gptools Documentation

Release 0.2.2

Mark Chilenski

January 17, 2017

Contents

Source home: https://github.com/markchil/gptools

Releases: https://pypi.python.org/pypi/gptools/

Installation is as simple as:

```
pip install gptools
```

A comprehensive demo is provided at https://github.com/markchil/gptools/blob/master/demo/demo.py, with extensive comments showing how the code functions on real data (also hosted on the github). This should be consulted in parallel with this manual.

CHAPTER 1

Overview

gptools is a Python package that provides a convenient, powerful and extensible implementation of Gaussian process regression (GPR). Central to *gptools*' implementation is support for derivatives and their variances. Furthermore, the implementation supports the incorporation of arbitrary linearly transformed quantities into the GP.

There are two key classes:

- *GaussianProcess* is the main class to represent a GP.
- *Kernel* (and its many subclasses) represents a covariance kernel, and must be provided when constructing a *GaussianProcess*. Separate kernels to describe the underlying signal and the noise are supported.

A third class, *JointPrior*, allows you to construct a hyperprior of arbitrary complexity to dictate how the hyperparameters are handled.

Creating a Gaussian process is as simple as:

```
import gptools
k = gptools.SquaredExponentialKernel()
gp = gptools.GaussianProcess(k)
```

But, the default bounds on the hyperparameters are very wide and can cause the optimizer/MCMC sampler to fail. So, it is usually a better idea to define the covariance kernel as:

```
k = gptools.SquaredExponentialKernel(param_bounds=[(0, 1e3), (0, 100)])
```

You will have to pick appropriate numbers by inspecting the typical range of your data.

Furthermore, you can include an explicit mean function by passing the appropriate *MeanFunction* instance into the *mu* keyword:

```
gp = gptools.GaussianProcess(k, mu=gptools.LinearMeanFunction())
```

This will enable you to perform inference (both empirical and full Bayes) for the hyperparameters of the mean function. Essentially, *gptools* can perform nonlinear Bayesian regression with a Gaussian process fit to the residuals.

You then add the training data using the *add_data()* method:

```
gp.add_data(x, y, err_y=stddev_y)
```

Here, *err_y* is the 1σ uncertainty on the observations y. For exact values, simply omit this keyword. Adding a derivative observation is as simple as specifying the derivative order with the *n* keyword:

```
gp.add_data(0, 0, n=1)
```

This will force the slope at $x = 0$ to be exactly zero. Quantities that represent an arbitrary linear transformation of the "basic" observations can be added by specifying the *T* keyword:

```
gp.add_data(x, y, T=T)
```

This will add the value(s) $y = TY(x)$ to the training data, where here Y represents the "basic" (untransformed) observations and y represents the transformed observations. This also supports the *err_y* and *n* keywords. Here, *err_y* is the error on the transformed quantity y. *n* applies to the latent variables $Y(x)$.

Once the GP has been populated with training data, there are two approaches supported to handle the hyperparameters.

The simplest approach is to use an empirical Bayes approach and compute the maximum a posteriori (MAP) estimate. This is accomplished using the *optimize_hyperparameters()* method of the *GaussianProcess* instance:

```
gp.optimize_hyperparameters()
```

This will randomly start the optimizer at points distributed according to the hyperprior several times in order to ensure that the global maximum is obtained. If you have a machine with multiple cores, these random starts will be performed in parallel. You can set the number of starts using the *random_starts* keyword, and you can set the number of processes used using the *num_proc* keyword.

For a more complete accounting of the uncertainties in the model, you can choose to use a fully Bayesian approach by using Markov chain Monte Carlo (MCMC) techniques to produce samples of the hyperposterior. The samples are produced directly with *sample_hyperparameter_posterior()*, though it will typically be more convenient to simply call *predict()* with the *use_MCMC* keyword set to True.

In order to make predictions, use the *predict()* method:

```
y_star, err_y_star = gp.predict(x_star)
```

By default, the mean and standard deviation of the GP posterior are returned. To compute only the mean and save some time, set the *return_std* keyword to False. To make predictions of derivatives, set the *n* keyword. To make a prediction of a linearly transformed quantity, set the *output_transform* keyword.

For a convenient wrapper for applying *gptools* to multivariate data, see `profiletools` at https://github.com/markchil/profiletools

CHAPTER 2

Kernels

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 $n < \nu$. Note that this class does not support arbitrary derivatives at this point. If you need this feature, try using *MaternKernelArb*, but note that this is very slow!
- *Matern52Kernel* implements a specialized Matern kernel with $\nu = \frac{5}{2}$ which efficiently supports 0th and 1st derivatives.
- *RationalQuadraticKernel* implements the rational quadratic kernel, which is a scale mixture over SE kernels.
- *GibbsKernel1d* and its subclasses implements the Gibbs kernel, which is a nonstationary form of the SE kernel.
- *MaskedKernel* creates a kernel that only operates on a subset of dimensions. Use this along with the sum and product operations to create kernels that encode different properties in different dimensions.
- *ArbitraryKernel* creates a kernel with an arbitrary covariance function and computes the derivatives as needed using `mpmath` to perform numerical differentiation. Naturally, this is very slow but is useful to let you explore the properties of arbitrary kernels without having to write a complete implementation.

In most 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 3

Notes

gptools has been developed and tested on Python 2.7 and scipy 0.14.0. It may work just as well on other versions, but has not been tested.

If you find this software useful, please be sure to cite it:

M.A. Chilenski et al. 2015 Nucl. Fusion 55 023012

http://stacks.iop.org/0029-5515/55/023012

Contents

4.1 gptools package

4.1.1 Subpackages

gptools.kernel package

Submodules

gptools.kernel.core module

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

class gptools.kernel.core.**Kernel**(*num_dim=1*, *num_params=0*, *initial_params=None*, *fixed_params=None*, *param_bounds=None*, *param_names=None*, *enforce_bounds=False*, *hyper-prior=None*)

Bases: object

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

Initialize the kernel with the given number of input dimensions.

When implementing an isotropic covariance kernel, the covariance length scales should be the last *num_dim* elements in *params*.

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. Default is 1.

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 hyperparameters 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 hyperparameters are fixed).

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

List of bounds for each of the hyperparameters. Each 2-tuple is of the form (*lower*, *upper*). If there is no bound in a given direction, it works best to set it to something big like 1e16. Default is (0.0, 1e16) for each hyperparameter. Note that this is overridden by the *hyperprior* keyword, if present.

param_names : list of str (*num_params*,), optional

List of labels for the hyperparameters. Default is all empty strings.

enforce_bounds : bool, optional

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

hyperprior : *JointPrior* instance or list, optional

Joint prior distribution for all hyperparameters. Can either be given as a `JointPrior` instance or a list of *num_params* callables or py:class:*rv_frozen* instances from `scipy.stats`, in which case a `IndependentJointPrior` is constructed with these as the independent priors on each hyperparameter. Default is a uniform PDF on all hyperparameters.

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_free_params	Returns the number of free parameters.
free_param_idxs	Returns the indices of the free parameters in the main arrays of parameters, etc.
free_params	Returns the values of the free hyperparameters.
free_param_bounds	Returns the bounds of the free hyperparameters.
free_param_names	Returns the names of the free hyperparameters.

num_params	(int) Number of parameters.
num_dim	(int) Number of dimensions.
param_names	(list of str, (*num_params*,)) List of the labels for the hyperparameters.
enforce_bounds	(bool) If True, do not allow hyperparameters to be set outside of their bounds.
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.
hyperprior	(*JointPrior* instance) Joint prior distribution for the hyperparameters.
param_bounds	(*CombinedBounds*) The bounds on the hyperparameters. Actually a getter method with a property decorator.

param_bounds

__call__(*Xi*, *Xj*, *ni*, *nj*, *hyper_deriv=None*, *symmetric=False*)
Evaluate the covariance between points *Xi* and *Xj* with derivative order *ni*, *nj*.

Note that this method only returns the covariance – the hyperpriors and potentials stored in this kernel must be applied separately.

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

M inputs with dimension *D*.

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

M inputs with dimension *D*.

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

M derivative orders for set *i*.

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

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.free_params`),)

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

num_free_params
Returns the number of free parameters.

free_param_idxs
Returns the indices of the free parameters in the main arrays of parameters, etc.

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.

free_param_names
Returns the names of the free hyperparameters.

Returns **free_param_names** : `Array`

Array of the names 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.

enforce_bounds
Boolean indicating whether or not the kernel will explicitly enforce the bounds defined by the hyperprior.

fixed_params

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.

free_param_names
Returns the names of the free hyperparameters.

Returns **free_param_names** : `Array`

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

params

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*, *D*)

M inputs with dimension *D*.

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

M inputs with dimension *D*.

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

M derivative orders for set *i*.

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

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.

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

The product of two kernels.

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

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

M inputs with dimension *D*.

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

M inputs with dimension *D*.

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

M derivative orders for set *i*.

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

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=1*, *num_params=0*, *initial_params=None*, *fixed_params=None*, *param_bounds=None*, *param_names=None*, *enforce_bounds=False*, *hyperprior=None*)

Bases: *gptools.kernel.core.Kernel*

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

Implementing classes should provide the following methods:

- *_compute_k(self, tau)*: Should return the value of *k(tau)* at the given values of *tau*, but without multiplying by *sigma_f*. This is done separately for efficiency.
- *_compute_y(self, tau, return_r2l2=False)*: Should return the "inner form" *y(tau)* to use with Faa di Bruno's formula. If *return_r2l2* is True, should also return the *r2l2* matrix from *self._compute_r2l2(tau)*.
- *_compute_dk_dy(self, y, n)*: Should compute the *n'th derivative of the kernel 'k* with respect to *y* at the locations *y*.
- *_compute_dy_dtau(self, tau, b, r2l2)*: Should compute the derivatives of *y* with respect to the elements of *tau* as indicated in *b*.

__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*, *D*)

M inputs with dimension *D*.

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

M inputs with dimension *D*.

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

M derivative orders for set *i*.

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

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**(*cov_func*, *num_dim=1*, *num_proc=0*, *num_params=None*, ***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, or with the num_params keyword.

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).

num_params : int or None, optional

Number of hyperparameters. If None, inspection will be used to infer the number of hyperparameters (but will fail if you used clever business with *args, etc.). Default is None (use inspection to find argument count).

****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*, *D*)

M inputs with dimension *D*.

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

M inputs with dimension *D*.

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

M derivative orders for set *i*.

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

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.

class gptools.kernel.core.**MaskedKernel**(*base, total_dim=2, mask=[0], scale=None*)

Bases: *gptools.kernel.core.Kernel*

Creates a kernel that is only masked to operate on certain dimensions, or has scaling/shifting.

This can be used, for instance, to put a squared exponential kernel in one direction and a Matern kernel in the other.

Overrides *__getattribute__()* and *__setattr__()* to make all setting/accessing go to the *base* kernel.

Parameters **base** : *Kernel* instance

The *Kernel* to apply in the dimensions specified in *mask*.

total_dim : int, optional

The total number of dimensions the masked kernel should have. Default is 2.

mask : list or other array-like, optional

1d list of indices of dimensions *X* to include when passing to the *base* kernel. Length must be *base.num_dim*. Default is [0] (i.e., just pass the first column of *X* to a univariate *base* kernel).

scale : list or other array-like, optional

1d list of scale factors to apply to the elements in *Xi*, *Xj*. Default is ones. Length must be equal to 2‘base.num_dim‘.

__getattribute__(*name*)

Gets all attributes from the base kernel.

The exceptions are ‘base’, ‘mask’, ‘maskC’, ‘num_dim’, ‘scale’ and any special method (i.e., a method/attribute having leading and trailing double underscores), which are taken from *MaskedKernel*.

__setattr__(*name*, *value*)

Sets all attributes in the base kernel.

The exceptions are ‘base’, ‘mask’, ‘maskC’, ‘num_dim’, ‘scale’ and any special method (i.e., a method/attribute having leading and trailing double underscores), which are set in *MaskedKernel*.

__call__(*Xi*, *Xj*, *ni*, *nj*, ***kwargs*)

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

Note that in the argument specifications, *D* is the *total_dim* specified in the constructor (i.e., `num_dim` for the *MaskedKernel* instance itself).

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

M inputs with dimension *D*.

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

M inputs with dimension *D*.

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

M derivative orders for set *i*.

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

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.

gptools.kernel.gibbs module

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

`gptools.kernel.gibbs.`**`tanh_warp_arb`**(*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.`**`gauss_warp_arb`**(*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.`**`GibbsFunction1dArb`**(*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.`**`GibbsKernel1dTanhArb`**(***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.**GibbsKernel1dGaussArb**(***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*.

class gptools.kernel.gibbs.**GibbsKernel1d**(*l_func*, *num_params=None*, ***kwargs*)
Bases: *gptools.kernel.core.Kernel*

Univariate Gibbs kernel with arbitrary length scale warping for low derivatives.

The covariance function is given by

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

The derivatives are hard-coded using expressions obtained from Mathematica.

Parameters **l_func** : callable

Function that dictates the length scale warping and its derivative. Must have fingerprint (*x*, *n*, *p1*, *p2*, ...) where *p1* is element one of the kernel's parameters (i.e., element zero is skipped).

num_params : int, optional

The number of parameters of the length scale function. If not passed, introspection will be used to determine this. This will fail if you have used the *args syntax in your function definition. This count should include sigma_f, even though it is not passed to the length scale function.

****kwargs**

All remaining arguments are passed to `Kernel`.

__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*, *D*)

M inputs with dimension *D*.

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

M inputs with dimension *D*.

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

M derivative orders for set *i*.

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

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.

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

Implements a tanh warping function and its derivative.

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

Parameters **x** : float or array of float

Locations to evaluate the function at.

n : int

Derivative order to take. Used for ALL of the points.

l1 : positive float

Left saturation value.

l2 : positive float

Right saturation value.

lw : positive float

Transition width.

x0 : float

Transition location.

Returns **l** : float or array

Warped length scale at the given locations.

Raises **NotImplementedError**

If $n > 1$.

class gptools.kernel.gibbs.**GibbsKernel1dTanh**(***kwargs*)

Bases: *gptools.kernel.gibbs.GibbsKernel1d*

Gibbs warped squared exponential covariance function in 1d.

