specgp

Release 0.1

Jul 21, 2020

User Guide

specgp enables 2D Gaussian process computations in exoplanet. This is accomplished by a new kernel term which combines a *celerite* term with a specification of the covariance for the second dimension.

While there are many uses for multidimensional Gaussian processes, one of special relevance in astronomy is simultaneously modeling variability in multiband light curves. Models of this type can be specified in *specgp* by a 1D *celerite* term to specify the common temporal covariance for each band and a vector, α, specifying a scaling relationship between the variability amplitudes in each band. Models of this type can be computed in $\mathcal{O}(NM)$ time for N observations taken in M bands.

specgp is also capable of computing 2D GP models with arbitrary covariance in the second dimension which can be specified with a user-supplied covariance matrix R. In this case the runtime scales as the cube of the size of the second dimension.

specgp is in active development on GitHub.

CHAPTER 1

Installation

specgp can be installed using `pip`:

```
pip install specgp
```

CHAPTER 2

Citation

```
@ARTICLE{2020arXiv200705799G,
        author = {{Gordon}, Tyler and {Agol}, Eric and {Foreman-Mackey}, Daniel},
        title = "{A Fast, 2D Gaussian Process Method Based on Celerite: Applications␣
↪to Transiting Exoplanet Discovery and Characterization}",
        journal = {arXiv e-prints},
        year = 2020,
        month = jul,
        url = {https://arxiv.org/abs/2007.05799}
}
```

CHAPTER 3

API documentation

class specgp.terms.**KronTerm**(*term*, ***kwargs*)

A Kronecker-structured covariance matrix of the form

$$K = \Sigma + T \otimes R$$

with `T` defined by a celerite term and `R` either an outer product of the form

$$R = \alpha\, \alpha^{\mathrm{T}}$$

or an arbitrary covariance matrix.

Parameters

- **term** (*Term*) – A celerite term.
- **alpha or R** (*tensor*) – a vector if alpha or matrix if R. If alpha is provided the matrix `R` is defined as the outer product of alpha with itself and the correlation between the GPs is a simple scaling relation with the scale factors given by the entries in alpha.

posdef(*x*, *diag*)

Check to determine postive definiteness of the Kronecker-structured covariance matrix. This operation is slow, and is thus not recommended to be called repeatedly as a check during optimization. Rather, the user should use this function as a guide to ensuring positive definiteness of the model for varying values of the kernel parameters.

Parameters

- **x** (*tensor*) – The input coordinates.
- **diag** (*tensor*) – The white noise variances. This should be an NxM array where N is the length of x and M is the size of alpha.

Returns

A boolean that is True if the covariance matrix is positive definite and False otherwise. The user will need to call `isposdef.eval()` to compute the returned value from the theano tensor variable.

Return type isposdef

psd(*omega*)
The power spectrum of the Kronecker-structured kernel.

Parameters **omega** (*tensor*) – A vector of frequencies.

Returns:

psd: A matrix with each row the power spectrum for one of the correlated processes.

value(*tau*)
This is not yet implemented, meaning that computing the GP prediction for a 2D GP won't work. This will be implemented in an upcoming release.

class specgp.distributions.**MvUniform**(*lower*, *upper*, **args*, ***kwargs*)
A multivariate uniform distribution.

Parameters

- **lower** – an array of lower bounds
- **upper** – an array of upper bounds

class specgp.means.**KronMean**(*values*)
A constant mean for use with Kronecker-structured kernels.

Parameters **values** (*tensor*) – A matrix with each row containing the mean for each of the correlated processes.

CHAPTER 4

Getting started

This tutorial covers the basics of computing 2D GPs with `exoplanet` using the classes provided by `specgp`. For a more complete tutorial on using `celerite` in `exoplanet`, check out the exoplanet docs. If you're not familiar with `exoplanet`, it might be helpful to take a look at some of those tutorials before working through these.

`exoplanet` and `specgp` are primarily intended for use with `pymc3`. For this reason all variables used by `specgp` are stored as nodes in a computational graph in the form of `theano` tensors. This means that in order to compute the actual value of a variable outside of the `pymc3` model context, we'll need to periodically call `.eval()` to get the actual values of these nodes.

To start, let's define a 1D `celerite` term:

[1]:
```
import numpy as np
import exoplanet as xo

term = xo.gp.terms.SHOTerm(log_S0=0.0, log_w0=1.0, log_Q=-np.log(np.sqrt(2)))
```

Here we've chosen the simple harmonic oscillator (SHO) term, which is a commonly used model for stellar variability. The choice is somewhat arbitrary; `specgp` works with any `celerite` term.

If we're only working in one dimension we can define the GP with this kernel alone. Here `diag` is the white noise *variance* for the GP and `J` is the width of the system (each `SHOTerm` contributes J=2 to the total width).

[2]:
```
t = np.linspace(0, 10, 1000)
gp = xo.gp.GP(
                x=t,
                kernel=term,
                diag=0.01 * np.ones_like(t),
                mean=xo.gp.means.Constant(0.0),
                J=2
              )
```

Let's take a look at a random realization of this GP:

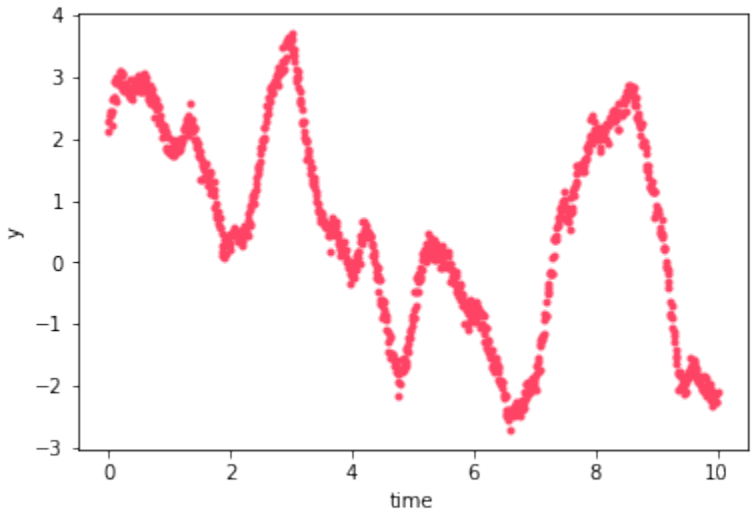
[3]:
```
import matplotlib.pyplot as pl
%matplotlib inline
```

(continues on next page)

(continued from previous page)

```
n = np.random.randn(len(t), 1)
y = gp.dot_l(n).eval()

pl.plot(t, y, '.', color='#FE4365')
pl.xlabel("time")
pl.ylabel("y")
```

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



What if we want to model two timeseries that have the same underlaying variability? This is where `specgp` comes in. It provides a class, `KronTerm` that defines a Kronecker-structured covariance matrix for computing a 2D GP (which can also be thought of as a collection of correlated 1D GPs).

