cmb-map-fit

December 8, 2020

1 Week 13 Problem Set

Ethan Marx, Haochen Wang, Geoffrey Mo 8.942 Fall 2020

Repo: https://github.com/EthanMarx/cmb-map-making

[1]:
```
import cmb_utils
import numpy as np
import pyfftw
import pyfftw.interfaces.numpy_fft as fft
from scipy.interpolate import interp1d
from scipy import linalg as LA
from scipy.integrate import trapz, quad
from scipy.special import jv
from scipy.optimize import minimize
import matplotlib.pyplot as plt

import camb
import fiducial_parameters as fp
import importlib
import emcee
import corner
```

[2]:
```
pix_width = 0.0015707 # radians
cmb_data = np.load('cmb_analysis_pset_data.npz')
```

[3]:
```
for key in cmb_data.keys():
    print(key)
```

```
test_signal
test_white_noise
test_red_noise
test_x
test_y
data_small
x_small
y_small
data_large
```

```
x_large
y_large
```

[4]:
```
cmb_data['test_red_noise'].shape
```

[4]:
```
(65536,)
```

2 1. Map making

2.1 1.1 Operators

All steps in this section are already provided in the starter code (cmb_utils). Section 1.1 in the pset is rather an explanation for what the starter code (more speficially the NoisePointingModel class) does.

2.2 1.2 Estimating the time stream noise power spectrum

First, we choose a data set to be processed.

[5]:
```
dt_test = cmb_data['test_signal'] + cmb_data['test_white_noise'] +␣
↪cmb_data['test_red_noise'] # test data
x_test = np.round(cmb_data['test_x']/pix_width).astype(int) # x coordinates of␣
↪the test data, starting with 0
y_test = np.round(cmb_data['test_y']/pix_width).astype(int) # y coordinates of␣
↪the test data, starting with 0
nx_test = np.amax(x_test) + 1 # Map size in x direction (plus 1 counting the␣
↪0th index)
ny_test = np.amax(y_test) + 1 # Map size in y direction (plus 1 counting the␣
↪0th index)
nt_test = len(x_test) # total number of data points
```

[6]:
```
print(nx_test, ny_test, nt_test)
```

```
32 32 65536
```

1. FFT d_t and divide by $\sqrt{n_t}$ to obtain d_ω.

[7]:
```
dt = dt_test
x = x_test
y = y_test
nx = nx_test
ny = ny_test
nt = nt_test
```

[8]:
```
d_omega = fft.rfft(dt)/np.sqrt(nt) # fft with real data as input
omega = fft.rfftfreq(nt)*2*np.pi # angular frequencies
```

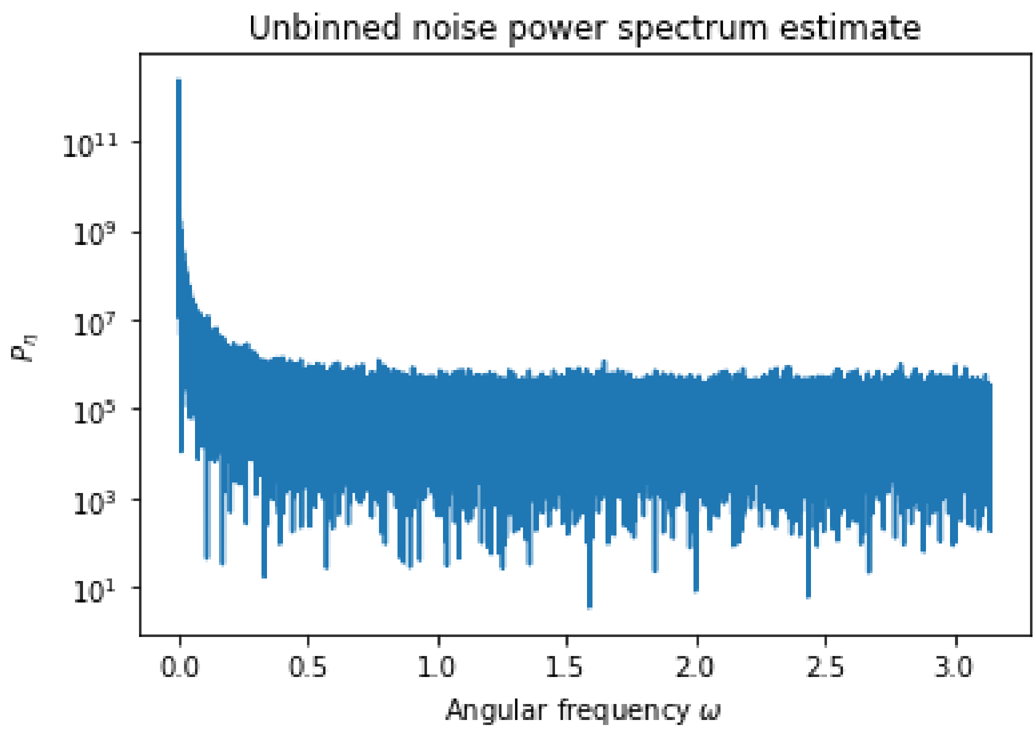
2. The quantity $d_\omega d^*_\omega$ is then a noisy estimate for $P_\eta(\omega)$.

