minimize()`. Refer to that function's docstring for valid options. Default is 'SLSQP'. See note above about behavior with older versions of `scipy`.

opt_kwargs : dict, optional

Dictionary of extra keywords to pass to `scipy.optimize.minimize()`. Refer to that function's docstring for valid options. Note that if you use *jac* = True (i.e., optimization function returns Jacobian) you should also set *args* = (True,) to tell `update_hyperparameters()` to compute and return the Jacobian. Default is: {}.

verbose : bool, optional

Whether or not the output should be verbose. If True, the entire `Result` object from `scipy.optimize.minimize()` is printed. If False, status information is only printed if the *success* flag from `minimize()` is False. Default is False.

predict(*Xstar*, *n=0*, *noise=False*, *return_cov=True*)

Predict the mean and covariance at the inputs *Xstar*.

The order of the derivative is given by *n*. The keyword *noise* sets whether or not noise is included in the prediction.

Parameters **Xstar** : `Array` or other Array-like, (*M*, *N*)

M test input values of dimension *N*.

n : `Matrix` or other Array-like, (*M*, *N*) or scalar, non-negative int, optional

Order of derivative to predict (0 is the base quantity). If *n* is scalar, the value is used for all points in *Xstar*. If non-integer values are passed, they will be silently rounded. Default is 0 (return base quantity).

noise : bool, optional

Whether or not noise should be included in the covariance. Default is False (no noise in covariance).

return_cov : bool, optional

Set to True to compute and return the covariance matrix for the predictions, False to skip this step. Default is True (return tuple of (*mean*, *cov*)).

Returns **mean** : `Array`, (*M*,)

Predicted GP mean.

covariance : `Matrix`, (*M*, *M*)

Predicted covariance matrix, only returned if *return_cov* is True.

Raises **ValueError** :

If *n* is not consistent with the shape of *Xstar* or is not entirely composed of non-negative integers.

compute_ll_matrix(*bounds*, *num_pts*)

Compute the log likelihood over the (free) parameter space.

Parameters **bounds** : 2-tuple or list of 2-tuples with length equal to the number of free parameters

Bounds on the range to use for each of the parameters. If a single 2-tuple is given, it will be used for each of the parameters.

num_pts : int or list of ints with length equal to the number of free parameters

If a single int is given, it will be used for each of the parameters.

Returns **ll_vals** : `Array`

The log likelihood for each of the parameter possibilities.

param_vals [List of `Array`] The parameter values used.

draw_sample(*Xstar*, *n=0*, *noise=False*, *num_samp=1*, *rand_vars=None*, *rand_type='standard normal'*, *diag_factor=1000.0*)

Draw a sample evaluated at the given points *Xstar*.

Parameters **Xstar** : `Matrix` or other Array-like, (*M*, *N*)

M test input values of dimension *N*.

n : `Matrix` or other Array-like, (*M*, *N*) or scalar, non-negative int, optional

Derivative order to evaluate at. Default is 0 (evaluate value).

noise : bool, optional

Whether or not to include the noise components of the kernel in the sample. Default is False (no noise in samples).

num_samp : Positive int, optional

Number of samples to draw. Default is 1. Cannot be used in conjunction with *rand_vars*: If you pass both *num_samp* and *rand_vars*, *num_samp* will be silently ignored.

rand_vars : `Matrix` or other Array-like (*M*, *P*), optional

Vector of random variables u to use in constructing the sample $y_* = f_* + Lu$, where $K = LL^T$. If None, values will be produced using `numpy.random.multivariate_normal()`. This allows you to use pseudo/quasi random numbers generated by an external routine. Default is None (use `multivariate_normal()` directly).

rand_type : {'standard normal', 'uniform'}, optional

Type of distribution the inputs are given with.

- 'standard normal': Standard (*mu* = 0, *sigma* = 1) normal distribution (this is the default)
- 'uniform': Uniform distribution on [0, 1). In this case the required Gaussian variables are produced with inversion.

diag_factor : float, optional

Number (times machine epsilon) added to the diagonal of the covariance matrix prior to computing its Cholesky decomposition. This is necessary as sometimes the decomposition will fail because, to machine precision, the matrix appears to not be positive definite. If you are getting errors from `scipy.linalg.cholesky()`, try increasing this an order of magnitude at a time. This parameter only has an effect when using rand_vars. Default value is 1e3.

Returns **samples** : `Array` (*M*, *P*) or (*M*, *num_samp*)

Samples evaluated at the *M* points.

class `gptools.gaussian_process.`**`Constraint`**(*gp*, *boundary_val=0.0*, *n=0*, *loc='min'*, *type_='gt'*, *bounds=None*)

Bases: `object`

Implements an inequality constraint on the value of the mean or its derivatives.

Provides a callable such as can be passed to SLSQP or COBYLA to implement the constraint when using `scipy.optimize.minimize()`.