[11]:
```
import specgp as sgp

# scaling factor for each process
alpha = [1, 2]
kernel = sgp.terms.KronTerm(term, alpha=alpha)
```

For the 2D GP, both `diag` (the white noise variances for each input coordinate) and `mean` (the value of the GP mean for each input coordinate) are two dimensional:

[5]:
```
# the white noise components for each process
# here we set the second process to have 100 times the white noise
# variance of the first.
diag = np.array([0.001, 0.1])
diag = diag[:, None] * np.ones_like(t)
print(diag)
```

```
[[0.001 0.001 0.001 ... 0.001 0.001 0.001]
 [0.1   0.1   0.1   ... 0.1   0.1   0.1  ]]
```

For the the GP mean we need to use the provided `KronMean` mean function which is compatible with the `KronTerm` kernel. `KronMean` defines a constant mean function. We give it two values, each being the (constant) mean of one of the processes:

[6]:
```
mu = sgp.means.KronMean(np.zeros((2, len(t))))
```

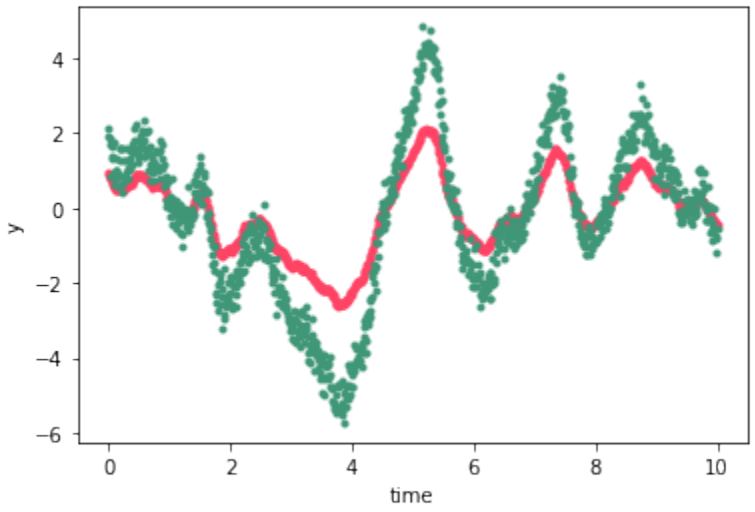
Finally, we can define the 2D GP:

[7]:
```
gp = xo.gp.GP(x=t, kernel=kernel, diag=diag, mean=mu, J=2)
```

Let's take a look at a realization of this GP:

[8]:
```
n = np.random.randn(2*len(t), 1)
y = gp.dot_l(n).eval()

pl.plot(t, y[::2], '.', color='#FE4365')
pl.plot(t, y[1::2], '.', color='#3F9778')
pl.xlabel("time")
pl.ylabel("y")
```

[8]:
```
Text(0, 0.5, 'y')
```

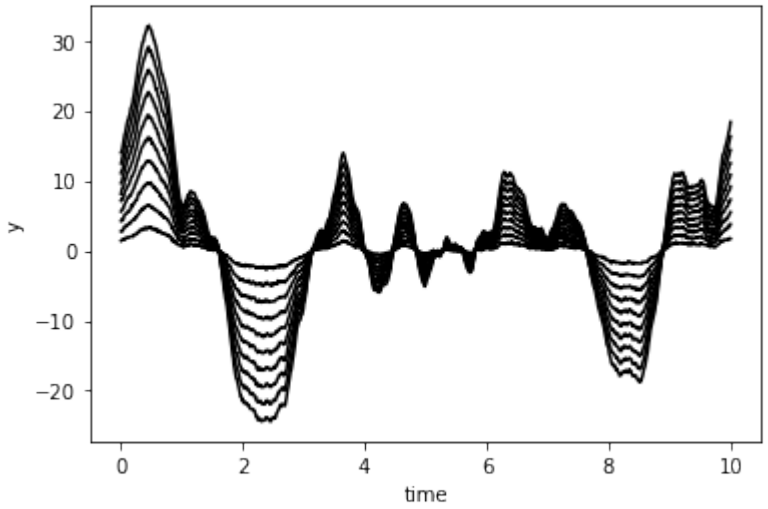


A quick note on the structure of the vector returned by `dot_l`: `y` is a 1D vector with the first 2 elements consisting of the first observation for each of the two processes, the next two being the second observation at each wavelength, and so forth. We could also write $y = vec(Y)$ where Y is a matrix of size $2 \times N$ with N the number of times and with each of the rows containing the timeseries of one of the correlated processes. For M correlated processes the first M elements of y are the observation at the first time for each process, and so forth:

[9]:
```
nprocesses = 10

alpha = np.linspace(1, 10, nprocesses)
kernel = sgp.terms.KronTerm(term, alpha=alpha)

diag = np.array([0.01] * nprocesses)
diag = diag[:, None] * np.ones_like(t)

mu = sgp.means.KronMean(np.zeros((2, len(t))))
gp = xo.gp.GP(x=t, kernel=kernel, diag=diag, mean=mu, J=2)

n = np.random.randn(nprocesses*len(t), 1)
y = gp.dot_l(n).eval()
```

[10]:
```
for i in range(nprocesses):
    pl.plot(t, y[i::nprocesses], '-', color='k')
pl.xlabel("time")
pl.ylabel("y")
```

[10]:
```
Text(0, 0.5, 'y')
```



CHAPTER 5

Sums of Kronecker terms

In some instances a single Kronecker term will not be enough to model a physical system. A star, for instance, may exhibit variability on timescales of days due to rotational modulation and variability and on timescales of minutes to hours due to surface convection. If these two different sources of variability shift the star's spectrum in different ways then the amplitude of the variability will differ with timescale. Modeling this situation will therefore require us to use two Kronecker terms, one to model the wavelength-dependence of the rotationally modulated variability and the other for the convection driven variability.

In this tutorial we will construct two different GP models out of sums of Kronecker-structured covariance matrices. In another tutorial we'll use a similar sum to fit solar variability.

To start, we'll define two `celerite` terms with different characteristic frequencies.