[9]:
```
P_eta_unbinned = d_omega*np.conjugate(d_omega)
```

[10]:
```
plt.plot(omega[:], P_eta_unbinned[:])
plt.title("Unbinned noise power spectrum estimate")
plt.yscale('log')
plt.xlabel(r'Angular frequency $\omega$')
plt.ylabel(r'$P_\eta$')
```

```
/Users/gmo/miniconda3/envs/cosmo-perturb/lib/python3.7/site-
packages/numpy/core/_asarray.py:83: ComplexWarning: Casting complex values to
real discards the imaginary part
  return array(a, dtype, copy=False, order=order)
```

[10]:
```
Text(0, 0.5, '$P_\\eta$')
```



3. Accumulate the estimate over bins in ω to reduce uncertainty

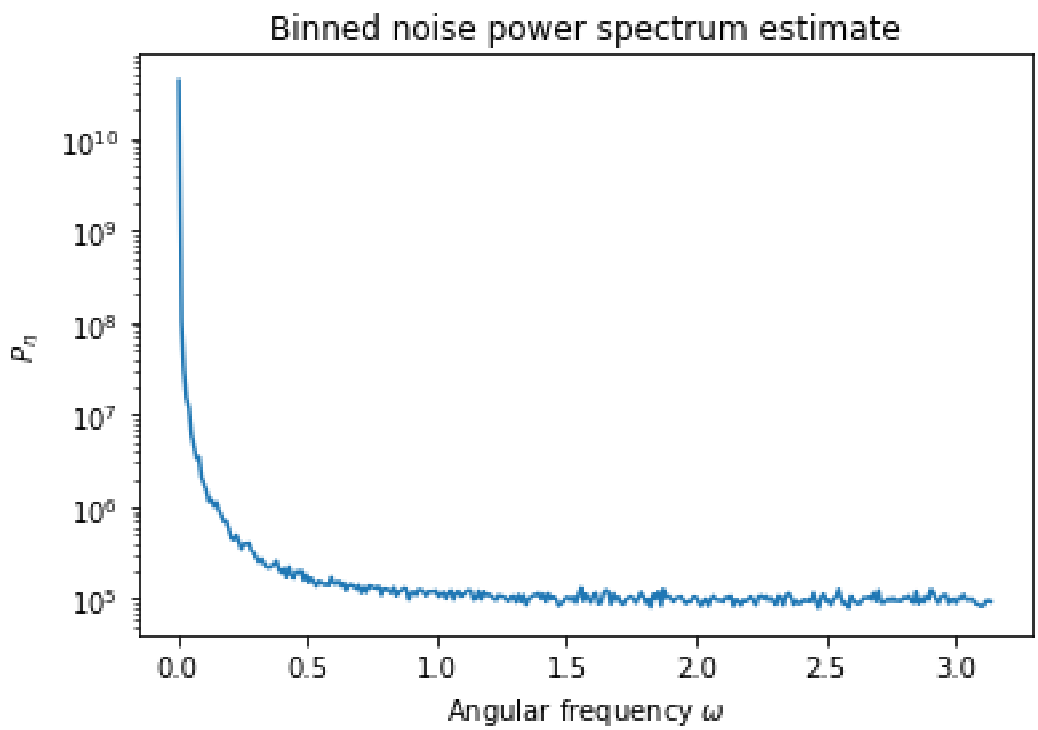
[11]:
```
n_omega = len(omega) # number of frequencies
bins = np.linspace(omega[0], omega[-1], num = int(n_omega/100), endpoint =␣
 ↪True) # bin edges, each bin contains about 100 points
P_eta_binned = np.histogram(omega, bins, weights=P_eta_unbinned)[0]/np.
 ↪histogram(omega, bins)[0]
```

```
bin_centers = np.array([(bins[i] + bins[i+1])/2 for i in range(len(bins) -1)])
```

[12]:
```
plt.plot(bin_centers[:], P_eta_binned[:])
plt.title("Binned noise power spectrum estimate")
plt.yscale('log')
plt.xlabel(r'Angular frequency $\omega$')
plt.ylabel(r'$P_\eta$')
```

```
/Users/gmo/miniconda3/envs/cosmo-perturb/lib/python3.7/site-
packages/numpy/core/_asarray.py:83: ComplexWarning: Casting complex values to
real discards the imaginary part
  return array(a, dtype, copy=False, order=order)
```