The function defaults implement a constraint that forces the mean value to be positive everywhere.

Parameters **gp** : `GaussianProcess`

The `GaussianProcess` instance to create the constraint on.

boundary_val : float, optional

Boundary value for the constraint. For *type_* = 'gt', this is the lower bound, for *type_* = 'lt', this is the upper bound. Default is 0.0.

n : non-negative int, optional

Derivative order to evaluate. Default is 0 (value of the mean). Note that non-int values are silently cast to int.

loc : {'min', 'max'}, float or Array-like of float (*num_dim*,), optional

Which extreme of the mean to use, or location to evaluate at.

- If 'min', the minimum of the mean (optionally over *bounds*) is used.
- If 'max', the maximum of the mean (optionally over *bounds*) is used.
- If a float (valid for *num_dim* = 1 only) or Array of float, the mean is evaluated at the given X value.

Default is 'min' (use function minimum).

type_ : {'gt', 'lt'}, optional

What type of inequality constraint to implement.

- If 'gt', a greater-than-or-equals constraint is used.
- If 'lt', a less-than-or-equals constraint is used.

Default is 'gt' (greater-than-or-equals).

bounds : 2-tuple of float or 2-tuple Array-like of float (*num_dim*,) or None, optional

Bounds to use when *loc* is 'min' or 'max'.

- If None, the bounds are taken to be the extremes of the training data. For multivariate data, "extremes" essentially means the smallest hypercube oriented parallel to the axes that encapsulates all of the training inputs. (I.e., `(gp.X.min(axis=0), gp.X.max(axis=0))`)
- If *bounds* is a 2-tuple, then this is used as (*lower*, *upper*) where lower' and *upper* are Array-like with dimensions (*num_dim*,).
- If *num_dim* is 1 then *lower* and *upper* can be scalar floats.

Default is None (use extreme values of training data).

Raises **TypeError** :

If *gp* is not an instance of `GaussianProcess`.

ValueError :

If *n* is negative.

ValueError :

If *loc* is not 'min', 'max' or an Array-like of the correct dimensions.

ValueError :

If *type_* is not 'gt' or 'lt'.

ValueError :

If *bounds* is not None or length 2 or if the elements of bounds don't have the right dimensions.

__call__(*params*)

Returns a non-negative number if the constraint is satisfied.

Parameters **params** : Array-like, length dictated by kernel

New parameters to use.

Returns **val** : float

Value of the constraint. `minimize` will attempt to keep this non-negative.

2.1.4 `utils` Module

Provides convenient utilities for working with the classes and results from `gptools`.

`gptools.utils.`**parallel_compute_ll_matrix**(*gp*, *bounds*, *num_pts*, *num_proc=None*)

Compute matrix of the log likelihood over the parameter space in parallel.

Parameters **bounds** : 2-tuple or list of 2-tuples with length equal to the number of free parameters

Bounds on the range to use for each of the parameters. If a single 2-tuple is given, it will be used for each of the parameters.

num_pts : int or list of ints with length equal to the number of free parameters

The number of points to use for each parameters. If a single int is given, it will be used for each of the parameters.

num_proc : Positive int or None, optional

Number of processes to run the parallel computation with. If set to None, ALL available cores are used. Default is None (use all available cores).

Returns **ll_vals** : `Array`

The log likelihood for each of the parameter possibilities.

param_vals : list of `Array`

The parameter values used.

`gptools.utils.`**`slice_plot`**(**args*, ***kwargs*)

Constructs a plot that lets you look at slices through a multidimensional array.

Parameters **vals** : `Array`, (*M*, *N*, *P*, ...)

Multidimensional array to visualize.

x_vals_1 : `Array`, (*M*,)

Values along the first dimension.

x_vals_2 : `Array`, (*N*,)

Values along the second dimension.

x_vals_3 : `Array`, (*P*,)

Values along the third dimension.

...and so on. At least four arguments must be provided.

names : list of strings, optional

Names for each of the parameters at hand. If None, sequential numerical identifiers will be used. Length must be equal to the number of dimensions of *vals*. Default is None.

n : Positive int, optional

Number of contours to plot. Default is 100.

Returns **f** : `Figure`

The Matplotlib figure instance created.

Raises **GPArgumentError** :

If the number of arguments is less than 4.

`gptools.utils.`**`arrow_respond`**(*slider*, *event*)

Event handler for arrow key events in plot windows.

Pass the slider object to update as a masked argument using a lambda function:

```
lambda evt: arrow_respond(my_slider, evt)
```

Parameters **slider** : Slider instance associated with this handler.

event : Event to be handled.

`gptools.utils.`**`incomplete_bell_poly`**(*n*, *k*, *x*)

Recursive evaluation of the incomplete Bell polynomial $B_{n,k}(x)$.

