gptide

Release 0.2.0

Matt Rayson; Lachlan Astfalck; Andrew Zulberti

Jul 31, 2023

CONTENTS

Gaussian Process regression toolkit for Transformation of Infrastructure through Digitial Engineering applications.

Gaussian Process regression (also called *Optimal Interpolation* or *Kriging*) is useful for fitting a continuous surface to sparse observations, i.e. making predictions. Its main use in environmental sciences, like oceanography, is for spatio-temporal modelling. This package provides a fairly simple API for making predictions AND for estimating kernel hyper-parameters. The hyper-parameter estimation has two main functions: one for Bayesians, one for frequentists. You choose.

Note that there are many other Gaussian Process packages on the world wide web - this package is yet another one. The selling point of this package is that the object is fairly straightforward and the kernel building is all done with functions, not abstract classes. The intention is to use this package as both a teaching and research tool.

Source code: https://github.com/tide-itrh/gptide

CHAPTER

ONE

CONTENTS

1.1 Examples

1.1.1 1D kernel basics

This example will cover:

- Initialising the GPtide class with a kernel and some 1D data
- Sampling from the prior
- Making a prediction at new points
- Sampling from the conditional distribution

[1]:
```
from gptide import cov
from gptide.gpscipy import GPtideScipy
import numpy as np
import matplotlib.pyplot as plt
```

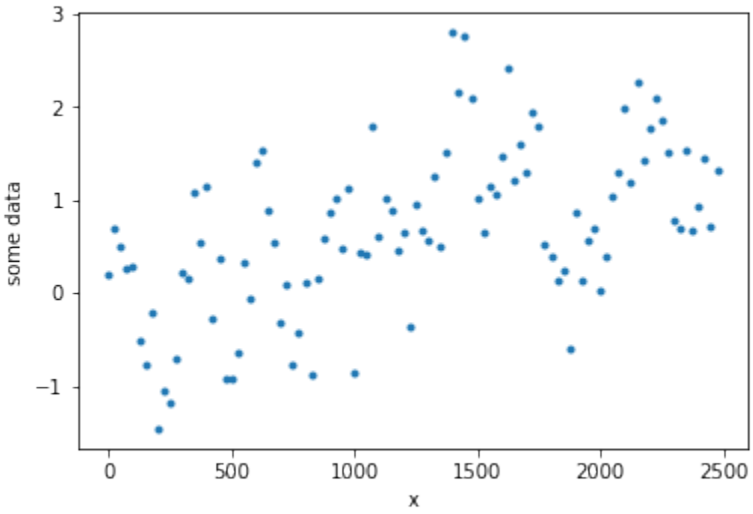
We use an exponential-quadratic kernel with a length scale ℓ = 100 and variance $\eta = 1.5^2$. The noise (σ) is 0.5. The total length of the domain is 2500 and we sample 100 data points.

[2]:
```
####
# These are our kernel input parameters
noise = 0.5
 = 1.5
 = 100
covfunc = cov.expquad_1d

###
# Domain size parameters
dx = 25.
N = 100
covparams = (, )

# Input data points
xd = np.arange(0,dx*N,dx)[:,None]
```

Initialise the GPtide object and sample from the prior

[3]:
```
GP = GPtideScipy(xd, xd, noise, covfunc, covparams)

# Use the .prior() method to obtain some samples
yd = GP.prior(samples=1)

plt.figure()
plt.plot(xd, yd,'.')
plt.ylabel('some data')
plt.xlabel('x')
```

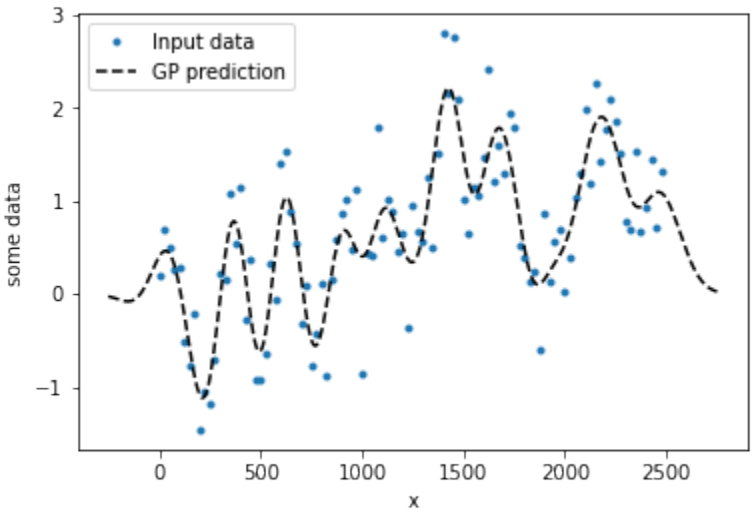
[3]:
```
Text(0.5, 0, 'x')
```



Make a prediction at new points

[4]:
```
# Output data points
xo = np.linspace(-10*dx,dx*N+dx*10,N*10)[:,None]

# Create a new object with the output points
GP2 = GPtideScipy(xd, xo, noise, covfunc, covparams)

# Predict the mean
y_mu = GP2(yd)

plt.figure()
plt.plot(xd, yd,'.')
plt.plot(xo, y_mu,'k--')

plt.legend(('Input data','GP prediction'))
plt.ylabel('some data')
plt.xlabel('x')
```

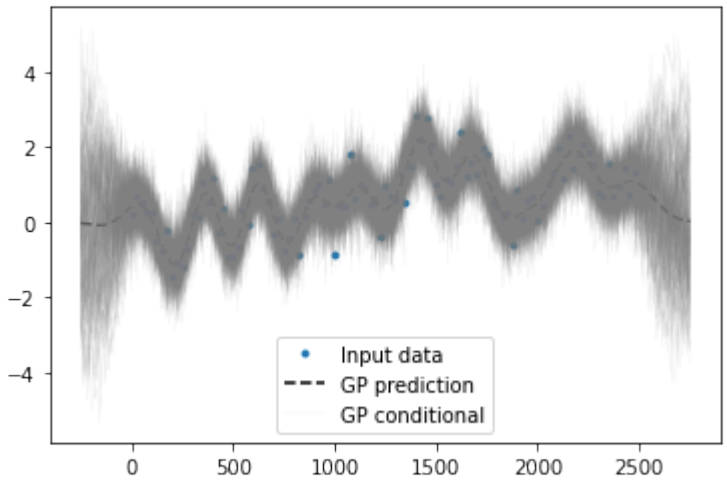
[4]:
```
Text(0.5, 0, 'x')
```



Make a prediction of the full conditional distribution at new points

[5]:
```
samples = 100
y_conditional = GP2.conditional(yd, samples=samples)

plt.figure()
plt.plot(xd, yd,'.')
plt.plot(xo, y_mu,'k--')

for ii in range(samples):
    plt.plot(xo[:,0], y_conditional[:,ii],'0.5',lw=0.2, alpha=0.2)

plt.legend(('Input data','GP prediction','GP conditional'))
```

[5]:
```
<matplotlib.legend.Legend at 0x18a6d743b80>
```



You can see from above how the mean prediction returns to zero in the extrapolation region whereas the conditional

samples reverts to the prior i.e. it goes all over the place.

1.1.2 1D parameter estimation using maximum likelihood estimation

This example will cover:

- Use maximum likelihood estimation to optimise kernel parameters

[1]:
```
from gptide import cov
from gptide import GPtideScipy
import numpy as np
import matplotlib.pyplot as plt
```

Generate some data

Start off with the same kernel as Example 1 and generate some data.

[2]:
```
####
# These are our kernel input parameters
np.random.seed(1)
noise = 0.5
 = 1.5
 = 100
covfunc = cov.expquad_1d

###
# Domain size parameters
dx = 25.
N = 100
covparams = (, )

# Input data points
xd = np.arange(0,dx*N,dx)[:,None]

GP = GPtideScipy(xd, xd, noise, covfunc, covparams)

# Use the .prior() method to obtain some samples
yd = GP.prior(samples=1)
```

Inference

We now use the `gptide.mle` function do the parameter estimation. This calls the `scipy.optimize.minimize` routine.