[12]: `Text(0, 0.5, '$P_\\eta$')`



4. Interpolate/extrapolate the result to any ω.

[13]:
```
# Before we interpolate, we have to add the first and last data point.␣
↪Otherwise interpolator will return "out of range"
first = np.sum(P_eta_unbinned[0:20])/20 # estimate for P_eta at omega[0], 20 is␣
↪arbitrarily chosen
last = np.sum(P_eta_unbinned[-20:])/20 # estimate for P_eta at omega[-1]
omega_for_interp = np.insert(np.append(bin_centers, omega[-1]), 0, omega[0])
```

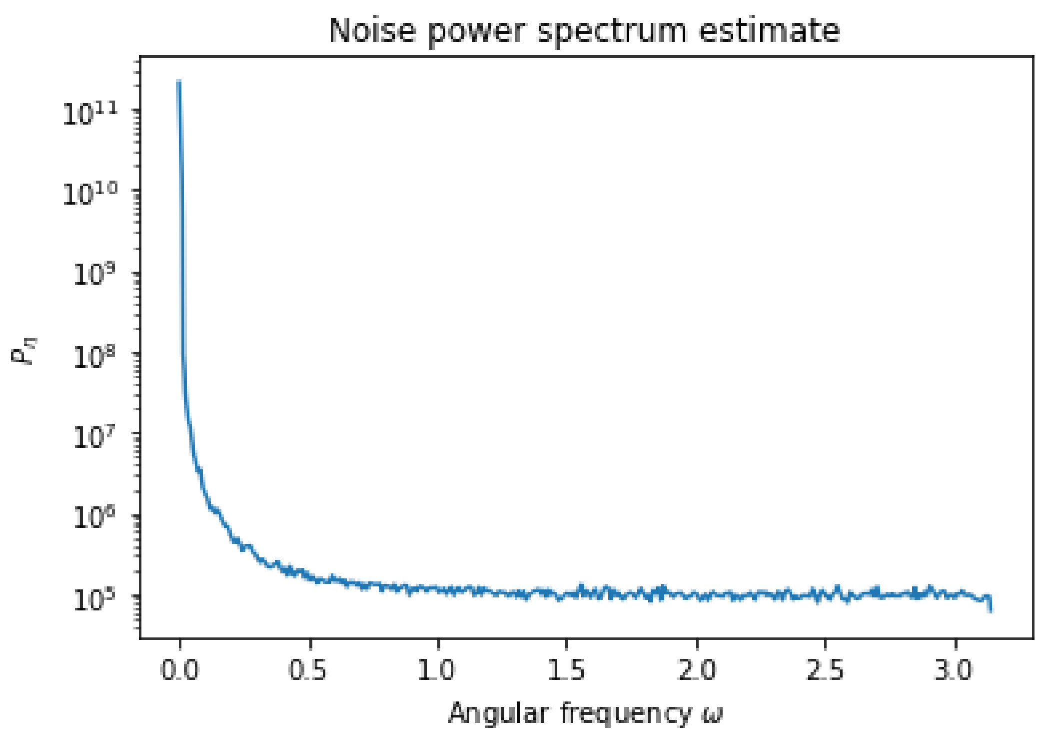
```
P_eta_for_interp = np.insert(np.append(P_eta_binned, last), 0, first)
```

[14]:
```
P_eta_interp = interp1d(omega_for_interp, P_eta_for_interp)
P_eta_final = P_eta_interp(omega) # use interpolator to get P_eta at orignial␣
↪omegas but now it is much smoother
```

[15]:
```
plt.plot(omega, P_eta_final)
plt.title("Noise power spectrum estimate")
plt.yscale('log')
plt.xlabel(r'Angular frequency $\omega$')
plt.ylabel(r'$P_\eta$')
```

```
/Users/gmo/miniconda3/envs/cosmo-perturb/lib/python3.7/site-
packages/numpy/core/_asarray.py:83: ComplexWarning: Casting complex values to
real discards the imaginary part
  return array(a, dtype, copy=False, order=order)
```

[15]:
```
Text(0, 0.5, '$P_\\eta$')
```



2.3 1.3 Noise covariance inverse

Obtain C_N^{-1}

[16]:
```python
model_test = cmb_utils.NoisePointingModel(x_test, y_test, nx_test, ny_test,␣
↪P_eta_final)
CN_inv_test = model_test.map_noise_inv()
CN_inv_test.shape
```

```
HBox(children=(HTML(value=''), FloatProgress(value=0.0, max=32.0),␣
↪HTML(value='')))
```

[16]:
```
(32, 32, 32, 32)
```

Reshape C_N^{-1} into a 2D matrix.