Evaluates the incomplete Bell polynomial $B_{n,k}(x_1, x_2, \ldots, x_{n-k+1})$, also known as the partial Bell polynomial or the Bell polynomial of the second kind. This polynomial is useful in the evaluation of (the univariate) Faa di Bruno's formula which generalizes the chain rule to higher order derivatives.

The implementation here is based on the implementation in: `sympy.functions.combinatorial.numbers.bell._bel`
Following that function's documentation, the polynomial is computed according to the recurrence formula:

$$B_{n,k}(x_1, x_2, \ldots, x_{n-k+1}) = \sum_{m=1}^{n-k+1} x_m \binom{n-1}{m-1} B_{n-m,k-1}(x_1, x_2, \ldots, x_{n-m-k})$$

The end cases are:
$B_{0,0} = 1$
$B_{n,0} = 0$ for $n \geq 1$
$B_{0,k} = 0$ for $k \geq 1$

Parameters **n** : scalar int

The first subscript of the polynomial.

k : scalar int

The second subscript of the polynomial.

x : `Array` of floats, (*p*, *n* - *k* + 1)

p sets of *n* - *k* + 1 points to use as the arguments to $B_{n,k}$. The second dimension can be longer than required, in which case the extra entries are silently ignored (this facilitates recursion without needing to subset the array *x*).

Returns **result** : `Array`, (*p*,)

Incomplete Bell polynomial evaluated at the desired values.

`gptools.utils.`**`generate_set_partition_strings`**(*n*)

Generate the restricted growth strings for all of the partitions of an *n*-member set.

Uses Algorithm H from page 416 of volume 4A of Knuth's *The Art of Computer Programming*. Returns the partitions in lexicographical order.

Parameters **n** : scalar int, non-negative

Number of (unique) elements in the set to be partitioned.

Returns **partitions** : list of `Array`

List has a number of elements equal to the *n*-th Bell number (i.e., the number of partitions for a set of size *n*). Each element has length *n*, the elements of which are the restricted growth strings describing the partitions of the set. The strings are returned in lexicographic order.

`gptools.utils.`**`generate_set_partitions`**(*set_*)

Generate all of the partitions of a set.

This is a helper function that utilizes the restricted growth strings from `generate_set_partition_strings()`. The partitions are returned in lexicographic order.

Parameters **set_** : `Array` or other Array-like, (*m*,)

The set to find the partitions of.

Returns **partitions** : list of lists of `Array`

The number of elements in the outer list is equal to the number of partitions, which is the len(*m*)^th Bell number. Each of the inner lists corresponds to a single possible partition. The length of an inner list is therefore equal to the number of blocks. Each of the arrays in an inner list is hence a block.

gptools.utils.**unique_rows**(*arr*)

Returns a copy of arr with duplicate rows removed.

From Stackoverflow "Find unique rows in numpy.array."

Parameters **arr** : Array, (*m*, *n*). The array to find the unique rows of.

Returns **unique** : Array, (*p*, *n*) where *p* <= *m*

The array *arr* with duplicate rows removed.

2.1.5 Subpackages

kernel Package

kernel Package

Subpackage containing a variety of covariance kernels and associated helpers.

core Module

Core kernel classes: contains the base Kernel class and helper subclasses.

class gptools.kernel.core.**Kernel**(*num_dim*, *num_params*, *initial_params=None*, *fixed_params=None*, *param_bounds=None*, *enforce_bounds=False*)

Bases: object

Covariance kernel base class. Not meant to be explicitly instantiated!

Initialize the kernel with the given number of input dimensions.

Parameters **num_dim** : positive int

Number of dimensions of the input data. Must be consistent with the *X* and *Xstar* values passed to the GaussianProcess you wish to use the covariance kernel with.

num_params : Non-negative int

Number of parameters in the model.

initial_params : Array or other Array-like, (*num_params*,), optional

Initial values to set for the hyperparameters. Default is None, in which case 1 is used for the initial values.

fixed_params : Array or other Array-like of bool, (*num_params*,), optional

Sets which parameters are considered fixed when optimizing the log likelihood. A True entry corresponds to that element being fixed (where the element ordering is as defined in the class). Default value is None (no parameters are fixed).

param_bounds : list of 2-tuples (*num_params*,), optional

List of bounds for each of the parameters. Each 2-tuple is of the form (*lower*, *upper*). If there is no bound in a given direction, set it to None. Default is (0.0, None) for each parameter.

enforce_bounds : bool, optional

If True, an attempt to set a parameter outside of its bounds will result in the parameter being set right at its bound. If False, bounds are not enforced inside the kernel. Default is False (do not enforce bounds).

Raises **ValueError** :

If *num_dim* is not a positive integer or the lengths of the input vectors are inconsistent.

GPArgumentError :

if *fixed_params* is passed but *initial_params* is not.