Uses hard-coded implementation up to first derivatives.

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*.

gptools.kernel.gibbs.**double_tanh_warp**(*x*, *n*, *lcore*, *lmid*, *ledge*, *la*, *lb*, *xa*, *xb*)

Implements a sum-of-tanh warping function and its derivative.

$$l = a \tanh\frac{x - x_a}{l_a} + b \tanh\frac{x - x_b}{l_b}$$

Parameters **x** : float or array of float

Locations to evaluate the function at.

n : int

Derivative order to take. Used for ALL of the points.

lcore : float

Core length scale.

lmid : float

Intermediate length scale.

ledge : float

Edge length scale.

la : positive float

Transition of first tanh.

lb : positive float

Transition of second tanh.

xa : float

Transition of first tanh.

xb : float

Transition of second tanh.

Returns **l** : float or array

Warped length scale at the given locations.

Raises **NotImplementedError**

If $n > 1$.

class gptools.kernel.gibbs.**GibbsKernel1dDoubleTanh**(***kwargs*)

Bases: *gptools.kernel.gibbs.GibbsKernel1d*

Gibbs warped squared exponential covariance function in 1d.

Uses hard-coded implementation up to first derivatives.

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 two hyperbolic tangents:

$$l = a \tanh\frac{x - x_a}{l_a} + b \tanh\frac{x - x_b}{l_b}$$

The order of the hyperparameters is:

0	sigmaf	Amplitude of the covariance function
1	lcore	Core length scale
2	lmid	Intermediate length scale
3	ledge	Edge length scale
4	la	Width of first tanh
5	lb	Width of second tanh
6	xa	Center of first tanh
7	xb	Center of second tanh

Parameters ****kwargs**

All parameters are passed to *Kernel*.

gptools.kernel.gibbs.**cubic_bucket_warp**(*x*, *n*, *l1*, *l2*, *l3*, *x0*, *w1*, *w2*, *w3*)

Warps the length scale with a piecewise cubic "bucket" shape.

Parameters **x** : float or array-like of float

Locations to evaluate length scale at.

n : non-negative int

Derivative order to evaluate. Only first derivatives are supported.

l1 : positive float

Length scale to the left of the bucket.

l2 : positive float

Length scale in the bucket.

l3 : positive float

Length scale to the right of the bucket.

x0 : float

Location of the center of the bucket.

w1 : positive float

Width of the left side cubic section.

w2 : positive float

Width of the bucket.

w3 : positive float

Width of the right side cubic section.

class gptools.kernel.gibbs.**GibbsKernel1dCubicBucket**(***kwargs*)

Bases: *gptools.kernel.gibbs.GibbsKernel1d*

Gibbs warped squared exponential covariance function in 1d.

Uses hard-coded implementation up to first derivatives.

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 "bucket" function with cubic joins.

The order of the hyperparameters is:

0	sigmaf	Amplitude of the covariance function
1	l1	Length scale to the left of the bucket.
2	l2	Length scale in the bucket.
3	l3	Length scale to the right of the bucket.
4	x0	Location of the center of the bucket.
5	w1	Width of the left side cubic section.
6	w2	Width of the bucket.
7	w3	Width of the right side cubic section.

Parameters ****kwargs**

All parameters are passed to *Kernel*.

gptools.kernel.gibbs.**quintic_bucket_warp**(*x*, *n*, *l1*, *l2*, *l3*, *x0*, *w1*, *w2*, *w3*)

Warps the length scale with a piecewise quintic "bucket" shape.

Parameters **x** : float or array-like of float

Locations to evaluate length scale at.

n : non-negative int

Derivative order to evaluate. Only first derivatives are supported.

l1 : positive float

Length scale to the left of the bucket.

l2 : positive float

Length scale in the bucket.

l3 : positive float

Length scale to the right of the bucket.

x0 : float

Location of the center of the bucket.

w1 : positive float

Width of the left side quintic section.

w2 : positive float

Width of the bucket.

w3 : positive float

Width of the right side quintic section.

class gptools.kernel.gibbs.**GibbsKernel1dQuinticBucket**(***kwargs*)

Bases: *gptools.kernel.gibbs.GibbsKernel1d*

Gibbs warped squared exponential covariance function in 1d.

Uses hard-coded implementation up to first derivatives.

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 "bucket" function with quintic joins.

The order of the hyperparameters is:

0	sigmaf	Amplitude of the covariance function
1	l1	Length scale to the left of the bucket.
2	l2	Length scale in the bucket.
3	l3	Length scale to the right of the bucket.
4	x0	Location of the center of the bucket.
5	w1	Width of the left side quintic section.
6	w2	Width of the bucket.
7	w3	Width of the right side quintic section.

Parameters ****kwargs**

All parameters are passed to *Kernel*.

gptools.kernel.gibbs.**exp_gauss_warp**(*X*, *n*, *l0*, **msb*)

Length scale function which is an exponential of a sum of Gaussians.

The centers and widths of the Gaussians are free parameters.

The length scale function is given by

$$l = l_0 \exp\left(\sum_{i=1}^{N} eta_i \exp\left(-\right.\right.$$

rac{(x-mu_i)^2}{2sigma_i^2} ight) ight)

The number of parameters is equal to the three times the number of Gaussians plus 1 (for l_0). This function is inspired by what Gibbs used in his PhD thesis.

Parameters **X** : 1d or 2d array of float

The points to evaluate the function at. If 2d, it should only have one column (but this is not checked to save time).

n [int] The derivative order to compute. Used for all *X*.

l0 [float] The covariance length scale at the edges of the domain.

***msb** [floats] Means, standard deviations and weights for each Gaussian, in that order.

class gptools.kernel.gibbs.**GibbsKernel1dExpGauss**(*n_gaussians*, ***kwargs*)
Bases: *gptools.kernel.gibbs.GibbsKernel1d*

Gibbs warped squared exponential covariance function in 1d.

Uses hard-coded implementation up to first derivatives.

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 an exponential of Gaussian basis functions.

The order of the hyperparameters is:

0	sigmaf	Amplitude of the covariance function.
1	l0	Length far away from the Gaussians.
2	mu1	Mean of first Gaussian.
3	mu2	And so on for all Gaussians...
4	sigma1	Width of first Gaussian.
5	sigma2	And so on for all Gaussians...
6	beta1	Amplitude of first Gaussian.
7	beta2	And so on for all Gaussians...

Parameters **n_gaussians** : int

The number of Gaussian basis functions to use.

****kwargs**

All keywords are passed to *Kernel*.

class gptools.kernel.gibbs.**BSplineWarp**(*k=3*)
Bases: object

Length scale function which is a B-spline.

The degree is fixed at creation, the knot locations and coefficients are free parameters.

Parameters **k** : int, optional

The polynomial degree to use. Default is 3 (cubic).

__call__ (*X*, *n*, **tC*)

Evaluate the length scale function with the given knots and coefficients.

If X is 2d, uses the first column.

Parameters **X** : array of float, (*N*,)

The points to evaluate the length scale function at.

n : int

The derivative order to compute.

***tC** : *2M + k - 1* floats

The *M* knots followed by the *M + k - 1* coefficients to use.

class gptools.kernel.gibbs.**GibbsKernel1dBSpline**(*nt*, *k=3*, ***kwargs*)

Bases: *gptools.kernel.gibbs.GibbsKernel1d*

Gibbs warped squared exponential covariance function in 1d.

Uses hard-coded implementation up to first derivatives.

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 B-spline with free knots but fixed order.

You should always put fixed boundary knots at or beyond the edge of your domain, otherwise the length scale will go to zero. You should always use hyperpriors which keep the coefficients positive, otherwise the length scale can go to zero/be negative.

The order of the hyperparameters is:

0	sigmaf	Amplitude of the covariance function.
1	t1	First knot locations.
2	t2	And so on for all *nt* knots...
3	C1	First coefficient.
4	C2	And so on for all *nt + k - 1* coefficients...

Parameters **nt** : int

Number of knots to use. Should be at least two (at the edges of the domain of interest).

k : int, optional

The polynomial degree to use. Default is 3 (cubic).

****kwargs**

All keywords are passed to *Kernel*.

class gptools.kernel.gibbs.**GPWarp**(*npts*, *k=None*)

Bases: object

Length scale function which is a Gaussian process.

Parameters **npts** : int

The number of points the GP's value will be specified on.

k : *Kernel* instance, optional

The covariance kernel to use for the GP. The default is to use a *SquaredExponentialKernel* with fixed σ_f (since it does not matter here) and broad bounds on l_1.

__call__ (*X*, *n*, **hpXy*)
Evaluate the length scale.

Parameters **X** : array of float

The points to evaluate the length scale at.

n : int

The order of derivative to compute.

***hpXy** : floats

The free hyperparameters of the GP, then the points to set the value at, then the values to use.

class gptools.kernel.gibbs.**GibbsKernel1dGP**(*npts*, *k=None*, ***kwargs*)
Bases: *gptools.kernel.gibbs.GibbsKernel1d*

Gibbs warped squared exponential covariance function in 1d.

Uses hard-coded implementation up to first derivatives.

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 process which interpolates the values specified at a set number of points. Both the values and the locations of the points can be treated as free parameters.

You should try to pick a hyperprior which keeps the outer points close to the edge of the domain, as otherwise the Gaussian process will try to go to zero there. You should put a hyperprior on the values at the points which keeps them positive, otherwise unphysical length scales will result.

The order of the hyperparameters is:

0	sigmaf	Amplitude of the covariance function.
1	hp1	First hyperparameter of the covariance function.
2	hp2	And so on for all free hyperparameters...
3	x1	Location of the first point to set the value at.
4	x2	And so on for all points the value is set at...
5	y1	Length scale at the first point.
6	y2	And so on for all points the value is set at...

Parameters **npts** : int

Number of points to use.

k : *Kernel* instance, optional

The covariance kernel to use for the GP. The default is to use a *SquaredExponentialKernel* with fixed σ_f (since it does not matter here) and broad bounds on l_1.

****kwargs**

All keywords are passed to *Kernel*.

gptools.kernel.matern module

Provides the *MaternKernel* class which implements the anisotropic Matern kernel.

gptools.kernel.matern.**matern_function**(*Xi*, *Xj*, **args*)

Matern covariance function of arbitrary dimension, for use with ArbitraryKernel.

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 **Xi, Xj** : Array, mpf, tuple or scalar float

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

***args**

Remaining arguments are the 2+num_dim hyperparameters as defined above.

class gptools.kernel.matern.**MaternKernelArb**(***kwargs*)

Bases: *gptools.kernel.core.ArbitraryKernel*

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

This version of the Matern kernel is painfully slow, but uses mpmath to ensure the derivatives are computed properly, since there may be issues with the regular *MaternKernel*.

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 ****kwargs**

All keyword parameters are passed to *ArbitraryKernel*.

nu

Returns the value of the order ν.

class gptools.kernel.matern.**MaternKernel1d**(***kwargs*)

Bases: *gptools.kernel.core.Kernel*

Matern covariance kernel. Only for univariate data. Only supports up to first 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

The kernel is defined as:

$$k_M = \sigma^2 \frac{2^{1-\nu}}{\Gamma(\nu)} \left(\sqrt{2\nu \left(\frac{\tau^2}{l_1^2} \right)} \right)^{\nu} K_\nu \left(\sqrt{2\nu \left(\frac{\tau^2}{l_1^2} \right)} \right)$$

where $\tau = X_i - X_j$.

Note that the expressions for the derivatives break down for $\nu < 1$.

Parameters ****kwargs**

All keyword parameters are passed to *Kernel*.

__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*, *D*)

M inputs with dimension *D*.

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

M inputs with dimension *D*.

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

M derivative orders for set *i*.

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

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.

class gptools.kernel.matern.**MaternKernel**(*num_dim=1*, ***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.

class gptools.kernel.matern.**Matern52Kernel**(*num_dim=1*, ***kwargs*)

Bases: `gptools.kernel.core.Kernel`

Matern 5/2 covariance kernel. Supports only 0th and 1st derivatives and is fixed at nu=5/2. Because of these limitations, it is quite a bit faster than the more general Matern kernels.

The Matern52 kernel has the following hyperparameters, always referenced in the order listed:

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

The kernel is defined as:

$$k_M(x, x') = \sigma^2 \left(1 + \sqrt{5r^2} + \frac{5}{3} r^2 \right) \exp(-\sqrt{5r^2})$$

$$r^2 = \sum_{d=1}^{D} \frac{(x_d - x'_d)^2}{l_d^2}$$

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*, *D*)

M inputs with dimension *D*.

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

M inputs with dimension *D*.

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

M derivative orders for set *i*.

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

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.

gptools.kernel.noise module

Provides classes for implementing uncorrelated noise.

class `gptools.kernel.noise.`**`DiagonalNoiseKernel`**(*num_dim=1*, *initial_noise=None*, *fixed_noise=False*, *noise_bound=None*, *n=0*, *hyperprior=None*)

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, optional

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

fixed_noise : bool, optional

Whether or not the noise is taken to be fixed when optimizing the log likelihood. Default is False (noise is not fixed).

noise_bound : 2-tuple, optional

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*, optional

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

hyperprior : callable, optional

Function that returns the prior log-density for a possible value of noise when called. Must also have an attribute called `bounds` that is the bounds on the noise and a method called `random_draw()` that yields a random draw. Default behavior is to assign a uniform prior.

__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*, *D*)

M inputs with dimension *D*.

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

M inputs with dimension *D*.

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

M derivative orders for set *i*.

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

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=1*)
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*, *D*)

M inputs with dimension *D*.

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

M inputs with dimension *D*.

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

M derivative orders for set *i*.

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

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.

gptools.kernel.rational_quadratic module

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

class gptools.kernel.rational_quadratic.**RationalQuadraticKernel**(*num_dim=1*, ***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.

gptools.kernel.squared_exponential module

Provides the *SquaredExponentialKernel* class that implements the anisotropic SE kernel.

class gptools.kernel.squared_exponential.**SquaredExponentialKernel**(*num_dim=1*, ***kwargs*)

Bases: *gptools.kernel.core.Kernel*

Squared exponential covariance kernel. Supports arbitrary derivatives.