[1]:
```
import numpy as np
import exoplanet as xo

term1 = xo.gp.terms.SHOTerm(log_S0=-3.0, log_w0=2.0, log_Q=-np.log(np.sqrt(2)))
term2 = xo.gp.terms.SHOTerm(log_S0=5.0, log_w0=-1.0, log_Q=-np.log(np.sqrt(2)))
```

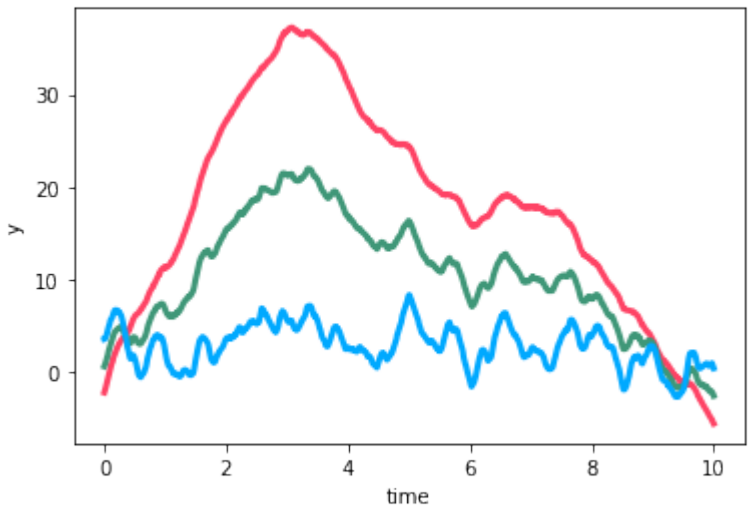
We can combine these with two different scaling vectors using `specgp` and define a GP from the resulting kernel:

[2]:
```
import specgp as sgp

kronterm1 = sgp.terms.KronTerm(term1, alpha=[1, 3, 5])
kronterm2 = sgp.terms.KronTerm(term2, alpha=[5, 3, 1])

kernel = kronterm1 + kronterm2

t = np.linspace(0, 10, 1000)
gp = xo.gp.GP(x=t, kernel=kernel, diag=0.001 * np.ones((3, len(t))), mean=sgp.means.
↪KronMean(np.zeros((2, len(t)))), J=4)
```

Now let's take a look at a random realization of the GP:

[3]:
```
import matplotlib.pyplot as pl
%matplotlib inline

n = np.random.randn(3*len(t), 1)
y = gp.dot_l(n).eval()

pl.plot(t, y[::3], color='#FE4365', linewidth=3)
pl.plot(t, y[1::3], color='#3F9778', linewidth=3)
pl.plot(t, y[2::3], color='#00A9FF', linewidth=3)
pl.ylabel('y')
pl.xlabel('time')
```

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



We can clearly see two different timescales of variability and that they have different wavelength-dependencies. On long timescales the amplitude of the variability increase as we go from the blue to the green to the red curve, but on short timescales the opposite is true: the amplitude of the short timescale variability decreases from blue to red. This behavior cannot be captured by a single `KronTerm`.

Another case in which one may want to combine multiple Kronecker terms is the case in which two processes share a common source of variability but also have their own independent variabilities. For instance, two stars that are observed at the same time via the same instrument will have the same systematics, but with their own variability superimposed. Let's see how we can reproduct this situation using a sum of `KronTerms`. We begin by defining `celerite` terms for the common variability and each of the independent processes:

[4]:
```
common_term = xo.gp.terms.SHOTerm(log_S0=10.0, log_w0=-2.0, log_Q=-np.log(np.sqrt(2)))
ind_term1 = xo.gp.terms.SHOTerm(log_S0=-2.0, log_w0=2.0, log_Q=-np.log(np.sqrt(2)))
ind_term2 = xo.gp.terms.SHOTerm(log_S0=-2.0, log_w0=2.0, log_Q=-np.log(np.sqrt(2)))
```

If we want the common term to have the same amplitude at each wavelength, then the scaling vector should be `[1, 1]`. For the independent terms, the scaling vectors will be `[1, 0]` and `[0, 1]` so that the amplitude of the unwanted term is zero'd out for each time series:

[5]:
```
common_alpha = [1, 1]
ind_alpha1 = [1, 0]
ind_alpha2 = [0, 1]

kernel = (sgp.terms.KronTerm(common_term, alpha=common_alpha) +
          sgp.terms.KronTerm(ind_term1, alpha=ind_alpha1) +
```

(continues on next page)

(continued from previous page)

```
            sgp.terms.KronTerm(ind_term2, alpha=ind_alpha2))
```

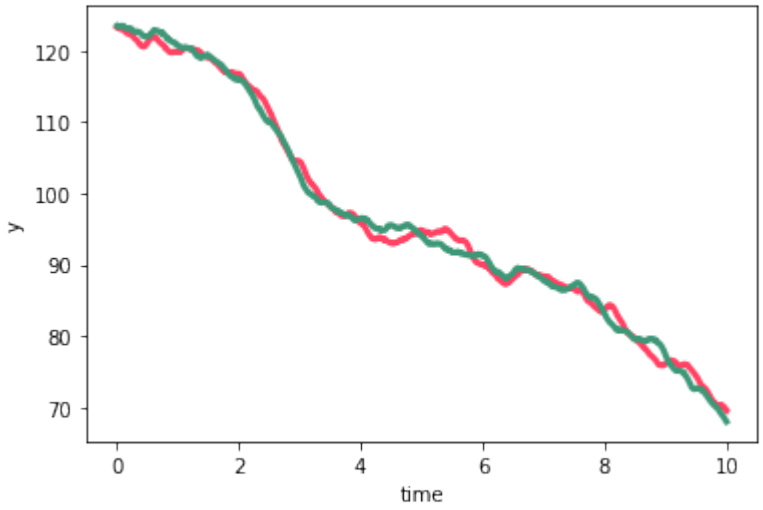
We define a GP with this kernel, this time setting J=6 (J increases by 2 for each SHOTerm).

[6]:
```
gp = xo.gp.GP(x=t,
              kernel=kernel,
              diag=0.001 * np.ones((2, len(t))),
              mean=sgp.means.KronMean(np.zeros((2, len(t)))),
              J=6)
```

As before we can visualize the GP is by examining a random realization:

[7]:
```
n = np.random.randn(2*len(t), 1)
y = gp.dot_l(n).eval()

pl.plot(t, y[::2], color='#FE4365', linewidth=3)
pl.plot(t, y[1::2], color='#3F9778', linewidth=3)
pl.ylabel('y')
pl.xlabel('time')
```

[7]:
```
Text(0.5, 0, 'time')
```



And, as constructed, we see that the two timeseries share the long-timescale, large-amplitude variability while the shorter timescale lower amplitude variability is unique to each timeseries.

CHAPTER 6

Solar variability

In this tutorial we use a three-term Kronecker-structured covariance kernel to model multiband solar variability as observed by the SOHO spacecraft.

We'll be modeling data from the SOHO spacecraft's three-channel sunphotometer. The data we use is available at https://sohowww.nascom.nasa.gov/data/archive.html as three separate fits files.