Attributes

num_params	int	Number of parameters
num_dim	int	Number of dimensions
params	`Array` of float, (*num_params*,)	Array of parameters.
fixed_params	`Array` of bool, (*num_params*,)	Array of booleans indicated which parameters in params are fixed.
param_bounds	list of 2-tuples, (*num_params*,)	List of bounds for the parameters in params.

`__call__` (*Xi*, *Xj*, *ni*, *nj*, *hyper_deriv=None*, *symmetric=False*)

Evaluate the covariance between points *Xi* and *Xj* with derivative order *ni*, *nj*.

Parameters **Xi** : `Matrix` or other Array-like, (*M*, *N*)

M inputs with dimension *N*.

Xj : `Matrix` or other Array-like, (*M*, *N*)

M inputs with dimension *N*.

ni : `Matrix` or other Array-like, (*M*, *N*)

M derivative orders for set *i*.

nj : `Matrix` or other Array-like, (*M*, *N*)

M derivative orders for set *j*.

hyper_deriv : Non-negative int or None, optional

The index of the hyperparameter to compute the first derivative with respect to. If None, no derivatives are taken. Default is None (no hyperparameter derivatives).

symmetric : bool, optional

Whether or not the input *Xi*, *Xj* are from a symmetric matrix. Default is False.

Returns **Kij** : `Array`, (*M*,)

Covariances for each of the *M* *Xi*, *Xj* pairs.

Notes

THIS IS ONLY A METHOD STUB TO DEFINE THE NEEDED CALLING FINGERPRINT!

`set_hyperparams` (*new_params*)

Sets the free hyperparameters to the new parameter values in new_params.

Parameters **new_params** : `Array` or other Array-like, (len(`self.params`),)

New parameter values, ordered as dictated by the docstring for the class.

free_params
Returns the values of the free hyperparameters.

Returns **free_params** : `Array`

Array of the free parameters, in order.

free_param_bounds
Returns the bounds of the free hyperparameters.

Returns **free_param_bounds** : `Array`

Array of the bounds of the free parameters, in order.

__add__(*other*)
Add two Kernels together.

Parameters **other** : `Kernel`

Kernel to be added to this one.

Returns **sum** : `SumKernel`

Instance representing the sum of the two kernels.

__mul__(*other*)
Multiply two Kernels together.

Parameters **other** : `Kernel`

Kernel to be multiplied by this one.

Returns **prod** : `ProductKernel`

Instance representing the product of the two kernels.

class gptools.kernel.core.**BinaryKernel**(*k1*, *k2*)
Bases: gptools.kernel.core.Kernel

Abstract class for binary operations on kernels (addition, multiplication, etc.).

Parameters **k1, k2** : `Kernel` instances to be combined

Notes

k1 and *k2* must have the same number of dimensions.

set_hyperparams(*new_params*)
Set the (free) hyperparameters.

Parameters **new_params** : `Array` or other Array-like

New values of the free parameters.

Raises **ValueError** :

If the length of *new_params* is not consistent with `self.params`.

class gptools.kernel.core.**SumKernel**(*k1*, *k2*)
Bases: gptools.kernel.core.BinaryKernel

The sum of two kernels.

__call__(**args*, ***kwargs*)
Evaluate the covariance between points *Xi* and *Xj* with derivative order *ni*, *nj*.

Parameters **Xi** : `Matrix` or other Array-like, (*M*, *N*)

M inputs with dimension *N*.

Xj : `Matrix` or other Array-like, (*M*, *N*)

M inputs with dimension *N*.

ni : `Matrix` or other Array-like, (*M*, *N*)

M derivative orders for set *i*.

nj : `Matrix` or other Array-like, (*M*, *N*)

M derivative orders for set *j*.

symmetric : bool, optional

Whether or not the input *Xi*, *Xj* are from a symmetric matrix. Default is False.

Returns **Kij** : `Array`, (*M*,)

Covariances for each of the *M* *Xi*, *Xj* pairs.

Raises **NotImplementedError** :

If the *hyper_deriv* keyword is given and is not None.

class gptools.kernel.core.**ProductKernel**(*k1*, *k2*)
Bases: gptools.kernel.core.BinaryKernel

The product of two kernels.

__call__(**args*, ***kwargs*)
Evaluate the covariance between points *Xi* and *Xj* with derivative order *ni*, *nj*.

Parameters **Xi** : `Matrix` or other Array-like, (*M*, *N*)

M inputs with dimension *N*.

Xj : `Matrix` or other Array-like, (*M*, *N*)

M inputs with dimension *N*.

ni : `Matrix` or other Array-like, (*M*, *N*)

M derivative orders for set *i*.

nj : `Matrix` or other Array-like, (*M*, *N*)

M derivative orders for set *j*.

symmetric : bool, optional

Whether or not the input *Xi*, *Xj* are from a symmetric matrix. Default is False.