Supports derivatives with respect to the hyperparameters.

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:

$$k_{SE} = \sigma^2 \exp\left(-\frac{1}{2}\sum_i \frac{\tau_i^2}{l_i^2}\right)$$

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*, *D*)

M inputs with dimension *D*.

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

M inputs with dimension *D*.

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

M derivative orders for set *i*.

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

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.

gptools.kernel.warping module

Classes and functions to warp inputs to kernels to enable fitting of nonstationary data. Note that this accomplishes a similar goal as the Gibbs kernel (which is a nonstationary version of the squared exponential kernel), but with the difference that the warpings in this module can be applied to any existing kernel. Furthermore, whereas the Gibbs kernel implements nonstationarity by changing the length scale of the covariance function, the warpings in the module act by transforming the input values directly.

The module contains two core classes that work together. *WarpingFunction* gives you a way to wrap a given function in a way that allows you to optimize/integrate over the hyperparameters that describe the warping. *WarpedKernel* is an extension of the `Kernel` base class and is how you apply a *WarpingFunction* to whatever kernel you want to warp.

Two useful warpings have been implemented at this point: *BetaWarpedKernel* warps the inputs using the CDF of the beta distribution (i.e., the regularized incomplete beta function). This requires that the starting inputs be constrained to the unit hypercube [0, 1]. In order to get arbitrary data to this form, *LinearWarpedKernel* allows you to apply a linear transformation based on the known bounds of your data.

So, for example, to make a beta-warped squared exponential kernel, you simply type:

```
k_SE = gptools.SquaredExponentialKernel()
k_SE_beta = gptools.BetaWarpedKernel(k_SE)
```

Furthermore, if your inputs X are not confined to the unit hypercube [0, 1], you should use a linear transformation to map them to it:

```
k_SE_beta_unit = gptools.LinearWarpedKernel(k_SE_beta, X.min(axis=0), X.max(axis=0))
```

class `gptools.kernel.warping.`**`WarpingFunction`**(*fun*, *num_dim=1*, *num_params=None*, *initial_params=None*, *fixed_params=None*, *param_bounds=None*, *param_names=None*, *enforce_bounds=False*, *hyperprior=None*)

Bases: `object`

Wrapper to interface a function with *WarpedKernel*.

This lets you define a simple function *fun(X, d, n, p1, p2, ...)* that operates on one dimension of X at a time and has several hyperparameters.

Parameters **fun** : callable

Must have fingerprint *fun(X, d, n, p1, p2, ...)* where X is an array of length M, d is the index of the dimension X is from, n is a non-negative integer representing the order of derivative to take and *p1*, *p2*, ... are the parameters of the warping. Note that this form assumes that the warping is applied independently to each dimension.

num_dim : positive int, optional

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 warping function with. Default is 1.

num_params : Non-negative int, optional

Number of parameters in the model. Default is to determine the number of parameters by inspection of *fun* or the other arguments provided.

initial_params : Array, (*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 of bool, (*num_params*,), optional

Sets which hyperparameters 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 hyperparameters are fixed).

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

List of bounds for each of the hyperparameters. Each 2-tuple is of the form (lower‘, *upper*). If there is no bound in a given direction, it works best to set it to something big like 1e16. Default is (0.0, 1e16) for each hyperparameter. Note that this is overridden by the *hyperprior* keyword, if present.

param_names : list of str (*num_params*,), optional

List of labels for the hyperparameters. Default is all empty strings.

enforce_bounds : bool, optional

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

hyperprior : `JointPrior` instance or list, optional

Joint prior distribution for all hyperparameters. Can either be given as a `JointPrior` instance or a list of *num_params* callables or `rv_frozen` instances from `scipy.stats`, in which case a `IndependentJointPrior` is constructed with these as the independent priors on each hyperparameter. Default is a uniform PDF on all hyperparameters.

__call__ (*X*, *d*, *n*)

Evaluate the warping function.

Parameters X : Array, (*M*,)

M inputs corresponding to dimension *d*.

d : non-negative int

Index of the dimension that *X* is from.

n : non-negative int

Derivative order to compute.

param_bounds

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.

num_free_params

Returns the number of free parameters.

free_param_idxs

Returns the indices of the free parameters in the main arrays of parameters, etc.

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.

free_param_names
Returns the names of the free hyperparameters.

Returns free_param_names : Array

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

gptools.kernel.warping.**beta_cdf_warp**(*X*, *d*, *n*, **args*)
Warp inputs that are confined to the unit hypercube using the regularized incomplete beta function.

Applies separately to each dimension, designed for use with *WarpingFunction*.

Assumes that your inputs *X* lie entirely within the unit hypercube [0, 1].

Note that you may experience some issues with constraining and computing derivatives at $x = 0$ when $\alpha < 1$ and at $x = 1$ when $\beta < 1$. As a workaround, try mapping your data to not touch the boundaries of the unit hypercube.

Parameters X : array, (*M*,)

M inputs from dimension *d*.

d : non-negative int

The index (starting from zero) of the dimension to apply the warping to.

n : non-negative int

The derivative order to compute.

***args** : 2N scalars

The remaining parameters to describe the warping, given as scalars. These are given as *alpha_i*, *beta_i* for each of the *D* dimensions. Note that these must ALL be provided for each call.

References

[R1]

gptools.kernel.warping.**linear_warp**(*X*, *d*, *n*, **args*)
Warp inputs with a linear transformation.

Applies the warping

$$w(x) = \frac{x - a}{b - a}$$

to each dimension. If you set *a=min(X)* and *b=max(X)* then this is a convenient way to map your inputs to the unit hypercube.

Parameters X : array, (*M*,)

M inputs from dimension *d*.

d : non-negative int

The index (starting from zero) of the dimension to apply the warping to.

n : non-negative int

The derivative order to compute.

***args** : 2N scalars

The remaining parameters to describe the warping, given as scalars. These are given as *a_i*, *b_i* for each of the *D* dimensions. Note that these must ALL be provided for each call.

class gptools.kernel.warping.**ISplineWarp**(*nt*, *k=3*)

Bases: object

Warps inputs with an I-spline.

Applies the warping

$$w(x) = \sum_{i=1}^{nt+k-2} C_i I_{i,k}(x|t)$$

to each dimension, where C_i are the coefficients and $I_{i,k}(x|t)$ are the I-spline basis functions with knot grid t.

Parameters **nt** : int or array of int (*D*,)

The number of knots. If this is a single int, it is used for all of the dimensions. If it is an array of ints, it is the number of knots for each dimension.

k : int, optional

The polynomial degree of I-spline to use. The same degree is used for all dimensions. The default is 3 (cubic I-splines).

__call__(*X*, *d*, *n*, **args*)

Evaluate the I-spline warping function.

Parameters **X** : array, (*M*,)

M inputs from dimension *d*.

d : non-negative int

The index (starting from zero) of the dimension to apply the warping to.

n : non-negative int

The derivative order to compute.

***args** : scalar flots

The remaining parameters to describe the warping, given as scalars. These are given as the knots followed by the coefficients, for each dimension. Note that these must ALL be provided for each call.

class gptools.kernel.warping.**WarpedKernel**(*k*, *w*)

Bases: *gptools.kernel.core.Kernel*

Kernel which has had its inputs warped through a basic, elementwise warping function.

In other words, takes $k(x_1, x_2, x_1', x_2')$ and turns it into $k(w_1(x_1), w_2(x_2), w_1(x_1'), w_2(x_2'))$.

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

w_func(*X*, *d*, *n*)

Evaluate the (possibly recursive) warping function and its derivatives.

Parameters **X** : array, (M,)

The points (from dimension d) to evaluate the warping function at.

d : int

The dimension to warp.

n : int

The derivative order to compute. So far only 0 and 1 are supported.

enforce_bounds
Boolean indicating whether or not the kernel will explicitly enforce the bounds defined by the hyperprior.

fixed_params

params

param_names

free_params

free_param_bounds

free_param_names

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.warping.`**BetaWarpedKernel**(*k*, ***w_kwargs*)
Bases: *gptools.kernel.warping.WarpedKernel*

Class to warp any existing `Kernel` with the beta CDF.

Assumes that your inputs *X* lie entirely within the unit hypercube [0, 1].

Note that you may experience some issues with constraining and computing derivatives at $x = 0$ when $\alpha < 1$ and at $x = 1$ when $\beta < 1$. As a workaround, try mapping your data to not touch the boundaries of the unit hypercube.

Parameters **k** : `Kernel`

The `Kernel` to warp.

****w_kwargs** : optional kwargs

All additional kwargs are passed to the constructor of *WarpingFunction*. If no hyperprior or param_bounds are provided, takes each α, β to follow the log-normal distribution.

References

[R2]

class gptools.kernel.warping.**LinearWarpedKernel**(*k*, *a*, *b*)
Bases: *gptools.kernel.warping.WarpedKernel*

Class to warp any existing `Kernel` with the linear transformation given in *linear_warp()*.

If you set *a* to be the minimum of your *X* inputs in each dimension and *b* to be the maximum then you can use this to map data from an arbitrary domain to the unit hypercube [0, 1], as is required for application of the *BetaWarpedKernel*, for instance.

Parameters **k** : `Kernel`

The `Kernel` to warp.

a : list

The *a* parameter in the linear warping defined in *linear_warp()*. This list must have length equal to *k.num_dim*.

b : list

The *b* parameter in the linear warping defined in *linear_warp()*. This list must have length equal to *k.num_dim*.

class gptools.kernel.warping.**ISplineWarpedKernel**(*k*, *nt*, *k_deg=3*, ***w_kwargs*)
Bases: *gptools.kernel.warping.WarpedKernel*

Class to warp any existing `Kernel` with the I-spline transformation given in *ISplineWarp()*.

Parameters **k** : `Kernel`

The `Kernel` to warp.

nt : int or array of int

The number of knots. If this is a single int, it is used for all of the dimensions. If it is an array of ints, it is the number of knots for each dimension.

k_deg : int, optional

The polynomial degree of I-spline to use. The same degree is used for all dimensions. The default is 3 (cubic I-splines).

****w_kwargs** : optional kwargs

All additional keyword arguments are passed to the constructor of *WarpingFunction*.

Module contents

Subpackage containing a variety of covariance kernels and associated helpers.

4.1.2 Submodules

4.1.3 gptools.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.

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

Exception class raised when parameters are not possible.

4.1.4 gptools.gaussian_process module

Provides the base *GaussianProcess* class.

class gptools.gaussian_process.**GaussianProcess**(*k*, *noise_k=None*, *X=None*, *y=None*, *err_y=0*, *n=0*, *T=None*, *diag_factor=100.0*, *mu=None*, *use_hyper_deriv=False*, *verbose=False*)
Bases: object

Gaussian process.

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

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.

mu : *MeanFunction* instance

The mean function of the Gaussian process. Default is None (zero mean prior).

NOTE

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

X : array, (N, D), optional

M input values of dimension D. Default value is None (no data).

y : array, (M,), optional

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

err_y : array, (M,), optional

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

n : array, (*N*, *D*) or scalar float, optional

Non-negative integer values only. Degree of derivative for each 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.

T : array, (*M*, *N*), optional

Linear transformation to get from latent variables to data in the argument *y*. When *T* is passed the argument *y* holds the transformed quantities *y=TY(X)* where *y* are the observed values of the transformed quantities, *T* is the transformation matrix and *Y(X)* is the underlying (untransformed) values of the function to be fit that enter into the transformation. When *T* is *M*-by-*N* and *y* has *M* elements, *X* and *n* will both be *N*-by-*D*. Default is None (no transformation).

use_hyper_deriv : bool, optional

If True, the elements needed to compute the derivatives of the log-likelihood with respect to the hyperparameters will be computed. Default is False (do not compute these elements). Derivatives with respect to hyperparameters are currently only supported for the squared exponential covariance kernel.

verbose : bool, optional

If True, warnings which are produced internally but are not critical will not be generated. This does not control when a warning happens in a routine which is called, however. Default is False (do not produce warnings).

Raises **TypeError**

k or *noise_k* is not an instance of `Kernel`.

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.

Attributes

`num_dim`	The number of dimensions of the input data.
`hyperprior`	Combined hyperprior for the kernel, noise kernel and (if present) mean function.
`params`	Combined hyperparameters for the kernel, noise kernel and (if present) mean function.
`param_bounds`	Combined bounds for the hyperparameters for the kernel, noise kernel and (if present) mean function.
`param_names`	Combined names for the hyperparameters for the kernel, noise kernel and (if present) mean function.
`fixed_params`	Combined fixed hyperparameter flags for the kernel, noise kernel and (if present) mean function.
`free_params`	Combined free hyperparameters for the kernel, noise kernel and (if present) mean function.
`free_param_bounds`	Combined free hyperparameter bounds for the kernel, noise kernel and (if present) mean function.
`free_param_names`	Combined free hyperparameter names for the kernel, noise kernel and (if present) mean function.

k	(*Kernel* instance) The non-noise portion of the covariance kernel.
noise_k	(*Kernel* instance) The noise portion of the covariance kernel.
mu	(*MeanFunction* instance) Parametric mean function.
X	(array, (*M*, *D*)) The *M* training input values, each of which is of dimension *D*.
n	(array, (*M*, *D*)) The orders of derivatives that each of the *M* training points represent, indicating the order of derivative with respect to each of the *D* dimensions.
T	(array, (*M*, *N*)) The transformation matrix applied to the training data. If this is not None, *X* and *n* will be *N*-by-*D*.
y	(array, (*M*,)) The *M* training target values.
err_y	(array, (*M*,)) The uncorrelated, possibly heteroscedastic uncertainty in the *M* training input values.
K	(array, (*M*, *M*)) Covariance matrix between all of the training inputs.
noise_K	(array, (*M*, *M*)) Noise portion of the covariance matrix between all of the training inputs. Note that if `T` is present this is applied between all of the training quadrature points, so additional noise on the transformed observations cannot be inferred.
L	(array, (*M*, *M*)) Lower-triangular Cholesky decomposition of the total covariance matrix between all of the training inputs.
alpha	(array, (*M*, 1)) Solution to $K\alpha = y$.
ll	(float) Log-posterior density of the model.
diag_factor	(float) The factor of `sys.float_info.epsilon` which is added to the diagonal of the `K` matrix to improve stability.
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()*.
use_hyper_deriv	(bool) Whether or not the derivative of the log-posterior with respect to the hyperparameters should be computed/used.
verbose	(bool) Whether or not to print non-critical, internally-generated warnings.

hyperprior
Combined hyperprior for the kernel, noise kernel and (if present) mean function.

fixed_params
Combined fixed hyperparameter flags for the kernel, noise kernel and (if present) mean function.

params
Combined hyperparameters for the kernel, noise kernel and (if present) mean function.

param_bounds
Combined bounds for the hyperparameters for the kernel, noise kernel and (if present) mean function.

param_names
Combined names for the hyperparameters for the kernel, noise kernel and (if present) mean function.

free_params
Combined free hyperparameters for the kernel, noise kernel and (if present) mean function.

free_param_bounds
Combined free hyperparameter bounds for the kernel, noise kernel and (if present) mean function.

free_param_names
Combined free hyperparameter names for the kernel, noise kernel and (if present) mean function.

add_data (*X*, *y*, *err_y=0*, *n=0*, *T=None*)
Add data to the training data set of the GaussianProcess instance.