[1]:
```
import numpy as np
from astropy.io import fits
from astropy.time import Time

blue = fits.open('../../specgp/soho/blue.fits')
green = fits.open('../../specgp/soho/green.fits')
red = fits.open('../../specgp/soho/red.fits')
rgb = red, green, blue

rgb = [f[0].data for f in rgb]
mask = np.all([np.isfinite(f) for f in rgb], axis=0)

start = blue[0].header['DATES'][0:9]
end = blue[0].header['DATES'][14:]
start, end = Time([start, end]).jd
t = np.linspace(start, end, np.shape(rgb)[1]) - start

t = t[mask]
rgb = [f[mask].astype('float64') for f in rgb]
flux = np.sum(rgb, axis=0)/np.shape(rgb)[0]

# choose an arbitrary starting index and number of points to
# select a segment of the (very large) SOHO timeseries
i = 98765
n = 1000
t = t[i:i+n]
# in units of parts per thousand
rgb = [f[i:i+n]/1e3 for f in rgb]
```

(continues on next page)

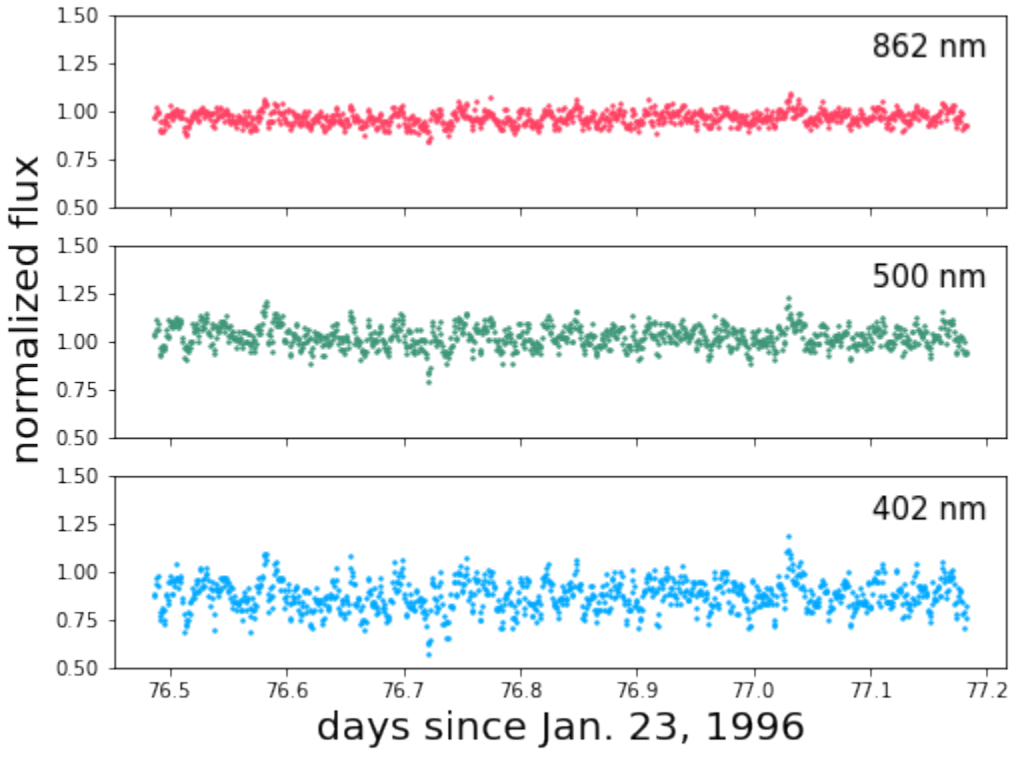
(continued from previous page)

```
# add artificial white noise to the SOHO data in order
# to make the model numerically stable.
rgb += np.random.randn(3, n) * np.exp(-4)
```

Let's plot the data and take a look at the power spectrum at each wavelength:

[2]:
```
import matplotlib.pyplot as pl
%matplotlib inline

colors = ['#FE4365', '#3F9778', '#00A9FF', '#ECA25C', '#005D7F']

fig, ax = pl.subplots(3, 1, figsize=(8, 6), sharex=True)
bands = ['862 nm', '500 nm', '402 nm']

[a.plot(t, 1+f, '.', color=colors[i], ms=3.0) for i, (a, f) in enumerate(zip(ax,␣
↪rgb))]
[a.annotate(b, xy=(0.85, 0.78), xycoords='axes fraction', fontsize=15)
     for a, b in zip(ax, bands)]
[ax.set_ylim(0.5, 1.5) for ax in ax]

ax[2].set_xlabel('days since Jan. 23, 1996', fontsize=20)
pl.annotate("normalized flux", xy=(0.025, 0.4),
            xycoords='figure fraction',
            rotation=90, fontsize=20)
```

[2]:
```
Text(0.025, 0.4, 'normalized flux')
```



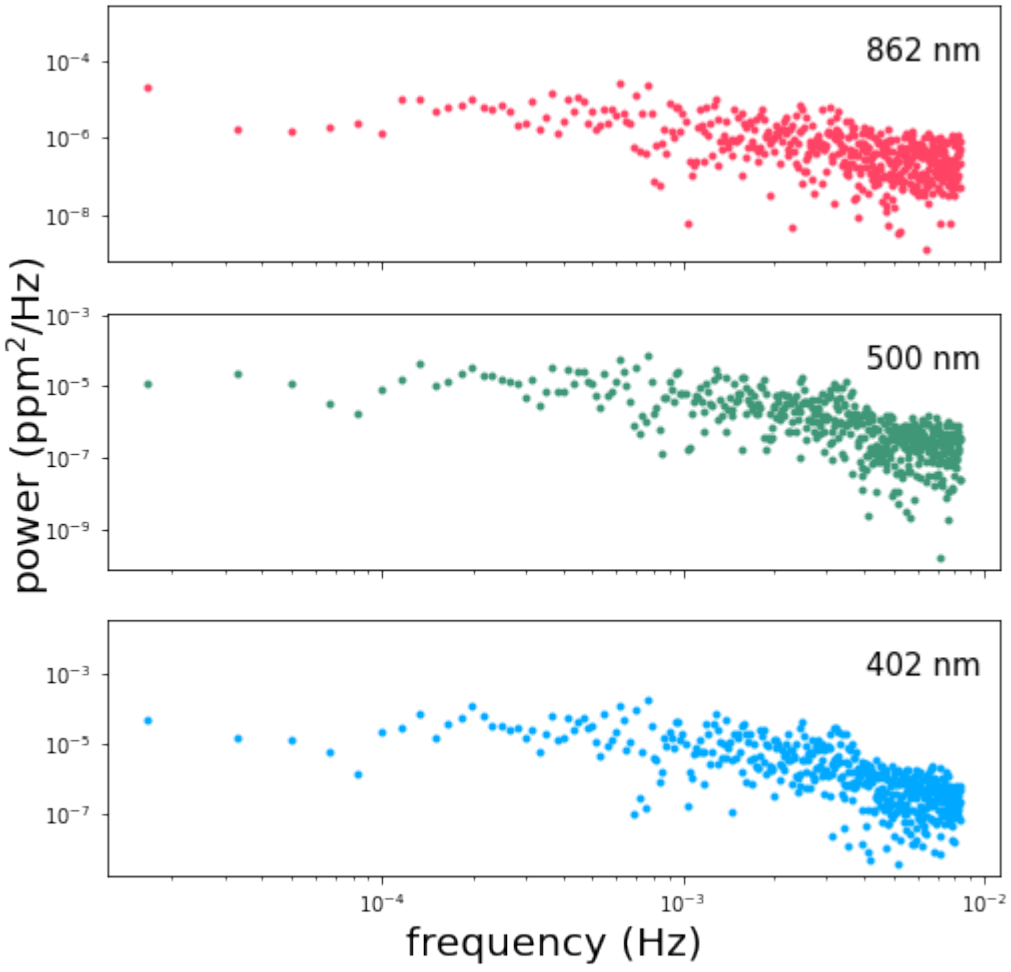
[3]:
```
f = np.fft.rfftfreq(len(t), t[1] - t[0])
f /= 60*60*24
```

