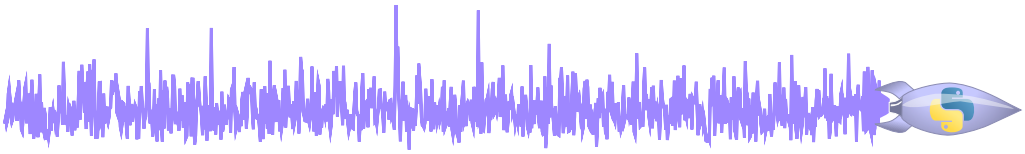
PyMC Gaussian process module User's guide

Anand Patil

January 29, 2010

CONTENTS



CHAPTER
ONE

The basics

Gaussian processes (GPs) are probability distributions for functions. In Bayesian statistics, they are often used as priors for functions whose forms are unknown. They can encode many types of knowledge about functions, yet remain much less restrictive than priors based on particular functional forms. GPs are not hard to understand at a conceptual level, but implementing them efficiently on a computer can require fairly involved linear algebra.

This package implements Gaussian processes as a set of Python classes that can support many types of usage, from intuitive exploration to embedding them in larger probability models and fitting them with MCMC.

All the code in the tutorial is in the folder '`pymc/examples/gp`' in the PyMC source tree.

1.1 Mathematical functions and Python functions

A mathematical function is a rule that associates a single output value with each input value in a set [**?**]. Examples are:

$$\begin{aligned} f(x) &= x^2 && \text{for each value } x \text{ in the real numbers} \\ f(x) &= \sin(x) && \text{for each value } x \text{ in the real numbers} \\ f(x,y) &= x^2 - (y-1)^2 && \text{for each pair of values } x, y, \text{ each in the real numbers.} \end{aligned}$$

Note the important distinction between the actual function f and the evaluation $f(x)$ of the function at a particular point. Each evaluation of a function is just a single output value, but a function itself is an (often infinite) table of values. If you 'look up' an input value in the table by evaluating the function, you'll find the corresponding output value.

One convenient way to visualize a function is by graphing it. The graphs of the first two functions above are curves. Their input and output values are single numbers, so the functions themselves are infinite tables of ordered pairs. The graph of the third function, on the other hand, is a surface. Its input value is a two-vector and its output value is a number, so the function itself is an infinite table of ordered triples. Ordered triples have to be plotted in three-dimensional space, so the graph ends up being a surface.

Python functions can emulate mathematical functions. Python representations of the mathematical functions above are:

```
def f(x):
    return x ** 2

def f(x):
    return sin(x)

def f(x,y):
    return x ** 2 - (y-1) ** 2
```

These Python functions act like very large tables of values. If you 'look up' an input value by passing it to the function as an argument, it will tell you the corresponding output value. It would be very inefficient to store output values corresponding to each input value on a computer, so the functions have to figure out the output values corresponding to arbitrary input values on demand.

1.2 A first look at Gaussian processes

Gaussian processes are probability distributions for mathematical functions. The statement 'random function f has a Gaussian process distribution with mean M and covariance C' is usually written as follows:

$$f \sim \mathrm{GP}(M, C). \tag{1.1}$$

Gaussian processes have two parameters, which are analogous to the parameters of the normal distribution:

- M is the mean function. Like the mean parameter of the normal distribution, M gives the central tendency for f. In Bayesian statistics, M is usually considered a prior guess for f.
- C is the covariance function. C takes twice as many arguments as f; if f is a function of one variable, C is a function of two variables. $C(x, y)$ gives the covariance of $f(x)$ and $f(y)$, and $C(x, x)$ gives the variance of $f(x)$. Its role is harder to understand than that of the mean function, but among other things it regulates:

- the amount by which f may deviate from M at any input value x,
- the roughnesss of f,
- the typical lengthscale of changes in f.

Section 1.3 will look at covariance functions in more depth; for the time being don't worry about them too much.

As with any probability distribution, random values can be drawn from a Gaussian process. However, these values (called 'realizations') are actually mathematical functions rather than the usual numbers or vectors. On the computer, the random values we draw will essentially be Python functions, with a few extra features.

1.2.1 What are Gaussian processes good for?

Mathematical functions are ubiquitous in science. A very short list of examples:

- Functional responses in predator-prey dynamics [**?**]. These functions associate a value for rate of prey capture with each value of predator population size.
- Transmission functions in epidemiology [**?**]. These functions associate a value for rate of new infections with each value of infected and uninfected population size.
- Transfer functions in engineering [**?**]. These functions associate a value for a ratio of Laplace transformed input and output signals with each value of the Laplace input variable s.
- Utility functions in microeconomics [**?**]. These functions associate a value for a person's satisfaction with each portfolio of goods.
- Action potentials in neuroscience [**?**]. These functions associate a value for transmembrane potential with each value of time since depolarization.
- The annual mean temperature at each point on the earth's surface in earth and atmospheric sciences.

The problem of estimating functions is just as widespread. In some cases, the phenomena underlying a function are understood well enough that the function can be derived up to a handful of parameters. For example, in Newtonian mechanics the height of a rock is known to be a parabolic function of the time since it was thrown, and the problem of inferring its trajectory reduces to the problem of inferring its initial height, its initial velocity and the acceleration due to gravity.

In many other cases it is not possible to deduce nearly as much about a function a priori. In some cases several candidate forms exist, but it is not possible to rule all of them out or even to ascertain that they are the only possibilities. As flexible and convenient probability distributions on function spaces, Gaussian processes are useful for Bayesian inference of functions without the need to reduce the problem to inference of a set of parameters.

1.2.2 Creating a Gaussian process

In the following subsections we will create objects representing a covariance function, a mean function, and finally several random functions drawn from the Gaussian process distribution defined by those objects.

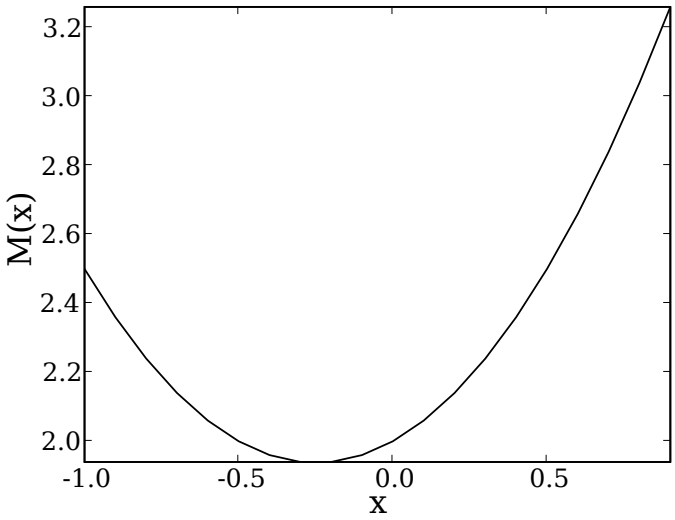


Figure 1.1: The mean function generated by 'examples/mean.py'.

Creating a mean function

The first component we will create is a mean function, represented by class `Mean`. The mean function of a univariate GP can be interpreted as a prior guess for the GP, so it is also a univariate function. The `Mean` class is a wrapper for an ordinary Python function. We will use the parabolic function

$$M(x) = ax^2 + bx + c. \tag{1.2}$$

The following code will produce an instance of class `Mean` called M:

```
from pymc.gp import *

# Generate mean
def quadfun(x, a, b, c):
    return (a * x ** 2 + b * x + c)

M = Mean(quadfun, a = 1., b = .5, c = 2.)

#### - Plot - ####
if __name__ == '__main__':
    from pylab import *
    x=arange(-1.,1.,.1)

    clf()
    plot(x, M(x), 'k-')
    xlabel('x')
    ylabel('M(x)')
    axis('tight')
    # show()
```

The first argument of `Mean`'s init method is the underlying Python function, in this case `quadfun`. The extra

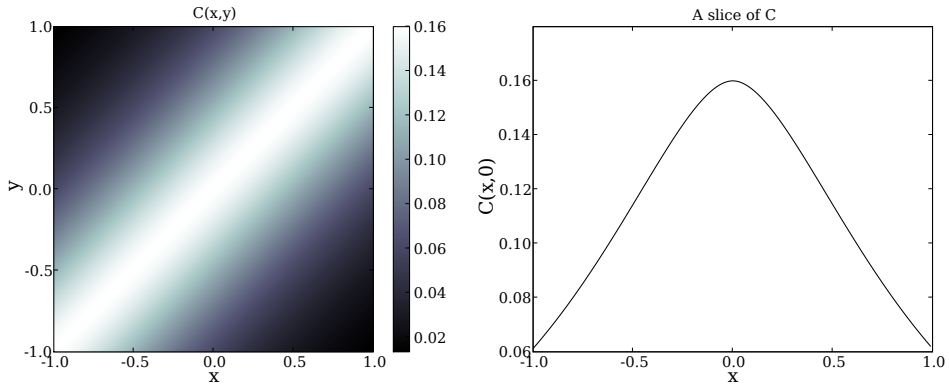


Figure 1.2: The covariance function generated by 'examples/cov.py'. On the left is the covariance function $C(x, y)$ evaluated over a square: $-1 \leq x \leq 1,\ -1 \leq y \leq 1$. On the right is a slice of the covariance: $C(x, 0)$ for $0 \leq x \leq 1$

arguments `a`, `b` and `c` will be memorized and passed to `quadfun` whenever M is called; the call `M(x)` in the plotting portion of the script does not need to pass them in.

Mean functions broadcast over their arguments in the same way as *NumPy universal functions* [10], which means that `M(x)` will return the vector

`[M(x[0]),..., M(x[N-1])].`

The last part of the code plots `M(x)` on $-1 < \mathtt{x} < 1$, and its output is shown in figure 1.1. As expected, the plot is a parabola.

Creating a covariance function

GP covariance functions are represented by the class `Covariance`, which like `Mean` is essentially a wrapper for ordinary Python functions. In this example we will use the popular Matèrn covariance function [3], which is provided in module `cov_funs`. In addition to the two arguments `x` and `y`, this function takes three tunable parameters: `amp` controls the amount by which realizations may deviate from their mean, `diff_degree` controls the roughness of realizations (the degree of differentiability), and `scale` controls the lengthscale over which realizations change.

You are free to write your own functions to wrap in `Covariance` objects. See section 3 for more information.

The code in 'examples/cov.py' will produce an instance of class `Covariance` called C.

```
from pymc.gp import *
from pymc.gp.cov_funs import matern
from numpy import *

C = Covariance(eval_fun = matern.euclidean, diff_degree = 1.4, amp = .4, scale = 1., rank_limit=100)
# C = FullRankCovariance(eval_fun = matern.euclidean, diff_degree = 1.4, amp = .4, scale = 1.)
# C = NearlyFullRankCovariance(eval_fun = matern.euclidean, diff_degree = 1.4, amp = .4, scale = 1.)

#### - Plot - ####
if __name__ == '__main__':
    from pylab import *
```

```
x=arange(-1.,1.,.01)
clf()

# Plot the covariance function
subplot(1,2,1)

contourf(x,x,C(x,x).view(ndarray),origin='lower',extent=(-1.,1.,-1.,1.),cmap=cm.bone)

xlabel('x')
ylabel('y')
title('C(x,y)')
axis('tight')
colorbar()

# Plot a slice of the covariance function
subplot(1,2,2)

plot(x,C(x,0).view(ndarray).ravel(),'k-')

xlabel('x')
ylabel('C(x,0)')
title('A slice of C')

# show()
```

The first argument to `Covariance`'s init method, `eval_fun`, gives the Python function from which the covariance function will be made. In this case, `eval_fun` is `matern.euclidean`. The extra arguments `diff_degree, amp` and `scale` will be passed to `matern.euclidean` every time C is called.

At this stage, the covariance function C exposes a very simple user interface. In fact, it behaves a lot like the ordinary Python function `matern.euclidean` that it wraps, except that like `Mean` it 'memorizes' the parameters `diff_degree, amp` and `scale` so that you don't need to pass them in when you call it. Covariance functions' calling conventions are slightly different from ordinary NumPy universal functions' [10]:

1. Broadcasting works differently. If C were a NumPy universal function, `C(x,y)` would return the following array:

   $$\texttt{[C(x[0],y[0])} \quad \ldots \quad \texttt{C(x[N-1],y[N-1])]},$$

   where `x` and `y` would need to be vectors of the same length. In fact `C(x,y)` returns a matrix:

   $$\begin{bmatrix} \texttt{C(x[0],y[0])} & \ldots & \texttt{C(x[0],y[Ny-1])} \\ \vdots & \ddots & \vdots \\ \texttt{C(x[Nx-1],y[0])} & \ldots & \texttt{C(x[Nx-1],y[Ny-1])} \end{bmatrix},$$

   and input arguments `x` and `y` don't need to be the same length.