[3]:
```
# mle function import
from gptide import mle
```

[4]:
```
# Initial guess of the noise and covariance parameters (these can matter)
noise_ic = 0.01
covparams_ic = [1., 10.]
```

(continues on next page)

(continued from previous page)

```
# There is no mean function in this case
# meanfunc = None
# meanparams_ic = ()


soln = mle(
    xd, yd,
    covfunc,
    covparams_ic,
    noise_ic,
    verbose=False)

print('Noise (true): {:3.2f}, |Noise| (mle): {:3.2f}'.format(noise, abs(soln['x'][0])))
print(' (true): {:3.2f},  (mle): {:3.2f}'.format(covparams[0], soln['x'][1]))
print(' (true): {:3.2f},  (mle): {:3.2f}'.format(covparams[1], soln['x'][2]))
```

```
Noise (true): 0.50, |Noise| (mle): 0.45
 (true): 1.50,  (mle): 1.21
 (true): 100.00,  (mle): 95.02
```

1.1.3 1D parameter estimation using MCMC

This example will cover:

- Use MCMC to infer kernel paramaters
- Finding sample with highest log-prob from the mcmc chain
- Visualising results of sampling
- Making predictions

[1]:
```
from gptide import cov
from gptide import GPtideScipy
import numpy as np
import matplotlib.pyplot as plt

import corner
import arviz as az

from scipy import stats
from gptide import stats as gpstats
```

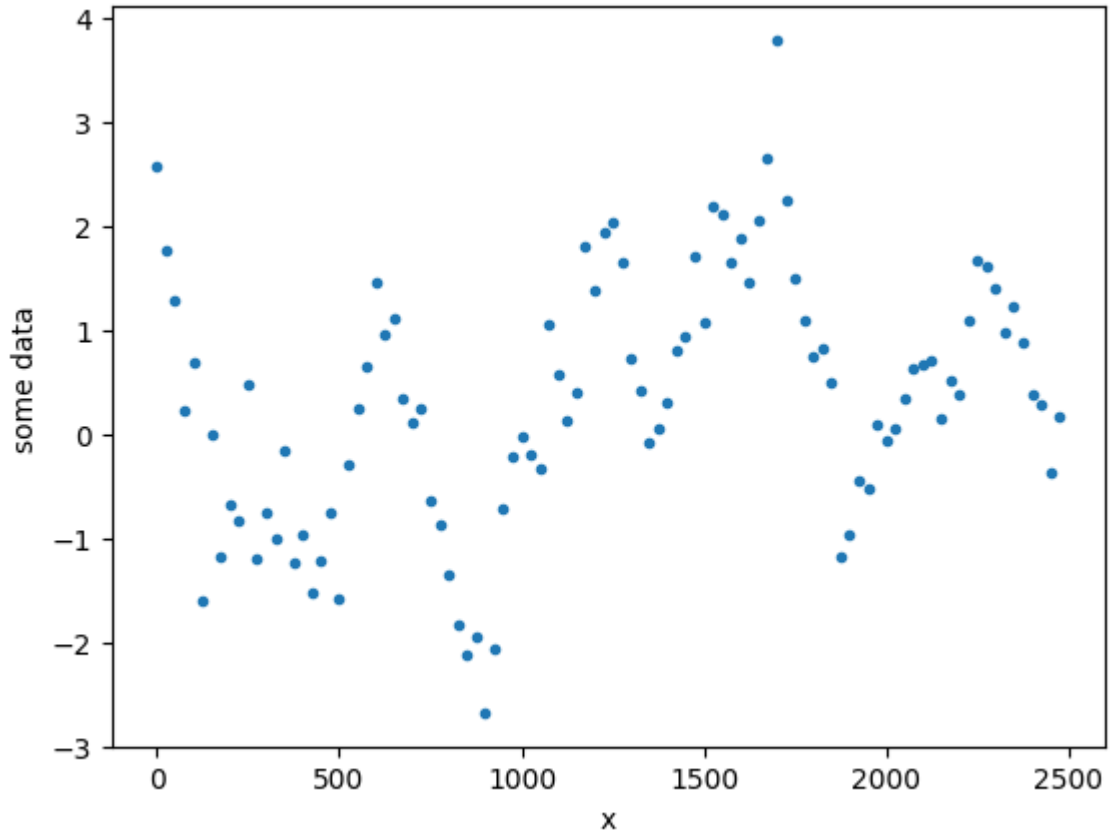
Generate some data

Start off with the same kernel as Example 1 and generate some data.

[2]:
```
####
# These are our kernel input parameters
np.random.seed(1)
noise = 0.5
 = 1.5
 = 100
covfunc = cov.expquad_1d

###
# Domain size parameters
dx = 25.
N = 100
covparams = (, )

# Input data points
xd = np.arange(0,dx*N,dx)[:,None]

GP = GPtideScipy(xd, xd, noise, covfunc, covparams)

# Use the .prior() method to obtain some samples
yd = GP.prior(samples=1)
```

[3]:
```
plt.figure()
plt.plot(xd, yd,'.')
plt.ylabel('some data')
plt.xlabel('x')
```

[3]:
```
Text(0.5, 0, 'x')
```



Inference

We now use the `gptide.mcmc` function do the parameter estimation. This uses the `emcee.EnsembleSampler` class.

[4]:
```
from gptide import mcmc
import importlib
importlib.reload(mcmc)
```

[4]:
```
<module 'gptide.mcmc' from '/mnt/c/Users/00071913/OneDrive - The University of Western␣
↪Australia/Zulberti/gptide/gptide/mcmc.py'>
```

[11]:
```
# Initial guess of the noise and covariance parameters (these can matter)

noise_prior      = gpstats.truncnorm(0.4, 0.25, 1e-15, 1e2)                # noise -␣
↪true value 0.5
covparams_priors = [gpstats.truncnorm(1, 1, 1e-15, 1e2),    #  - true value 1.5
#                   gpstats.truncnorm(10, 1, 1e-15, 1e4) #  - true value 100
                    gpstats.truncnorm(125, 50, 1e-15, 1e4)  #  - true value 100
                   ]

samples, log_prob, priors_out, sampler = mcmc.mcmc( xd,
                                                    yd,
                                                    covfunc,
                                                    covparams_priors,
```

(continues on next page)

(continued from previous page)

```
                                                    noise_prior,
                                                    nwarmup=50,
                                                    niter=50,
                                                    verbose=False)
```

```
Running burn-in...
```

```
100%|| 50/50 [00:09<00:00,  5.36it/s]
```

```
Running production...
```

```
100%|| 50/50 [00:07<00:00,  7.06it/s]
```

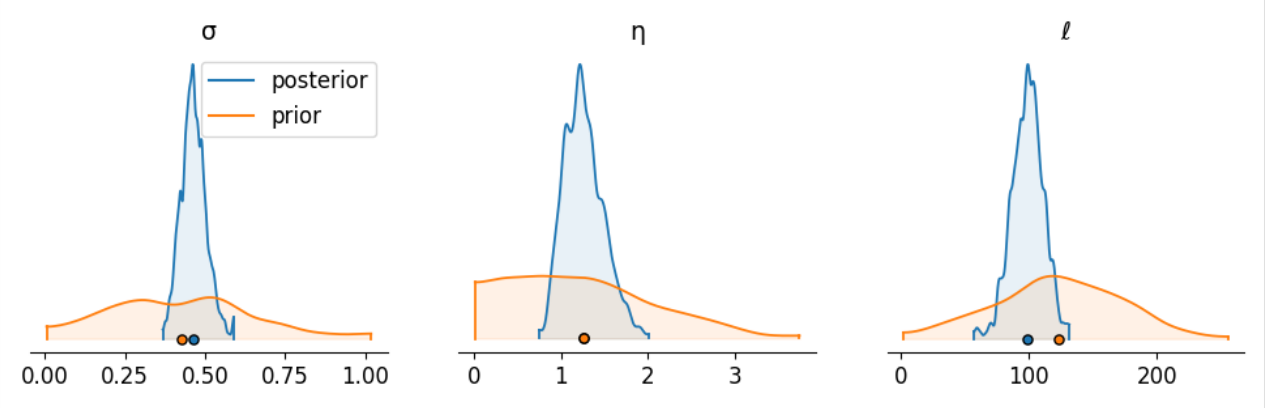
Find sample with highest log prob

[12]:
```
i = np.argmax(log_prob)
MAP = samples[i, :]

print('Noise (true): {:3.2f}, Noise (mcmc): {:3.2f}'.format(noise, MAP[0]))
print(' (true): {:3.2f},  (mcmc): {:3.2f}'.format(covparams[0],  MAP[1]))
print(' (true): {:3.2f},  (mcmc): {:3.2f}'.format(covparams[1],  MAP[2]))
```

```
Noise (true): 0.50, Noise (mcmc): 0.47
 (true): 1.50,  (mcmc): 1.03
 (true): 100.00,  (mcmc): 92.19
```