[17]:
```python
CN_inv_test_reshape = np.reshape(CN_inv_test, (nx*ny, nx*ny))
CN_inv_test_reshape.shape
```

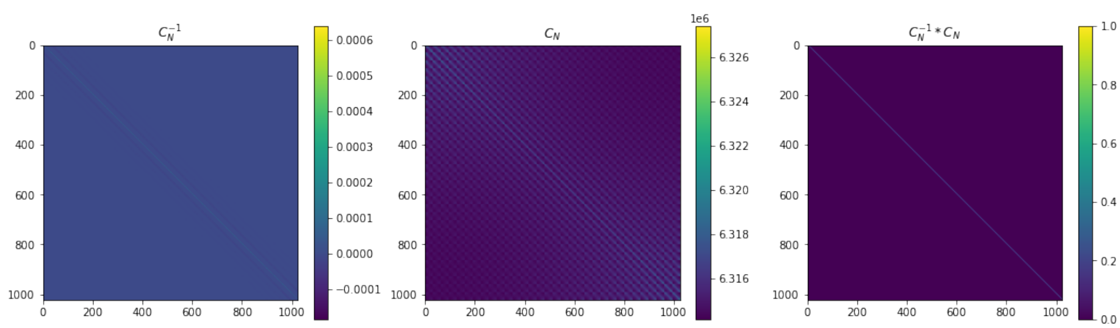
[17]:
```
(1024, 1024)
```

Invert to obtain C_N.

[18]:
```python
CN_test = LA.inv(CN_inv_test_reshape)
```

[19]:
```python
plt.figure(figsize=(18,5))
plt.subplot(131)
plt.imshow(CN_inv_test_reshape)
plt.colorbar()
plt.title(r'$C_N^{-1}$')
plt.subplot(132)
plt.imshow(CN_test)
plt.colorbar()
plt.title(r'$C_N$')
plt.subplot(133)
plt.imshow(np.dot(CN_test,CN_inv_test_reshape))
plt.colorbar()
plt.title(r'$C_N^{-1}*C_N$')
```

[19]:
```
Text(0.5, 1.0, '$C_N^{-1}*C_N$')
```



Get the signal estimate with D&S (14.30)

[20]:
```
N_inv_d_test = model_test.apply_noise_weights(dt)
out = np.zeros((nx, ny), dtype=float)
P_T_N_inv_d_test = model_test.grid_data(N_inv_d_test, out)
```