2. You can call covariance functions with just one argument. `C(x)` returns

   $$\texttt{[C(x[0],x[0])} \quad \ldots \quad \texttt{C(x[Nx-1],x[Nx-1])]} = \texttt{diag(C(x,x))},$$

   but is computed much faster than `diag(C(x,x))` would be.

Most of the code in 'examples/cov.py' is devoted to output, which is shown in figure 1.2. It plots the covariance function `C(x,x)` evaluated over a square, and also the 'slice' `C(x,0)` over an interval. You'll notice that the graph of the full covariance function resembles a rounded A-frame tent.

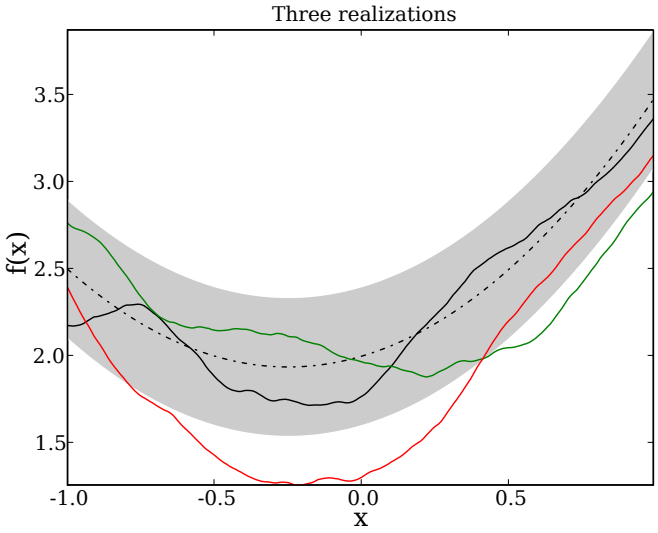


Figure 1.3: Three realizations from a Gaussian process displayed with mean ± 1 sd envelope. Generated by 'examples/realizations.py'.

Drawing realizations

Finally, we will generate some realizations (draws) from the Gaussian process defined by M and C and take a look at them. The following code will generate a list of `Realization` objects:

```
# Import the mean and covariance
from mean import M
from cov import C
from pymc.gp import *

# Generate realizations
f_list=[]
for i in range(3):
    f = Realization(M, C)
    f_list.append(f)

#### - Plot - ####
if __name__ == '__main__':
    from pylab import *

    x=arange(-1.,1.,.01)
    clf()

    plot_envelope(M, C, x)

    for f in f_list:
        plot(x, f(x))

    xlabel('x')
    ylabel('f(x)')
    title('Three realizations of the GP')
```

```
axis('tight')

# show()
```

The init method of `Realization` takes only two required arguments, a `Mean` object and a `Covariance` object. Each element of `f_list` is a Gaussian process realization, which is essentially a randomly-generated Python function. Like `Mean` objects, `Realization` objects use the same broadcasting rules as NumPy universal functions. Typing `f(x)` will return the vector

$$[\texttt{f(x[0])} \ldots \texttt{f(x[N-1])}].$$

The plotting portion of the code calls the function `plot_envelope`, which summarizes some aspects of the distribution. The dashdot black line in the middle is M, and the gray band is the ± 1 standard deviation envelope for f, generated by C. Each of the three realizations in `f_list` is a callable function, and they are plotted superimposed on the envelope. The plot output is shown in figure 1.3.

1.3 The role of the covariance function

The following covariance functions for Euclidean coordinates are included in the module `cov_funs`:

- `matern.euclidean`
- `sphere.euclidean`
- `pow_exp.euclidean`
- `gaussian.euclidean`
- `quadratic.euclidean`

See section 2.1.3 of *Banerjee et al.* [3] for more information on each of these. Each covariance function takes at least two parameters, called `amp` and `scale`. The following covariance functions take extra parameters:

`matern: diff_degree`

`pow_exp: pow`

`quadratic: phi`.

In this section we'll focus on the Matèrn family, as it is the 'go-to' covariance function for many applications. Its popularity is due to the fact that it has three parameters, each of which clearly controls one of three important properties of realizations: roughness, lengthscale of changes and amplitude.

In this section we will set the mean function to zero (more precisely, a function whose output value is zero regardless of input) in order to focus on the covariance. This:

$$f \sim \mathrm{GP}(M, C)$$

is equivalent to this:

$$\begin{aligned} f &= M + g, \\ g &\sim \mathrm{GP}(0, C), \end{aligned}$$

so it is not difficult to adapt the intuition we gain in this section to GPs with nontrivial mean functions.

The covariance functions listed above are *stationary* and *isotropic*. Intuitively, that means our a priori expectation of how f will deviate from its mean doesn't vary with location or with the direction in which we look (for functions of several variables). Section 3 describes how these restrictions can be relaxed.

All the figures in this section were produced using the file 'examples/cov_params.py'. You can follow along by editing the line of that file which reads

```
C = Covariance(eval_fun = matern.euclidean, diff_degree = 1.4, amp = 1., scale = 1.)
```

The actual formulas for the covariance functions are given in section 3, but the Matèrn formula is fairly inscrutable (though its Fourier transform is much more readable [**?**]). This section will try to help you understand it graphically.

The `amp` and `scale` parameters

These parameters are common to all covariance functions provided by this package, not just `matern.euclidean`. Please see section 3 if you are planning to write your own covariance functions, as this package provides utilities that will endow them with these parameters.

As mentioned in section 1.2.2, the covariance function plotted in figure 1.2 resembles a rounded A-frame tent. The width of this tent controls how tightly nearby evaluations of a realization f will be coupled to each other. If the tent is wide, $f(x)$ and $f(y)$ will tend to have similar values when x and y are close to one another. If the tent is narrower, $f(x)$ and $f(y)$ will not be as tightly correlated. The height of the tent controls the overall amplitude of f's deviation from its mean.

The mathematical definition of the covariance function is as follows:

$$C(x, y) = \mathrm{cov}(f(x), f(y)). \tag{1.3}$$

That implies the following:

$$\mathrm{var}(f(x)) = C(x, x).$$

By the definition of variance, for any real number a

$$\mathrm{var}(af(x)) = a^2 C(x, x).$$

The covariance function is multiplied by $\texttt{amp}^2$, and this effectively multiplies realizations by `amp`. In other words, a larger `amp` parameter means that realizations will deviate further from their mean. The effects of changing the `amp` parameter are illustrated in figure 1.4.

Whereas the `amp` parameter stretches the y axis, the `scale` parameter stretches the x axis. Say C's value can be computed from the difference in its arguments, $x - y$, rather than the arguments themselves. In that case, substituting $(x - y)/s$ for $x - y$, where s is greater than 1, will make the input points 'appear' closer together.

Whenever a call $C(x, y)$ is made, the distances between elements of the input arrays x and y are divided by the `scale` parameter before being passed to the underlying covariance function. In our one-dimensional example this effectively stretches the realizations in the x direction. If `scale` is large, the function will be correlated over a larger distance and will not 'wiggle' as quickly. Figure 1.5 illustrates the effects of the scale parameter.

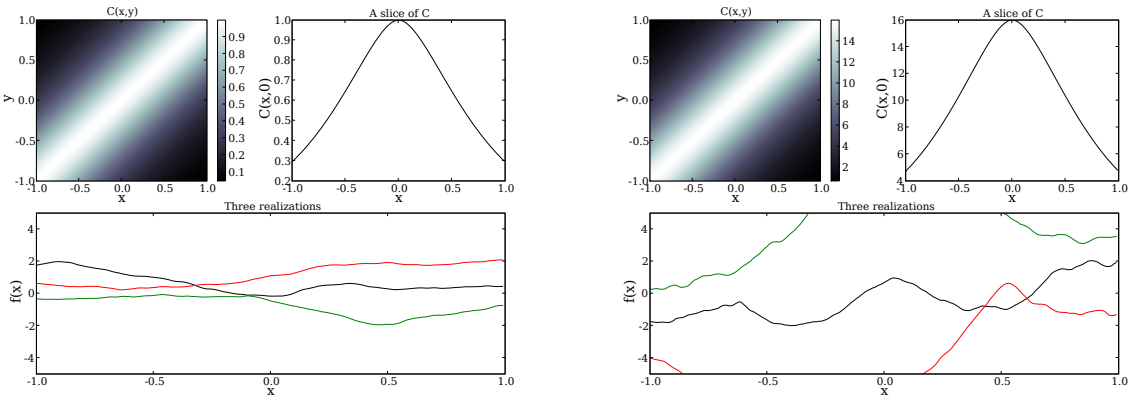


Figure 1.4: Matèrn covariances with `diff_degree=1.4`, `scale=1`, and `amp=1` (left) and `amp=4` (right), and corresponding realizations. The amplitude of realizations tends to scale with `amp`, and the amplitude of the covariance function scales with `amp`^{2}. Note the scales on the plots of the covariance functions.

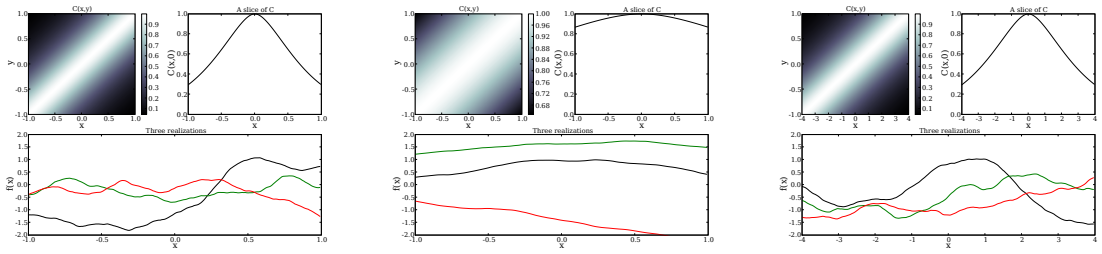


Figure 1.5: Matèrn covariances with `diff_degree=1.4`, `amp=1` and `scale=1` (left) and `scale=4` (center and right), and corresponding realizations. A larger value of `scale` stretches both covariance functions and realizations (center), so that realizations don't wiggle as rapidly. When the stretched functions are plotted on commensurately stretched axes (right), they look like the unstretched functions again.

1.3.1 The `diff_degree` parameter

The `diff_degree` parameter, denoted ν, is unique (amongst the covariance functions provided by this package) to the Matèrn family of covariance functions. It controls the sharpness of the ridge of the covariance function, which controls the roughness/smoothness of realizations.

More specifically, look at the slices $C(x, 0)$ that are shown in the upper right-hand panels of the subfigures in figures 1.4 and 1.5. If `diff_degree` is greater than an integer n, this slice is $2n$ times differentiable at $x = 0$. It turns out that this means realizations will be n times differentiable.

It is natural to ask what happens when `diff_degree` is not an integer. There is such a thing as *fractional calculus*, which deals with things such as taking half a derivative. Stein [**?**] discusses the connection briefly. See also Miller and Ross [**?**].

For our purposes, it is safe to say that `diff_degree` is a roughness index that can be interpreted as a degree of differentiability when it is an integer. Figure 1.6 illustrates the effects of changing this parameter:

0 (not shown): $C(x, y) = 1$ if $y = x$, 0 if `x` $\neq$ `y`. Not only are realizations not differentiable, they are not continuous. $f(x)$ is an independent normal random variable for each value of x.

.2: $C(x, 0)$ is not even one time differentiable at its peak, where $x = 0$. Realizations are very rough, but continuous.

.5: If `diff_degree` is just larger than .5, $C(x, 0)$ is differentiable at $x = 0$. Realizations, however, are not differentiable; their roughness is comparable to trajectories of Brownian particles. When `diff_degree` is equal to `.5`, `matern` is equivalent to another covariance function, `pow_exp`, with extra argument `pow=1`.

1: If `diff_degree` is just larger than 1, $C(x, 0)$ is twice differentiable at $x = 0$, and realizations are differentiable.

1.4: The value from figures 1.4, 1.2 and 1.5 is shown for comparison.

2: If `diff_degree` is just larger than 2, $C(x, 0)$ is four times differentiable at $x = 0$, and realizations are twice differentiable.

10: Realizations are very smooth. As `diff_degree` approaches infinity, `matern` gets closer to `gaussian`. Realizations from GPs with Gaussian covariances are infinitely differentiable. In fact, if `diff_degree` is larger than 10, `matern` simply calls `gaussian`, as it is much faster.