Parameters **X** : array, (*M*, *D*)

M input values of dimension *D*.

y : array, (*M*,)

M target values.

err_y : array, (*M*,) or scalar float, optional

Non-negative values only. Error given as standard deviation) in the *M* 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 : array, (*M*, *D*) or scalar float, optional

Non-negative integer values only. Degree of derivative for each 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.

T : array, (*M*, *N*), optional

Linear transformation to get from latent variables to data in the argument *y*. When *T* is passed the argument *y* holds the transformed quantities *y=TY(X)* where *y* are the observed values of the transformed quantities, *T* is the transformation matrix and *Y(X)* is the underlying (untransformed) values of the function to be fit that enter into the transformation. When *T* is *M*-by-*N* and *y* has *M* elements, *X* and *n* will both be *N*-by-*D*. Default is None (no transformation).

Raises **ValueError**

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

condense_duplicates()

Condense duplicate points using a transformation matrix.

This is useful if you have multiple non-transformed points at the same location or multiple transformed points that use the same quadrature points.

Won't change the GP if all of the rows of [X, n] are unique. Will create a transformation matrix T if necessary. Note that the order of the points in [X, n] will be arbitrary after this operation.

If there are any transformed quantities (i.e., *self.T* is not None), it will also remove any quadrature points for which all of the weights are zero (even if all of the rows of [X, n] are unique).

remove_outliers(*thresh=3*, ***predict_kwargs*)

Remove outliers from the GP with very simplistic outlier detection.

Removes points that are more than *thresh* * *err_y* away from the GP mean. Note that this is only very rough in that it ignores the uncertainty in the GP mean at any given point. But you should only be using this as a rough way of removing bad channels, anyways!

Returns the values that were removed and a boolean array indicating where the removed points were.

Parameters **thresh** : float, optional

The threshold as a multiplier times *err_y*. Default is 3 (i.e., throw away all 3-sigma points).

****predict_kwargs** : optional kwargs

All additional kwargs are passed to *predict()*. You can, for instance, use this to make it use MCMC to evaluate the mean. (If you don't use MCMC, then the current value of the hyperparameters is used.)

Returns **X_bad** : array

Input values of the bad points.

y_bad : array

Bad values.

err_y_bad : array

Uncertainties on the bad values.

n_bad : array

Derivative order of the bad values.

bad_idxs : array

Array of booleans with the original shape of X with True wherever a point was taken to be bad and subsequently removed.

T_bad : array

Transformation matrix of returned points. Only returned if `T` is not None for the instance.

optimize_hyperparameters(*method='SLSQP'*, *opt_kwargs={}*, *verbose=False*, *random_starts=None*, *num_proc=None*, *max_tries=1*)

Optimize the hyperparameters by maximizing the log-posterior.

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.

If `use_hyper_deriv` is True the optimizer will attempt to use the derivatives of the log-posterior with respect to the hyperparameters to speed up the optimization. Note that only the squared exponential covariance kernel supports hyperparameter derivatives at present.

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. 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.

random_starts : non-negative int, optional

Number of times to randomly perturb the starting guesses (distributed according to the hyperprior) in order to seek the global minimum. If None, then *num_proc* random starts will be performed. Default is None (do number of random starts equal to the number of processors allocated). Note that for *random_starts* != 0, the initial state of the hyperparameters is not actually used.

num_proc : non-negative int or None, optional

Number of processors to use with random starts. If 0, processing is not done in parallel. If None, all available processors are used. Default is None (use all available processors).

max_tries : int, optional

Number of times to run through the random start procedure if a solution is not found. Default is to only go through the procedure once.

predict(*Xstar*, *n=0*, *noise=False*, *return_std=True*, *return_cov=False*, *full_output=False*, *return_samples=False*, *num_samples=1*, *samp_kwargs={}*, *return_mean_func=False*, *use_MCMC=False*, *full_MC=False*, *rejection_func=None*, *ddof=1*, *output_transform=None*, ***kwargs*)

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, (*M*, *D*)

M test input values of dimension *D*.

n : array, (*M*, *D*) 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_std : bool, optional

Set to True to compute and return the standard deviation for the predictions, False to skip this step. Default is True (return tuple of (*mean*, *std*)).

return_cov : bool, optional

Set to True to compute and return the full covariance matrix for the predictions. This overrides the *return_std* keyword. If you want both the standard deviation and covariance matrix pre-computed, use the *full_output* keyword.

full_output : bool, optional

Set to True to return the full outputs in a dictionary with keys:

mean	mean of GP at requested points
std	standard deviation of GP at requested points
cov	covariance matrix for values of GP at requested points
samp	random samples of GP at requested points (only if *return_samples* is True)
mean_func	mean function of GP (only if *return_mean_func* is True)
cov_func	covariance of mean function of GP (zero if not using MCMC)
std_func	standard deviation of mean function of GP (zero if not using MCMC)
mean_without_func	mean of GP minus mean function of GP
cov_without_func	covariance matrix of just the GP portion of the fit
std_without_func	standard deviation of just the GP portion of the fit

return_samples : bool, optional

Set to True to compute and return samples of the GP in addition to computing the mean. Only done if *full_output* is True. Default is False.

num_samples : int, optional

Number of samples to compute. If using MCMC this is the number of samples per MCMC sample, if using present values of hyperparameters this is the number of samples actually returned. Default is 1.

samp_kwargs : dict, optional

Additional keywords to pass to `draw_sample()` if *return_samples* is True. Default is {}.

return_mean_func : bool, optional

Set to True to return the evaluation of the mean function in addition to computing the mean of the process itself. Only done if *full_output* is True and *self.mu* is not None. Default is False.

use_MCMC : bool, optional

Set to True to use `predict_MCMC()` to evaluate the prediction marginalized over the hyperparameters.

full_MC : bool, optional

Set to True to compute the mean and covariance matrix using Monte Carlo sampling of the posterior. The samples will also be returned if full_output is True. The sample mean and covariance will be evaluated after filtering through *rejection_func*, so conditional means and covariances can be computed. Default is False (do not use full sampling).

rejection_func : callable, optional

Any samples where this function evaluates False will be rejected, where it evaluates True they will be kept. Default is None (no rejection). Only has an effect if *full_MC* is True.

ddof : int, optional

The degree of freedom correction to use when computing the covariance matrix when *full_MC* is True. Default is 1 (unbiased estimator).

output_transform: array, (‘L‘, ‘M‘), optional

Matrix to use to transform the output vector of length M to one of length L. This can, for instance, be used to compute integrals.

****kwargs** : optional kwargs

All additional kwargs are passed to `predict_MCMC()` if *use_MCMC* is True.

Returns **mean** : array, (M,)

Predicted GP mean. Only returned if *full_output* is False.

std : array, (M,)

Predicted standard deviation, only returned if *return_std* is True, *return_cov* is False and *full_output* is False.

cov : array, (M, M)

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

full_output : dict

Dictionary with fields for mean, std, cov and possibly random samples and the mean function. Only returned if *full_output* is True.

Raises **ValueError**

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

plot(*X=None, n=0, ax=None, envelopes=[1, 3], base_alpha=0.375, return_prediction=False, return_std=True, full_output=False, plot_kwargs={}, **kwargs*)

Plots the Gaussian process using the current hyperparameters. Only for num_dim <= 2.

Parameters **X** : array-like (*M*,) or (*M*, *num_dim*), optional

The values to evaluate the Gaussian process at. If None, then 100 points between the minimum and maximum of the data's X are used for a univariate Gaussian process and a 50x50 grid is used for a bivariate Gaussian process. Default is None (use 100 points between min and max).

n : int or list, optional

The order of derivative to compute. For num_dim=1, this must be an int. For num_dim=2, this must be a list of ints of length 2. Default is 0 (don't take derivative).

ax : axis instance, optional

Axis to plot the result on. If no axis is passed, one is created. If the string 'gca' is passed, the current axis (from plt.gca()) is used. If X_dim = 2, the axis must be 3d.

envelopes: list of float, optional

+/-n*sigma envelopes to plot. Default is [1, 3].

base_alpha : float, optional

Alpha value to use for +/-1*sigma envelope. All other envelopes *env* are drawn with *base_alpha/env*. Default is 0.375.

return_prediction : bool, optional

If True, the predicted values are also returned. Default is False.

return_std : bool, optional

If True, the standard deviation is computed and returned along with the mean when *return_prediction* is True. Default is True.

full_output : bool, optional

Set to True to return the full outputs in a dictionary with keys:

mean	mean of GP at requested points
std	standard deviation of GP at requested points
cov	covariance matrix for values of GP at requested points
samp	random samples of GP at requested points (only if *return_sample* is True)

plot_kwargs : dict, optional

The entries in this dictionary are passed as kwargs to the plotting command used to plot the mean. Use this to, for instance, change the color, line width and line style.

****kwargs** : extra arguments for predict, optional

Extra arguments that are passed to `predict()`.

Returns **ax** : axis instance

The axis instance used.

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

Predicted GP mean. Only returned if *return_prediction* is True and *full_output* is False.

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

Predicted standard deviation, only returned if *return_prediction* and *return_std* are True and *full_output* is False.

full_output : dict

Dictionary with fields for mean, std, cov and possibly random samples. Only returned if *return_prediction* and *full_output* are True.

draw_sample(*Xstar*, *n=0*, *num_samp=1*, *rand_vars=None*, *rand_type='standard normal'*, *diag_factor=1000.0*, *method='cholesky'*, *num_eig=None*, *mean=None*, *cov=None*, *modify_sign=None*, ***kwargs*)

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

Note that this function draws samples from the GP given the current values for the hyperparameters (which may be in a nonsense state if you just created the instance or called a method that performs MCMC sampling). If you want to draw random samples from MCMC output, use the *return_samples* and *full_output* keywords to `predict()`.

Parameters **Xstar** : array, (*M*, *D*)

M test input values of dimension *D*.

n : array, (*M*, *D*) 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 : array, (*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. Note that, when *method* is 'eig', the eigenvalues are in *ascending* order. 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.

method : {'cholesky', 'eig'}, optional

Method to use for constructing the matrix square root. Default is 'cholesky' (use lower-triangular Cholesky decomposition).

- 'cholesky': Perform Cholesky decomposition on the covariance matrix: $K = LL^T$, use L as the matrix square root.
- 'eig': Perform an eigenvalue decomposition on the covariance matrix: $K = Q\Lambda Q^{-1}$, use $Q\Lambda^{1/2}$ as the matrix square root.

num_eig : int or None, optional

Number of eigenvalues to compute. Can range from 1 to *M* (the number of test points). If it is None, then all eigenvalues are computed. Default is None (compute all eigenvalues). This keyword only has an effect if *method* is 'eig'.

mean : array, (*M*,), optional

If you have pre-computed the mean and covariance matrix, then you can simply pass them in with the *mean* and *cov* keywords to save on having to call *predict()*.

cov : array, (*M*, *M*), optional

If you have pre-computed the mean and covariance matrix, then you can simply pass them in with the *mean* and *cov* keywords to save on having to call *predict()*.

modify_sign : {None, 'left value', 'right value', 'left slope', 'right slope', 'left concavity', 'right concavity'}, optional

If None (the default), the eigenvectors as returned by `scipy.linalg.eigh()` are used without modification. To modify the sign of the eigenvectors (necessary for some advanced use cases), set this kwarg to one of the following:

- 'left value': forces the first value of each eigenvector to be positive.
- 'right value': forces the last value of each eigenvector to be positive.
- 'left slope': forces the slope to be positive at the start of each eigenvector.
- 'right slope': forces the slope to be positive at the end of each eigenvector.
- 'left concavity': forces the second derivative to be positive at the start of each eigenvector.
- 'right concavity': forces the second derivative to be positive at the end of each eigenvector.

****kwargs** : optional kwargs

All extra keyword arguments are passed to *predict()* when evaluating the mean and covariance matrix of the GP.

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

Samples evaluated at the *M* points.

Raises **ValueError**

If rand_type or method is invalid.

update_hyperparameters(*new_params*, *hyper_deriv_handling='default'*, *exit_on_bounds=True*, *inf_on_error=True*)

Update the kernel's hyperparameters to the new parameters.

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

Note that if this method crashes and the *hyper_deriv_handling* keyword was used, it may leave `use_hyper_deriv` in the wrong state.

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

New parameters to use.

hyper_deriv_handling : {'default', 'value', 'deriv'}, optional

Determines what to compute and return. If 'default' and `use_hyper_deriv` is True then the negative log-posterior and the negative gradient of the log-posterior with respect to the hyperparameters is returned. If 'default' and `use_hyper_deriv` is False or 'value' then only the negative log-posterior is returned. If 'deriv' then only the negative gradient of the log-posterior with respect to the hyperparameters is returned.

exit_on_bounds : bool, optional

If True, the method will automatically exit if the hyperparameters are impossible given the hyperprior, without trying to update the internal state. This is useful during MCMC sampling and optimization. Default is True (don't perform update for impossible hyperparameters).

inf_on_error : bool, optional

If True, the method will return *scipy.inf* if the hyperparameters produce a linear algebra error upon trying to update the Gaussian process. Default is True (catch errors and return infinity).