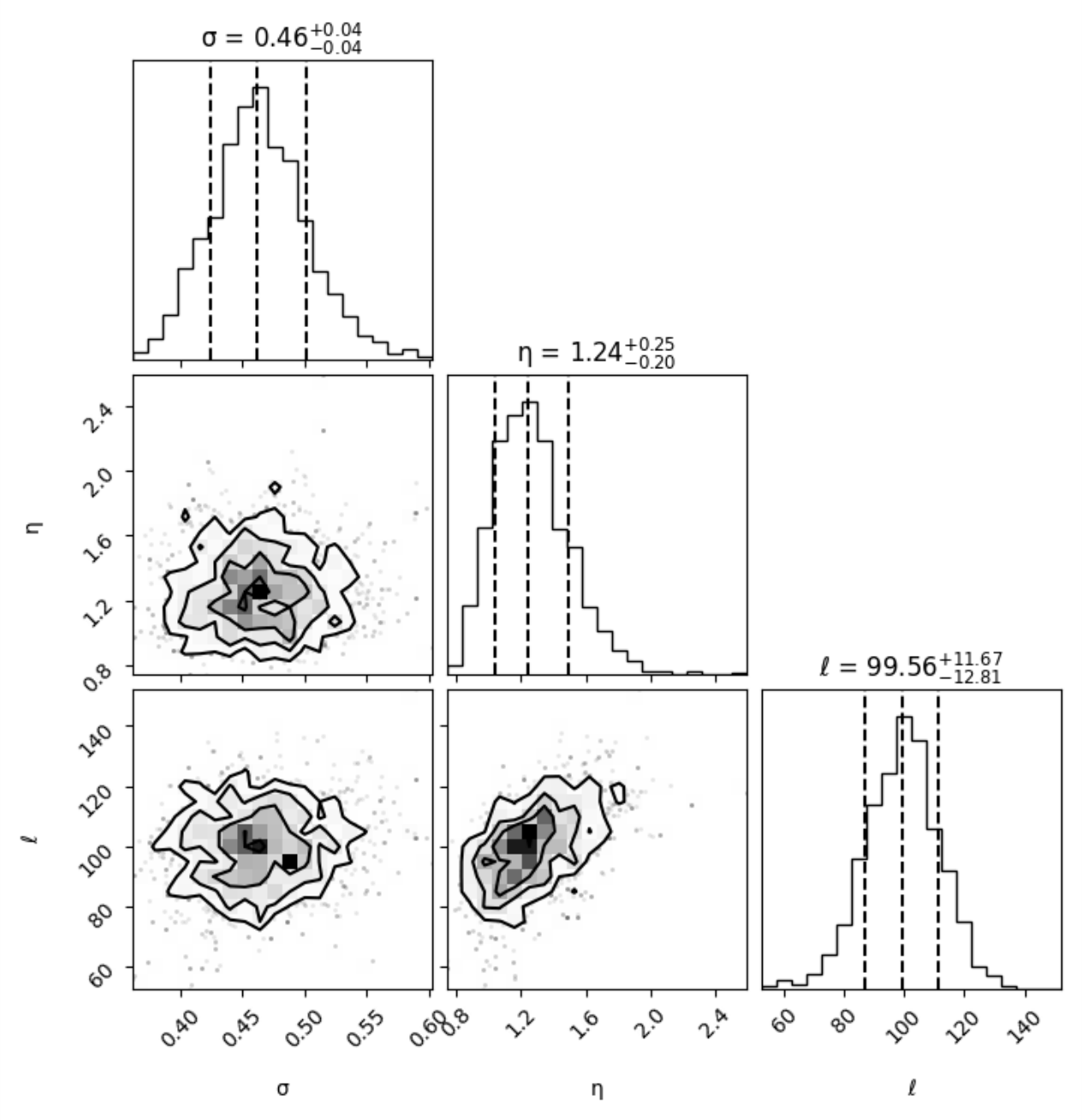
Posterior density plot

[13]:
```
labels = ['','','']
def convert_to_az(d, labels):
    output = {}
    for ii, ll in enumerate(labels):
        output.update({ll:d[:,ii]})
    return az.convert_to_dataset(output)

priors_out_az = convert_to_az(priors_out, labels)
samples_az    = convert_to_az(samples, labels)

axs = az.plot_density(   [samples_az[labels],
                          priors_out_az[labels]],
                          shade=0.1,
                          grid=(1, 3),
                          textsize=12,
                          figsize=(12,3),
                          data_labels=('posterior','prior'),
                          hdi_prob=0.995)
```



Posterior corner plot

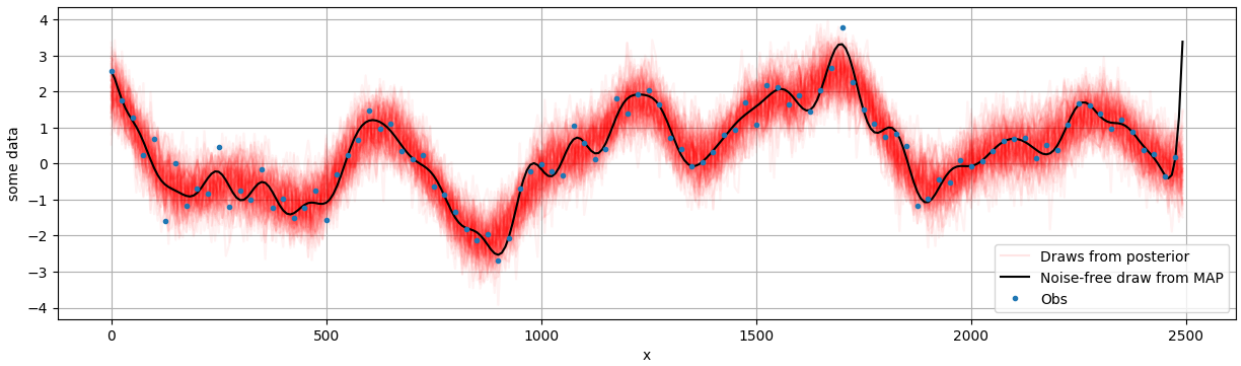
[14]:
```
fig = corner.corner(samples,
                    show_titles=True,
                    labels=labels,
                    plot_datapoints=True,
                    quantiles=[0.16, 0.5, 0.84])
```



Condition and make predictions

[15]:
```
xo = np.arange(0,dx*N,dx/3)[:,None]

OI = GPtideScipy(xd, xo, MAP[0], covfunc, MAP[1:],
                 P=1, mean_func=None)

out_map = OI.conditional(yd)
```

[16]:
```
plt.figure(figsize=(15, 4))
plt.ylabel('some data')
plt.xlabel('x')

for i, draw in enumerate(np.random.uniform(0, samples.shape[0], 100).astype(int)):
    sample = samples[draw, :]
    OI = GPtideScipy(xd, xo, sample[0], covfunc, sample[1:],
             P=1, mean_func=None)
    out_samp = OI.conditional(yd)
    plt.plot(xo, out_samp, 'r', alpha=0.05, label=None)

plt.plot(xo, out_samp, 'r', alpha=0.1, label='Draws from posterior') # Just for legend

OI = GPtideScipy(xd, xo, 0, covfunc, MAP[1:],
             P=1, mean_func=None)
out_map = OI.conditional(yd)

plt.plot(xo, out_map, 'k', label='Noise-free draw from MAP')
plt.plot(xd, yd,'.', label='Obs')
plt.legend()
plt.grid()
```



[ ]:

1.1.4 1D parameter estimation using MCMC: kernel multplication

This example will cover:

- Use MCMC to infer kernel paramaters
- Finding sample with highest log-prob from the mcmc chain
- Visualising results of sampling
- Making predictions

[1]:
```
from gptide import cov
from gptide import GPtideScipy
import numpy as np
import matplotlib.pyplot as plt
```

(continues on next page)

(continued from previous page)