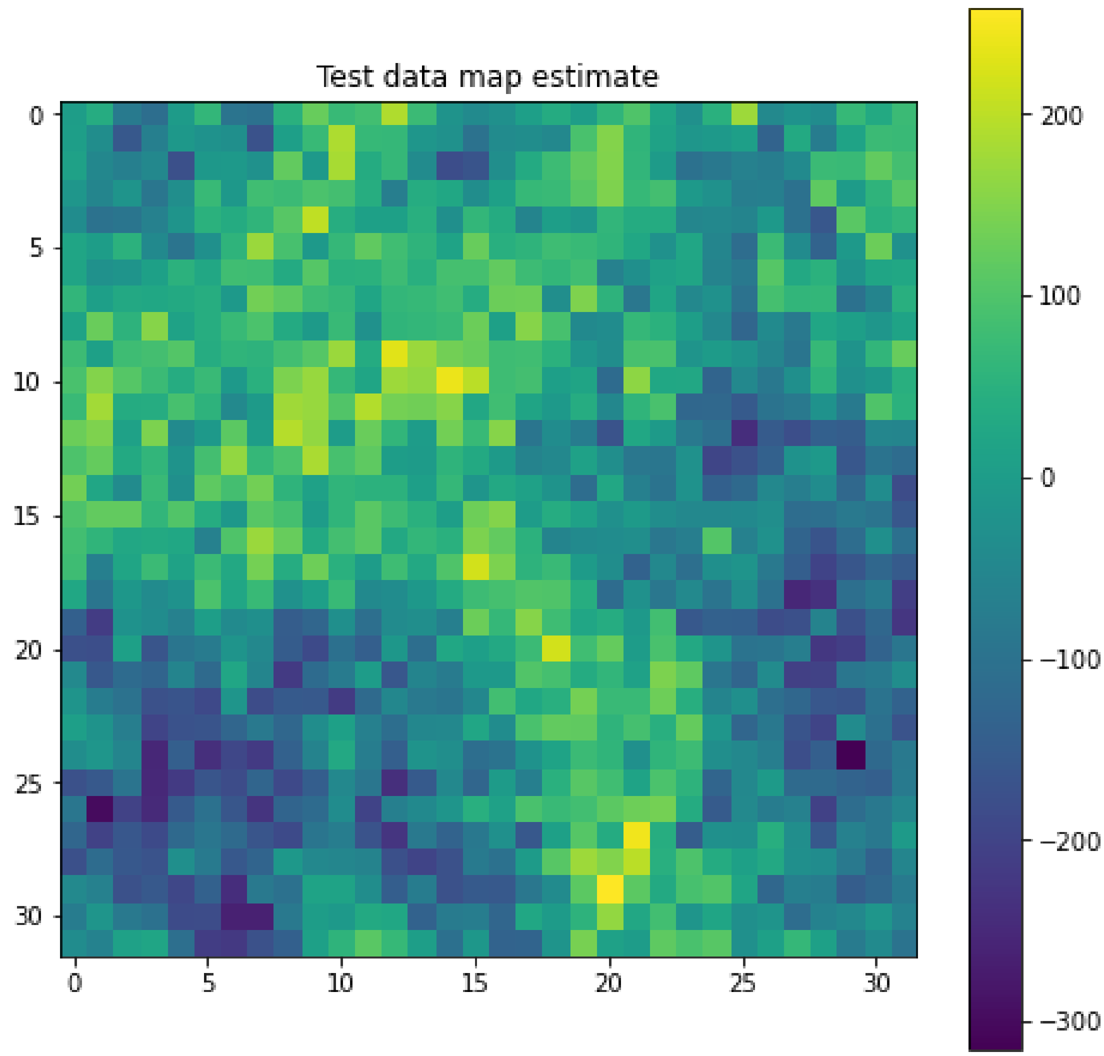
[21]:
```
P_T_N_inv_d_test.shape
```

[21]:
```
(32, 32)
```

[22]:
```
s_hat_test = np.dot(CN_test, P_T_N_inv_d_test.flatten())
s_hat_test = np.reshape(s_hat_test, (nx, ny))
```

[23]:
```
plt.figure(figsize=(8,8))
plt.imshow(s_hat_test)
plt.colorbar()
plt.title('Test data map estimate')
```

[23]:
```
Text(0.5, 1.0, 'Test data map estimate')
```



3 1.4 Testing

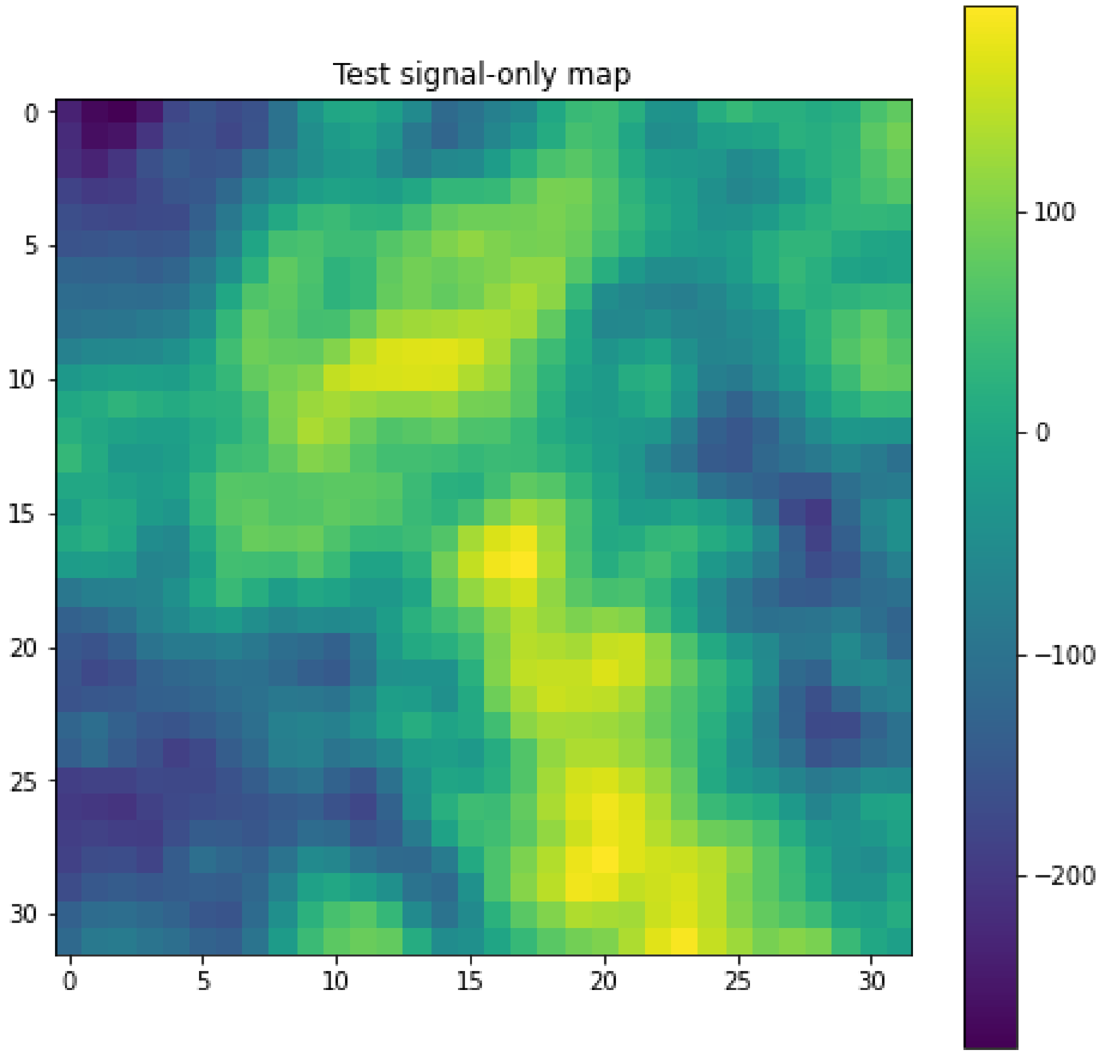
Check the result with test signal only and construct the map with D&S (14.33)