Returns **-1*ll** : float

The updated log posterior.

-1*ll_deriv : array of float, (*num_params*,)

The gradient of the log posterior. Only returned if `use_hyper_deriv` is True or *hyper_deriv_handling* is set to 'deriv'.

compute_K_L_alpha_ll()

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.

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.

compute_Kij(*Xi*, *Xj*, *ni*, *nj*, *noise=False*, *hyper_deriv=None*, *k=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** : array, (*M*, *D*)

M input values of dimension *D*.

Xj : array, (*P*, *D*)

P input values of dimension *D*.

ni : array, (*M*, *D*), non-negative integers

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

nj : array, (*P*, *D*), 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, optional

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

k : `Kernel` instance, optional

The covariance kernel to used. Overrides *noise* if present.

Returns **Kij** : array, (*M*, *P*)

Covariance matrix between *Xi* and *Xj*.

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.

sample_hyperparameter_posterior(*nwalkers=200*, *nsamp=500*, *burn=0*, *thin=1*, *num_proc=None*, *sampler=None*, *plot_posterior=False*, *plot_chains=False*, *sampler_type='ensemble'*, *ntemps=20*, *sampler_a=2.0*, ***plot_kwargs*)

Produce samples from the posterior for the hyperparameters using MCMC.

Returns the sampler created, because storing it stops the GP from being pickleable. To add more samples to a previous sampler, pass the sampler instance in the *sampler* keyword.

Parameters **nwalkers** : int, optional

The number of walkers to use in the sampler. Should be on the order of several hundred. Default is 200.

nsamp : int, optional

Number of samples (per walker) to take. Default is 500.

burn : int, optional

This keyword only has an effect on the corner plot produced when *plot_posterior* is True and the flattened chain plot produced when *plot_chains* is True. To perform computations with burn-in, see `compute_from_MCMC()`. The number of samples to discard at the beginning of the chain. Default is 0.

thin : int, optional

This keyword only has an effect on the corner plot produced when *plot_posterior* is True and the flattened chain plot produced when *plot_chains* is True. To perform computations with thinning, see `compute_from_MCMC()`. Every *thin*-th sample is kept. Default is 1.

num_proc : int or None, optional

Number of processors to use. If None, all available processors are used. Default is None (use all available processors).

sampler : `Sampler` instance

The sampler to use. If the sampler already has samples, the most recent sample will be used as the starting point. Otherwise a random sample from the hyperprior will be used.

plot_posterior : bool, optional

If True, a corner plot of the posterior for the hyperparameters will be generated. Default is False.

plot_chains : bool, optional

If True, a plot showing the history and autocorrelation of the chains will be produced.

sampler_type : str, optional

The type of sampler to use. Valid options are "ensemble" (affine- invariant ensemble sampler) and "pt" (parallel-tempered ensemble sampler).

ntemps : int, optional

Number of temperatures to use with the parallel-tempered ensemble sampler.

sampler_a : float, optional

Scale of the proposal distribution.

plot_kwargs : additional keywords, optional

Extra arguments to pass to `plot_sampler()`.

compute_from_MCMC(*X*, *n=0*, *return_mean=True*, *return_std=True*, *return_cov=False*, *return_samples=False*, *return_mean_func=False*, *num_samples=1*, *noise=False*, *samp_kwargs={}*, *sampler=None*, *flat_trace=None*, *burn=0*, *thin=1*, ***kwargs*)

Compute desired quantities from MCMC samples of the hyperparameter posterior.

The return will be a list with a number of rows equal to the number of hyperparameter samples. The columns depend on the state of the boolean flags, but will be some subset of (mean, stddev, cov, samples),

in that order. Samples will be the raw output of *draw_sample()*, so you will need to remember to convert to an array and flatten if you want to work with a single sample.

Parameters **X** : array-like (*M*,) or (*M*, *num_dim*)

The values to evaluate the Gaussian process at.

n : non-negative int or list, optional

The order of derivative to compute. For num_dim=1, this must be an int. For num_dim=2, this must be a list of ints of length 2. Default is 0 (don't take derivative).

return_mean : bool, optional

If True, the mean will be computed at each hyperparameter sample. Default is True (compute mean).

return_std : bool, optional

If True, the standard deviation will be computed at each hyperparameter sample. Default is True (compute stddev).

return_cov : bool, optional

If True, the covariance matrix will be computed at each hyperparameter sample. Default is True (compute stddev).

return_samples : bool, optional

If True, random sample(s) will be computed at each hyperparameter sample. Default is False (do not compute samples).

num_samples : int, optional

Compute this many samples if *return_sample* is True. Default is 1.

noise : bool, optional

If True, noise is included in the predictions and samples. Default is False (do not include noise).

samp_kwargs : dict, optional

If *return_sample* is True, the contents of this dictionary will be passed as kwargs to *draw_sample()*.

sampler : `Sampler` instance or None, optional

`Sampler` instance that has already been run to the extent desired on the hyperparameter posterior. If None, a new sampler will be created with *sample_hyperparameter_posterior()*. In this case, all extra kwargs will be passed on, allowing you to set the number of samples, etc. Default is None (create sampler).

flat_trace : array-like (*nsamp*, *ndim*) or None, optional

Flattened trace with samples of the free hyperparameters. If present, overrides *sampler*. This allows you to use a sampler other than the ones from `emcee`, or to specify arbitrary values you wish to evaluate the curve at. Note that this WILL be thinned and burned according to the following two kwargs. "Flat" refers to the fact that you must have combined all chains into a single one. Default is None (use *sampler*).

burn : int, optional

The number of samples to discard at the beginning of the chain. Default is 0.

thin : int, optional

Every *thin*-th sample is kept. Default is 1.

num_proc : int, optional

The number of processors to use for evaluation. This is used both when calling the sampler and when evaluating the Gaussian process. If None, the number of available processors will be used. If zero, evaluation will proceed in parallel. Default is to use all available processors.

****kwargs** : extra optional kwargs

All additional kwargs are passed to `sample_hyperparameter_posterior()`.

Returns **out** : dict

A dictionary having some or all of the fields 'mean', 'std', 'cov' and 'samp'. Each entry is a list of array-like. The length of this list is equal to the number of hyperparameter samples used, and the entries have the following shapes:

mean	(*M*,)
std	(*M*,)
cov	(*M*, *M*)
samp	(*M*, *num_samples*)

compute_l_from_MCMC(*X*, *n=0*, *sampler=None*, *flat_trace=None*, *burn=0*, *thin=1*, ***kwargs*)

Compute desired quantities from MCMC samples of the hyperparameter posterior.

The return will be a list with a number of rows equal to the number of hyperparameter samples. The columns will contain the covariance length scale function.

Parameters **X** : array-like (*M*,) or (*M*, *num_dim*)

The values to evaluate the Gaussian process at.

n : non-negative int or list, optional

The order of derivative to compute. For num_dim=1, this must be an int. For num_dim=2, this must be a list of ints of length 2. Default is 0 (don't take derivative).

sampler : `Sampler` instance or None, optional

`Sampler` instance that has already been run to the extent desired on the hyperparameter posterior. If None, a new sampler will be created with `sample_hyperparameter_posterior()`. In this case, all extra kwargs will be passed on, allowing you to set the number of samples, etc. Default is None (create sampler).

flat_trace : array-like (*nsamp*, *ndim*) or None, optional

Flattened trace with samples of the free hyperparameters. If present, overrides *sampler*. This allows you to use a sampler other than the ones from `emcee`, or to specify arbitrary values you wish to evaluate the curve at. Note that this WILL be thinned and burned according to the following two kwargs. "Flat" refers to the fact that you must have combined all chains into a single one. Default is None (use *sampler*).

burn : int, optional

The number of samples to discard at the beginning of the chain. Default is 0.

thin : int, optional

Every *thin*-th sample is kept. Default is 1.

num_proc : int, optional

The number of processors to use for evaluation. This is used both when calling the sampler and when evaluating the Gaussian process. If None, the number of available processors will be used. If zero, evaluation will proceed in parallel. Default is to use all available processors.

****kwargs** : extra optional kwargs

All additional kwargs are passed to *sample_hyperparameter_posterior()*.

Returns **out** : array of float

Length scale function at the indicated points.

compute_w_from_MCMC(*X*, *n=0*, *sampler=None*, *flat_trace=None*, *burn=0*, *thin=1*, ***kwargs*)

Compute desired quantities from MCMC samples of the hyperparameter posterior.

The return will be a list with a number of rows equal to the number of hyperparameter samples. The columns will contain the warping function.

Parameters **X** : array-like (*M*,) or (*M*, *num_dim*)

The values to evaluate the Gaussian process at.

n : non-negative int or list, optional

The order of derivative to compute. For num_dim=1, this must be an int. For num_dim=2, this must be a list of ints of length 2. Default is 0 (don't take derivative).

sampler : `Sampler` instance or None, optional

`Sampler` instance that has already been run to the extent desired on the hyperparameter posterior. If None, a new sampler will be created with *sample_hyperparameter_posterior()*. In this case, all extra kwargs will be passed on, allowing you to set the number of samples, etc. Default is None (create sampler).

flat_trace : array-like (*nsamp*, *ndim*) or None, optional

Flattened trace with samples of the free hyperparameters. If present, overrides *sampler*. This allows you to use a sampler other than the ones from `emcee`, or to specify arbitrary values you wish to evaluate the curve at. Note that this WILL be thinned and burned according to the following two kwargs. "Flat" refers to the fact that you must have combined all chains into a single one. Default is None (use *sampler*).

burn : int, optional

The number of samples to discard at the beginning of the chain. Default is 0.

thin : int, optional

Every *thin*-th sample is kept. Default is 1.

num_proc : int, optional

The number of processors to use for evaluation. This is used both when calling the sampler and when evaluating the Gaussian process. If None, the number of available processors will be used. If zero, evaluation will proceed in parallel. Default is to use all available processors.

****kwargs** : extra optional kwargs

All additional kwargs are passed to *sample_hyperparameter_posterior()*.

Returns **out** : array of float

Length scale function at the indicated points.

predict_MCMC(*X*, *ddof=1*, *full_MC=False*, *rejection_func=None*, ***kwargs*)

Make a prediction using MCMC samples.

This is essentially a convenient wrapper of *compute_from_MCMC()*, designed to act more or less interchangeably with *predict()*.

Computes the mean of the GP posterior marginalized over the hyperparameters using iterated expectations. If *return_std* is True, uses the law of total variance to compute the variance of the GP posterior marginalized over the hyperparameters. If *return_cov* is True, uses the law of total covariance to compute the entire covariance of the GP posterior marginalized over the hyperparameters. If both *return_cov* and *return_std* are True, then both the covariance matrix and standard deviation array will be returned.

Parameters **X** : array-like (*M*,) or (*M*, *num_dim*)

The values to evaluate the Gaussian process at.

ddof : int, optional

The degree of freedom correction to use when computing the variance. Default is 1 (standard Bessel correction for unbiased estimate).

return_std : bool, optional

If True, the standard deviation is also computed. Default is True.

full_MC : bool, optional

Set to True to compute the mean and covariance matrix using Monte Carlo sampling of the posterior. The samples will also be returned if full_output is True. Default is False (don't use full sampling).

rejection_func : callable, optional

Any samples where this function evaluates False will be rejected, where it evaluates True they will be kept. Default is None (no rejection). Only has an effect if *full_MC* is True.

ddof : int, optional

****kwargs** : optional kwargs

All additional kwargs are passed directly to *compute_from_MCMC()*.

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.

4.1.5 gptools.gp_utils module

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

This module specifically contains utilities that need to interact directly with the GaussianProcess object, and hence can present circular import problems when incorporated in the main utils submodule.

gptools.gp_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.gp_utils.**slice_plot**(**args*, ***kwargs*)

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

Parameters **vals** : array, (*M*, *D*, *P*, ...)

Multidimensional array to visualize.

x_vals_1 : array, (*M*,)

Values along the first dimension.

x_vals_2 : array, (*D*,)

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.gp_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.

4.1.6 gptools.mean module

Provides classes for defining explicit, parametric mean functions.

To provide the necessary hooks to optimize/sample the hyperparameters, your mean function must be wrapped with *MeanFunction* before being passed to `GaussianProcess`. The function must have the calling fingerprint *fun(X, n, p1, p2, ...)*, where *X* is an array with shape *(M, N)*, *n* is a vector with length *D* and *p1*, *p2*, ... are the (hyper)parameters of the mean function, given as individual arguments.

class `gptools.mean.`**`MeanFunction`**(*fun*, *num_params=None*, *initial_params=None*, *fixed_params=None*, *param_bounds=None*, *param_names=None*, *enforce_bounds=False*, *hyperprior=None*)

Bases: `object`

Wrapper to turn a function into a form useable by `GaussianProcess`.

This lets you define a simple function *fun(X, n, p1, p2, ...)* that operates on an (*M*, *D*) array *X*, taking the derivatives indicated by the vector *n* with length *D* (one derivative order for each dimension). The function should evaluate this derivative at all points in *X*, returning an array of length *M*. *MeanFunction* takes care of looping over the different derivatives requested by `GaussianProcess`.

Parameters **fun** : callable

Must have fingerprint *fun(X, n, p1, p2, ...)* where *X* is an array with shape (*M*, *D*), *n* is an array of non-negative integers with length *D* representing the order of derivative orders to take for each dimension and *p1*, *p2*, ... are the parameters of the mean function.

num_params : Non-negative int, optional

Number of parameters in the model. Default is to determine the number of parameters by inspection of *fun* or the other arguments provided.

initial_params : Array, (*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 of bool, (*num_params*,), optional

Sets which hyperparameters 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 hyperparameters are fixed).