(continues on next page)

(continued from previous page)

```
fft = [np.fft.rfft(x) for x in rgb]
fft = [x*np.conj(x) for x in fft]
powerfft = [x.real / len(t)**2 for x in fft]

fig, ax = pl.subplots(3, 1, figsize=(8, 8), sharex=True)
ax[0].loglog(f, powerfft[0], '.', color=colors[0])
ax[1].loglog(f, powerfft[1], '.', color=colors[1])
ax[2].loglog(f, powerfft[2], '.', color=colors[2])

[a.annotate(b, xy=(0.85, 0.78), xycoords='axes fraction', fontsize=15)
     for a, b in zip(ax, bands)]
ax[2].set_xlabel('frequency (Hz)', fontsize=20)
pl.annotate(r"power (ppm$^2$/Hz)", xy=(0.025, 0.4),
            xycoords='figure fraction',
            rotation=90, fontsize=20)
```

[3]:
```
Text(0.025, 0.4, 'power (ppm$^2$/Hz)')
```



The shape of this power spectrum gives us an idea of how we should go about modeling the variability. Depending on the segment of the timeseries that we've chosen, we might observe a slight negative slope in the spectrum at the lowest frequencies, especially in the red band. This is due to trends in the timeseries on scales of days, probably due to rotationally modulated variability. The first term in our covariance kernel is meant to capture this part of the power spectrum.

We see a slight break in the slope of the spectrum right around 10^{-3} Hz, possibly associated with the characteristic timescales of convective granulation on the Sun's surface. The second term in our covariance kernel is meant to capture that feature.

Lastly, we observe a peak in the power at $0.003 - 0.004$ Hz possibly resulting from a superposition of asteroseismic modes. The third term in our covariance kernel is intended to model this feature.

We use celerite's builtin `SHOTerm` to capture the time-correlation for each term in our kernel. For the first two terms we hold the `log_Q` parameter constant at a value of $-\log(\sqrt{2})$, corresponding to a critically-damped oscillator. The third term we allow to vary in order to model the observed peak in the spectrum which results from coherent oscillations in the Sun's brightness.

This next cell is where we define and do inference on our model, so there's a lot to explain. Go ahead and run the code first, and we'll discuss the details afterwards. The last section of this cell runs the mcmc. It usually takes less than 10 minutes on my computer but the runtime can vary by quite a lot.

[7]:
```
import pymc3 as pm
import theano.tensor as tt

import exoplanet as xo
import specgp as sgp

# construct the vector of observations
y = np.vstack(rgb).T.reshape(3*len(t),)

with pm.Model() as model:

    logS0 = sgp.distributions.MvUniform("logS0", lower=[-20]*2, upper=[0.0]*2,␣
↪testval=[-15, -17])
    logw = sgp.distributions.MvUniform("logw", lower=[5, 7], upper=[7, 10],␣
↪testval=[6.5, 7.5])
    logQ = pm.Uniform("logQ", lower=0.0, upper=3.0, testval=1.5)
    alpha1 =  sgp.distributions.MvUniform("alpha1", lower=[0]*2, upper=[2]*2,␣
↪testval=[0.5]*2)
    alpha2 =  sgp.distributions.MvUniform("alpha2", lower=[0]*2, upper=[2]*2,␣
↪testval=[0.5]*2)
    mean = sgp.distributions.MvUniform("mean", lower=[-1]*3, upper=[1]*3, testval=tt.
↪mean(rgb, axis=1))
    logsig = sgp.distributions.MvUniform("logsig", lower=[-15]*3, upper=[0.0]*3,␣
↪testval=[-4]*3)

    term1 = xo.gp.terms.SHOTerm(
        log_S0 = logS0[0],
        log_w0 = logw[0],
        log_Q = -np.log(np.sqrt(2))
    )
    term2 = xo.gp.terms.SHOTerm(
        log_S0 = logS0[1],
        log_w0 = logw[1],
        log_Q = logQ
    )

    a1 = tt.exp(tt.stack([0.0, alpha1[0], alpha2[0]]))
    a2 = tt.exp(tt.stack([0.0, alpha1[1], alpha2[1]]))

    kernel = (sgp.terms.KronTerm(term1, alpha=a1) +
              sgp.terms.KronTerm(term2, alpha=a2))
```

(continues on next page)

(continued from previous page)

```
    yerr = tt.exp(2 * logsig)
    yerr = yerr[:, None] * tt.ones(len(t))

    mean = mean[:, None] * tt.ones(len(t))

    gp = xo.gp.GP(kernel, t, yerr, J=4, mean=sgp.means.KronMean(mean))
    gp.marginal("gp", observed = y)

    start = model.test_point
    map_soln = xo.optimize(start=start, verbose=True)
    start = map_soln

    # comment this out if you don't want to run the mcmc right now.
    trace = pm.sample(
        tune=500,
        draws=500,
        start=start,
        cores=2,
        chains=2,
        step=xo.get_dense_nuts_step(target_accept=0.9)
    )
```

```
optimizing logp for variables: [logsig, mean, alpha2, alpha1, logQ, logw, logS0]
```

```
HBox(children=(IntProgress(value=1, bar_style='info', max=1), HTML(value='')))
```

```

```

```
message: Desired error not necessarily achieved due to precision loss.
logp: 7437.022694511439 -> 8356.637427172627
Multiprocess sampling (2 chains in 2 jobs)
NUTS: [logsig, mean, alpha2, alpha1, logQ, logw, logS0]
Sampling 2 chains, 33 divergences: 100%|| 2000/2000 [02:06<00:00, 15.75draws/s]
There were 14 divergences after tuning. Increase `target_accept` or reparameterize.
There were 19 divergences after tuning. Increase `target_accept` or reparameterize.
```

The very first step in the code block above structures the input (observations) vector the way it needs to be for input to the Kronecker-structured GP. y is a 1D vector with the first 3 elements consisting of the first observation at each of the 3 wavelengths, the next 3 being the second observation at each wavelength, and so forth. We could also write $y = vec(Y)$ where Y is a matrix of size $3 \times N$ with N the number of times and with each of the three rows containing an entire timeseries observed at one wavelength.