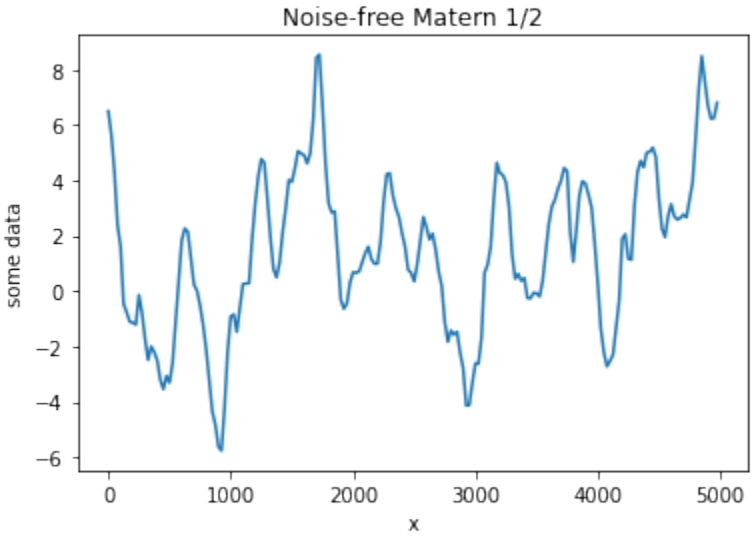
```
import corner
import arviz as az

from scipy import stats
from gptide import stats as gpstats
```

Generate some data

[2]:
```
####
# These are our kernel input parameters
np.random.seed(1)
noise = 0
_m = 4
_m = 150


covfunc = cov.matern32_1d

###
# Domain size parameters
dx = 25.
N = 200
covparams = (_m, _m)

# Input data points
xd = np.arange(0,dx*N,dx)[:,None]

GP = GPtideScipy(xd, xd, noise, covfunc, covparams)

# Use the .prior() method to obtain some samples
yd = GP.prior(samples=1)
```

[3]:
```
plt.figure()
plt.plot(xd, yd)
plt.ylabel('some data')
plt.xlabel('x')
plt.title('Noise-free Matern 1/2')
```

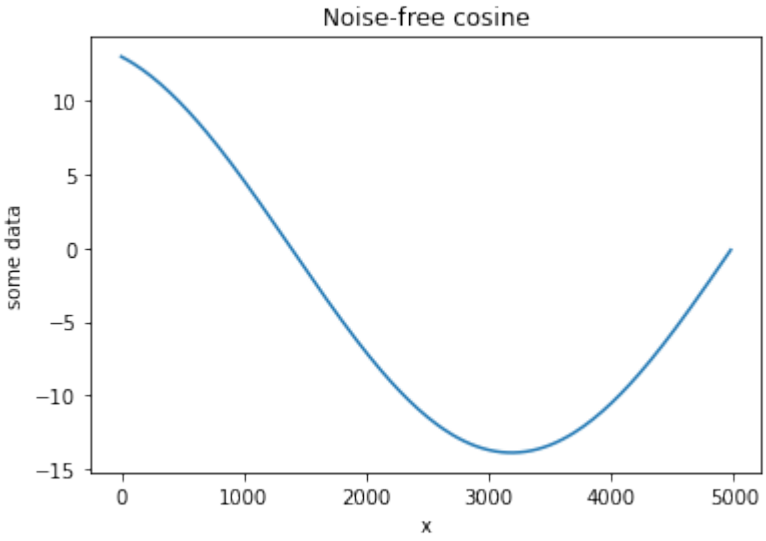
[3]:
```
Text(0.5, 1.0, 'Noise-free Matern 1/2')
```



[4]:
```
np.random.seed(1)
noise = 0

_p = 60
_p = 8

covfunc = cov.cosine_1d

covparams = (_p, _p)

# Input data points
xd = np.arange(0,dx*N,dx)[:,None]

GP = GPtideScipy(xd, xd, noise, covfunc, covparams)

# Use the .prior() method to obtain some samples
yd = GP.prior(samples=1)

def sm(x, xpr, params):

    _m, _m, x_p, _p = params

    return cov.matern32_1d(x, xpr, (_m, _m)) * cov.cosine(x, xpr, (x_p, _p))
```

[5]:
```
plt.figure()
plt.plot(xd, yd)
plt.ylabel('some data')
plt.xlabel('x')
plt.title('Noise-free cosine')
```

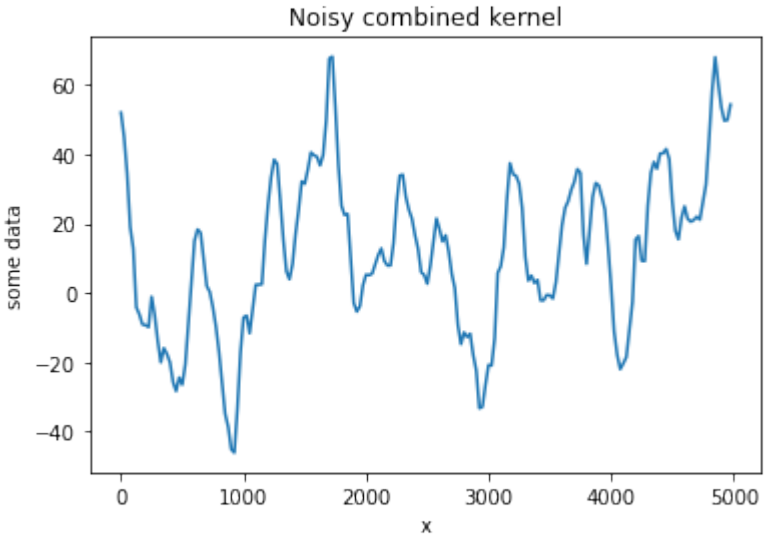
[5]:
```
Text(0.5, 1.0, 'Noise-free cosine')
```



[6]:
```
np.random.seed(1)
noise = 0.5

def sm(x, xpr, params):

    _m, _m, _p, _p = params


    return cov.matern32_1d(x, xpr, (_m, _m)) * cov.cosine_1d(x, xpr, (_p, _p))

covfunc = sm

covparams = (_m, _m, _p, _p)

# Input data points
xd = np.arange(0,dx*N,dx)[:,None]

GP = GPtideScipy(xd, xd, noise, covfunc, covparams)

# Use the .prior() method to obtain some samples
yd = GP.prior(samples=1)
```

[7]:
```
plt.figure()
plt.plot(xd, yd)
plt.ylabel('some data')
plt.xlabel('x')
plt.title('Noisy combined kernel')
```

[7]:
```
Text(0.5, 1.0, 'Noisy combined kernel')
```



Inference

We now use the `gptide.mcmc` function do the parameter estimation. This uses the `emcee.EnsembleSampler` class.

[8]:
```
from gptide import mcmc
n = len(xd)
```

[11]:
```
# Initial guess of the noise and covariance parameters (these can matter)

noise_prior      = gpstats.truncnorm(0.4, 0.25, 1e-15, 1e2)         # noise - true value␣
↪0.5
covparams_priors = [gpstats.truncnorm(1, 1, 1e-15, 1e2),              # _m - true value 4
                    gpstats.truncnorm(125, 50, 1e-15, 1e4),           # _m - true value␣
↪150
                    gpstats.truncnorm(2, 2, 1e-15, 1e4),              # _p - true value 8
                    gpstats.truncnorm(50, 10, 1e-15, 1e4)             # _p - true value␣
↪60
                   ]

samples, log_prob, priors_out, sampler = mcmc.mcmc( xd,
                                                    yd,
                                                    covfunc,
                                                    covparams_priors,
                                                    noise_prior,
                                                    nwarmup=100,
                                                    niter=50,
                                                    verbose=False)
```

```
Running burn-in...
```

```
100%|| 100/100 [01:54<00:00,  1.14s/it]
```

```
Running production...
```

```
100%|| 50/50 [01:00<00:00,  1.20s/it]
```