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

List of bounds for each of the hyperparameters. Each 2-tuple is of the form (lower‘, *upper*). If there is no bound in a given direction, it works best to set it to something big like 1e16. Default is (0.0, 1e16) for each hyperparameter. Note that this is overridden by the *hyperprior* keyword, if present.

param_names : list of str (*num_params*,), optional

List of labels for the hyperparameters. Default is all empty strings.

enforce_bounds : bool, optional

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

hyperprior : `JointPrior` instance or list, optional

Joint prior distribution for all hyperparameters. Can either be given as a `JointPrior` instance or a list of *num_params* callables or `rv_frozen` instances from `scipy.stats`, in which case a `IndependentJointPrior` is constructed with these as the independent priors on each hyperparameter. Default is a uniform PDF on all hyperparameters.

__call__ (*X*, *n*, *hyper_deriv=None*)

Evaluate the mean function at the given points with the current parameters.

Parameters X : array, (*N*,) or (*N*, *D*)

Points to evaluate the mean function at.

n : array, (*N*,) or (*N*, *D*)

Derivative orders for each point.

hyper_deriv : int or None, optional

Index of parameter to take derivative with respect to.

param_bounds

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.

num_free_params

Returns the number of free parameters.

free_param_idxs

Returns the indices of the free parameters in the main arrays of parameters, etc.

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.

free_param_names
Returns the names of the free hyperparameters.

Returns **free_param_names** : Array

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

gptools.mean.**constant**(*X*, *n*, *mu*, *hyper_deriv=None*)
Function implementing a constant mean suitable for use with *MeanFunction*.

class gptools.mean.**ConstantMeanFunction**(***kwargs*)
Bases: *gptools.mean.MeanFunction*

Class implementing a constant mean function suitable for use with GaussianProcess.

All kwargs are passed to *MeanFunction*. If you do not pass *hyperprior* or *param_bounds*, the hyperprior for the mean is taken to be uniform over [-1e3, 1e3].

gptools.mean.**mtanh**(*alpha*, *z*)
Modified hyperbolic tangent function mtanh(z; alpha).

Parameters **alpha** : float

The core slope of the mtanh.

z : float or array

The coordinate of the mtanh.

gptools.mean.**mtanh_profile**(*X*, *n*, *x0*, *delta*, *alpha*, *h*, *b*, *hyper_deriv=None*)
Profile used with the mtanh function to fit profiles, suitable for use with *MeanFunction*.

Only supports univariate data!

Parameters **X** : array, (*M*, 1)

The points to evaluate at.

n : array, (1,)

The order of derivative to compute. Only up to first derivatives are supported.

x0 : float

Pedestal center

delta : float

Pedestal halfwidth

alpha : float

Core slope

h : float

Pedestal height

b : float

Pedestal foot

hyper_deriv : int or None, optional

The index of the parameter to take a derivative with respect to.

class gptools.mean.**MtanhMeanFunction1d**(***kwargs*)
Bases: *gptools.mean.MeanFunction*

Profile with mtanh edge, suitable for use with GaussianProcess.

All kwargs are passed to *MeanFunction*. If *hyperprior* and *param_bounds* are not passed then the hyperprior is taken to be uniform over the following intervals:

x0	0.98	1.1
delta	0.0	0.1
alpha	-0.5	0.5
h	0	5
b	0	0.5

gptools.mean.**linear**(*X*, *n*, **args*, ***kwargs*)
Linear mean function of arbitrary dimension, suitable for use with *MeanFunction*.

The form is $m_0 * X[:, 0] + m_1 * X[:, 1] + \cdots + b$.

Parameters **X** : array, (*M*, *D*)

The points to evaluate the model at.

n : array of non-negative int, (*D*)

The derivative order to take, specified as an integer order for each dimension in *X*.

***args** : num_dim+1 floats

The slopes for each dimension, plus the constant term. Must be of the form *m0, m1, ..., b*.

class gptools.mean.**LinearMeanFunction**(*num_dim=1*, ***kwargs*)
Bases: *gptools.mean.MeanFunction*

Linear mean function suitable for use with GaussianProcess.

Parameters **num_dim** : positive int, optional

The number of dimensions of the input data. Default is 1.

****kwargs** : optional kwargs

All extra kwargs are passed to *MeanFunction*. If *hyperprior* and *param_bounds* are not specified, all parameters are taken to have a uniform hyperprior over [-1e3, 1e3].

4.1.7 gptools.utils module

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

class gptools.utils.**JointPrior**(*i=1.0*)
Bases: object

Abstract class for objects implementing joint priors over hyperparameters.

In addition to the abstract methods defined in this template, implementations should also have an attribute named *bounds* which contains the bounds (for a prior with finite bounds) or the 95%% interval (for a prior which is unbounded in at least one direction).

__call__(*theta*, *hyper_deriv=None*)
Evaluate the prior log-PDF at the given values of the hyperparameters, theta.

Parameters **theta** : array-like, (*num_params*,)

The hyperparameters to evaluate the log-PDF at.

hyper_deriv : int or None, optional

If present, return the derivative of the log-PDF with respect to the variable with this index.

random_draw(*size=None*)
Draw random samples of the hyperparameters.

Parameters **size** : None, int or array-like, optional

The number/shape of samples to draw. If None, only one sample is returned. Default is None.

sample_u(*q*)
Extract a sample from random variates uniform on $[0, 1]$.

For a univariate distribution, this is simply evaluating the inverse CDF. To facilitate efficient sampling, this function returns a *vector* of PPF values, one value for each variable. Basically, the idea is that, given a vector q of *num_params* values each of which is distributed uniformly on $[0, 1]$, this function will return corresponding samples for each variable.

Parameters **q** : array-like, (*num_params*,)

Values between 0 and 1 to evaluate inverse CDF at.

elementwise_cdf(*p*)
Convert a sample to random variates uniform on $[0, 1]$.

For a univariate distribution, this is simply evaluating the CDF. To facilitate efficient sampling, this function returns a *vector* of CDF values, one value for each variable. Basically, the idea is that, given a vector q of *num_params* values each of which is distributed according to the prior, this function will return variables uniform on $[0, 1]$ corresponding to each variable. This is the inverse operation to *sample_u()*.

Parameters **p** : array-like, (*num_params*,)

Values to evaluate CDF at.

__mul__(*other*)
Multiply two *JointPrior* instances together.

class gptools.utils.**CombinedBounds**(*l1*, *l2*)
Bases: object

Object to support reassignment of the bounds from a combined prior.

Works for any types of arrays.

Parameters **l1** : array-like

The first list.

l2 : array-like

The second list.

__getitem__(*pos*)
Get the item(s) at *pos*.

pos can be a basic slice object. But, the method is implemented by turning the internal array-like objects into lists, so only the basic indexing capabilities supported by the list data type can be used.

__setitem__(*pos*, *value*)
Set the item at location pos to value.

Only works for scalar indices.

__len__()
Get the length of the combined arrays.

__invert__()
Return the elementwise inverse.

__str__()
Get user-friendly string representation.

__repr__()
Get exact string representation.

class gptools.utils.**MaskedBounds**(*a*, *m*)
Bases: object

Object to support reassignment of free parameter bounds.

Parameters **a** : array

The array to be masked.

m : array of int

The indices in *a* which are to be accessible.

__getitem__(*pos*)
Get the item(s) at location *pos* in the masked array.

__setitem__(*pos*, *value*)
Set the item(s) at location *pos* in the masked array.

__len__()
Get the length of the masked array.

__str__()
Get user-friendly string representation.

__repr__()
Get exact string representation.

class gptools.utils.**ProductJointPrior**(*p1*, *p2*)
Bases: *gptools.utils.JointPrior*

Product of two independent priors.

Parameters **p1, p2: :py:class:'JointPrior' instances**

The two priors to merge.

i

bounds

__call__(*theta*, *hyper_deriv=None*)
Evaluate the prior log-PDF at the given values of the hyperparameters, theta.

The log-PDFs of the two priors are summed.

Parameters **theta** : array-like, (*num_params*,)

The hyperparameters to evaluate the log-PDF at.

sample_u(*q*)
Extract a sample from random variates uniform on $[0, 1]$.

For a univariate distribution, this is simply evaluating the inverse CDF. To facilitate efficient sampling, this function returns a *vector* of PPF values, one value for each variable. Basically, the idea is that, given a vector q of *num_params* values each of which is distributed uniformly on $[0, 1]$, this function will return corresponding samples for each variable.

Parameters **q** : array-like, (*num_params*,)

Values between 0 and 1 to evaluate inverse CDF at.

elementwise_cdf(*p*)

Convert a sample to random variates uniform on $[0, 1]$.

For a univariate distribution, this is simply evaluating the CDF. To facilitate efficient sampling, this function returns a *vector* of CDF values, one value for each variable. Basically, the idea is that, given a vector q of *num_params* values each of which is distributed according to the prior, this function will return variables uniform on $[0, 1]$ corresponding to each variable. This is the inverse operation to *sample_u()*.

Parameters **p** : array-like, (*num_params*,)

Values to evaluate CDF at.

random_draw(*size=None*)

Draw random samples of the hyperparameters.

The outputs of the two priors are stacked vertically.

Parameters **size** : None, int or array-like, optional

The number/shape of samples to draw. If None, only one sample is returned. Default is None.

class gptools.utils.**UniformJointPrior**(*bounds*, *ub=None*, ***kwargs*)

Bases: *gptools.utils.JointPrior*

Uniform prior over the specified bounds.

Parameters **bounds** : list of tuples, (*num_params*,)

The bounds for each of the random variables.

ub : list of float, (*num_params*,), optional

The upper bounds for each of the random variables. If present, *bounds* is then taken to be a list of float with the lower bounds. This gives *UniformJointPrior* a similar calling fingerprint as the other *JointPrior* classes.

__call__(*theta*, *hyper_deriv=None*)

Evaluate the prior log-PDF at the given values of the hyperparameters, theta.

Parameters **theta** : array-like, (*num_params*,)

The hyperparameters to evaluate the log-PDF at.

sample_u(*q*)

Extract a sample from random variates uniform on $[0, 1]$.

For a univariate distribution, this is simply evaluating the inverse CDF. To facilitate efficient sampling, this function returns a *vector* of PPF values, one value for each variable. Basically, the idea is that, given a vector q of *num_params* values each of which is distributed uniformly on $[0, 1]$, this function will return corresponding samples for each variable.

Parameters **q** : array of float

Values between 0 and 1 to evaluate inverse CDF at.

elementwise_cdf(*p*)
Convert a sample to random variates uniform on $[0, 1]$.

For a univariate distribution, this is simply evaluating the CDF. To facilitate efficient sampling, this function returns a *vector* of CDF values, one value for each variable. Basically, the idea is that, given a vector q of *num_params* values each of which is distributed according to the prior, this function will return variables uniform on $[0, 1]$ corresponding to each variable. This is the inverse operation to *sample_u()*.

Parameters **p** : array-like, (*num_params*,)

Values to evaluate CDF at.

random_draw(*size=None*)
Draw random samples of the hyperparameters.

Parameters **size** : None, int or array-like, optional

The number/shape of samples to draw. If None, only one sample is returned. Default is None.

class gptools.utils.**CoreEdgeJointPrior**(*bounds*, *ub=None*, ***kwargs*)
Bases: *gptools.utils.UniformJointPrior*

Prior for use with Gibbs kernel warping functions with an inequality constraint between the core and edge length scales.

__call__(*theta*, *hyper_deriv=None*)
Evaluate the prior log-PDF at the given values of the hyperparameters, theta.

Parameters **theta** : array-like, (*num_params*,)

The hyperparameters to evaluate the log-PDF at.

sample_u(*q*)
Extract a sample from random variates uniform on $[0, 1]$.

For a univariate distribution, this is simply evaluating the inverse CDF. To facilitate efficient sampling, this function returns a *vector* of PPF values, one value for each variable. Basically, the idea is that, given a vector q of *num_params* values each of which is distributed uniformly on $[0, 1]$, this function will return corresponding samples for each variable.

Parameters **q** : array of float

Values between 0 and 1 to evaluate inverse CDF at.

elementwise_cdf(*p*)
Convert a sample to random variates uniform on $[0, 1]$.

For a univariate distribution, this is simply evaluating the CDF. To facilitate efficient sampling, this function returns a *vector* of CDF values, one value for each variable. Basically, the idea is that, given a vector q of *num_params* values each of which is distributed according to the prior, this function will return variables uniform on $[0, 1]$ corresponding to each variable. This is the inverse operation to *sample_u()*.

Parameters **p** : array-like, (*num_params*,)

Values to evaluate CDF at.

random_draw(*size=None*)
Draw random samples of the hyperparameters.

Parameters **size** : None, int or array-like, optional

The number/shape of samples to draw. If None, only one sample is returned. Default is None.

class gptools.utils.**CoreMidEdgeJointPrior**(*bounds*, *ub=None*, ***kwargs*)
Bases: *gptools.utils.UniformJointPrior*

Prior for use with Gibbs kernel warping functions with an inequality constraint between the core, mid and edge length scales and the core-mid and mid-edge joins.

__call__(*theta*, *hyper_deriv=None*)
Evaluate the prior log-PDF at the given values of the hyperparameters, theta.

Parameters **theta** : array-like, (*num_params*,)

The hyperparameters to evaluate the log-PDF at.

sample_u(*q*)
Extract a sample from random variates uniform on $[0, 1]$.

For a univariate distribution, this is simply evaluating the inverse CDF. To facilitate efficient sampling, this function returns a *vector* of PPF values, one value for each variable. Basically, the idea is that, given a vector q of *num_params* values each of which is distributed uniformly on $[0, 1]$, this function will return corresponding samples for each variable.

Parameters **q** : array of float

Values between 0 and 1 to evaluate inverse CDF at.

elementwise_cdf(*p*)
Convert a sample to random variates uniform on $[0, 1]$.

For a univariate distribution, this is simply evaluating the CDF. To facilitate efficient sampling, this function returns a *vector* of CDF values, one value for each variable. Basically, the idea is that, given a vector q of *num_params* values each of which is distributed according to the prior, this function will return variables uniform on $[0, 1]$ corresponding to each variable. This is the inverse operation to *sample_u()*.

Parameters **p** : array-like, (*num_params*,)

Values to evaluate CDF at.

random_draw(*size=None*)
Draw random samples of the hyperparameters.