The next important thing to notice comes when we define our probabilistic variables. We use the `MvUniform` distribution from `MvUniform` to define a uniform distribution for multiple variables. This distribution is provided purely as a convenience for organizing large numbers of parameters. For instance, we use the `MvUniform` distribution to define one `log_S0` variable that contains an element for each of our three celerite terms. We could just as easily have defined three separate univariate normal distributions.

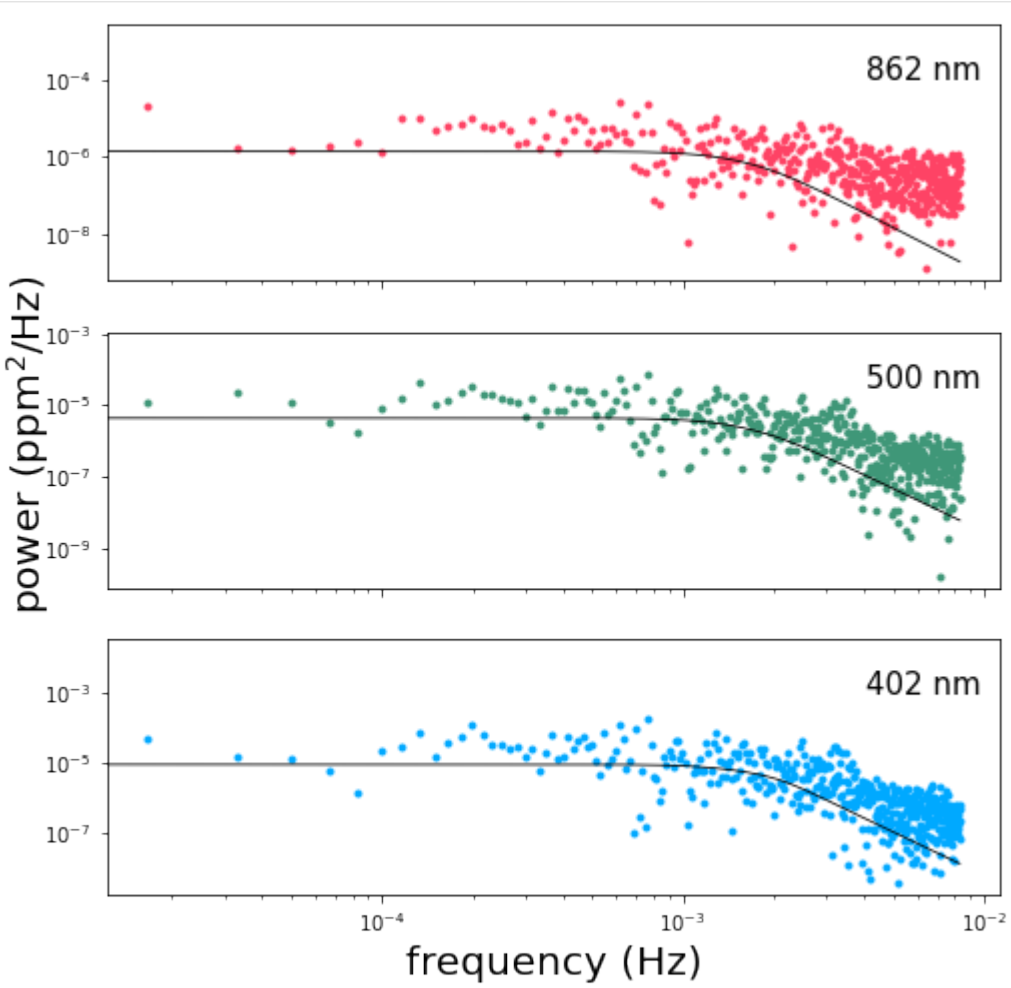
After defining our variables we construct the GP model in much the same way as for a 1D celerite model in exoplanet, the difference being that we wrap each celerite term in a `KronTerm` which combines it with the α vector that defines the covariance for the wavelength dimension.

The final important thing to note is that we must specify a mean consistent with the Kronecker-structured kernel function. For this purpose we provide the `KronMean` mean which takes as its single argument an array of (in this case) 3 constant mean values, one for each wavelength.

Now let's take a look at the psd of the maximum-likelihood solution (as found via `xo.optimize` above, which wraps `scipy.optimize.minimize`):

[8]:
```
fdays = f * 60*60*24
with model:
    psd = xo.eval_in_model(
        kernel.psd(2*np.pi*fdays), map_soln)

fig, ax = pl.subplots(3, 1, figsize=(8, 8), sharex=True)
ax[0].loglog(f, powerfft[0], '.', color=colors[0])
ax[1].loglog(f, powerfft[1], '.', color=colors[1])
ax[2].loglog(f, powerfft[2], '.', color=colors[2])

ax[0].loglog(f, psd.T[0], color='k', linewidth=0.8)
ax[1].loglog(f, psd.T[1], color='k', linewidth=0.8)
ax[2].loglog(f, psd.T[2], color='k', linewidth=0.8)

[a.annotate(b, xy=(0.85, 0.78), xycoords='axes fraction', fontsize=15)
     for a, b in zip(ax, bands)]
ax[2].set_xlabel('frequency (Hz)', fontsize=20)
pl.annotate(r"power (ppm$^2$/Hz)", xy=(0.025, 0.4),
            xycoords='figure fraction',
            rotation=90, fontsize=20)
```