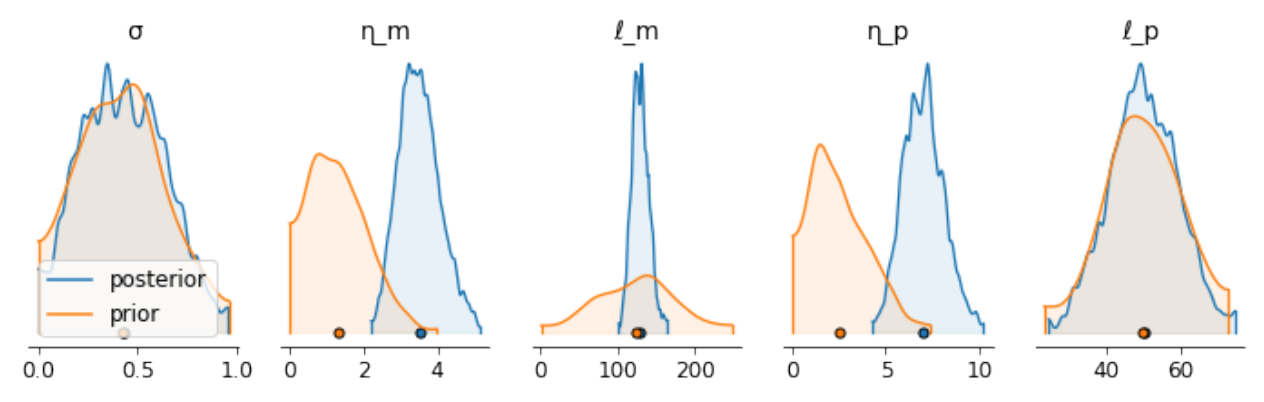
Find sample with highest log prob

[12]:
```
i = np.argmax(log_prob)
MAP = samples[i, :]

print('Noise (true): {:3.2f},  Noise (mcmc): {:3.2f}'.format(noise, MAP[0]))
print('_m (true):   {:3.2f},  _m   (mcmc): {:3.2f}'.format(covparams[0],  MAP[1]))
print('_m (true):   {:3.2f},  _m   (mcmc): {:3.2f}'.format(covparams[1],  MAP[2]))
print('_p (true):   {:3.2f},  _p   (mcmc): {:3.2f}'.format(covparams[2],  MAP[3]))
print('_p (true):   {:3.2f},  _p   (mcmc): {:3.2f}'.format(covparams[3],  MAP[4]))
```

```
Noise (true): 0.50,  Noise (mcmc): 0.77
_m (true):   4.00,  _m   (mcmc): 3.28
_m (true):   150.00,  _m   (mcmc): 142.82
_p (true):   8.00,  _p   (mcmc): 7.81
_p (true):   60.00,  _p   (mcmc): 48.30
```

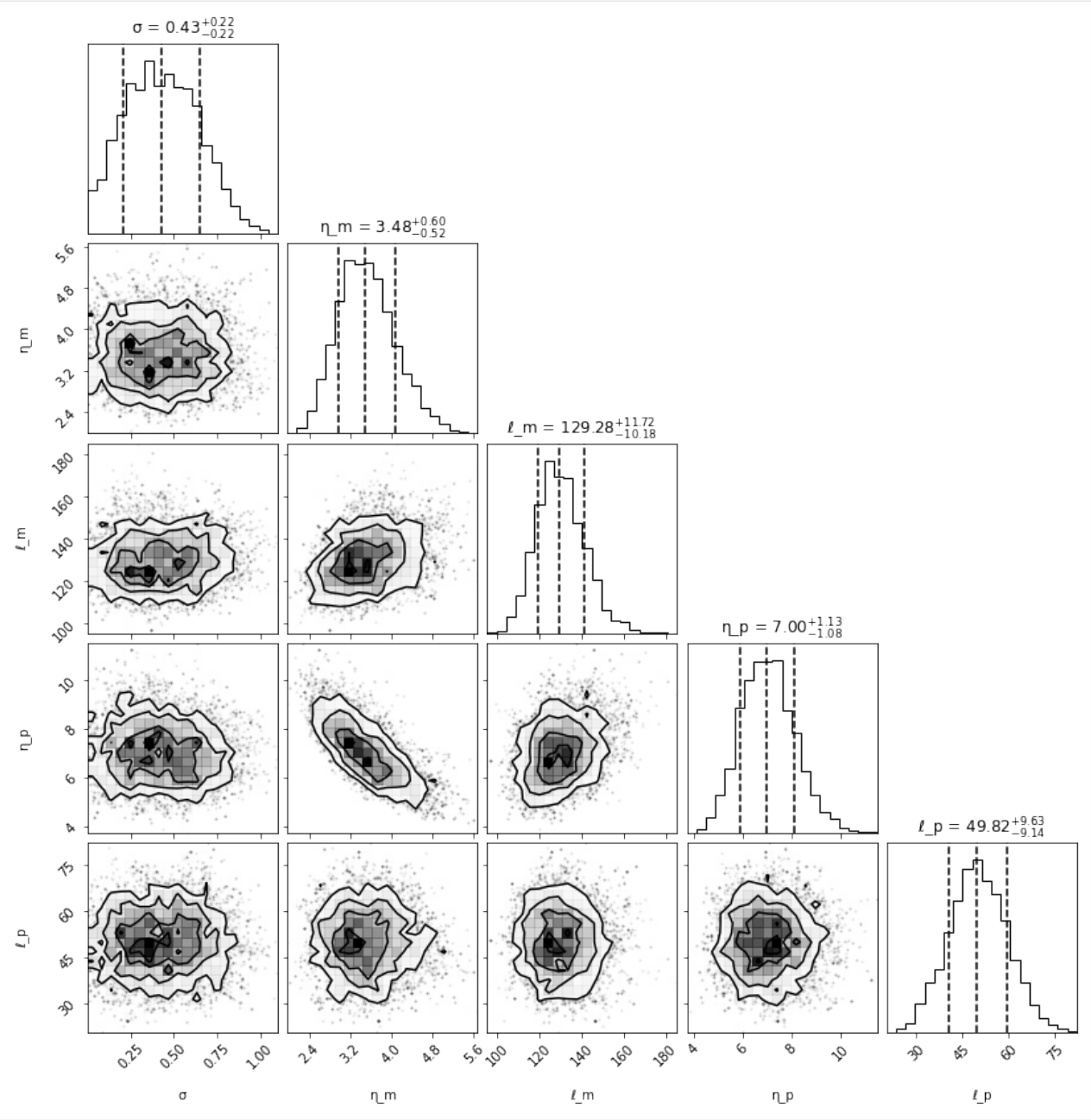
Posterior density plot

[13]:
```
labels = ['','_m','_m','_p','_p']
def convert_to_az(d, labels):
    output = {}
    for ii, ll in enumerate(labels):
        output.update({ll:d[:,ii]})
    return az.convert_to_dataset(output)

priors_out_az = convert_to_az(priors_out, labels)
samples_az    = convert_to_az(samples, labels)

axs = az.plot_density(   [samples_az[labels],
                          priors_out_az[labels]],
                          shade=0.1,
                          grid=(1, 5),
                          textsize=12,
                          figsize=(12,3),
                          data_labels=('posterior','prior'),
                          hdi_prob=0.995)
```



Posterior corner plot

[14]:
```
fig = corner.corner(samples,
                    show_titles=True,
                    labels=labels,
                    plot_datapoints=True,
                    quantiles=[0.16, 0.5, 0.84])
```



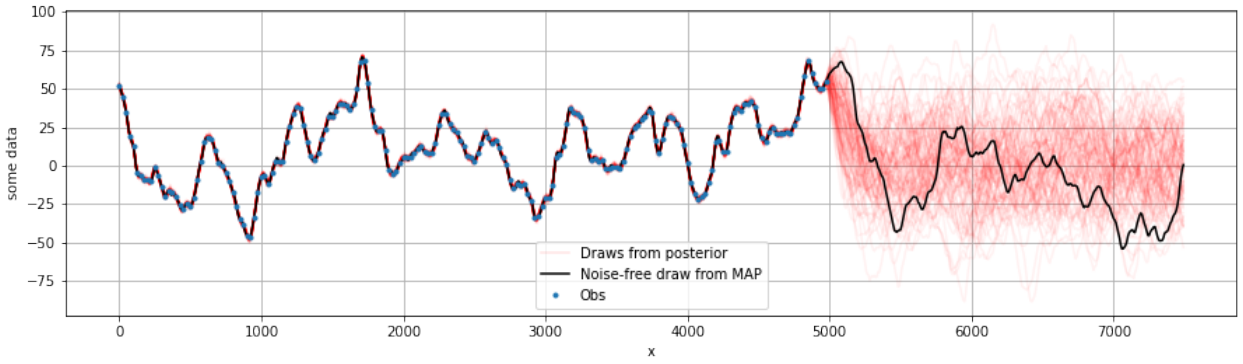
Condition and make predictions

[15]:
```
plt.figure(figsize=(15, 4))
plt.ylabel('some data')
plt.xlabel('x')

xo = np.arange(0,dx*N*1.5,dx/3)[:,None]

for i, draw in enumerate(np.random.uniform(0, samples.shape[0], 100).astype(int)):
    sample = samples[draw, :]
```