[24]:
```
dt_signal = cmb_data['test_signal']
```

[25]:
```
s_hat_test_signal = np.zeros((nx, ny), dtype=float)
m = np.zeros((nx, ny), dtype=float)
np.add.at(s_hat_test_signal, (x, y), dt_signal)
np.add.at(m, (x, y), 1)
s_hat_test_signal = s_hat_test_signal/m
```

[26]:
```
plt.figure(figsize=(8,8))
plt.imshow(s_hat_test_signal)
plt.colorbar()
plt.title('Test signal-only map')
```

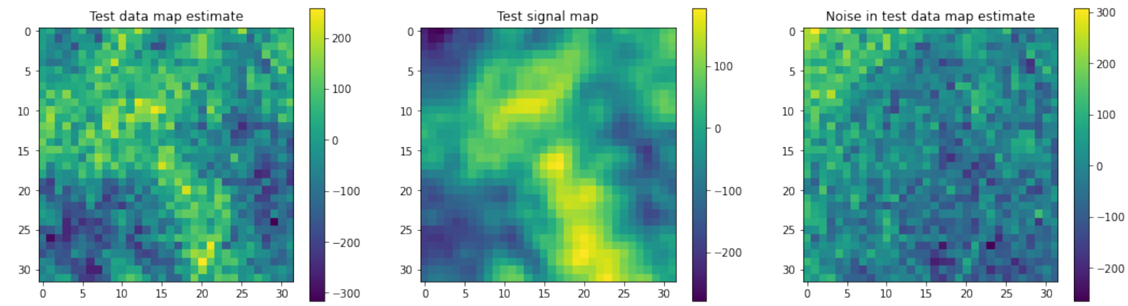
[26]: Text(0.5, 1.0, 'Test signal-only map')



[27]:
```
plt.figure(figsize=(18,5))
plt.subplot(131)
plt.imshow(s_hat_test)
plt.colorbar()
plt.title('Test data map estimate')
plt.subplot(132)
plt.imshow(s_hat_test_signal)
```

```
plt.colorbar()
plt.title('Test signal map')
plt.subplot(133)
plt.imshow(s_hat_test - s_hat_test_signal)
plt.colorbar()
plt.title('Noise in test data map estimate')
```

[27]: Text(0.5, 1.0, 'Noise in test data map estimate')



4 2. Power Spectrum estimation

In this section, we are looking to estimate the band powers, c^{α}

Applying Dodelson 1st edition eqn 11.94 in this context gives:

$$\hat{c^{\alpha}} = c_0^{\alpha} + F_{\alpha\beta}^{-1}\frac{sC^{-1}C_{,\beta}C^{-1}s - Tr(C^{-1}C_{,\beta})}{2} \tag{1}$$

where

$$F_{\alpha\beta} = \frac{Tr(C_{,\alpha}C^{-1}C_{,\beta}C^{-1})}{2} \tag{2}$$

$$C = C_S + C_N \tag{3}$$

s is our signal map created in the previous part, and c_0^{α} is calculated from a fiducial model, in our case, CAMB

We have already calculated C_N in the previous part, we just need C_S, which enters into the equation for $\hat{c^{\alpha}}$ in two parts:

1. C^{-1}
2. $C_{,\alpha}$

4.1 2.1 Signal Covariance Matrix

To account for C_S in the calculation of C^{-1}, we will use our fiducial cosmology to fix C_l and apply:

$$C_S = \bar{T}^2 \omega(|\theta_i - \theta_j|, \lambda_\alpha) \tag{4}$$

with

$$\omega(\theta, \lambda_\alpha) = \int_0^\infty \frac{dl}{2\pi} C_l(\lambda_\alpha) J_0(l\theta) \tag{5}$$