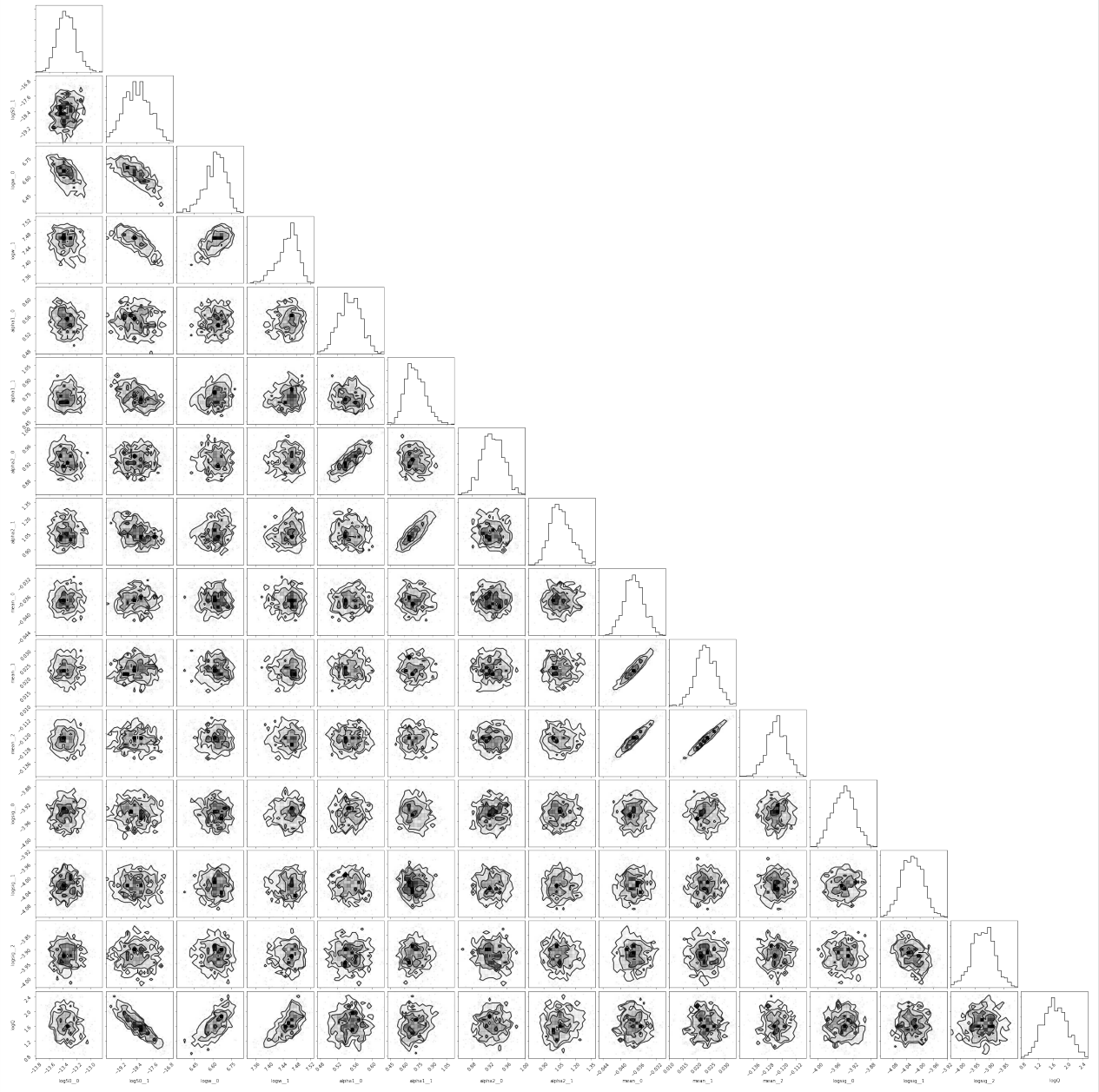
[8]:
```
Text(0.025, 0.4, 'power (ppm$^2$/Hz)')
```



A straightforward optimization doesn't always give us the best solution for the system, but sometimes it does alright.

If you ran MCMC above, then you can also examine the corner plot for the system:

[9]:
```
import corner
samples = pm.trace_to_dataframe(trace)
_ = corner.corner(samples)
```



We can also take a closer look at a random subset of samples from the MCMC chains:

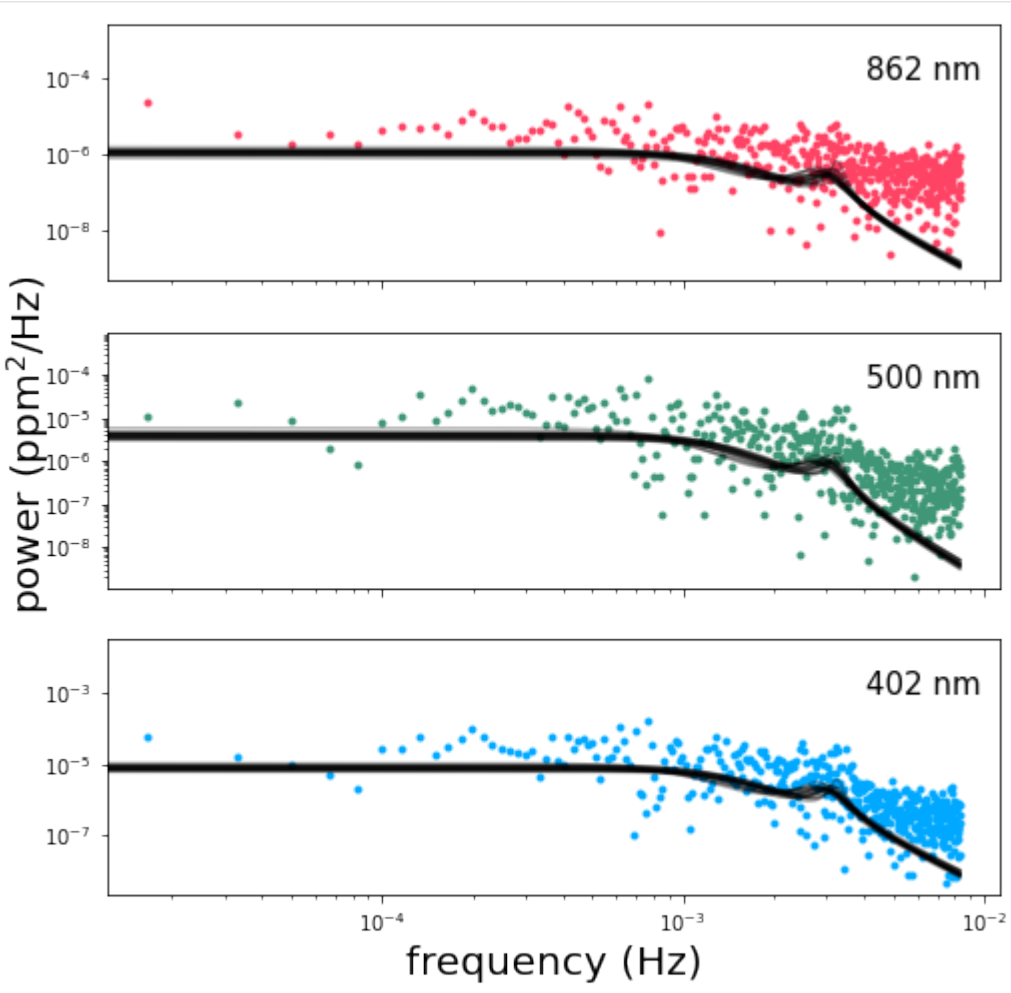
[7]:
```
nsamp = 20
inds = np.random.randint(len(trace), size=nsamp)
with model:
    power = [xo.eval_in_model(
        kernel.psd(2*np.pi*fdays), trace[i])
        for i in inds
    ]
```