(continues on next page)

(continued from previous page)

```
    OI = GPtideScipy(xd, xo, sample[0], covfunc, sample[1:],
             P=1, mean_func=None)
    out_samp = OI.conditional(yd)
    plt.plot(xo, out_samp, 'r', alpha=0.05, label=None)

plt.plot(xo, out_samp, 'r', alpha=0.1, label='Draws from posterior') # Just for legend

OI = GPtideScipy(xd, xo, 0, covfunc, MAP[1:],
             P=1, mean_func=None)
out_map = OI.conditional(yd)

plt.plot(xo, out_map, 'k', label='Noise-free draw from MAP')
plt.plot(xd, yd,'.', label='Obs')
plt.legend()
plt.grid()
```



[ ]:

1.1.5 2D parameter estimation using MCMC

This example will cover:

- Use MCMC to infer kernel paramaters
- Finding sample with highest log-prob from the mcmc chain
- Visualising results of sampling
- Making predictions

[1]:
```
from gptide import cov
from gptide import GPtideScipy
import numpy as np
import matplotlib.pyplot as plt

import corner
import arviz as az

from scipy import stats
```

(continues on next page)

(continued from previous page)

```
from gptide import stats as gpstats
```

Generate some data

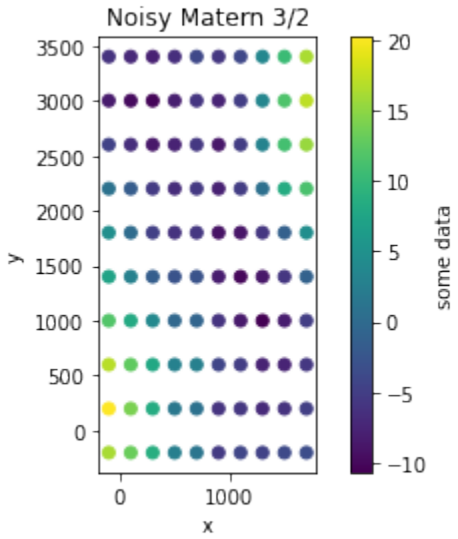
[2]:
```
####
# These are our kernel input parameters
np.random.seed(1)
noise = 0.5
 = 10
_x = 900
_y = 1800

dx = 200.
dy = 400.

def kernel_2d(x, xpr, params):
    """
    2D kernel

    Inputs:
        x: matrices input points [N,3]
        xpr: matrices output points [M,3]
        params: tuple length 3
            eta: standard deviation
            lx: x length scale
            ly: y length scale

    """
    eta, lx, ly = params

    # Build the covariance matrix
    C  = cov.matern32(x[:,1,None], xpr.T[:,1,None].T, ly)
    C *= cov.matern32(x[:,0,None], xpr.T[:,0,None].T, lx)
    C *= eta**2

    return C

covfunc = kernel_2d

###
# Domain size parameters
N = 10

covparams = (, _x, _y)

# Input data points
xd = np.arange(0,dx*N,dx)[:,None]-dx/2
yd = np.arange(0,dy*N,dy)[:,None]-dy/2
```

(continues on next page)

(continued from previous page)

```
# Make a grid
Xg, Yg = np.meshgrid(xd, yd)

# Vectorise grid and stack
Xv = Xg.ravel()
Yv = Yg.ravel()
X = np.hstack([Xv[:,None], Yv[:,None]])

GP = GPtideScipy(X, X.copy(), noise, covfunc, covparams)

# Use the .prior() method to obtain some samples
zd = GP.prior(samples=1)
zg = zd.reshape(Xg.shape)
```

[3]:
```
plt.figure()
plt.scatter(Xg, Yg, c=zg)
plt.ylabel('y')
plt.xlabel('x')
plt.title('Noisy Matern 3/2')
plt.colorbar(label='some data')
plt.gca().set_aspect('equal')
```



Inference

We now use the `gptide.mcmc` function do the parameter estimation. This uses the `emcee.EnsembleSampler` class.

[4]:
```
from gptide import mcmc
n = len(xd)
covparams
```

[4]:
```
(10, 900, 1800)
```

[6]:
```
# Initial guess of the noise and covariance parameters (these can matter)

noise_prior      = gpstats.truncnorm(0.4, 0.25, 1e-15, 1)           # noise - true value␣
↪0.5
covparams_priors = [gpstats.truncnorm(8, 3, 2, 14),                   # eta   - true␣
↪value 10
                    gpstats.truncnorm(600, 200, 1e-15, 1e4),           # _x - true value␣
↪900
                    gpstats.truncnorm(1400, 250, 1e-15, 1e4)           # _y - true value␣
↪1800
                   ]

samples, log_prob, priors_out, sampler = mcmc.mcmc( X,
                                                    zd,
                                                    covfunc,
                                                    covparams_priors,
                                                    noise_prior,
                                                    nwarmup=30,
                                                    niter=100,
                                                    verbose=False)
```

```
Running burn-in...
```

```
100%|| 30/30 [00:16<00:00,  1.81it/s]
```

```
Running production...
```

```
100%|| 100/100 [00:56<00:00,  1.77it/s]
```

Find sample with highest log prob

[7]:
```
i = np.argmax(log_prob)
MAP = samples[i, :]

print('Noise (true): {:3.2f},  Noise (mcmc): {:3.2f}'.format(noise, MAP[0]))
print('   (true):   {:3.2f},       (mcmc): {:3.2f}'.format(covparams[0],  MAP[1]))
print('_x (true):   {:3.2f},  _x   (mcmc): {:3.2f}'.format(covparams[1],  MAP[2]))
print('_y (true):   {:3.2f},  _y   (mcmc): {:3.2f}'.format(covparams[2],  MAP[3]))
```

```
Noise (true): 0.50,  Noise (mcmc): 0.29
   (true):   10.00,       (mcmc): 8.34
_x (true):   900.00,  _x   (mcmc): 705.59
_y (true):   1800.00,  _y   (mcmc): 1692.21
```

[8]:
```
i = np.argmax(log_prob)
MAP = samples[i, :]

print('Noise (true): {:3.2f},  Noise (mcmc): {:3.2f}'.format(noise, MAP[0]))
print('   (true):   {:3.2f},       (mcmc): {:3.2f}'.format(covparams[0],  MAP[1]))
print('_x (true):   {:3.2f},  _x   (mcmc): {:3.2f}'.format(covparams[1],  MAP[2]))
```

(continues on next page)

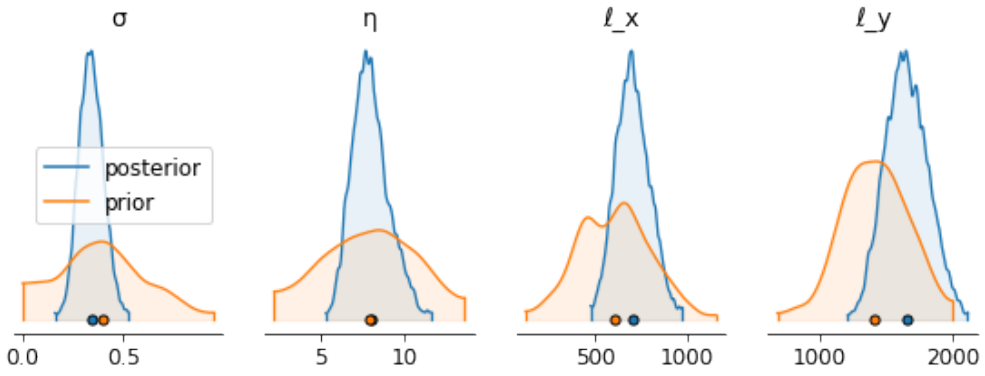
(continued from previous page)

```
print('_y (true):   {:3.2f},  _y   (mcmc): {:3.2f}'.format(covparams[2],  MAP[3]))
```

```
Noise (true): 0.50,  Noise (mcmc): 0.29
   (true):   10.00,        (mcmc): 8.34
_x (true):   900.00,  _x   (mcmc): 705.59
_y (true):   1800.00,  _y   (mcmc): 1692.21
```