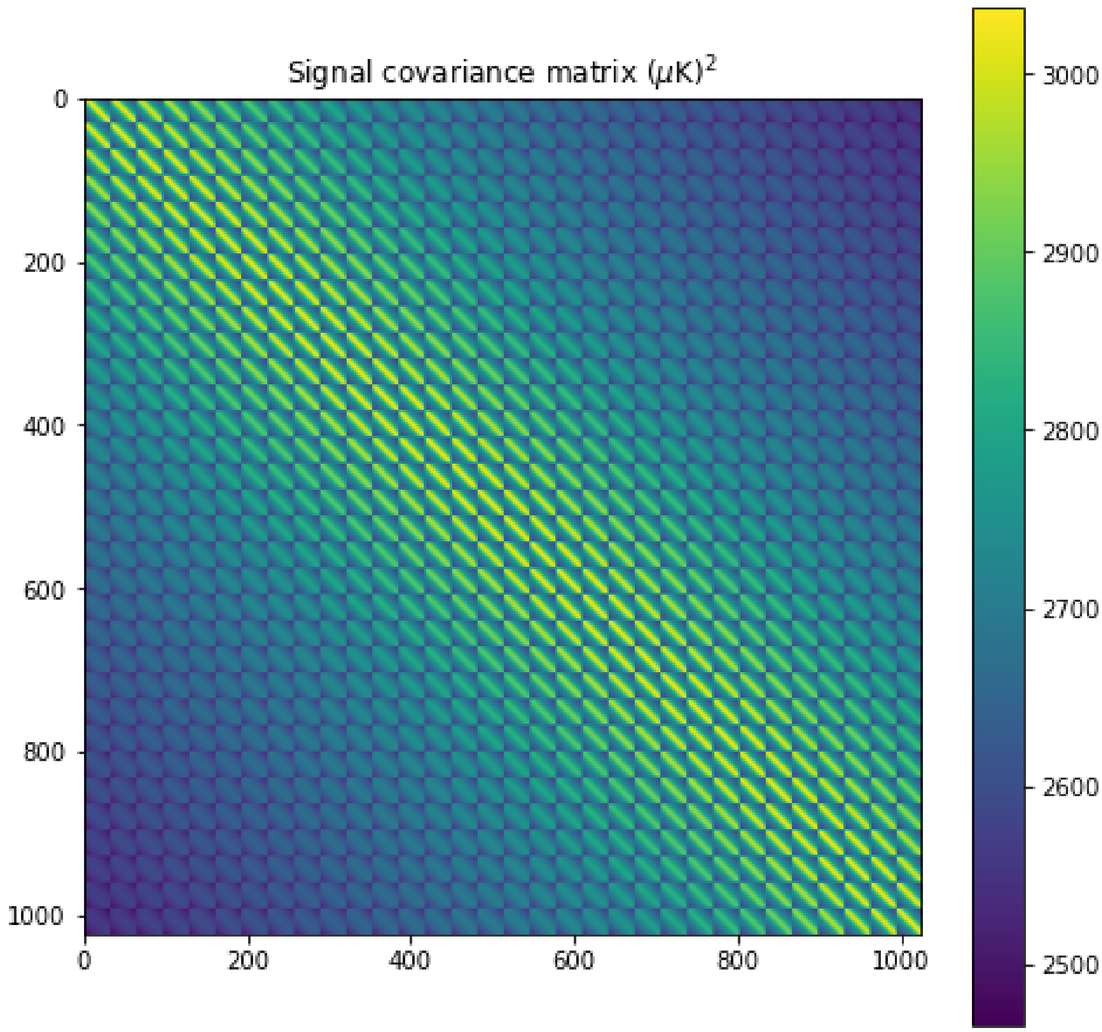
[30]:
```
Tbar = 2.725e6 # known value in microK
theta_beam = 0.000667 # radians, as specified in pdf

# Get C_l from CAMB
pars = cmb_utils.fast_camb_settings() # Use this from what Kiyo gave us
results = camb.get_results(pars)

#get dictionary of CAMB power spectra
powers = results.get_cmb_power_spectra(pars, CMB_unit='muK')
totCL = powers['total']
ls = np.arange(totCL.shape[0])[1:] # get rid of l = 0
Cls_withfactors = totCL[:,0][1:] # this goes out to l = 1500 per the settings,␣
↪and we get rid of l=0
Cls = Cls_withfactors / (ls * (ls + 1) * Tbar**2 / (2 * np.pi))
Cls_obs = Cls * np.exp(-ls**2 * theta_beam**2)
```

[31]:
```
# calculate angular correlation function
max_theta = pix_width * np.linalg.norm([nx, ny])
sampled_thetas = np.linspace(0, max_theta, 300)
w_theta_data = np.array([trapz(jv(0, ls * theta) * Cls_obs) for theta in␣
↪sampled_thetas])
w_theta = interp1d(sampled_thetas, w_theta_data)
```