1.3.2 Suggestions for further experimentation

- Change the parameters of the Matèrn covariance function, and try to guess how realizations will look.
- Differentiate realizations numerically. Recall that

$$\frac{df}{dx} = \lim_{h \to 0} \frac{f(x + h) - f(x)}{h},$$

  so you can take an approximate 'numerical derivative' by evaluating the fraction on the right hand side with h equal to a small number.
- Replace `matern.euclidean` in 'examples/cov_params.py' with the Euclidean version of one of the other covariance functions and experiment with the parameters. See Banerjee [3] for their interesting properties.
- Replace `zero_fun` in 'cov_params.py' with a nontrivial mean function and repeat the above.

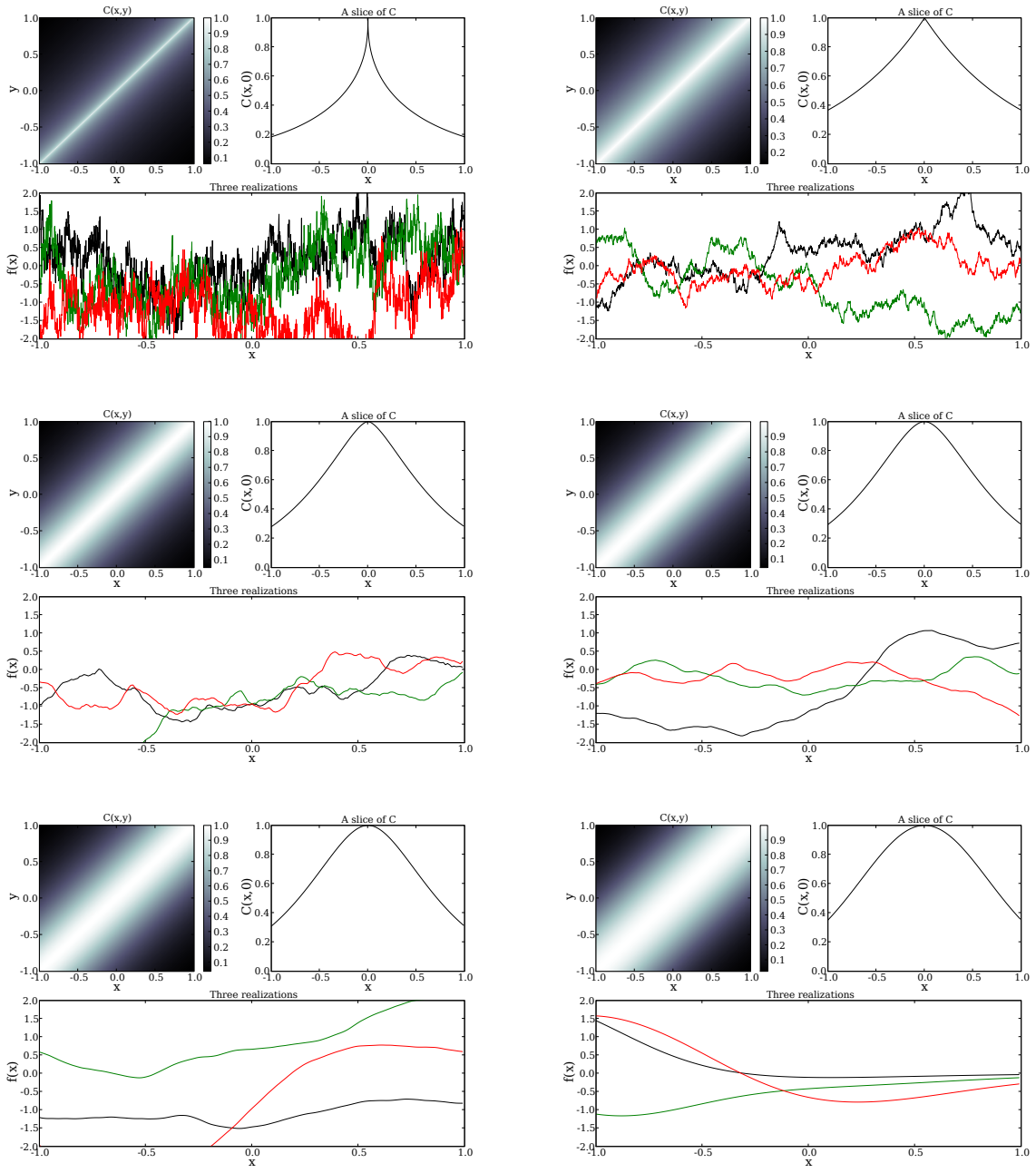


Figure 1.6: Matern draws for various `diff_degree` parameters. In increasing orders of smoothness: `.2`, `.5`, (equivalent to `pow_exp` with `pow=1`, roughness similar to Brownian motion), `1`, `1.4` (the examples given so far), `2`, `10` (nearly equivalent to `gaussian`).

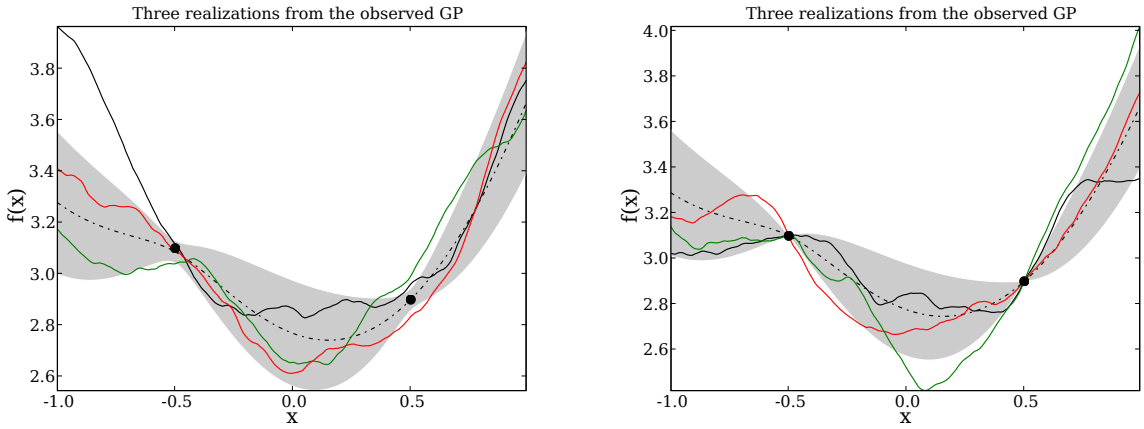


Figure 1.7: The output of 'examples/observations.py': the observed GP with `obs_V = .002` (left) and `obs_V = 0` (right). Note that in the conditioned case, the ± 1 SD envelope shrinks to zero at the points where the observations were made, and all realizations pass through the observed values. Compare these plots to those in figure 1.3.

1.4 Nonparametric regression: observing Gaussian processes

Consider the following common statistical situation: You decide on a GP prior for an unknown function f, then you observe the value of f at N input points $[o_0 \dots o_{N-1}]$, possibly with uncertainty. If the observation error is normally distributed, it turns out that f's posterior distribution given the new information is another Gaussian process, with new mean and covariance functions.

The probability model that represents this situation is as follows:

$$\left.\begin{array}{l} \text{data}_i \overset{\text{ind}}{\sim} \text{N}(f(o_i), V_i) \\ f \sim \text{GP}(M, C) \end{array}\right\} \Rightarrow f|\text{data} \sim \text{GP}(M_o, C_o). \tag{1.4}$$

This package provides a function called `observe` that imposes normally-distributed observations on Gaussian process distributions. This function converts f's prior to its posterior by transforming M and C in equation 1.4 to M_o and C_o:

The following code (from 'observation.py') imposes the observations

$$f(-.5) = 3.1$$
$$f(.5) = 2.9$$

with observation variance $V = .002$ on the GP distribution defined in 'mean.py' and 'cov.py':

```
# Import the mean and covariance
from mean import M
from cov import C
from pymc.gp import *
from numpy import *

# Impose observations on the GP
obs_x = array([-.5,.5])
V = array([.002,.002])
```

```
data = array([3.1, 2.9])
observe(M=M,
        C=C,
        obs_mesh=obs_x,
        obs_V = V,
        obs_vals = data)

# Generate realizations
f_list=[]
for i in range(3):
    f = Realization(M, C)
    f_list.append(f)

x=arange(-1.,1.,.01)

#### - Plot - ####
if __name__ == '__main__':
    from pylab import *

    x=arange(-1.,1.,.01)

    clf()

    plot_envelope(M, C, mesh=x)

    for f in f_list:
        plot(x, f(x))

    xlabel('x')
    ylabel('f(x)')
    title('Three realizations of the observed GP')
    axis('tight')


    # show()
```

The function `observe` takes a covariance C and a mean M as arguments, and essentially tells them that their realizations' values on `obs_mesh` have been observed to be `obs_vals` with variance `obs_V`. If `obs_V` is `None`, `observe` assumes that the observation precision was infinite; i.e. the realizations' values on `obs_mesh` were observed with no uncertainty. Making (or pretending to make) observations with infinite precision is sometimes called *conditioning*, and can be a valuable tool for modifying GP priors; for example, if a rate function is known to be zero when a population's size is zero.

The output of the code is shown in figure 1.7, along with the output with `obs_V=None`. Compare these to the analogous figure for the unobserved GP, figure 1.3. The covariance after observation is visualized in figure 1.8. The covariance 'tent' has been pressed down at points where $x \approx \pm.5$ and/or $y \approx \pm.5$, which are the values where the observations were made.

1.4.1 Example: Salmonid stock-recruitment functions

Munch, Kottas and Mangel [**?**] use Gaussian process priors to infer various *stock-recruitment (SR) functions*. An important concept in fishery science, SR functions relate the size of a fish stock to the number or biomass of recruits to the fishery each year. In other words, they relate population size or biomass to number or biomass of new fish produced. The authors argue that model uncertainty is endemic in stock-recruitment

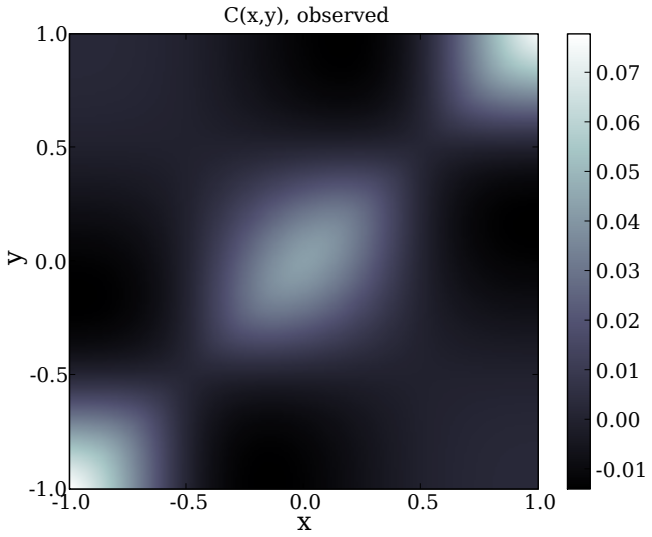


Figure 1.8: The covariance function from 'observation.py' after observation. Compare this with the covariance function before observation, visualized in figure 1.2

theory, and that in this situation GP priors are a sensible alternative to particular functional forms.

We do not have the tools yet to fully duplicate Munch, Kottas and Mangel's results; this will have to wait until chapter 2. However, we can fit a simpler version of their model now by using the `observe` function. Specifically, we will fit the data in figure 6 of their paper [**?**], which is for three salmonids: chum (*Onchorhynchus keta*), pink (*Onchorhynchus gorbuscha*) and sockeye (*Onchorhynchus nerka*).

The code is in the script 'examples/gp/more_examples/MMKsalmon/regression.py'. The script begins by importing the `salmon` class from 'salmon.py' in the same directory. The `salmon` class does the following:

- Reads data from a csv file.
- Creates a GP prior with a Matèrn covariance function and a linear mean function. The parameters are chosen fairly arbitrarily at this stage. In chapter 2, we'll look at how to place priors on these parameters and infer them along with the unknown function itself.

  Note also that I've specified the prior using the data; for instance, the `scale` parameter depends on the maximum observed abundance. Some would consider this cheating.
- 'Observes' the unknown function's value to be zero at the origin with no uncertainty. No matter what the data are, every draw from the posterior will have $f(0) = 0$. This isn't really an observation, it's just a convenient way to incorporate the knowledge that if there is no stock, there will be no recruitment.
- Provides a `plot` method, which just plots the posterior envelope, data and three realizations from the posterior.

The main script, 'examples/more_examples/MKMsalmon/salmon.py', creates three `salmon` instances called `chum`, `pink` and `sockeye`, then imposes the data for each species on its prior to obtain its posterior. Like the prior parameters, the observation variance I used was chosen fairly arbitrarily. We will look at inferring it in chapter 2 also. Finally, each species' `plot` method is called. Output is shown in figure 1.9.