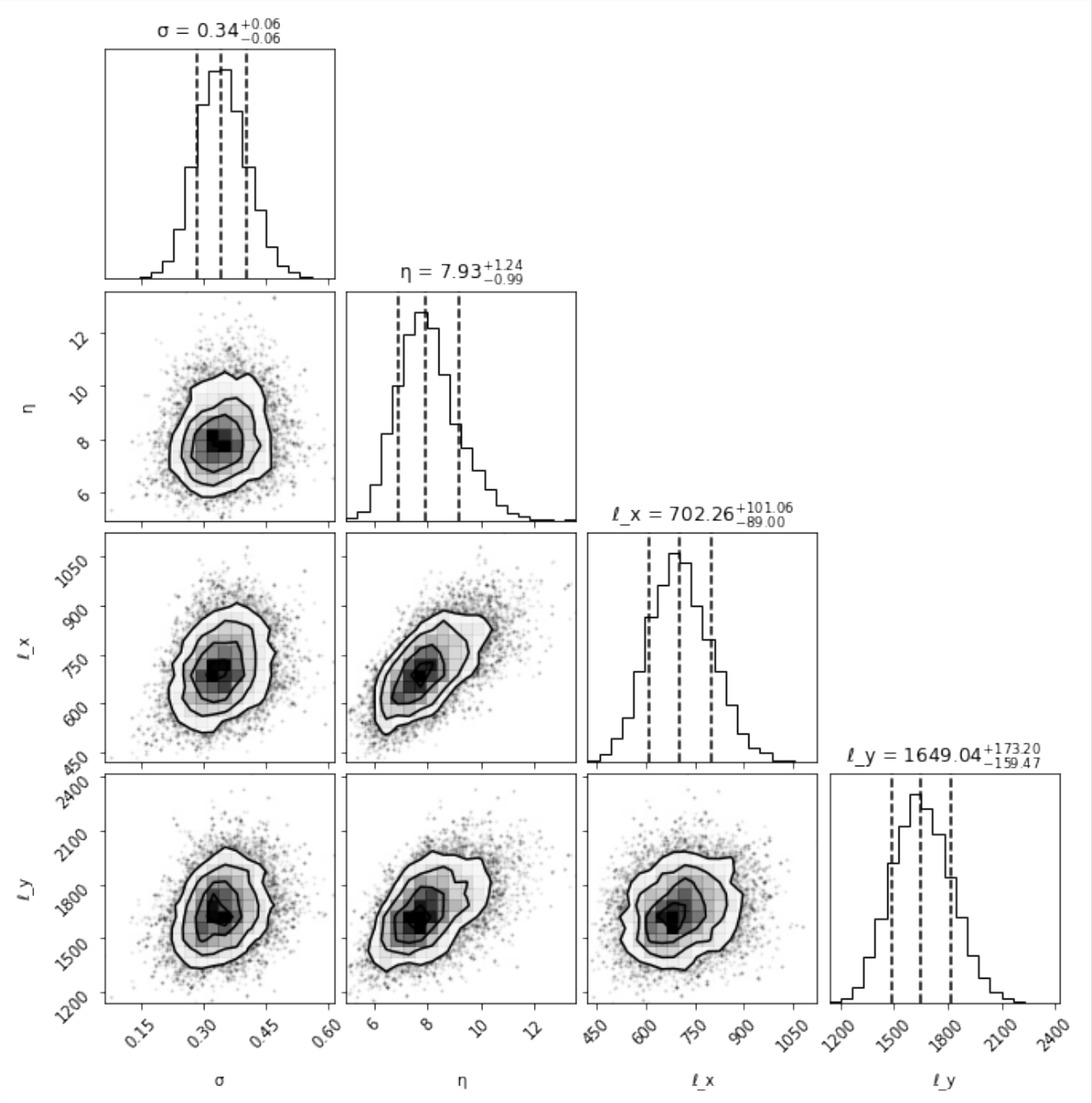
Posterior density plot

[9]:
```
labels = ['','','_x', '_y']
def convert_to_az(d, labels):
    output = {}
    for ii, ll in enumerate(labels):
        output.update({ll:d[:,ii]})
    return az.convert_to_dataset(output)

priors_out_az = convert_to_az(priors_out, labels)
samples_az    = convert_to_az(samples, labels)

axs = az.plot_density(   [samples_az[labels],
                          priors_out_az[labels]],
                          shade=0.1,
                          grid=(1, 5),
                          textsize=12,
                          figsize=(12,3),
                          data_labels=('posterior','prior'),
                          hdi_prob=0.995)
```



Posterior corner plot

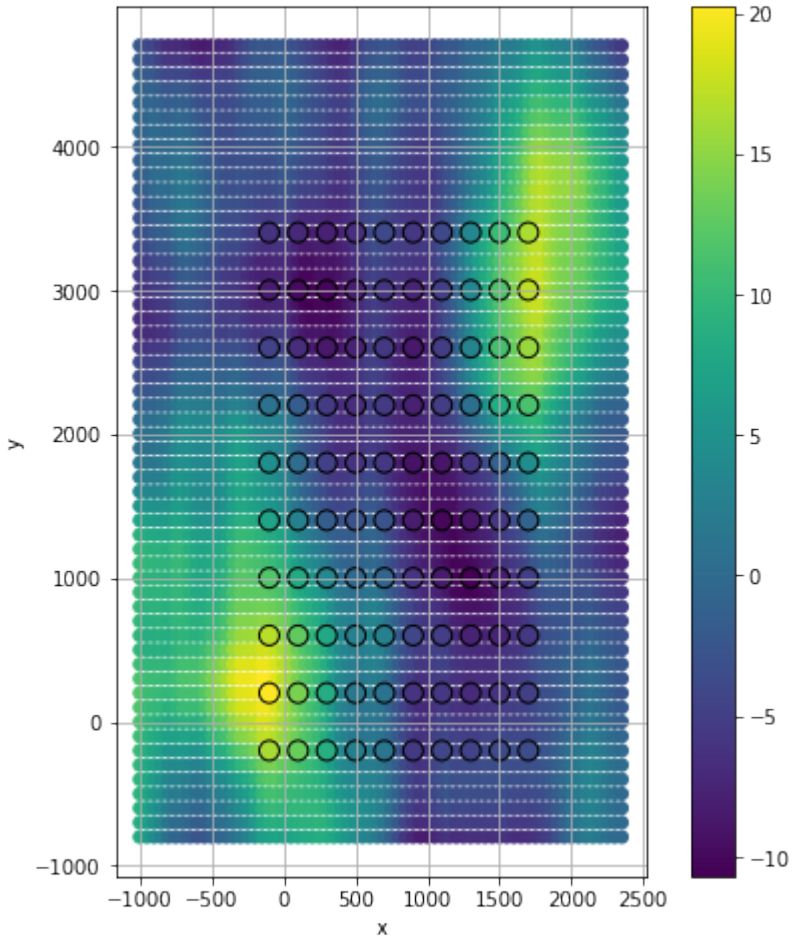
[10]:
```
fig = corner.corner(samples,
                    show_titles=True,
                    labels=labels,
                    plot_datapoints=True,
                    quantiles=[0.16, 0.5, 0.84])
```



Condition and make predictions

[11]:
```
plt.figure(figsize=(8, 8))
plt.ylabel('y')
plt.xlabel('x')

# xo = np.arange(0,dx*N,dx/3)[:,None]
xdo = np.arange(-dx*0.5*N, dx*1.2*N, dx/4)[:,None]
ydo = np.arange(-dy*0.2*N, dy*1.2*N, dy/4)[:,None]

# Make a grid
Xgo, Ygo = np.meshgrid(xdo, ydo)

# Vectorise grid and stack
Xvo = Xgo.ravel()
Yvo = Ygo.ravel()
Xo = np.hstack([Xvo[:,None], Yvo[:,None]])

OI = GPtideScipy(X, Xo, 0, covfunc, MAP[1:],
             P=1, mean_func=None)
out_map = OI.conditional(zd)

plt.scatter(Xgo, Ygo, c=out_map, alpha=1)
plt.scatter(Xg, Yg,   c=zg, s=100, edgecolors='k')

plt.grid()
plt.gca().set_aspect('equal')
plt.colorbar()
```

[11]:
```
<matplotlib.colorbar.Colorbar at 0x1ef276883a0>
```



[ ]:

1.2 API

1.2.1 Main Class

class gptide.gpscipy.**GPtideScipy**(*xd*, *xm*, *sd*, *cov_func*, *cov_params*, ***kwargs*)