Returns **Kij** : `Array`, (*M*,)

Covariances for each of the *M* *Xi*, *Xj* pairs.

Raises **NotImplementedError** :

If the *hyper_deriv* keyword is given and is not None.

class gptools.kernel.core.**ChainRuleKernel**(*num_dim*, *num_params*, *initial_params=None*, *fixed_params=None*, *param_bounds=None*, *enforce_bounds=False*)

Bases: gptools.kernel.core.Kernel

Abstract class for the common methods in creating kernels that require application of Faa di Bruno's formula.

__call__(*Xi*, *Xj*, *ni*, *nj*, *hyper_deriv=None*, *symmetric=False*)

Evaluate the covariance between points *Xi* and *Xj* with derivative order *ni*, *nj*.

Parameters **Xi** : Matrix or other Array-like, (*M*, *N*)

M inputs with dimension *N*.

Xj : Matrix or other Array-like, (*M*, *N*)

M inputs with dimension *N*.

ni : Matrix or other Array-like, (*M*, *N*)

M derivative orders for set *i*.

nj : Matrix or other Array-like, (*M*, *N*)

M derivative orders for set *j*.

hyper_deriv : Non-negative int or None, optional

The index of the hyperparameter to compute the first derivative with respect to. If None, no derivatives are taken. Hyperparameter derivatives are not supported at this point. Default is None.

symmetric : bool

Whether or not the input *Xi*, *Xj* are from a symmetric matrix. Default is False.

Returns **Kij** : Array, (*M*,)

Covariances for each of the *M* *Xi*, *Xj* pairs.

Raises **NotImplementedError** :

If the *hyper_deriv* keyword is not None.

class gptools.kernel.core.**ArbitraryKernel**(*num_dim*, *cov_func*, *num_proc=0*, ***kwargs*)

Bases: gptools.kernel.core.Kernel

Covariance kernel from an arbitrary covariance function.

Computes derivatives using mpmath.diff() and is hence in general much slower than a hard-coded implementation of a given kernel.

Parameters **num_dim** : positive int

Number of dimensions of the input data. Must be consistent with the *X* and *Xstar* values passed to the GaussianProcess you wish to use the covariance kernel with.

cov_func : callable, takes >= 2 args

Covariance function. Must take arrays of *Xi* and *Xj* as the first two arguments. The subsequent (scalar) arguments are the hyperparameters. The number of parameters is found by inspection of *cov_func* itself.

num_proc : int or None, optional

Number of procs to use in evaluating covariance derivatives. 0 means to do it in serial, None means to use all available cores. Default is 0 (serial evaluation).

****kwargs** :

All other keyword parameters are passed to `Kernel`.

Attributes

cov_func	callable	The covariance function
num_proc	non-negative int	Number of processors to use in evaluating covariance derivatives. 0 means serial.

__call__ (*Xi*, *Xj*, *ni*, *nj*, *hyper_deriv=None*, *symmetric=False*)

Evaluate the covariance between points *Xi* and *Xj* with derivative order *ni*, *nj*.

Parameters **Xi** : `Matrix` or other Array-like, (*M*, *N*)

M inputs with dimension *N*.

Xj : `Matrix` or other Array-like, (*M*, *N*)

M inputs with dimension *N*.

ni : `Matrix` or other Array-like, (*M*, *N*)

M derivative orders for set *i*.

nj : `Matrix` or other Array-like, (*M*, *N*)

M derivative orders for set *j*.

hyper_deriv : Non-negative int or None, optional

The index of the hyperparameter to compute the first derivative with respect to. If None, no derivatives are taken. Hyperparameter derivatives are not supported at this point. Default is None.

symmetric : bool, optional

Whether or not the input *Xi*, *Xj* are from a symmetric matrix. Default is False.

Returns **Kij** : `Array`, (*M*,)

Covariances for each of the *M* *Xi*, *Xj* pairs.

Raises **NotImplementedError** :

If the *hyper_deriv* keyword is not None.

gibbs Module

Provides classes and functions for creating SE kernels with warped length scales.

gptools.kernel.gibbs.**tanh_warp** (*X*, *l1*, *l2*, *lw*, *x0*)

Warps the *X* coordinate with the tanh model

$$l = \frac{l_1 + l_2}{2} - \frac{l_1 - l_2}{2} \tanh \frac{x - x_0}{l_w}$$

Parameters **X** : `Array`, (*M*,) or scalar float

M locations to evaluate length scale at.

l1 : positive float

Small-*X* saturation value of the length scale.

l2 : positive float

Large-*X* saturation value of the length scale.

lw : positive float

Length scale of the transition between the two length scales.

x0 : float

Location of the center of the transition between the two length scales.