Parameters **size** : None, int or array-like, optional

The number/shape of samples to draw. If None, only one sample is returned. Default is None.

class gptools.utils.**IndependentJointPrior**(*univariate_priors*)
Bases: *gptools.utils.JointPrior*

Joint prior for which each hyperparameter is independent.

Parameters **univariate_priors** : list of callables or rv_frozen, (*num_params*,)

The univariate priors for each hyperparameter. Entries in this list can either be a callable that takes as an argument the entire list of hyperparameters or a frozen instance of a distribution from `scipy.stats`.

__call__(*theta*, *hyper_deriv=None*)
Evaluate the prior log-PDF at the given values of the hyperparameters, theta.

Parameters **theta** : array-like, (*num_params*,)

The hyperparameters to evaluate the log-PDF at.

bounds
The bounds of the random variable.

Set *self.i=0.95* to return the 95% interval if this is used for setting bounds on optimizers/etc. where infinite bounds may not be useful.

sample_u(*q*)
Extract a sample from random variates uniform on $[0, 1]$.

For a univariate distribution, this is simply evaluating the inverse CDF. To facilitate efficient sampling, this function returns a *vector* of PPF values, one value for each variable. Basically, the idea is that, given a vector q of *num_params* values each of which is distributed uniformly on $[0, 1]$, this function will return corresponding samples for each variable.

Parameters **q** : array of float

Values between 0 and 1 to evaluate inverse CDF at.

elementwise_cdf(*p*)
Convert a sample to random variates uniform on $[0, 1]$.

For a univariate distribution, this is simply evaluating the CDF. To facilitate efficient sampling, this function returns a *vector* of CDF values, one value for each variable. Basically, the idea is that, given a vector q of *num_params* values each of which is distributed according to the prior, this function will return variables uniform on $[0, 1]$ corresponding to each variable. This is the inverse operation to *sample_u()*.

Parameters **p** : array-like, (*num_params*,)

Values to evaluate CDF at.

random_draw(*size=None*)
Draw random samples of the hyperparameters.

Parameters **size** : None, int or array-like, optional

The number/shape of samples to draw. If None, only one sample is returned. Default is None.

class gptools.utils.**NormalJointPrior**(*mu*, *sigma*, ***kwargs*)
Bases: *gptools.utils.JointPrior*

Joint prior for which each hyperparameter has a normal prior with fixed hyper-hyperparameters.

Parameters **mu** : list of float, same size as *sigma*

Means of the hyperparameters.

sigma : list of float

Standard deviations of the hyperparameters.

__call__(*theta*, *hyper_deriv=None*)
Evaluate the prior log-PDF at the given values of the hyperparameters, theta.

Parameters **theta** : array-like, (*num_params*,)

The hyperparameters to evaluate the log-PDF at.

bounds
The bounds of the random variable.

Set *self.i=0.95* to return the 95% interval if this is used for setting bounds on optimizers/etc. where infinite bounds may not be useful.

sample_u(*q*)
Extract a sample from random variates uniform on $[0, 1]$.

For a univariate distribution, this is simply evaluating the inverse CDF. To facilitate efficient sampling, this function returns a *vector* of PPF values, one value for each variable. Basically, the idea is that, given a

vector q of *num_params* values each of which is distributed uniformly on $[0, 1]$, this function will return corresponding samples for each variable.

Parameters **q** : array of float

Values between 0 and 1 to evaluate inverse CDF at.

elementwise_cdf(*p*)

Convert a sample to random variates uniform on $[0, 1]$.

For a univariate distribution, this is simply evaluating the CDF. To facilitate efficient sampling, this function returns a *vector* of CDF values, one value for each variable. Basically, the idea is that, given a vector q of *num_params* values each of which is distributed according to the prior, this function will return variables uniform on $[0, 1]$ corresponding to each variable. This is the inverse operation to *sample_u()*.

Parameters **p** : array-like, (*num_params*,)

Values to evaluate CDF at.

random_draw(*size=None*)

Draw random samples of the hyperparameters.

Parameters **size** : None, int or array-like, optional

The number/shape of samples to draw. If None, only one sample is returned. Default is None.

class gptools.utils.**LogNormalJointPrior**(*mu*, *sigma*, ***kwargs*)

Bases: *gptools.utils.JointPrior*

Joint prior for which each hyperparameter has a log-normal prior with fixed hyper-hyperparameters.

Parameters **mu** : list of float, same size as *sigma*

Means of the logarithms of the hyperparameters.

sigma : list of float

Standard deviations of the logarithms of the hyperparameters.

__call__(*theta*, *hyper_deriv=None*)

Evaluate the prior log-PDF at the given values of the hyperparameters, theta.

Parameters **theta** : array-like, (*num_params*,)

The hyperparameters to evaluate the log-PDF at.

bounds

The bounds of the random variable.

Set *self.i=0.95* to return the 95% interval if this is used for setting bounds on optimizers/etc. where infinite bounds may not be useful.

sample_u(*q*)

Extract a sample from random variates uniform on $[0, 1]$.

For a univariate distribution, this is simply evaluating the inverse CDF. To facilitate efficient sampling, this function returns a *vector* of PPF values, one value for each variable. Basically, the idea is that, given a vector q of *num_params* values each of which is distributed uniformly on $[0, 1]$, this function will return corresponding samples for each variable.

Parameters **q** : array of float

Values between 0 and 1 to evaluate inverse CDF at.

elementwise_cdf(*p*)
Convert a sample to random variates uniform on $[0, 1]$.

For a univariate distribution, this is simply evaluating the CDF. To facilitate efficient sampling, this function returns a *vector* of CDF values, one value for each variable. Basically, the idea is that, given a vector q of *num_params* values each of which is distributed according to the prior, this function will return variables uniform on $[0, 1]$ corresponding to each variable. This is the inverse operation to *sample_u()*.

Parameters **p** : array-like, (*num_params*,)

Values to evaluate CDF at.

random_draw(*size=None*)
Draw random samples of the hyperparameters.

Parameters **size** : None, int or array-like, optional

The number/shape of samples to draw. If None, only one sample is returned. Default is None.

class gptools.utils.**GammaJointPrior**(*a*, *b*, ***kwargs*)
Bases: *gptools.utils.JointPrior*

Joint prior for which each hyperparameter has a gamma prior with fixed hyper-hyperparameters.

Parameters **a** : list of float, same size as *b*

Shape parameters.

b : list of float

Rate parameters.

__call__(*theta*, *hyper_deriv=None*)
Evaluate the prior log-PDF at the given values of the hyperparameters, theta.

Parameters **theta** : array-like, (*num_params*,)

The hyperparameters to evaluate the log-PDF at.

bounds
The bounds of the random variable.

Set *self.i=0.95* to return the 95% interval if this is used for setting bounds on optimizers/etc. where infinite bounds may not be useful.

sample_u(*q*)
Extract a sample from random variates uniform on $[0, 1]$.

For a univariate distribution, this is simply evaluating the inverse CDF. To facilitate efficient sampling, this function returns a *vector* of PPF values, one value for each variable. Basically, the idea is that, given a vector q of *num_params* values each of which is distributed uniformly on $[0, 1]$, this function will return corresponding samples for each variable.

Parameters **q** : array of float

Values between 0 and 1 to evaluate inverse CDF at.

elementwise_cdf(*p*)
Convert a sample to random variates uniform on $[0, 1]$.

For a univariate distribution, this is simply evaluating the CDF. To facilitate efficient sampling, this function returns a *vector* of CDF values, one value for each variable. Basically, the idea is that, given a vector q of *num_params* values each of which is distributed according to the prior, this function will return variables uniform on $[0, 1]$ corresponding to each variable. This is the inverse operation to *sample_u()*.

Parameters **p** : array-like, (*num_params*,)

Values to evaluate CDF at.

random_draw(*size=None*)
Draw random samples of the hyperparameters.

Parameters **size** : None, int or array-like, optional

The number/shape of samples to draw. If None, only one sample is returned. Default is None.

class gptools.utils.**GammaJointPriorAlt**(*m*, *s*, *i=1.0*)
Bases: *gptools.utils.GammaJointPrior*

Joint prior for which each hyperparameter has a gamma prior with fixed hyper-hyperparameters.

This is an alternate form that lets you specify the mode and standard deviation instead of the shape and rate parameters.

Parameters **m** : list of float, same size as *s*

Modes

s : list of float

Standard deviations

a

b

class gptools.utils.**SortedUniformJointPrior**(*num_var*, *lb*, *ub*, ***kwargs*)
Bases: *gptools.utils.JointPrior*

Joint prior for a set of variables which must be strictly increasing but are otherwise uniformly-distributed.

Parameters **num_var** : int

The number of variables represented.

lb : float

The lower bound for all of the variables.

ub : float

The upper bound for all of the variables.

__call__(*theta*, *hyper_deriv=None*)
Evaluate the log-probability of the variables.

Parameters **theta** : array

The parameters to find the log-probability of.

bounds

sample_u(*q*)
Extract a sample from random variates uniform on $[0, 1]$.

For a univariate distribution, this is simply evaluating the inverse CDF. To facilitate efficient sampling, this function returns a *vector* of PPF values, one value for each variable. Basically, the idea is that, given a vector q of *num_params* values each of which is distributed uniformly on $[0, 1]$, this function will return corresponding samples for each variable.

Parameters **q** : array of float

Values between 0 and 1 to evaluate inverse CDF at.

elementwise_cdf(*p*)
Convert a sample to random variates uniform on $[0, 1]$.

For a univariate distribution, this is simply evaluating the CDF. To facilitate efficient sampling, this function returns a *vector* of CDF values, one value for each variable. Basically, the idea is that, given a vector q of *num_params* values each of which is distributed according to the prior, this function will return variables uniform on $[0, 1]$ corresponding to each variable. This is the inverse operation to *sample_u()*.

Parameters **p** : array-like, (*num_params*,)

Values to evaluate CDF at.

random_draw(*size=None*)
Draw random samples of the hyperparameters.

Parameters **size** : None, int or array-like, optional

The number/shape of samples to draw. If None, only one sample is returned. Default is None.

gptools.utils.**wrap_fmin_slsqp**(*fun*, *guess*, *opt_kwargs={}*)
Wrapper for `fmin_slsqp()` to allow it to be called with `minimize()`-like syntax.

This is included to enable the code to run with `scipy` versions older than 0.11.0.

Accepts *opt_kwargs* in the same format as used by `scipy.optimize.minimize()`, with the additional precondition that the keyword *method* has already been removed by the calling code.

Parameters **fun** : callable

The function to minimize.

guess : sequence

The initial guess for the parameters.

opt_kwargs : dict, optional

Dictionary of extra keywords to pass to `scipy.optimize.minimize()`. Refer to that function's docstring for valid options. The keywords 'jac', 'hess' and 'hessp' are ignored. Note that if you were planning to use *jac* = True (i.e., optimization function returns Jacobian) and have set *args* = (True,) to tell `update_hyperparameters()` to compute and return the Jacobian this may cause unexpected behavior. Default is: {}.

Returns **Result** : namedtuple

`namedtuple` that mimics the fields of the `Result` object returned by `scipy.optimize.minimize()`. Has the following fields:

status	int	Code indicating the exit mode of the optimizer (*imode* from `fmin_slsqp()`)
success	bool	Boolean indicating whether or not the optimizer thinks a minimum was found.
fun	float	Value of the optimized function (-1*LL).
x	ndarray	Optimal values of the hyperparameters.
message	str	String describing the exit state (*smode* from `fmin_slsqp()`)
nit	int	Number of iterations.

Raises **ValueError**

Invalid constraint type in *constraints*. (See documentation for `scipy.optimize.minimize()`.)

`gptools.utils.`**`fixed_poch`**`(a, n)`

Implementation of the Pochhammer symbol $(a)_n$ which handles negative integer arguments properly.

Need conditional statement because scipy's impelementation of the Pochhammer symbol is wrong for negative integer arguments. This function uses the definition from http://functions.wolfram.com/GammaBetaErf/Pochhammer/02/

Parameters **a** : float

The argument.

n : nonnegative int

The order.

`gptools.utils.`**`Kn2Der`**`(nu, y, n=0)`

Find the derivatives of $K_\nu(y^{1/2})$.

Parameters **nu** : float

The order of the modified Bessel function of the second kind.

y : array of float

The values to evaluate at.

n : nonnegative int, optional

The order of derivative to take.

`gptools.utils.`**`yn2Kn2Der`**`(nu, y, n=0, tol=0.0005, nterms=1, nu_step=0.001)`

Computes the function $y^{\nu/2}K_\nu(y^{1/2})$ and its derivatives.

Care has been taken to handle the conditions at $y = 0$.

For *n=0*, uses a direct evaluation of the expression, replacing points where *y=0* with the appropriate value. For *n>0*, uses a general sum expression to evaluate the expression, and handles the value at *y=0* using a power series expansion. Where it becomes infinite, the infinities will have the appropriate sign for a limit approaching zero from the right.

Uses a power series expansion around $y = 0$ to avoid numerical issues.

Handles integer *nu* by performing a linear interpolation between values of *nu* slightly above and below the requested value.

Parameters **nu** : float

The order of the modified Bessel function and the exponent of *y*.

y : array of float

The points to evaluate the function at. These are assumed to be nonegative.

n : nonnegative int, optional

The order of derivative to take. Set to zero (the default) to get the value.

tol : float, optional

The distance from zero for which the power series is used. Default is 5e-4.

nterms : int, optional

The number of terms to include in the power series. Default is 1.

nu_step : float, optional

The amount to vary *nu* by when handling integer values of *nu*. Default is 0.001.

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.`**`powerset`**(*[1,2,3]) –> () (1,) (2,) (3,) (1,2) (1,3) (2,3) (1,2,3*)

From itertools documentation, https://docs.python.org/2/library/itertools.html.

`gptools.utils.`**`unique_rows`**(*arr*, *return_index=False*, *return_inverse=False*)

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.

return_index : bool, optional

If True, the indices of the unique rows in the array will also be returned. I.e., unique = arr[idx]. Default is False (don't return indices).

return_inverse: bool, optional

If True, the indices in the unique array to reconstruct the original array will also be returned. I.e., arr = unique[inv]. Default is False (don't return inverse).

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

The array *arr* with duplicate rows removed.