Gaussian Process regression class

Uses scipy to do the heavy lifting

Parameters

xd: numpy.ndarray [N, D]
Input data locations

xm: numpy.ndarray [M, D]
Output/target point locations

sd: float
Data noise parameter

cov_func: callable function
Function used to compute the covariance matrices

cov_params: tuple
Parameters passed to *cov_func*

Other Parameters

P: int (default=1)
number of output dimensions

cov_kwargs: dictionary, optional
keyword arguments passed to *cov_func*

mean_func: callable function
Returns the mean function

mean_params: tuple
parameters passed to the mean function

mean_kwargs: dict
kwargs passed to the mean function

Attributes

mean_func

Methods

__call__(yd)	Predict the GP posterior mean given data
conditional(yd[, samples])	Sample from the conidtional distribution
log_marg_likelihood(yd)	Compute the log of the marginal likelihood
prior([samples, noise])	Sample from the prior distribution
update_xm(xm)	Update the output locations and the covariance kernel

__call__(*yd*)

Predict the GP posterior mean given data

Parameters

yd: numpy.ndarray [N,1]
Observed data

Returns

numpy.ndarray
Prediction

__init__(*xd*, *xm*, *sd*, *cov_func*, *cov_params*, ***kwargs*)

Initialise GP object and evaluate mean and covatiance functions.

conditional(*yd*, *samples=1*)

Sample from the conidtional distribution

Parameters

yd: numpy.ndarray [N,1]
Observed data

samples: int, optional (default=1)
number of samples

Returns

conditional_sample: numpy.ndarray [N,samples]
output array

`log_marg_likelihood`(*yd*)

Compute the log of the marginal likelihood

`prior`(*samples=1*, *noise=0.0*)

Sample from the prior distribution

Parameters

samples: int, optional (default=1)
number of samples

Returns

prior_sample: numpy.ndarray [N,samples]
array of output samples

`update_xm`(*xm*)

Update the output locations and the covariance kernel

1.2.2 Parent Class

Classes for Gaussian Process regression

`class gptide.gp.GPtide`(*xd*, *xm*, *sd*, *cov_func*, *cov_params*, ***kwargs*)

Gaussian Process base class

Intended as a placeholder for classes built with other libraries (scipy, jax)

Attributes

mean_func

Methods

`__call__`(yd)	Placeholder
conditional(yd[, samples])	Placeholder
log_marg_likelihood(yd)	Placeholder
prior([samples])	Placeholder
update_xm(xm)	Update the output locations and the covariance kernel

`conditional`(*yd*, *samples=1*)

Placeholder

`log_marg_likelihood`(*yd*)

Placeholder

`prior`(*samples=1*)

Placeholder

update_xm(*xm*)

Update the output locations and the covariance kernel

1.2.3 Covariance functions

Covariance functions for optimal interpolation

gptide.cov.**cosine**(*x*, *xpr*, *l*)

Cosine base function

gptide.cov.**cosine_rw06**(*x*, *xpr*, *l*)

Cosine base function

gptide.cov.**expquad**(*x*, *xpr*, *l*)

Exponential quadration base function/Squared exponential/RBF

gptide.cov.**matern12**(*x*, *xpr*, *l*)

Matern 1/2 base function

gptide.cov.**matern32**(*x*, *xpr*, *l*)

Matern 3/2 base function

gptide.cov.**matern32_matrix**(*d*, *l*)

Non scaled [var 1] Matern 3/2 that takes a distance martrix as an input (i.e. not a vector of coordinates as above)

Parameters

l: Matern length scale
d: distance matrix

gptide.cov.**matern52**(*x*, *xpr*, *l*)

Matern 5/2 base function

gptide.cov.**matern_general**(*dx*, *eta*, *nu*, *l*)

General Matern base function

gptide.cov.**matern_spectra**(*f*, *eta*, *nu*, *l*, *n=1*)

Helper routine to compute the power spectral density of a general Matern function

gptide.cov.**periodic**(*x*, *xpr*, *l*, *p*)

Periodic base function

1.2.4 Inference

MCMC parameter estimation using emcee

gptide.mcmc.**mcmc**(*xd*, *yd*, *covfunc*, *cov_priors*, *noise_prior*, *meanfunc=None*, *mean_priors=[]*, *mean_kwargs={}*, *GPclass=<class 'gptide.gpscipy.GPtideScipy'>*, *gp_kwargs={}*, *nwalkers=None*, *nwarmup=200*, *niter=20*, *nprior=500*, *parallel=False*, *verbose=False*, *progress=True*)

Main MCMC function

Run MCMC using emcee.EnsembleSampler and return posterior samples, log probability of your MCMC chain, samples from your priors and the actual emcee.EnsembleSampler [for testing].

Parameters

xd: numpy.ndarray [N, D]
Input data locations / predictor variable(s)

yd: numpy.ndarray [N,1]
Observed data

covfunc: function
Covariance function

cov_priors: list of scipy.stats.rv_continuous objects
List containing prior probability distribution for each parameter of the covfunc

noise_priors: scipy.stats.rv_continuous object
Prior for I.I.D. noise

Returns

samples:
MCMC chains after burn in

log_prob:
Log posterior probability for each sample in the MCMC chain after burn in

p0:
Samples from the prior distributions

sampler: emcee.EnsembleSampler
The actual emcee.EnsembleSampler used

Other Parameters

meanfunc: function [None]
Mean function

mean_priors: list of scipy.stats.rv_continuous objects
List containing prior probability distribution for each parameter of the meanfunc

mean_kwargs: dict
Key word arguments for the mean function

GPclass: gptide.gp.GPtide class [GPtideScipy]
The GP class used to estimate the log marginal likelihood

gp_kwargs: dict
Key word arguments for the GPclass initialisation

nwalkers: int or None
see emcee.EnsembleSampler. If None it will be 20 times the number of parameters.

nwarmup: int
see emcee.EnsembleSampler

niter: int
see emcee.EnsembleSampler.run_mcmc

nprior: int
number of samples from the prior distributions to output

parallel: bool [False]
Set to true to run parallel

verbose: bool [False]
Set to true for more output

gptide.mle.**mle**(*xd*, *yd*, *covfunc*, *covparams_ic*, *noise_ic*, *meanfunc=None*, *meanparams_ic=[]*, *mean_kwargs={}*, *GPclass=<class 'gptide.gpscipy.GPtideScipy'>*, *gp_kwargs={}*, *priors=None*, *method='L-BFGS-B'*, *bounds=None*, *options=None*, *callback=None*, *verbose=False*)

Main MLE function

Optimise the GP kernel parameters by minimising the negative log marginal likelihood/probability using scipy.optimize.minimize. If priors are specified the log marginal probability is optimised, otherwise the log marginal likelihood is minimised.

Parameters

xd: numpy.ndarray [N, D]
Input data locations / predictor variable(s)

yd: numpy.ndarray [N,1]
Observed data

covfunc: function
Covariance function

covparams_ic: numeric
Initial guess for the covariance function parameters

noise_ic: scipy.stats.rv_continuous object
Initial guess for the I.I.D. noise

Returns

res: OptimizeResult
Result of the scipy.optimize.minimize call

Other Parameters

meanfunc: function [None]
Mean function

meanparams_ic: scipy.stats.rv_continuous object
Initial guess for the mean function parameters

mean_kwargs: dict
Key word arguments for the mean function

GPclass: gptide.gp.GPtide class [GPtideScipy]
The GP class used to estimate the log marginal likelihood

gp_kwargs: dict
Key word arguments for the GPclass initialisation

priors:
List containing prior probability distribution for each parameter of the noise, covfunc and meanfunc. If specified the log marginal probability is optimised, otherwise the log marginal likelihood is minimised.

verbose: bool [False]
Set to true for more output

method: str ['L-BFGS-B']
see scipy.optimize.minimize

bounds: sequence or Bounds, optional [None]
see scipy.optimize.minimize

options: dict, optional [None]
see scipy.optimize.minimize

callback: callable, optional [None]
see scipy.optimize.minimize

1.3 What's new

1.3.1 v0.3.0

Added a Toeplitz solver for evenly spaced 1D input/output problems

1.3.2 v0.2.0

Version with most of the API and more example documentation

1.3.3 v0.1.0

Basic untested version with the main classes and some basic docs

PYTHON MODULE INDEX

g

INDEX

Symbols

C

E

G

L

M

P

U