Returns **l** : `Array`, (*M*,) or scalar float

The value of the length scale at the specified point.

`gptools.kernel.gibbs.`**`spline_warp`**(*X*, *l1*, *l2*, *lw*, *x0*)

Warps the *X* coordinate with a "divot" spline shape.

Warning: Broken! Do not use!

Parameters **X** : `Array`, (*M*,) or scalar float

M locations to evaluate length scale at.

l1 : positive float

Global value of the length scale.

l2 : positive float

Pedestal value of the length scale.

lw : positive float

Width of the dip.

x0 : float

Location of the center of the dip in length scale.

Returns **l** : `Array`, (*M*,) or scalar float

The value of the length scale at the specified point.

`gptools.kernel.gibbs.`**`gauss_warp`**(*X*, *l1*, *l2*, *lw*, *x0*)

Warps the *X* coordinate with a Gaussian-shaped divot.

$$l = l_1 - (l_1 - l_2)\exp\left(-4\ln 2\frac{(X - x_0)^2}{l_w^2}\right)$$

Parameters **X** : `Array`, (*M*,) or scalar float

M locations to evaluate length scale at.

l1 : positive float

Global value of the length scale.

l2 : positive float

Pedestal value of the length scale.

lw : positive float

Width of the dip.

x0 : float

Location of the center of the dip in length scale.

Returns **l** : `Array`, (*M*,) or scalar float

The value of the length scale at the specified point.

class `gptools.kernel.gibbs.`**`GibbsFunction1d`**(*warp_function*)
Bases: `object`

Wrapper class for the Gibbs covariance function, permits the use of arbitrary warping.

The covariance function is given by

$$k = \left(\frac{2l(x)l(x')}{l^2(x)+l^2(x')}\right)^{1/2} \exp\left(-\frac{(x-x')^2}{l^2(x)+l^2(x')}\right)$$

Parameters **warp_function** : callable

The function that warps the length scale as a function of X. Must have the fingerprint (*Xi*, *l1*, *l2*, *lw*, *x0*).

`__call__`(*Xi*, *Xj*, *sigmaf*, *l1*, *l2*, *lw*, *x0*)
Evaluate the covariance function between points *Xi* and *Xj*.

Parameters **Xi, Xj** : `Array`, `mpf` or scalar float

Points to evaluate covariance between. If they are `Array`, `scipy` functions are used, otherwise `mpmath` functions are used.

sigmaf : scalar float

Prefactor on covariance.

l1, l2, lw, x0 : scalar floats

Parameters of length scale warping function, passed to `warp_function`.

Returns **k** : `Array` or `mpf`

Covariance between the given points.

class `gptools.kernel.gibbs.`**`GibbsKernel1dtanh`**(***kwargs*)
Bases: `gptools.kernel.core.ArbitraryKernel`

Gibbs warped squared exponential covariance function in 1d.

Computes derivatives using `mpmath.diff()` and is hence in general much slower than a hard-coded implementation of a given kernel.

The covariance function is given by

$$k = \left(\frac{2l(x)l(x')}{l^2(x)+l^2(x')}\right)^{1/2} \exp\left(-\frac{(x-x')^2}{l^2(x)+l^2(x')}\right)$$

Warps the length scale using a hyperbolic tangent:

$$l = \frac{l_1+l_2}{2} - \frac{l_1-l_2}{2}\tanh\frac{x-x_0}{l_w}$$

The order of the hyperparameters is:

0	sigmaf	Amplitude of the covariance function
1	l1	Small-X saturation value of the length scale.
2	l2	Large-X saturation value of the length scale.
3	lw	Length scale of the transition between the two length scales.
4	x0	Location of the center of the transition between the two length scales.

Parameters **kwargs :

All parameters are passed to `Kernel`.

class `gptools.kernel.gibbs.`**`GibbsKernel1dSpline`**(***kwargs*)
Bases: `gptools.kernel.core.ArbitraryKernel`

Gibbs warped squared exponential covariance function in 1d.

Warning: Broken! Do not use!

Computes derivatives using `mpmath.diff()` and is hence in general much slower than a hard-coded implementation of a given kernel.

The covariance function is given by

$$k = \left(\frac{2l(x)l(x')}{l^2(x) + l^2(x')}\right)^{1/2} \exp\left(-\frac{(x-x')^2}{l^2(x) + l^2(x')}\right)$$

Warps the length scale using a spline.

The order of the hyperparameters is:

0	sigmaf	Amplitude of the covariance function
1	l1	Global value of the length scale.
2	l2	Pedestal value of the length scale.
3	lw	Width of the dip.
4	x0	Location of the center of the dip in length scale.

Parameters **kwargs :

All parameters are passed to `Kernel`.

class `gptools.kernel.gibbs.`**`GibbsKernel1dGauss`**(***kwargs*)
Bases: `gptools.kernel.core.ArbitraryKernel`

Gibbs warped squared exponential covariance function in 1d.