`gptools.utils.`**`compute_stats`**(*vals*, *check_nan=False*, *robust=False*, *axis=1*, *plot_QQ=False*, *bins=15*, *name=''*)

Compute the average statistics (mean, std dev) for the given values.

Parameters **vals** : array-like, (*M*, *D*)

Values to compute the average statistics along the specified axis of.

check_nan : bool, optional

Whether or not to check for (and exclude) NaN's. Default is False (do not attempt to handle NaN's).

robust : bool, optional

Whether or not to use robust estimators (median for mean, IQR for standard deviation). Default is False (use non-robust estimators).

axis : int, optional

Axis to compute the statistics along. Presently only supported if *robust* is False. Default is 1.

plot_QQ : bool, optional

Whether or not a QQ plot and histogram should be drawn for each channel. Default is False (do not draw QQ plots).

bins : int, optional

Number of bins to use when plotting histogram (for plot_QQ=True). Default is 15

name : str, optional

Name to put in the title of the QQ/histogram plot.

Returns **mean** : ndarray, (*M*,)

Estimator for the mean of *vals*.

std : ndarray, (*M*,)

Estimator for the standard deviation of *vals*.

Raises **NotImplementedError**

If *axis* != 1 when *robust* is True.

NotImplementedError

If *plot_QQ* is True.

gptools.utils.**univariate_envelope_plot**(*x, mean, std, ax=None, base_alpha=0.375, envelopes=[1, 3], lb=None, ub=None, expansion=10, **kwargs*)

Make a plot of a mean curve with uncertainty envelopes.

gptools.utils.**summarize_sampler**(*sampler*, *weights=None*, *burn=0*, *ci=0.95*, *chain_mask=None*)

Create summary statistics of the flattened chain of the sampler.

The confidence regions are computed from the quantiles of the data.

Parameters **sampler** : `emcee.Sampler` instance or array, (*n_temps*, *n_chains*, *n_samp*, *n_dim*), (*n_chains*, *n_samp*, *n_dim*) or (*n_samp*, *n_dim*)

The sampler to summarize the chains of.

weights : array, (*n_temps*, *n_chains*, *n_samp*), (*n_chains*, *n_samp*) or (*n_samp*,), optional

The weight for each sample. This is useful for post-processing the output from MultiNest sampling, for instance.

burn : int, optional

The number of samples to burn from the beginning of the chain. Default is 0 (no burn).

ci : float, optional

A number between 0 and 1 indicating the confidence region to compute. Default is 0.95 (return upper and lower bounds of the 95% confidence interval).

chain_mask : (index) array, optional

Mask identifying the chains to keep before plotting, in case there are bad chains. Default is to use all chains.

Returns **mean** : array, (num_params,)

Mean values of each of the parameters sampled.

ci_l : array, (num_params,)

Lower bounds of the *ci*100%* confidence intervals.

ci_u : array, (num_params,)

Upper bounds of the *ci*100%* confidence intervals.

gptools.utils.**plot_sampler**(*sampler, suptitle=None, labels=None, bins=50, plot_samples=False, plot_hist=True, plot_chains=True, burn=0, chain_mask=None, temp_idx=0, weights=None, cutoff_weight=None, cmap='gray_r', hist_color='k', chain_alpha=0.1, points=None, covs=None, colors=None, ci=[0.95], max_hist_ticks=None, max_chain_ticks=6, label_chain_y=False, hide_chain_yticklabels=False, chain_ytick_pad=2.0, label_fontsize=None, ticklabel_fontsize=None, chain_label_fontsize=None, chain_ticklabel_fontsize=None, xticklabel_angle=90.0, bottom_sep=0.075, suptitle_space=0.1, fixed_height=None, fixed_width=None, l=0.1, r=0.9, t1=None, b1=None, t2=0.2, b2=0.1, ax_space=0.1*)

Plot the results of MCMC sampler (posterior and chains).

Loosely based on triangle.py. Provides extensive options to format the plot.

Parameters **sampler** : `emcee.Sampler` instance or array, (*n_temps*, *n_chains*, *n_samp*, *n_dim*), (*n_chains*, *n_samp*, *n_dim*) or (*n_samp*, *n_dim*)

The sampler to plot the chains/marginals of. Can also be an array of samples which matches the shape of the *chain* attribute that would be present in a `emcee.Sampler` instance.

suptitle : str, optional

The figure title to place at the top. Default is no title.

labels : list of str, optional

The labels to use for each of the free parameters. Default is to leave the axes unlabeled.

bins : int, optional

Number of bins to use for the histograms. Default is 50.

plot_samples : bool, optional

If True, the samples are plotted as individual points. Default is False.

plot_hist : bool, optional

If True, histograms are plotted. Default is True.

plot_chains : bool, optional

If True, plot the sampler chains at the bottom. Default is True.

burn : int, optional

The number of samples to burn before making the marginal histograms. Default is zero (use all samples).

chain_mask : (index) array, optional

Mask identifying the chains to keep before plotting, in case there are bad chains. Default is to use all chains.

temp_idx : int, optional

Index of the temperature to plot when plotting a `emcee.PTSampler`. Default is 0 (samples from the posterior).

weights : array, (*n_temps*, *n_chains*, *n_samp*), (*n_chains*, *n_samp*) or (*n_samp*,), optional

The weight for each sample. This is useful for post-processing the output from MultiNest sampling, for instance. Default is to not weight the samples.

cutoff_weight : float, optional

If *weights* and *cutoff_weight* are present, points with *weights* < *cutoff_weight* * *weights.max()* will be excluded. Default is to plot all points.

cmap : str, optional

The colormap to use for the histograms. Default is 'gray_r'.

hist_color : str, optional

The color to use for the univariate histograms. Default is 'k'.

chain_alpha : float, optional

The transparency to use for the plots of the individual chains. Setting this to something low lets you better visualize what is going on. Default is 0.1.

points : array, (*D*,) or (*N*, *D*), optional

Array of point(s) to plot onto each marginal and chain. Default is None.

covs : array, (*D*, *D*) or (*N*, *D*, *D*), optional

Covariance matrix or array of covariance matrices to plot onto each marginal. If you do not want to plot a covariance matrix for a specific point, set its corresponding entry to *None*. Default is to not plot confidence ellipses for any points.

colors : array of str, (*N*,), optional

The colors to use for the points in *points*. Default is to use the standard matplotlib RGBCMYK cycle.

ci : array, (*num_ci*,), optional

List of confidence intervals to plot for each non-*None* entry in *covs*. Default is 0.95 (just plot the 95 percent confidence interval).

max_hist_ticks : int, optional

The maximum number of ticks for the histogram plots. Default is None (no limit).

max_chain_ticks : int, optional

The maximum number of y-axis ticks for the chain plots. Default is 6.

label_chain_y : bool, optional

If True, the chain plots will have y axis labels. Default is False.

hide_chain_yticklabels : bool, optional

If True, hide the y axis tick labels for the chain plots. Default is False (show y tick labels).

chain_ytick_pad : float, optional

The padding (in points) between the y-axis tick labels and the axis for the chain plots. Default is 2.0.

label_fontsize : float, optional

The font size (in points) to use for the axis labels. Default is *axes.labelsize*.

ticklabel_fontsize : float, optional

The font size (in points) to use for the axis tick labels. Default is *xtick.labelsize*.

chain_label_fontsize : float, optional

The font size (in points) to use for the labels of the chain axes. Default is *axes.labelsize*.

chain_ticklabel_fontsize : float, optional

The font size (in points) to use for the chain axis tick labels. Default is *xtick.labelsize*.

xticklabel_angle : float, optional

The angle to rotate the x tick labels, in degrees. Default is 90.

bottom_sep : float, optional

The separation (in relative figure units) between the chains and the marginals. Default is 0.075.

suptitle_space : float, optional

The amount of space (in relative figure units) to leave for a figure title. Default is 0.1.

fixed_height : float, optional

The desired figure height (in inches). Default is to automatically adjust based on *fixed_width* to make the subplots square.

fixed_width : float, optional

The desired figure width (in inches). Default is *figure.figsize[0]*.

l : float, optional

The location (in relative figure units) of the left margin. Default is 0.1.

r : float, optional

The location (in relative figure units) of the right margin. Default is 0.9.

t1 : float, optional

The location (in relative figure units) of the top of the grid of histograms. Overrides *suptitle_space* if present.

b1 : float, optional

The location (in relative figure units) of the bottom of the grid of histograms. Overrides *bottom_sep* if present. Defaults to 0.1 if *plot_chains* is False.

t2 : float, optional

The location (in relative figure units) of the top of the grid of chain plots. Default is 0.2.

b2 : float, optional

The location (in relative figure units) of the bottom of the grid of chain plots. Default is 0.1.

ax_space : float, optional

The *w_space* and *h_space* to use (in relative figure units). Default is 0.1.

gptools.utils.**plot_sampler_fingerprint**(*sampler*, *hyperprior*, *weights=None*, *cutoff_weight=None*, *nbins=None*, *labels=None*, *burn=0*, *chain_mask=None*, *temp_idx=0*, *points=None*, *plot_samples=False*, *sample_color='k'*, *point_color=None*, *point_lw=3*, *title=''*, *rot_x_labels=False*, *figsize=None*)

Make a plot of the sampler's "fingerprint": univariate marginal histograms for all hyperparameters.

The hyperparameters are mapped to [0, 1] using `hyperprior.elementwise_cdf()`, so this can only be used with prior distributions which implement this function.

Returns the figure and axis created.

Parameters **sampler** : `emcee.Sampler` instance or array, (*n_temps*, *n_chains*, *n_samp*, *n_dim*), (*n_chains*, *n_samp*, *n_dim*) or (*n_samp*, *n_dim*)

The sampler to plot the chains/marginals of. Can also be an array of samples which matches the shape of the *chain* attribute that would be present in a `emcee.Sampler` instance.

hyperprior : *JointPrior* instance

The joint prior distribution for the hyperparameters. Used to map the values to [0, 1] so that the hyperparameters can all be shown on the same axis.

weights : array, (*n_temps*, *n_chains*, *n_samp*), (*n_chains*, *n_samp*) or (*n_samp*,), optional

The weight for each sample. This is useful for post-processing the output from MultiNest sampling, for instance.

cutoff_weight : float, optional

If *weights* and *cutoff_weight* are present, points with *weights* < *cutoff_weight* * *weights.max()* will be excluded. Default is to plot all points.

nbins : int or array of int, (*D*,), optional

The number of bins dividing [0, 1] to use for each histogram. If a single int is given, this is used for all of the hyperparameters. If an array of ints is given, these are the numbers of bins for each of the hyperparameters. The default is to determine the number of bins using the Freedman-Diaconis rule.

labels : array of str, (*D*,), optional

The labels for each hyperparameter. Default is to use empty strings.

burn : int, optional

The number of samples to burn before making the marginal histograms. Default is zero (use all samples).

chain_mask : (index) array, optional

Mask identifying the chains to keep before plotting, in case there are bad chains. Default is to use all chains.

temp_idx : int, optional

Index of the temperature to plot when plotting a `emcee.PTSampler`. Default is 0 (samples from the posterior).

points : array, (*D*,) or (*N*, *D*), optional

Array of point(s) to plot as horizontal lines. Default is None.

plot_samples : bool, optional

If True, the samples are plotted as horizontal lines. Default is False.

sample_color : str, optional

The color to plot the samples in. Default is 'k', meaning black.

point_color : str or list of str, optional

The color to plot the individual points in. Default is to loop through matplotlib's default color sequence. If a list is provided, it will be cycled through.

point_lw : float, optional

Line width to use when plotting the individual points.

title : str, optional

Title to use for the plot.

rot_x_labels : bool, optional

If True, the labels for the x-axis are rotated 90 degrees. Default is False (do not rotate labels).

figsize : 2-tuple, optional

The figure size to use. Default is to use the matplotlib default.

gptools.utils.**plot_sampler_cov**(*sampler*, *method='corr'*, *weights=None*, *cutoff_weight=None*, *labels=None*, *burn=0*, *chain_mask=None*, *temp_idx=0*, *cbar_label=None*, *title=''*, *rot_x_labels=False*, *figsize=None*, *xlabel_on_top=True*)

Make a plot of the sampler's correlation or covariance matrix.

Returns the figure and axis created.

Parameters **sampler** : `emcee.Sampler` instance or array, (*n_temps*, *n_chains*, *n_samp*, *n_dim*), (*n_chains*, *n_samp*, *n_dim*) or (*n_samp*, *n_dim*)

The sampler to plot the chains/marginals of. Can also be an array of samples which matches the shape of the *chain* attribute that would be present in a `emcee.Sampler` instance.

method : {'corr', 'cov'}

Whether to plot the correlation matrix ('corr') or the covariance matrix ('cov'). The covariance matrix is often not useful because different parameters have wildly different scales. Default is to plot the correlation matrix.

labels : array of str, (*D*,), optional

The labels for each hyperparameter. Default is to use empty strings.

burn : int, optional

The number of samples to burn before making the marginal histograms. Default is zero (use all samples).

chain_mask : (index) array, optional

Mask identifying the chains to keep before plotting, in case there are bad chains. Default is to use all chains.

temp_idx : int, optional

Index of the temperature to plot when plotting a `emcee.PTSampler`. Default is 0 (samples from the posterior).

cbar_label : str, optional

The label to use for the colorbar. The default is chosen based on the value of the *method* keyword.

title : str, optional

Title to use for the plot.

rot_x_labels : bool, optional

If True, the labels for the x-axis are rotated 90 degrees. Default is False (do not rotate labels).

figsize : 2-tuple, optional

The figure size to use. Default is to use the matplotlib default.

xlabel_on_top : bool, optional

If True, the x-axis labels are put on top (the way mathematicians present matrices). Default is True.

4.1.8 Module contents

gptools - Gaussian process regression with support for arbitrary derivatives

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

[R1] J. Snoek, K. Swersky, R. Zemel, R. P. Adams, “Input Warping for Bayesian Optimization of Non-stationary Functions” ICML (2014)

[R2] J. Snoek, K. Swersky, R. Zemel, R. P. Adams, “Input Warping for Bayesian Optimization of Non-stationary Functions” ICML (2014)

g

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

W

Y

Z