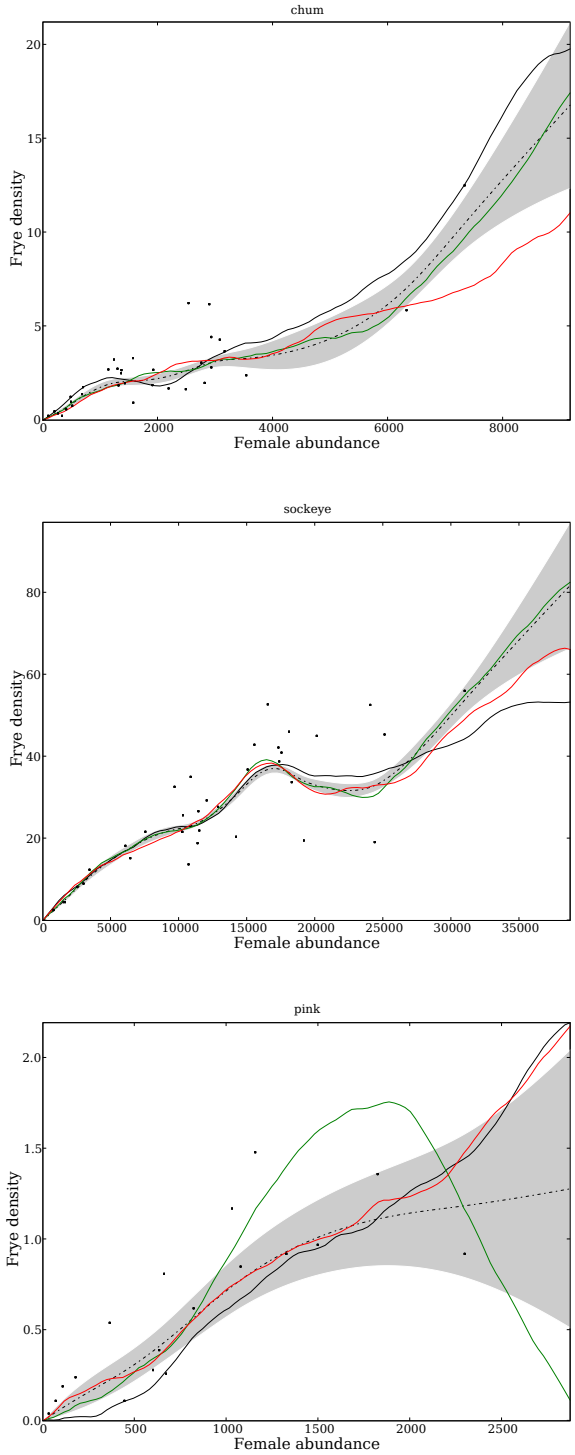


Figure 1.9: Fits to the stock-recruitment data in Munch, Kottas and Mangel' [**?**] Figure 6 using a simple nonparametric regression.

To reiterate, there are some major drawbacks to this simple model. The observation variance may not actually be known, and we may not be comfortable specifying a single value for each of the prior parameters. Because of these considerations, Munch, Kottas and Mangel opt for a more sophisticated statistical model that has to be fit using MCMC. We will follow their example in section 2.5.

1.5 Higher-dimensional GPs

In addition to functions of one variable such as $f(x)$, this package supports Gaussian process priors for functions of many variables such as $f(\mathbf{x})$, where $\mathbf{x} = [x_0 \dots x_{n-1}]$. This is useful for modeling dynamical or biological functions of many variables as well as for spatial statistics.

Any time you pass an array into a `Mean`, `Covariance` or `Realization`'s init method or evaluate one of these objects on an array, the convention is that the array's last index iterates over spatial dimension. To evaluate a covariance C on the ordered pairs `(0,1)`, `(2,3)`, `(4,5)` and `(6,7)`, you could pass in the following two-dimensional array:

```
[[0,1]
 [2,3]
 [4,5]
 [6,7]]
```

or the following three-dimensional array:

```
[[[0,1]
  [2,3]],

  [4,5]
  [6,7]]]
```

Either is fine, since in both the last index iterates over elements of the ordered pairs.

The exception to this rule is one-dimensional input arrays. The array

```
[0, 1, 2, 3, 4, 5, 6, 7]
```

is interpreted as an array of eight one-dimensional values, whereas the array

```
[[0, 1, 2, 3, 4, 5, 6, 7]]
```

is interpreted as a single eight-dimensional value according to the convention above.

Means and covariances learn their spatial dimension the first time they are called or observed. Some covariances, such as those specified in geographic coordinates, have an intrinsic spatial dimension. Realizations inherit their spatial dimension from their means and covariances when possible, otherwise they learn it the first time they are called. If one of these objects is subsequently called with an input of a different dimension, it raises an error.

1.5.1 Covariance function bundles and coordinate systems

The examples so far, starting with '`examples/cov.py`', have used the covariance function `matern.euclidean`. This function is an attribute of the `matern` object, which is an instance of class `covariance_function_bundle`.

Instances of `covariance_function_bundle` have three attributes, `euclidean`, `geo_deg` and `geo_rad`, which correspond to standard coordinate systems:

- `euclidean`: n-dimensional Euclidean coordinates.
- `geo_deg`: Geographic coordinates (longitude, latitude) in degrees, with radius 1.
- `geo_rad`: Geographic coordinates (longitude, latitude) in radians, with radius 1.

Note that you can effectively change the radius of the geographic coordinate systems using the `scale` parameter.

Covariance function bundles are described in more detail in section 3.

1.5.2 Multithreading GP operations

This package can use multi-core systems to speed up two kinds of computations:

- filling in covariance matrices,
- linear algebra for observing GP's or drawing realizations.

If you've built NumPy against multithreaded linear algebra libraries, all the linear algebra will be parallelized automatically. The functions contained in covariance function bundles (section 3), which include all the covariance functions distributed with this package, are multithreaded. The number of threads they use is controlled by the environment variable `OMP_NUM_THREADS`.

On a quad-core system, evaluating a covariance function on large input vectors when `OMP_NUM_THREADS` is equal to 4 will use all the cores. Note that there's no point setting `OMP_TUM_THREADS` to 5 on a quad-core system, because only four cores are available.

1.6 Basis covariances

It is possible to create random functions from linear combinations of finite sets of basis functions $\{e\}$ with random coefficients $\{c\}$:

$$f(x) = M(x) + \sum_{i_0=0}^{n_0-1} \cdots \sum_{i_{N-1}=0}^{n_{N-1}-1} c_{i_1 \ldots i_{N-1}} e_{i_1 \ldots i_{N-1}}(x), \quad \{c\} \sim \text{some distribution.}$$

If the distribution is multivariate normal with mean zero, f is a Gaussian process with mean M and covariance defined by

$$C(x,y) = \sum_{i_0=0}^{n_0-1} \cdots \sum_{i_{N-1}=0}^{n_{N-1}-1} \sum_{j_0=0}^{n_0-1} \cdots \sum_{j_{N-1}=0}^{n_{N-1}-1} e_{i_0 \ldots i_{N-1}}(x) e_{j_1 \ldots j_{N-1}}(x) K_{i_0 \ldots i_{N-1}, j_1 \ldots j_{N-1}},$$

where K is the covariance of the coefficients c.

Particularly successful applications of this general idea (shown in one dimension) are:

Random Fourier series: $e_i(x) = \sin(i\pi x/L)$ or $\cos(i\pi x/L)$, for instance [5].

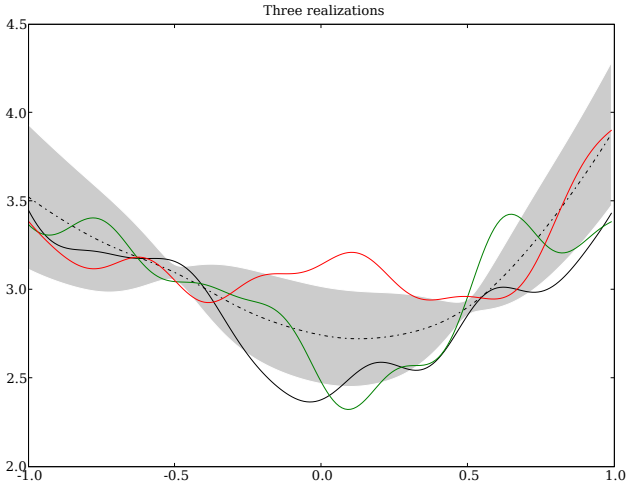


Figure 1.10: Three realizations of an observed Gaussian process whose covariance is an instance of `Basis-Covariance`. The basis in this case is function `fourier_basis` from module `cov_funs`. 25 basis functions are used.

Gaussian process convolutions: $e_i(x) = \exp(-(x-\mu_n)^2)$, for instance [9].

B-splines: $e_i(x) =$ a polynomial times an interval indicator. See *Wikipedia*'s article.

Such representations can be very efficient when there are many observations in a low-dimensional space, but are relatively inflexible in that they generally produce realizations that are infinitely differentiable. In some applications, this tradeoff makes sense.

This package supports basis representations via the `BasisCovariance` class:

```
C = BasisCovariance(basis, cov, **basis_params)
```

The arguments are:

`basis`: Must be an array of functions, of any shape. Each basis function will be evaluated at `x` with the extra parameters. The basis functions should obey the same calling conventions as mean functions: return values should have shape `x.shape[:-1]` unless `x` is one-dimensional, in which case return values should be of the same shape as `x`. Note that each function should take the entire input array as an argument.

`cov`: An array whose shape is either:

- Of the same shape as `basis`. In this case the coefficients are assumed independent, and `cov[i[0],...,i[N-1]]` (an N-dimensional index) simply gives the prior variance of the corresponding coefficient.
- Of shape `basis.shape * 2`, using Python's convention for tuple multiplication. In this case `cov[i[0],...,i[N-1], j[0],...,j[N-1]]` (a $2N$-dimensional index) gives the covariance of $c_{i_0 \ldots i_{N-1}}$ and $c_{j_1 \ldots j_{N-1}}$.

Internally, the basis array is ravelled and this covariance tensor is reshaped into a matrix. The input convention is this way in order to make it easier to keep track of which covariance value corresponds to which coefficients. The covariance tensor must be symmetric (`cov[i[0],...,i[N-1], j[0],...,j[N-1]] = cov[j[0],...,j[N-1], i[0],...,i[N-1]]`), and positive semidefinite when reshaped to a matrix.

`basis_params`: Any extra parameters required by the basis functions.

1.7 Separable bases

Many bases, such as Fourier series, can be decomposed into products of functions as follows:

$$e_{i_0 \ldots i_{N-1}}(x) = e^0_{i_0}(x) \ldots e^{N-1}_{i_{N-1}}(x)$$

Basis covariances constructed using such bases can be represented more efficiently using `SeparableBasisCovariance` objects. These objects are constructed just like `BasisCovariance` objects, but instead of an $n_0 \times \ldots \times n_{N-1}$ array of basis functions they take a nested lists of functions as follows:

```
basis = [ [e[0][0], ... ,e[0][n[0]-1]]
                    ...
          [e[N-1][0], ... ,e[N-1][n[N-1]-1]] ].
```

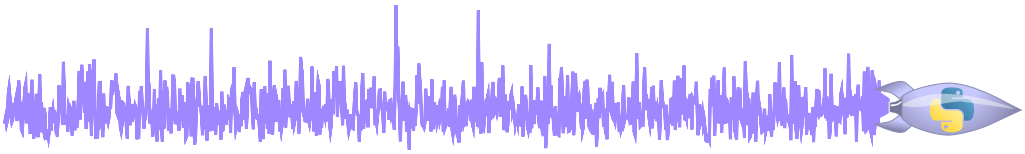
For an N-dimensional Fourier basis, each of the `e`'s would be a sine or cosine; frequency would increase with the second index. As with `BasisCovariance`, each basis needs to take the entire input array `x` and `basis_params` as arguments. See `fourier_basis` in `examples/gp/basiscov.py` for an example.