(continues on next page)

(continued from previous page)

```
fig, ax = pl.subplots(3, 1, figsize=(8, 8), sharex=True)
ax[0].loglog(f, powerfft[0], '.', color=colors[0])
ax[1].loglog(f, powerfft[1], '.', color=colors[1])
ax[2].loglog(f, powerfft[2], '.', color=colors[2])

for p in power:
    ax[0].loglog(f, p[:,0], color='k', alpha=0.3)
    ax[1].loglog(f, p[:,1], color='k', alpha=0.3)
    ax[2].loglog(f, p[:,2], color='k', alpha=0.3)

[a.annotate(b, xy=(0.85, 0.78), xycoords='axes fraction', fontsize=15)
     for a, b in zip(ax, bands)]
ax[2].set_xlabel('frequency (Hz)', fontsize=20)
pl.annotate(r"power (ppm$^2$/Hz)", xy=(0.025, 0.4),
            xycoords='figure fraction',
            rotation=90, fontsize=20)
```

[7]:
```
Text(0.025, 0.4, 'power (ppm$^2$/Hz)')
```



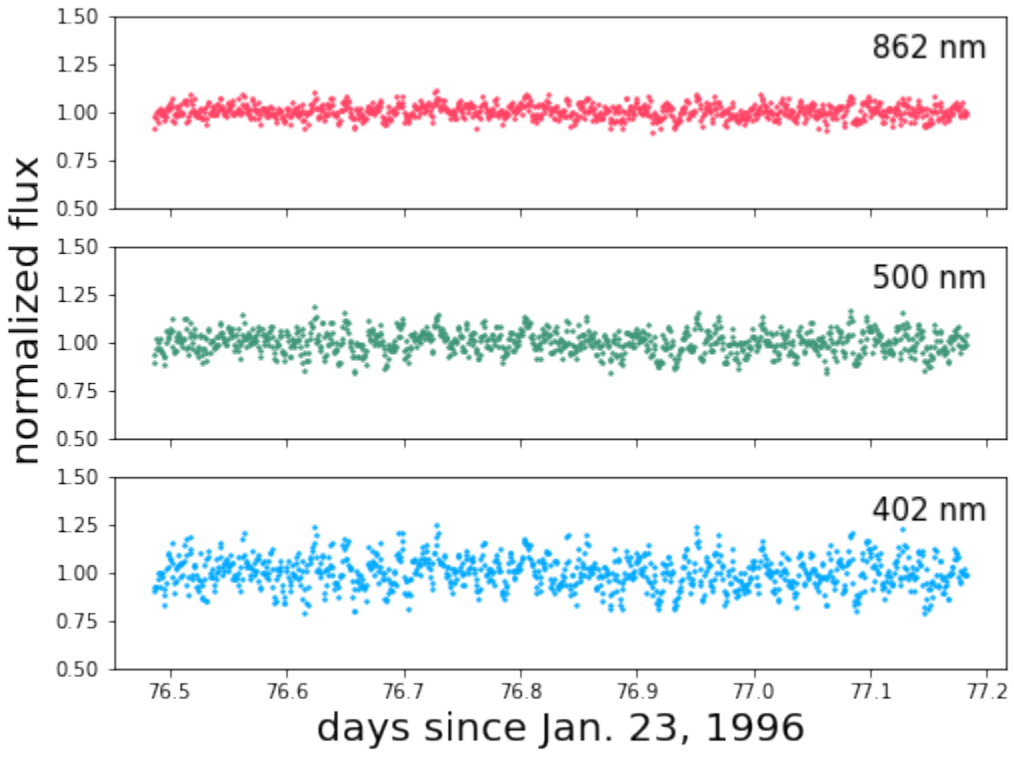
As you can see, the full MCMC analysis is much better at converging towards the maximum likelihood solution.

Finally, if you prefer to look at timeseries in, well, time-space rather than frequency-space, we can plot a random realization of the GP for a random sample from the MCMC to verify that it looks like it has the same noise properties

as the original SOHO data:

[8]:
```
# random index for a random trace
i = np.random.randint(len(trace))

# an array of normally-distributed random numbers to seed
# the GP sample
n = np.random.randn(3*len(t), 1)

with model:
    samp = xo.eval_in_model(gp.dot_l(n), trace[i])
samp_rgb = np.reshape(samp.T, (len(t), 3)).T

fig, ax = pl.subplots(3, 1, figsize=(8, 6), sharex=True)
[a.plot(t, 1+f, '.', color=colors[i], ms=3.0) for i, (a, f) in enumerate(zip(ax, samp_
↪rgb))]
[a.annotate(b, xy=(0.85, 0.78), xycoords='axes fraction', fontsize=15)
     for a, b in zip(ax, bands)]
[ax.set_ylim(0.5, 1.5) for ax in ax]

ax[2].set_xlabel('days since Jan. 23, 1996', fontsize=20)
pl.annotate("normalized flux", xy=(0.025, 0.4),
            xycoords='figure fraction',
            rotation=90, fontsize=20)
```

[8]:
```
Text(0.025, 0.4, 'normalized flux')
```



We end with a few warnings for anyone interested in exploring this example more on their own. As for all covariance matrices, the Kronecker-structured matrices we deal with here must be positive definite for the model to be computed. Unfortunately, positive definiteness is difficult to prove for matrices of this type. We provide a function, `KronTerm.posdef(x, diag)` which takes a vector of input coordinates, `x` and a vector of white noise variances `diag`, and returns a boolean indicating whether the model is positive definite or not for that specific combination of parameters

(those of the celerite term, the vector passed into the sgp.terms.KronTerm, and the white noise variance). This function is slow to compute, and should therefore only be used to explore the positive definiteness of specific models, rather than as a condition within a loop.

In the code above we add white noise to the SOHO data. This is because the SOHO data is very high precision, and without the inclusion of artificial white noise the model easily slips into a part of parameter space where the covariance matrix is not positive definite. In general, a model that is not positive definite can be made so with the addition of white noise or the reduction of the correlated variability amplitude relative to the existing white noise amplitude. Our recommended procedure is to add a known quantity of white noise to the observations such that it stabilizes the model without obscuring features of interest. This should only be necessary for very high precision photometry.

[9]:
```
pm.save_trace(trace)
```

[9]:
```
'.pymc_4.trace'
```

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Index