Computes derivatives using `mpmath.diff()` and is hence in general much slower than a hard-coded implementation of a given kernel.

The covariance function is given by

$$k = \left(\frac{2l(x)l(x')}{l^2(x) + l^2(x')}\right)^{1/2} \exp\left(-\frac{(x-x')^2}{l^2(x) + l^2(x')}\right)$$

Warps the length scale using a gaussian:

$$l = l_1 - (l_1 - l_2)\exp\left(-4\ln 2\frac{(X-x_0)^2}{l_w^2}\right)$$

The order of the hyperparameters is:

0	sigmaf	Amplitude of the covariance function
1	l1	Global value of the length scale.
2	l2	Pedestal value of the length scale.
3	lw	Width of the dip.
4	x0	Location of the center of the dip in length scale.

Parameters **kwargs :

All parameters are passed to `Kernel`.

matern Module

Provides the MaternKernel class which implements the anisotropic Matern kernel.

class gptools.kernel.matern.**MaternKernel**(*num_dim*, ***kwargs*)

Bases: gptools.kernel.core.ChainRuleKernel

Matern covariance kernel. Supports arbitrary derivatives. Treats order as a hyperparameter.

The Matern kernel has the following hyperparameters, always referenced in the order listed:

0	sigma	prefactor
1	nu	order of kernel
2	l1	length scale for the first dimension
3	l2	...and so on for all dimensions

The kernel is defined as:

$$k_M = \sigma^2 \frac{2^{1-\nu}}{\Gamma(\nu)} \left(\sqrt{2\nu} \sum_i \left(\frac{\tau_i^2}{l_i^2} \right) \right)^{\nu} K_\nu \left(\sqrt{2\nu} \sum_i \left(\frac{\tau_i^2}{l_i^2} \right) \right)$$

Parameters **num_dim** : int

Number of dimensions of the input data. Must be consistent with the *X* and *Xstar* values passed to the GaussianProcess you wish to use the covariance kernel with.

****kwargs** :

All keyword parameters are passed to ChainRuleKernel.

Raises **ValueError** :

If *num_dim* is not a positive integer or the lengths of the input vectors are inconsistent.

GPArgumentError :

If *fixed_params* is passed but *initial_params* is not.

nu

Returns the value of the order ν.

noise Module

Provides classes for implementing uncorrelated noise.

class gptools.kernel.noise.**DiagonalNoiseKernel**(*num_dim*, *initial_noise=None*, *fixed_noise=None*, *noise_bound=None*, *n=0*)

Bases: gptools.kernel.core.Kernel

Kernel that has constant, independent noise (i.e., a diagonal kernel).

Parameters **num_dim** : positive int

Number of dimensions of the input data.

initial_noise : float

Initial value for the noise standard deviation. Default value is None (noise gets set to 1).

fixed_noise : bool

Whether or not the noise is taken to be fixed when optimizing the log likelihood.

noise_bound : 2-tuple

The bounds for the noise when optimizing the log likelihood with `scipy.optimize.minimize()`. Must be of the form (*lower*, *upper*). Set a given entry to None to put no bound on that side. Default is None, which gets set to (0, None).

n : non-negative int or tuple of non-negative ints with length equal to *num_dim*

Indicates which derivative this noise is with respect to. Default is 0 (noise applies to value).

__call__(*Xi*, *Xj*, *ni*, *nj*, *hyper_deriv=None*, *symmetric=False*)

Evaluate the covariance between points *Xi* and *Xj* with derivative order *ni*, *nj*.

Parameters **Xi** : `Matrix` or other Array-like, (*M*, *N*)

M inputs with dimension *N*.

Xj : `Matrix` or other Array-like, (*M*, *N*)

M inputs with dimension *N*.

ni : `Matrix` or other Array-like, (*M*, *N*)

M derivative orders for set *i*.

nj : `Matrix` or other Array-like, (*M*, *N*)

M derivative orders for set *j*.

hyper_deriv : Non-negative int or None, optional

The index of the hyperparameter to compute the first derivative with respect to. Since this kernel only has one hyperparameter, 0 is the only valid value. If None, no derivatives are taken. Default is None (no hyperparameter derivatives).

symmetric : bool, optional

Whether or not the input *Xi*, *Xj* are from a symmetric matrix. Default is False.

Returns **Kij** : `Array`, (*M*,)

Covariances for each of the *M* *Xi*, *Xj* pairs.

class `gptools.kernel.noise.`**ZeroKernel**(*num_dim*)

Bases: `gptools.kernel.noise.DiagonalNoiseKernel`

Kernel that always evaluates to zero, used as the default noise kernel.

Parameters **num_dim** : positive int

The number of dimensions of the inputs.