1.8 Example

Once created, a `BasisCovariance` or `SeparableBasisCovariance` object behaves just like a `Covariance` object, but it and any `Mean` and `Realization` objects associated with it will take advantage of the efficient basis representation in their internal computations. An example of `SeparableBasisCovariance` usage is given in 'examples/basis_cov.py', shown below. Compare its output in figure 1.10 to that of 'examples/observation.py'.

```
from pymc.gp.cov_funs import *
from numpy import *
from copy import copy
from pymc.gp import *

N=100

# Generate mean
def quadfun(x, a, b, c):
    return (a * x ** 2 + b * x + c)

M = Mean(eval_fun = quadfun, a = 1., b = .5, c = 2.)

# Generate basis covariance
coef_cov = ones(2*N+1, dtype=float)
for i in xrange(1,len(coef_cov)):
    coef_cov[i] = 1./int(((i+1)/2))**2.5
```

```
basis = fourier_basis([N])

C = SeparableBasisCovariance(basis,coef_cov,xmin = [-2.], xmax = [2.])

obs_x = array([-.5,.5])
V = array([.002,.002])
data = array([3.1, 2.9])

observe(M=M,
        C=C,
        obs_mesh=obs_x,
        obs_V = V,
        obs_vals = data)

if __name__ == '__main__':
    from pylab import *

    close('all')
    x=arange(-1.,1.,.01)

    figure()
    # Plot the covariance function
    subplot(1,2,1)


    contourf(x,x,C(x,x).view(ndarray),origin='lower',extent=(-1.,1.,-1.,1.),cmap=cm.bone)

    xlabel('x')
    ylabel('y')
    title('C(x,y)')
    axis('tight')
    colorbar()

    # Plot a slice of the covariance function
    subplot(1,2,2)

    plot(x,C(x,0.).view(ndarray).ravel(),'k-')

    xlabel('x')
    ylabel('C(x,.5)')
    title('A slice of C')

    figure()
    plot_envelope(M,C,x)
    for i in range(3):
        f = Realization(M,C)
        plot(x,f(x))
        title('Three realizations')


    # show()
```



CHAPTER
TWO

Incorporating Gaussian processes in PyMC probability models

This chapter will show you how to build and fit statistical models that go beyond simple nonparametric regression.

2.1 Gaussian process submodels

This package represents a Gaussian process $f \sim \mathrm{GP}(M, C)$ as a `GaussianProcess` object which, as you might expect, is a PyMC `stochastic` whose value is a `Realization` object. It is not feasible to endow a full `GaussianProcess` with a `logp` attribute, so `GaussianProcess` objects cannot be handled by PyMC's standard MCMC machinery.

However, the evaluation $f(x_*)$ on a mesh x_* is a simple multivariate normal random variable, which can be handled by the standard machinery. If $f(x_*)$ is incorporated in the model as a variable, a minor extension to the standard machinery (section 2.3) makes it possible to handle f itself as well.

Pairs of f and $f(x_*)$ variables are created by `GPSubmodel` objects, which are containers for PyMC variables. `GaussianProcess` objects can only be incorporated in PyMC probability models via `GPSubmodel` objects.

2.2 Example: nonparametric regression with unknown mean and covariance parameters

A GP submodel is created in 'examples/gp/PyMCmodel.py' with the following call:

```
sm = gp.GPSubmodel('sm',M,C,fmesh)
```

There are two stochastic variables in the submodel: `smf` and `smf_eval`. The first is the actual Gaussian process f: a stochastic variable valued as a `Realization` object. The second is $f(x_*)$, where x_* is the input argument `fmesh`.

Once the GP submodel is created, we can create other variables that depend on f and $f(x_*)$. In 'PyMCmodel.py', the observation d depends on $f(x_*)$:

```
d = pymc.Normal('d',mu=smf_eval, tau=1./V, value=init_val, observed=True)
```

The full probability model is shown as a directed acyclic graph in figure 2.1. It illustrates the dependency relationships between the variables in a GP submodel.

The file 'examples/gp/MCMC.py' fits the probability model created in 'PyMCmodel.py' using MCMC. The 'business part' of the file is very simple:

```
GPSampler = MCMC(PyMCmodel)
GPSampler.isample(iter=5000,burn=1000,thin=100)
```

Most of the code in the file is devoted to plotting, and its output is shown in figure 2.2. Note that after the MCMC run `GPSampler.trace('sm_f')[:]` yields a sequence of `Realization` objects, which can be evaluated on new arrays for plotting. GP realizations can even be tallied on disk using the HDF5 backend (see PyMC's user guide).

2.3 Step methods

Since f has no `logp` attribute, the Metropolis-Hastings family of step methods cannot be used to update f, $f(x_*)$ or any of their mean or covariance parameters. This package uses a relatively simple work-around that will be described here. Throughout this section, the parents of f are denoted P and the children K.

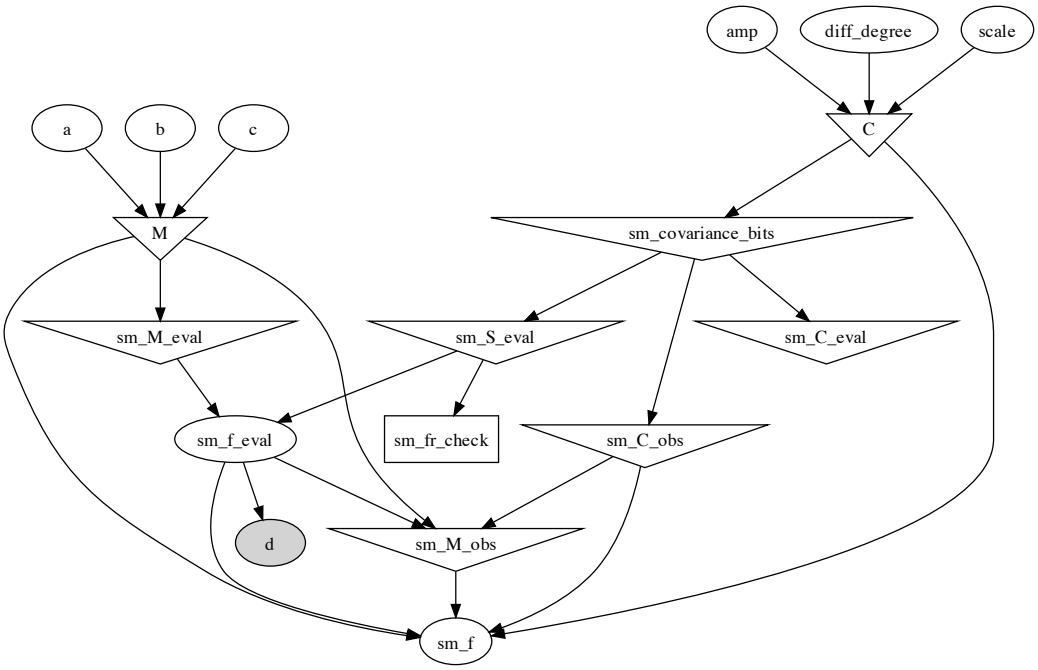


Figure 2.1: The PyMC-generated directed acyclic graph representation of the extended nonparametric regression model created by 'examples/gp/PyMCModel.py'. Ellipses represent `Stochastic` objects (variables whose values are unknown even if their parents' values are known), triangles represent `Deterministic` objects (variables whose values can be determined if their parents' values are known), and rectangles represent `Stochastic` objects with the `isdata` flag set to `True` (data). Rectangles represent potentials. Arrows point from parent to child. The submodel contains the Gaussian process `sm_f` and its evaluation `sm_f_eval` on input array `sm_mesh`. It also contains the mean `sm_M_eval` of `sm_f_eval` and the lower-triangular Cholesky factor `sm_S_eval` of its covariance matrix, and a potential `sm_fr_check` that forces that covariance matrix to remain positive definite. The actual covariance evaluation `sm_C_eval` is not needed by the model, but it is exposed for use by Gibbs step methods.

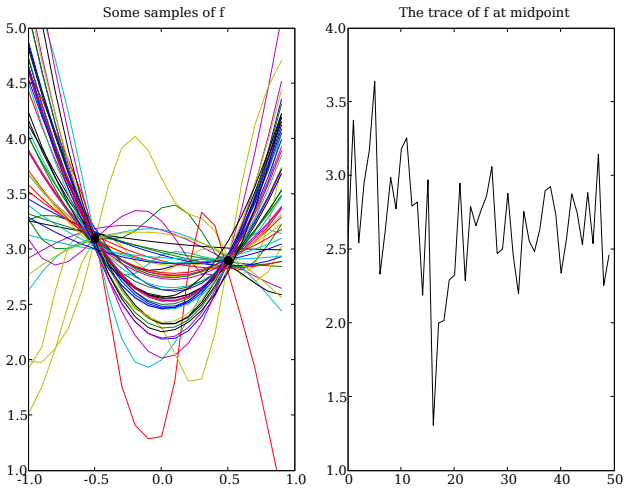


Figure 2.2: The output of 'examples/gp/MCMC.py'. The left-hand panel shows all the samples generated for the Gaussian process f, and the right-hand panel shows the trace of $f(0)$.

2.3.1 Step methods that handle parents of Gaussian processes

If we could come up with a probability density function for f, the Metropolis-Hastings acceptance ratio for a proposed value P_p of the parents *and* a proposed value $\tilde{f}$ for f would be:

$$\frac{p(K|f_p)\ p(f_p|P_p)\ q(P)}{p(K|f)\ p(f|P)\ q(P_p)}$$

where q denotes the proposal density. Now, suppose we proposed a value for f conditional on the proposed values for the parents P *and* conditional on $f(x_*)$. The new acceptance ratio would become

$$\frac{p(K|f_p)\ p(f_p|f(x_*),P_p)\ p(f(x_*)|P_p)\ q(f_p|f(x_*),f_p,P_p)\ q(P)}{p(K|f)\ p(f|f(x_*),P)\ p(f(x_*)|P)\ q(f_p|f(x_*),f,P)\ q(P_p)}$$

We want to avoid computing all terms with f or f_p in the consequent position:

$$\begin{aligned} p(f_p|f(x_*),P),\\ q(f|f(x_*),f_p,P),\\ p(f|f(x_*),P),\\ q(f_p|f(x_*),f,P_p), \end{aligned}$$

but all other terms are fine. We can make the problem terms cancel by choosing our proposal distribution as follows:

$$q(f_p|f(x_*),f,P_p) = p(f_p|f(x_*),P).$$

In other words, if we propose f from its prior distribution conditional on $f(x_*)$ and its parents whenever we propose $f(x_*)$, we don't have to worry about computing the intractable terms. This argument can be made more rigorous by replacing f with its evaluation at all the points at which we would ever want to know it.

By choosing the same proposal distribution for $\tilde{f}$ as above, we again avoid having to compute the intractable terms. In other words, every time a value is proposed for a `GP`'s parent, a value must be proposed for the `GP` conditional on its value's evaluation on its mesh, and the prior probability of the `GP`'s children must be included in the acceptance ratio.

To summarize, any Metropolis-Hastings step method can handle the parents of f, as well as $f(x_*)$, if it proposes values for f jointly with its target variable as outlined above.

This minor alteration can be done using the function `wrap_metropolis_for_gp_parents`, which takes a subclass of `Metropolis` as an argument and returns a new step method class with altered `propose` and `reject` methods. The function automatically produces modified versions of all Metropolis step methods in PyMC's step method registry (`Metropolis`, `AdaptiveMetropolis`, etc.). The modified step methods are automatically assigned to parents of Gaussian processes.

2.3.2 Choosing a mesh

The mesh points x_* are the points where Metropolis-Hastings step methods can 'grab' the value of f to moderate the variance of its proposal distribution. If x_* is an empty array, f's value will be proposed from its prior, and rejection rates are likely to be quite large. If x_* is too dense, on the other hand, computation of the log-probability of $f(x_*)$ will be expensive, as it scales as the cube of the number of points in the mesh.This continuum is illustrated in figure 2.3. Finding the happy medium requires some experimentation.

Another important point to bear in mind is that if f's children depend on its value only via its evaluation on the mesh, the likelihood terms $p(K|f_p)$ and $p(K|f)$ will cancel. In other words, if the mesh is chosen so that $p(K|f) = p(K|f(x_*))$ then the proposed value of f will have no bearing on the acceptance probability of the proposed value of $f(x_*)$ or of the parents P. This is the situation in 'PyMCModel.py'. Such a mesh choice will generally improve the acceptance rate.

2.3.3 The `GPEvaluationGibbs` class

If all of f's children K, depend on it as follows:

$$K_i|f \overset{\text{ind}}{\sim} \text{Normal}(f(x_{*i}), V_i)$$

then $f(x_*)$ can be handled by the `GPEvaluationGibbs` step method. This step method is used in 'MCMC.py':

```
GPSampler.use_step_method(gp.GPEvaluationGibbs, GPSampler.submod, GPSampler.V, GPSampler.d)
```

The initialization arguments are the Gaussian process submodel that contains f, the observation variance of f's children, and the children, in this case the vector-valued normal variable d.

`GPEvaluationGibbs` covers the standard submodel encountered in geostatistics, but there are many conjugate situations to which it does not apply. If necessary, special step methods can be written to handle these situations. If `GPEvaluationGibbs` is not assigned manually, $f(x_*)$ will generally be handled by a wrapped version of `AdaptiveMetropolis`.

2.4 Geostatistical example

Bayesian geostatistics is demonstrated in the folder 'examples/gp/more_examples/Geostatistics'. File 'getdata.py' downloads the Walker Lake dataset of Isaaks and Srivastava [8] from the internet and manipulates the x and

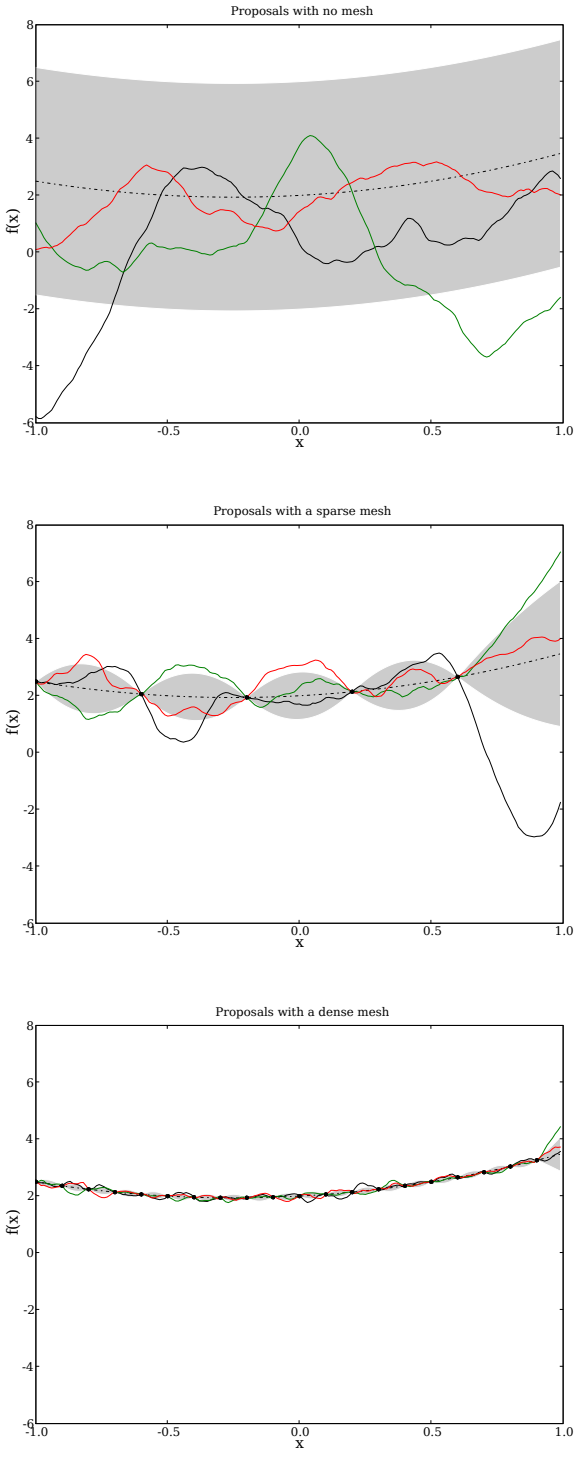


Figure 2.3: Several possible proposals of f (curves) given proposed values for $f(x_*)$ (heavy dots) with no mesh (top), a sparse mesh (middle), and a dense mesh (bottom). Proposal distributions' envelopes are shown as shaded regions, with means shown as broken lines. With no mesh, f is proposed from its prior and the acceptance rate will be very low. A denser mesh permits a high degree of control over f, but computing the log-probability will be more expensive.

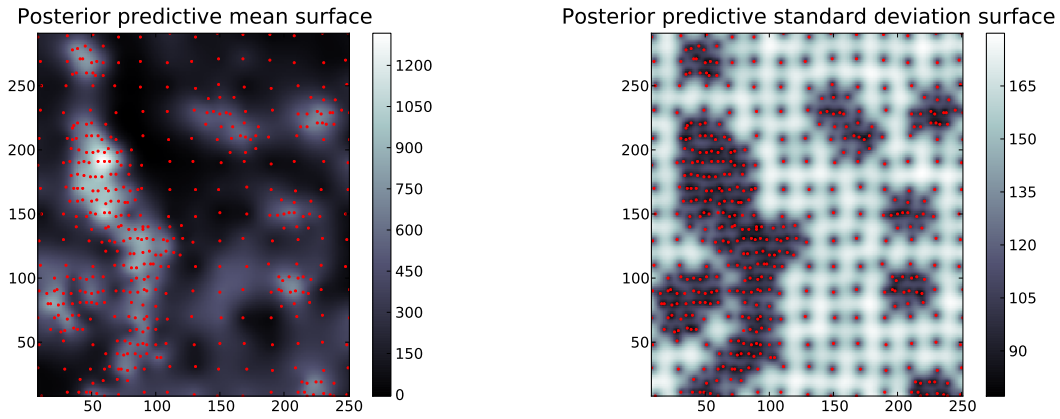


Figure 2.4: The posterior mean and variance surfaces for the v variable of the Walker lake example. The posterior variance is relatively small in the neighborhood of observations, but large in regions where no observations were made.

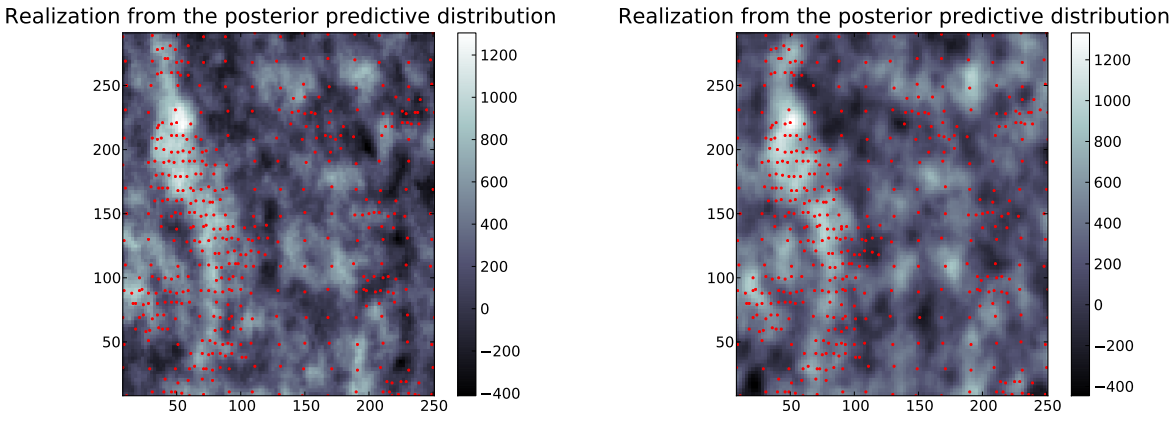


Figure 2.5: Two realizations from the posterior distribution of the v surface for the Walker Lake example. Elevation is measured in meters.

y coordinates into the array format described in section 1.5. File 'model.py' contains the geostatistical model specification, which is

$$
\begin{aligned}
d|f &\sim \text{Normal}(f(x), V) \\
f|M, C &\sim \text{GP}(M, C) \\
M &: x \to m \\
C : x, y, \texttt{amp}, \texttt{scale}, \texttt{diff\_degree} &\to \texttt{matern.euclidean}(x, y; \texttt{amp}, \texttt{scale}, \texttt{diff\_degree}) \\
p(m) &\propto 1 \\
\texttt{amp} &\sim \text{Exponential}(7e-5) \\
\texttt{scale} &\sim \text{Exponential}(4e-3) \\
\texttt{diff\_degree} &\sim \text{Uniform}(.5, 2) \\
V &\sim \text{Exponential}(5e-9)
\end{aligned}
$$

File 'mcmc.py' fits the model and produces output maps. The output of 'mcmc.py' is shown in figures 2.4 and 2.5. Figure 2.4 shows the posterior mean and variance of the v variable of the dataset, which is a function of elevation (see Isaaks and Srivastava [8], appendix A). The mean and variance surfaces are generated conveniently from the trace as follows:

```
Msurf = zeros(dplot.shape[:2])
E2surf = zeros(dplot.shape[:2])
for i in xrange(n):
    WalkerSampler.remember(i)
    Msurf_i = WalkerSampler.walker_v.M_obs.value(dplot)
    Msurf += Msurf_i / n
    E2surf += (WalkerSampler.walker_v.C_obs.value(dplot) + Msurf_i**2) / n
```

The call `WalkerSampler.remember(i)` resets all the variables in the model to their values at frame i of the MCMC trace. The surfaces $E[v]$ (`Msurf`) and $E[v^2]$ (`E2surf`) are easily accumulated by stepping through the trace using `remember` and evaluating the `M_obs` and `C_obs` attributes of the Gaussian process submodel on input array `dplot`. The standard deviation surface is generated using the standard formula:

```
Vsurf = E2surf - Msurf**2
SDsurf = sqrt(Vsurf)
```

Figure 2.5 shows two realizations from the posterior predictive distribution of the v surface. They are much rougher than their mean surface, and also much more expensive to compute. However, they are equally convenient to evaluate:

```
indices = random_integers(n,size=2)
for j,i in enumerate(indices):
    WalkerSampler.remember(i)
    R = WalkerSampler.walker_v.f.value(dplot)
```

Again, `remember` is used to reset the model to a randomly-selected frame of the trace, and the Gaussian process f is evaluated on `dplot` in that frame.

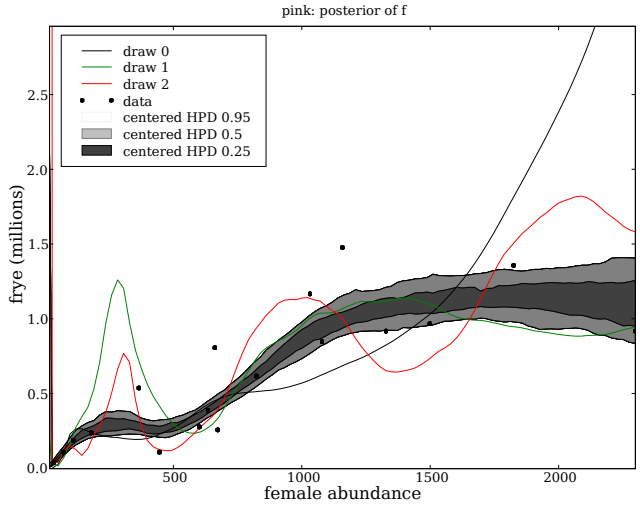


Figure 2.6: The posterior of the stock-recruitment function for pink salmon (*Onchorhynchus gorbuscha*). The data are shown as heavy black dots. The centered 95%, 50% and 25% posterior probability intervals are shown as shaded regions. Three draws from the posterior are plotted.

2.5 Biological example: Munch, Kottas and Mangel's stock-recruitment study

We now have the tools to duplicate Munch, Kottas and Mangel's [**?**] stock-recruitment results. The full probability we will use follows. It is like Munch, Kottas and Mangel's probability model, but we will use a Matèrn covariance function. Here `SR` is the stock-recruitment function, C is its covariance and M is its mean.

$$\begin{aligned}
&\texttt{frye}[i] \overset{\text{ind}}{\sim} \mathrm{N}(exp(\texttt{SR}(\log(\texttt{abundance}[i]))), V), && i = 1 \ldots n \\
&\texttt{SR} \sim \mathrm{GP}(M, C) \\
&M : x \rightarrow \beta_0 + \beta_1 \log(x) \\
&C : x, y \rightarrow \texttt{matern.euclidean}(x, y;\ \texttt{diff_degree}, \texttt{amp}, \texttt{scale}) \\
&V,\ \beta_0,\ \beta_1,\ \texttt{diff_degree},\ \texttt{amp},\ \texttt{scale} \sim \texttt{priors}.
\end{aligned} \tag{2.1}$$

The priors can be found by reading their paper (except the prior on `diff_degree`, which I chose) or by reading the file 'examples/more_examples/MMKSalmon/salmon_sampler.py'. This file contains a PyMC `MCMC` subclass called `SalmonSampler`, which reads in the data, creates a probability model incorporating the data, and provides some plotting methods.

Note that this model differs from the model in section 1.4.1 in that we're putting a Gaussian process prior on the stock-recruitment function in log-log space, so conditioning its value at zero is not an option.

Munch, Kottas and Mangel specify priors for $\texttt{amp}^{-2}$ and V^{-1}, and we have used PyMC deterministic variables to conveniently implement the transformations rather than changing variables manually. The posterior distribution of `SR` for the pink salmon (*Onchorhynchus gorbuscha*) is shown in figure 2.6, and the posterior of the mean and covariance parameters for the same species are shown in figure 2.7.