__call__(*Xi*, *Xj*, *ni*, *nj*, *hyper_deriv=None*, *symmetric=False*)

Return zeros the same length as the input Xi.

Ignores all other arguments.

Parameters **Xi** : `Matrix` or other Array-like, (*M*, *N*)

M inputs with dimension *N*.

Xj : `Matrix` or other Array-like, (*M*, *N*)

M inputs with dimension *N*.

ni : `Matrix` or other Array-like, (*M*, *N*)

M derivative orders for set *i*.

nj : `Matrix` or other Array-like, (*M*, *N*)

M derivative orders for set *j*.

hyper_deriv : Non-negative int or None, optional

The index of the hyperparameter to compute the first derivative with respect to. Since this kernel only has one hyperparameter, 0 is the only valid value. If None, no derivatives are taken. Default is None (no hyperparameter derivatives).

symmetric : bool, optional

Whether or not the input *Xi*, *Xj* are from a symmetric matrix. Default is False.

Returns **Kij** : `Array`, (*M*,)

Covariances for each of the *M* *Xi*, *Xj* pairs.

`rational_quadratic` Module

Provides the `RationalQuadraticKernel` class which implements the anisotropic rational quadratic (RQ) kernel.

class `gptools.kernel.rational_quadratic.`**`RationalQuadraticKernel`**(*num_dim*, ***kwargs*)

Bases: `gptools.kernel.core.ChainRuleKernel`

Rational quadratic (RQ) covariance kernel. Supports arbitrary derivatives.

The RQ kernel has the following hyperparameters, always referenced in the order listed:

0	sigma	prefactor.
1	alpha	order of kernel.
2	l1	length scale for the first dimension.
3	l2	...and so on for all dimensions.

The kernel is defined as:

$$k_{RQ} = \sigma^2 \left(1 + \frac{1}{2\alpha} \sum_i \frac{\tau_i^2}{l_i^2}\right)^{-\alpha}$$

Parameters **num_dim** : int

Number of dimensions of the input data. Must be consistent with the *X* and *Xstar* values passed to the `GaussianProcess` you wish to use the covariance kernel with.

****kwargs** :

All keyword parameters are passed to `ChainRuleKernel`.

Raises **ValueError** :

If *num_dim* is not a positive integer or the lengths of the input vectors are inconsistent.

GPArgumentError :

If *fixed_params* is passed but *initial_params* is not.

squared_exponential Module

Provides the SquaredExponentialKernel class that implements the anisotropic SE kernel.

class gptools.kernel.squared_exponential.**SquaredExponentialKernel**(*num_dim*, ***kwargs*)

Bases: gptools.kernel.core.Kernel

Squared exponential covariance kernel. Supports arbitrary derivatives.

The squared exponential has the following hyperparameters, always referenced in the order listed:

0	sigma	prefactor on the SE
1	l1	length scale for the first dimension
2	l2	...and so on for all dimensions

The kernel is defined as:

$$k_{SE} = \sigma^2 \exp\left(-\sum_i \frac{\tau_i^2}{l_i^2}\right)$$

Parameters **num_dim** : int

Number of dimensions of the input data. Must be consistent with the *X* and *Xstar* values passed to the GaussianProcess you wish to use the covariance kernel with.

****kwargs** :

All keyword parameters are passed to Kernel.

Raises **ValueError** :

If *num_dim* is not a positive integer or the lengths of the input vectors are inconsistent.

GPArgumentError :

If *fixed_params* is passed but *initial_params* is not.

__call__(*Xi*, *Xj*, *ni*, *nj*, *hyper_deriv=None*, *symmetric=False*)

Evaluate the covariance between points *Xi* and *Xj* with derivative order *ni*, *nj*.

Parameters **Xi** : Matrix or other Array-like, (*M*, *N*)

M inputs with dimension *N*.

Xj : Matrix or other Array-like, (*M*, *N*)

M inputs with dimension *N*.

ni : Matrix or other Array-like, (*M*, *N*)

M derivative orders for set *i*.

nj : Matrix or other Array-like, (*M*, *N*)

M derivative orders for set *j*.

hyper_deriv : Non-negative int or None, optional

The index of the hyperparameter to compute the first derivative with respect to. If None, no derivatives are taken. Default is None (no hyperparameter derivatives). Hyperparameter derivatives are not support for $n > 0$ at this time.

symmetric : bool, optional

Whether or not the input *Xi*, *Xj* are from a symmetric matrix. Default is False.

Returns **Kij** : `Array`, (*M*,)

Covariances for each of the *M Xi*, *Xj* pairs.

Raises **NotImplementedError** :

If hyper_deriv is not None and $n > 0$.

CHAPTER

THREE

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX

g

PYTHON MODULE INDEX

g

INDEX

Symbols

A

B

C

D

F

G

I

K

M

N

O

P

R

S

T

U

Z