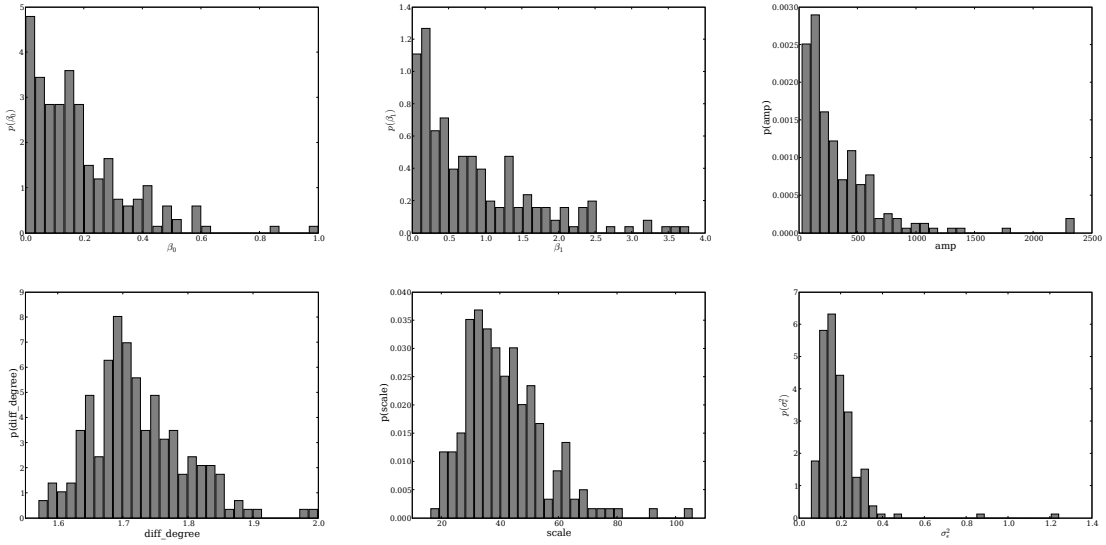
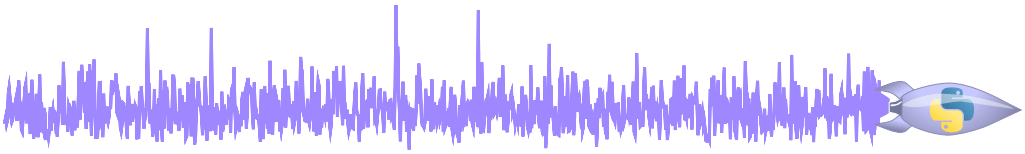


Figure 2.7: The marginal posterior distributions of the mean and covariance parameters of the stock-recruitment function for pink salmon (*Onchorhynchus gorbuscha*).



CHAPTER
THREE

Extending the covariance functions: Writing your own, using alternate coordinate systems, building in anisotropy and nonstationarity

This section will give you a brief tour of the `cov_funs` module and point out plugs where you can add functionality that isn't included in this package.

Note that you don't need to understand this section; any covariance function that satisfies the calling convention described in section 1.2.2 will work. The `cov_funs` module is convenient, but it is admittedly complicated.

The `covariance_function_bundle` class includes versions of each covariance function using several different distance metrics. For example, the Euclidean version of the Matèrn covariance function is `matern.euclidean` and the Gaussian covariance function in geographic coordinates (in radians) is `gaussian.geo_rad`.

In addition, it has two attributes that are included for extensibility: the `raw` attribute, which exposes the basic Fortran implementation of each function, and `add_distance_metric` method, which combines the `raw` attribute with distance metrics. This section will describe these attributes in more detail.

Class `covariance_function_bundle`'s init method takes three arguments:

`cov_fun_name` The name of the covariance function.

`cov_fun_module` The name of the module in which the covariance function can be found. This must be somewhere on your `PYTHONPATH`.

`extra_cov_params` A dictionary whose keys are the extra parameters the covariance function takes, and whose values are brief sentences explaining the roles of those parameters. This dictionary will be used to generate the docstring of the bundle.

The names of the function and module are used rather than the function object itself in order to allow covariance objects to be pickled. This, in turn, allows covariance-valued stochastic variables to be stored in PyMC's `pickle` and `hdf52` database backends.

3.1 The components of a covariance function bundle

3.2 The actual covariance functions

The covariance functions contained in a bundle are named by the distance metric they use. The `matern` bundle, for instance, contains the following covariance functions:

- `matern.euclidean`
- `matern.geo_rad`
- `matern.geo_deg`
- `matern.aniso_geo_rad`
- `matern.aniso_geo_deg`
- `matern.paniso_geo_rad`
- `matern.paniso_geo_deg`

Each covariance function takes at least two arguments, x and y, which are two-dimensional arrays in which the first index iterates over separate points and the second iterates over coordinates as in section 2.4. For the geographic distance functions, the first coordinate is longitude and the second is latitude. Covariance functions also take arguments `amp` and `scale`. Finally, they take optional argument `symm`, which indicates whether x and y the same array.

Some of the covariance functions take extra arguments for the distance metric and/or actual covariance function. For example, `matern.aniso_geo_rad` takes extra arguments `inc` and `ecc` for the distance function and `diff_degree` for the covariance function.

When `matern.euclidean`, for example, is called with input locations x and y, the following events place:

- The output matrix is allocated.
- The output matrix is filled with the Euclidean distances between the elements of x and the elements of y.
- The output matrix is overwritten with the covariances between the elements of x and the elements of y.

The last two steps will be executed in parallel if environment variable `OMP_NUM_THREADS` is greater than 1 and the size of the output matrix is sufficiently large.

The easiest way to deal with different coordinate systems or to build in anisotropy and nonstationarity using a deformation approach [**?**] is to write your own distance function and add it to the covariance function by calling the `add_distance_metric` method. See 3.4 for information on how to use your distance function with an existing covariance function, or with one of your own devising. See 3.5 for the calling conventions required from distance functions.

3.3 The `raw` attribute

The `raw` attribute of each bundle is the underlying covariance function. These functions take distance matrices as arguments and overwrite them in-place as covariance matrices. If you write your own raw covariance function, it should conform to the standard described in section 3.5.

The following raw functions are implemented in Fortran in 'isotropic_cov_funs.f' and wrapped as covariance function bundles in `cov_funs`. Here t denotes a single element of a distance matrix, $|x_i - y_j|$. Each function is equal to 1 when $t = 0$.

`matern(`ν, t`):` The argument `diff_degree` is written in the following formula as ν for readability. K_ν is a modified Bessel function of the third kind of order ν.

$$\frac{(2\sqrt{\nu}t)^\nu}{2^{\nu-1}\Gamma(\nu)} K_\nu(2\sqrt{\nu}t)$$

`quadratic(phi,`t`):`

$$1 - \frac{t^2}{1 + \phi t^2}$$

`gaussian(`t`):`

$$e^{-t^2}$$

`pow_exp(pow,`t`):` The argument `pow` is written as p.

$$e^{-|t|^p}$$

`sphere(t)`:

$$\begin{cases} 1 - \frac{3}{2}t + \frac{1}{2}t^3 & t \leq 1 \\ 0 & t > 1 \end{cases} \tag{3.1}$$

See *Banerjee et al.* [3] , *Diggle and Ribeiro* [**?**] and *Stein* [**?**]for more discussion of the properties of these covariance functions. The init method of `covariance_function_bundle` takes just one argument, a raw covariance function.

3.4 The `add_distance_metric` method and the `apply_distance` function

`apply_distance` takes two arguments, a distance function and a raw covariance function, and returns a covariance function suitable for wrapping in a `Covariance` object.

It endows the new function with a docstring that combines the raw covariance function's information with the distance function's information. It's possible to include information about extra parameters in the auto-generated docstring. To wrap a raw covariance function with a distance function called `my_dist_fun` with extra parameters called `eta` and `gamma` you could add explanations to `my_dist_fun` before calling `apply_distance`:

```
my_dist_fun.extra_params = {'eta': 'The eta parameter.',
                            'gamma': 'The gamma parameter.'}
```

The new parameters will be included in the docstring.

Covariance functions generated by `apply_distance` take the following arguments:

- Input arrays x and y. These high-level covariance functions are more forgiving than the distance functions; x and y do not actually need to be two-dimensional, the only requirement is that their last index must iterate over coordinates as in section 2.4. The exception to this rule is if x and y are only one dimensional, in which case they are assumed to have only one coordinate.
- Scaling arguments `amp` and `scale`. Parameter `amp` is the standard deviation of $f(x)$ for arbitrary x, regardless of coordinate system, before observation (the prior standard deviation). Parameter `scale` controls the lengthscale of decay of the correlation, which controls the wiggliness of f. It effectively multiplies distances, so that large values yield quick decay and more wiggliness. These scaling arguments are sometimes referred to as the 'sill' and 'range' parameters.
- Extra arguments for the covariance and/or distance functions.

The `add_distance_metric` method of `covariance_function_bundle` is just a thin wrapper for the function `apply_distance`, which is in 'cov_utils.py'. Since `covariance_function_bundle` objects already have a raw covariance function, this method only needs a distance function.

3.5 Calling conventions for new covariance and distance functions

User-defined covariance and distance functions should always take the following arguments:

C A Fortran-contiguous (column major) array of dimension (`x.shape[0]`,`y.shape[0]`). This should be overwritten in-place.

x, y Arrays of input locations. These will be regularized: they will be two-dimensional, with the first index iterating over points and the second over coordinates.

`cmin`$= 0$, `cmax`$= -1$ Optional arguments. If non-default values are provided, only the slice `C[:,cmin:cmax]` of C should be overwritten.

`symm=False` An optional argument indicating whether x and y are the same array. If `True`, C will be square and only the upper triangle of C should be overwritten.

They can take any other arguments they need, of course.

If parallel computation is desired, covariance and distance functions must release the *global interpreter lock*. That means they have to be written in Fortran or C. *f2py* extensions can release the global interpreter lock by simply including the statement `cf2py threadsafe`; other types of extensions should call the Python C-API macros `Py_BEGIN_ALLOW_THREADS` and `Py_END_ALLOW_THREADS`.

The high-level covariance functions produced by `apply_distance` will be more forgiving than these low-level distance and covariance functions; x and y will not actually need to be two-dimensional, the only requirement is that their last index must iterate over coordinates as in section 2.4. The exception to this rule is if x and y are only one dimensional, in which case they are assumed to have only one coordinate. These high-level functions will regularize x and y before passing them into your distance and covariance functions.

User-supplied covariance functions should be expressed in their simplest forms: their amplitude and input scalings ('sill' and 'range', when applicable) should be set to 1. No 'nugget' parameter should be used. Numerically singular covariances are helpful when using `Covariance` objects, because they have low-rank Cholesky factors that can be used for efficient computation. The effect of a nugget can be obtained by adding iid normal variables to realizations. `FullRankCovariance` objects take a nugget argument regardless of their underlying covariance functions.

Distance functions should be symmetric, meaning the distance from point x to point y is the same as the distance from y to x. Since raw covariance functions overwrite distance matrices, the output of an `apply_distance`-generated covariance function will be symmetric if its distance function is symmetric.

To be a valid covariance function, an `apply_distance`-generated covariance function C must be positive semidefinite, meaning $C(x, x)$ is a positive semidefinite matrix for any mesh x. The formal test for positive semidefiniteness is Bochner's theorem [**?**].

3.6 Summary

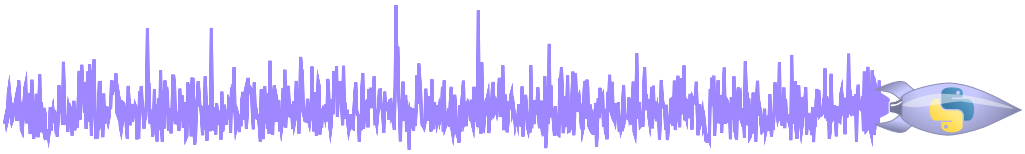
To use a new functional form: Write a new raw covariance function and wrap it in a `covariance_function_bundle` object.

To use a new coordinate system: Write a new distance function and use the `add_distance_metric` method of an existing `covariance_function_bundle`.

To build in anisotropy/ nonstationarity using a deformation approach like that of Sampson and Guttorp [**?**], you can either write a new distance function and use `add_distance_metric` or you can actually implement the deformation before passing in the x and y arguments, possibly using PyMC `Deterministics`.

To use a new functional form *and* a new distance function: Wrap the covariance function as a `covariance_function_bundle` and apply the `add_distance_metric` method to the distance function, or just apply `apply_distance` to both of them at once.

The easiest way to implement a more advanced extension like that of Paciorek and Schervish [**?**] , is to write a new wrapper like `apply_distance`. You'll probably still be able to take advantage of the raw covariance functions, but not the distance functions. If you do this and don't mind sharing your work, please email me.



CHAPTER
FOUR

Overview of algorithms

More detail on these algorithms, as well as the internal workings of the mean, covariance and realization objects and the observe function, are given in the algorithm documentation.

4.1 Gaussian processes and the multivariate normal distribution

Gaussian processes generalize the multivariate normal distribution from vectors to functions, like the multivariate normal distribution generalizes the univariate normal distribution from scalars to vectors. The progression is as follows:

$$\begin{aligned} y \sim \mathrm{N}(\mu, V): &\quad y,\ \mu,\ V \text{ are scalars} \\ \vec{y} \sim \mathrm{N}(\vec{\mu}, C): &\quad \vec{y} \text{ and } \vec{\mu} \text{ are vectors, } C \text{ is a matrix} \\ f \sim \mathrm{GP}(M, C): &\quad f \text{ and } M \text{ are functions of one variable, } C \text{ is a function of two variables} \end{aligned} \tag{4.1}$$

One of the nice things about all Gaussian distributions (parameterized by mean and covariance) is that they're easy to marginalize. For example, each element of a vector with a multivariate normal distribution has a univariate normal distribution:

$$\begin{aligned} \vec{y} &\sim \mathrm{N}(\vec{\mu}, C) \\ \Rightarrow \vec{y}_i &\sim \mathrm{N}(\vec{\mu}_i, C_{i,i}), \end{aligned}$$

and any subvector of a vector with a multivariate normal distribution has a multivariate normal distribution:

$$\begin{aligned} \vec{y} &\sim \mathrm{N}(\vec{\mu}, C) \\ \Rightarrow \vec{y}_{i_1 \ldots i_2} &\sim \mathrm{N}(\vec{\mu}_{i_1 \ldots i_2}, C_{i_1 \ldots i_2, i_1 \ldots i_2}). \end{aligned}$$

This marginalizability applies to GP's as well. If $\vec{x}$ is a vector of values,

$$\begin{aligned} f &\sim \mathrm{GP}(M, C) \\ \Rightarrow f(\vec{x}) &\sim \mathrm{N}(M(\vec{x}), C(\vec{x}, \vec{x})). \end{aligned} \tag{4.2}$$

In other words, any evaluation of a Gaussian process realization has a multivariate normal distribution. Its mean is the corresponding evaluation of the associated mean function, and its covariance is the corresponding evaluation of the associated covariance function. You can probably start to see why this fact is important for working with GPs on a computer.

4.1.1 Observations

As mentioned earlier, if f has a GP prior and normally-distributed observations of f are made at a finite set of values, f's posterior is a Gaussian process also:

$$\left.\begin{array}{l} d_i \overset{\text{ind}}{\sim} \mathrm{N}(f(o_i), V_i) \\ f \sim \mathrm{GP}(M, C) \end{array}\right\} \Rightarrow f|d \sim \mathtt{GP}(M_o, C_o).$$

Denoting by o the array of observation values $[o_0 \ldots o_{N-1}]$ and V a matrix with observation variances $[V_0 \ldots V_{N-1}]$ on its diagonal, $M_o(x)$ and $C_o(x, y)$ are as follows for arbitrary input vectors x and y:

$$\begin{aligned} M_o(x) &= M(x) + C(x, o)[C(o, o) + V]^{-1}(f(o) - M(o)) \\ C_o(x, y) &= C(x, y) - C(x, o)[C(o, o) + V]^{-1}C(o, y). \end{aligned}$$

4.1.2 Low-rank observations

If $C(o,o)+V$ is singular, some elements of $f(o)$ can be computed from others with no uncertainty. In other words, there exists a partition $[o_*, o_{**}]$ of o and corresponding partition of the data such that $C(o_*,o_*)+V$ is full-rank and

$$f(o_{**}) = M_{o_*}(o_{**}),$$

where M_{o_*} can be computed from the formula above.

In such cases, this package's strategy is to observe f at a subvector o_* of o, such that $C(o_*,o_*)+V$ is full-rank but if any elements were added to o_* it would pick up a very small eigenvalue. The function `predictive_check` optionally checks the remaining data values d_{**} against $M_{o_*}(o_{**})$, and raises an error if the two aren't equal up to a user-defined threshold.

This threshold is parameter `relative_precision` from `Covariance`'s init method, multiplied by the largest value on the diagonal of $C(o_*,o_*)+V$; intuitively, the maximal variance of $f(o_{**})$ given $f(o_*)$ is a multiple of the maximal variance of $f(o_*)$.

4.2 Incomplete Cholesky factorizations

In addition to numpy's linear algebra support, this package uses some Fortran subroutines wrapped using *f2py*. They require *BLAS* and *LAPACK* libraries (eg, *ATLAS*) on your system, and the 'setup.py' script will attempt to find these. If you don't have optimized BLAS and LAPACK installed, it's a good idea to install them; this package and numpy in general will be much faster. Some operating systems (such as Mac OS X) ship with optimized BLAS and LAPACK libraries included.

The following incomplete Cholesky factorization functions are found in the Fortran files 'linalg_utils.f' and 'incomplete_chol.f' :

`U, m, piv = ichol(diag, reltol, rowfun):` An implementation of the incomplete Cholesky decomposition, based on a port of one of the functions in the *chol_incomplete* package by Matthias Seeger.

The arguments are:

`diag:` The diagonal of an `n` by `n` covariance matrix C.

`reltol:` If the ratio of the `i`'th pivot to the maximum pivot is found to be less than `reltol`, the `i`'th pivot is assumed to be zero.

`rowfun:` A Python function. The call `rowfun(i,p,rowvec)` should perform the update `rowvec[i,i+1:]=C[i,p[i+1:]]` in-place.

The outputs are:

M: The rank of C. Note that M≤`n`.

`piv:` A length-`n` vector of pivots.

`U:` An M-by-`n` upper-triangular matrix that satisfies `U[:,argsort(piv)].T * U[:,argsort(piv)] = C`.

Because the full matrix C does not need to be computed ahead of time, the algorithm is able to factor C in $O(m^2 n)$ operations [**?**]. The algorithm is implemented in Fortran, but a Python version would be as follows:

```
def swap(vec,i,j):
    temp = vec[i]
    vec[i] = vec[j]
    vec[j] = temp

def ichol(diag, reltol, rowfun):

    piv = arange(n)
    U = zeros((n,n),dtype=float).view(matrix)
    rowvec = zeros(n,dtype=float)

    for i in range(n):
        l = diag[i:].argmax()
        maxdiag = diag[l]

        if maxdiag < reltol:
            m=i
            return U[:m,:], m, piv

        swap(diag,i,l)
        swap(p,i,l)

        temp = U[:i,i]
        U[:i,i] = U[:i,l]
        U[:i,l] = temp


        U[i,i] = sqrt(diag[i])
        rowvec[i:] = C[i,piv[i+1:]]

        if i > 0:
            rowvec -= U[:i,i].T * U[:i,i+1:]

        U[i,i+1:] = rowvec[i+1:] / U[i,i]
        diag[i+1:] -= U[i,i+1:].view(ndarray) ** 2

    m=n
    return U, m, piv
```

Function `ichol` is wrapped by the `Covariance` method `cholesky`.

`m, piv = ichol_continue(U, diag, reltol, rowfun, piv)`: This function computes the Cholesky factorization of an `n` by `n` covariance matrix C from the factor of its upper-left $\mathtt{n}_*$ by $\mathtt{n}_*$ submatrix C_*. Its input arguments are as follows:

`U`: Unlike `ichol`, this function overwrites a matrix in-place. Suppose the Cholesky factor of C_*, U_*, is of rank M_*. On input, `U` must be an `[m + (n-n*)]`-by-`n` matrix arranged like this:

$$\begin{bmatrix} U_*[:,:m_*] & U_*[:,:m_*].T.I\ C[:m_*,n_*:] & U_*[:,m_*:] \\ 0 & 0 & 0 \end{bmatrix}$$

On exit, `U` will be an M-by-`n` upper-triangular matrix that satisfies `U[:,argsort(piv)].T * U[:,argsort(piv)]=C`.

`piv`: Denote by $\texttt{piv}_*$ the pivot vector associated with $\texttt{U}_*$ On input, `piv` must be a length-`n` vector laid out like this:

$$[\texttt{piv}_*\texttt{[:m}_*\texttt{]} \quad \texttt{arange(n-n}_*\texttt{)} \quad \texttt{piv}_*\texttt{[m}_*\texttt{:]}]$$

`diag`: The length $\texttt{n-n}_*$ diagonal of $\texttt{C[n}_*\texttt{:,n}_*\texttt{]}$.

`rowfun`: The call `rowfun(i,p,rowvec)` should perform the update `rowvec[i,i+1:] = C[i,p[i+1:]]` in-place.

The input parameter `reltol`, as well as the output parameters `piv` and M and the updated matrix `U`, should be interpreted just like their counterparts in `ichol`.

The algorithm is just like the algorithm in ichol, but the index `i` iterates over $\texttt{range(m}_*\texttt{,m}_*\texttt{+n-n}_*\texttt{)}$ (and the index of `diag` is downshifted appropriately).

`ichol_continue` is wrapped by the `Covariance` method `continue_cholesky`, so it should rarely be necessary to call it directly.

`U, m, piv = ichol_full(c, reltol)`: Just like `ichol`, but instead of a diagonal and a 'get-row' function a full covariance matrix C is required as an input.

`U, m, piv = ichol_full(basis, nug, reltol)`: Just like `ichol`, but the following arguments are required:

`basis`: The evaluation of a basis function on the mesh, multiplied by the Cholesky factor of the coefficients' covariance matrix. This is itself a square root of the covariance matrix, but running it through `ichol_basis` allows for observations with nonzero variance and essentially pivots for better accuracy even in the zero-variance case.

`nug`: A vector that will effectively be added to the diagonal of the covariance matrix.

BIBLIOGRAPHY

[] Roy M. Anderson and Robeert M. May. *Infectious diseases of humans: dynamics and control*. Oxford Science Publications, 1992.

[2] Francis R. Bach and Michael I. Jordan. Predictive low-rank decomposition for kernel methods. In *Proceedings of the 22nd International Conference on Machine learning*, 2005.

[3] Sudipto Banerjee, Bradley P. Carlin, and Alan E. Gelfand. *Heirarchical modeling and analysis for spatial data*. Chapman & Hall / CRC, 2003.

[] Donald A. Berry. *Statistics: a Bayesian perspective*. Duxbury Press, 1996.

[] Peter J. Diggle and Paulo J. Ribeiro. *Model-based geostatistics*. Springer, 2007.

[] Dean G. Duffy. *Advanced Engineering Mathematics*. CRC Press, 1997.

[] Stephen P. Ellner, Yodit Seifu, and Robert H. Smith. Fitting population dynamic models to time-series data by gradient matching. *Ecology*, 83(8):2256–2270, 2002.

[] Shai Fine and Katya Scheinberg. Efficient svm training using low-rank kernel representations. *Journal of Machine Learning Research*, 2:243–264, 2001.

[] Andrew Gelman, John B. Carlin, Hal S. Stern, and Donald B. Rubin. *Bayesian data analysis*. Chapman & Hall / CRC, 2004.

[5] Roger G. Ghanem and Pol D. Spanos. *Stochastic finite elements: a spectral approach*. Dover Publications, 1991.

[8] Edward H. Isaaks and R. Mohan Srivastava. *An introduction to applied geostatistics*. Oxford University Press, 1989.

[9] Herbert K. H. Lee, Dave M. Higdon, Catherine A. Calder, and Christopher H. Holloman. Efficient models for correlated data via convolutions of intrinsic processes. *Statistical Modelling*, 5(1):53–74, 2005.

[] Irwin B. Levitan and Leonard K. Kaczmarek. *The neuron: cell and molecular biology*. Oxford University Press, 1997.

[] Kenneth S. Miller and Betram Ross. *An introduction to the fractional calculus and fractional differential equations*. John Wiley and Sons, 1993.

[] Stephan B. Munch, Athanasios Kottas, and Marc Mangel. Bayesian nonparametric analysis of stock-recruitment relationships. *Canadian Journal of Fisheries and Aquatic Sciences*, 62:1808–1821, 2005.

[] Walter Nicholson. *Microeconomic Theory*. Thomson Learning, eighth edition, 2002.

[10] Travis E. Oliphant. *Guide to NumPy*. Trelgol Publishing, 2006.

[] Christopher J. Paciorek and Mark J. Schervish. Nonstationary covariance functions for gaussian process regression. In *NIPS conference proceedings*, 2004.

[] Walter Rudin. *Real & complex analysis*. McGraw-Hill, 1966.

[] P. D. Sampson and Peter Guttorp. Nonparametric estimation of nonstationary spatial covariance structure. *Journal of the American Statistical Association*, 87(417):108–119, 1992.

[] Michael L. Stein. *Interpolation of spatial data: some theory for kriging*. Springer, 1999.

[] John H. Vandermeer. *Elementary mathematical ecology*. Krieger Publication Co., 1990.