[32]:
```
# calculate theta values between all pixels
thetas = np.zeros((nx,ny,nx,ny), dtype=float) # initalize array

# angle between pixel a = (ax, ay) and pixel b = (bx, by) stored at (ax, ay,␣
↪bx, by)
# angle in radians is distance (in pixels) between pixels times pix_width
for ax in range(thetas.shape[0]):
    for ay in range(thetas.shape[1]):
        for bx in range(thetas.shape[2]):
            for by in range(thetas.shape[3]):
                thetas[ax, ay, bx, by] = pix_width * np.sqrt(np.abs(bx - ax)**2␣
↪+ np.abs(by - ay)**2)
```

```
thetas = thetas.flatten()
CS_test = Tbar**2 * w_theta(thetas)
CS_test = np.reshape(CS_test, (nx*ny,nx*ny))
```

[67]:
```
plt.figure(figsize=(8,8))
plt.imshow(CS_test)
plt.colorbar()
plt.title('Signal covariance matrix ($\mu$K)$^2$');
```



This covariance matrix looks as we expect, with the pixels next to each other having the highest covariance.

4.2 2.2 Band Powers

To account for C_S in the calculation of $C_{,\alpha}$ we will use band powers c^α as our parameters:

$$C_l^{obs} \approx \sum_\alpha c^\alpha E_\alpha(l) \tag{6}$$

where

$$E_\alpha(l) = \begin{cases} 1 & l_\alpha^{low} \leq l < l_{\alpha+1}^{low} \\ 0 & otherwise \end{cases} \tag{7}$$

It can then be shown that

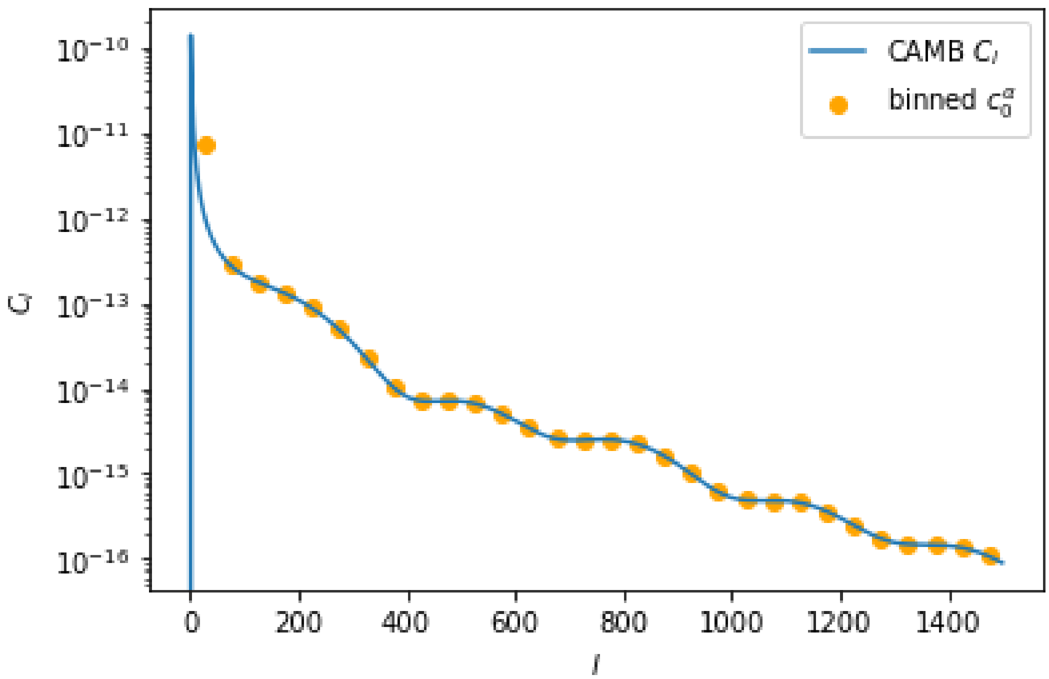
$$C_{,\alpha} = \int_0^\infty \frac{dl}{2\pi} E_\alpha(l) J_0(l\theta) \tag{8}$$

[34]:
```
# create logarithmically spaced l bins to use for bandpowers
l_min = min(ls)
l_max = pars.max_l # get from CAMB parameters
num_bins = 30
l_bin_edges = np.linspace(l_min, l_max, num_bins+1, endpoint=True) # 31 numbers␣
↪for 30 bins,  for test data
```

[60]:
```
# eq 8 basically changes the bounds of integration for eq 9

# array to store C_ , alpha values
C_alphas = np.array([])

# array to store fiducial c_alpha_0
c_alpha_0s = np.array([])

# for each alpha
for alpha in range(num_bins):
    # integration only within l bin
    ls_int = ls[ np.logical_and(ls >= l_bin_edges[alpha],ls <␣
↪l_bin_edges[alpha+1]) ]

    C_alpha_data = np.array([trapz(jv(0, ls_int*theta)) for theta in␣
↪sampled_thetas])
    C_alpha_interp = interp1d(sampled_thetas, C_alpha_data)

    C_alphas = np.append(C_alphas, C_alpha_interp(thetas))

    # also bin and avereage the Cls from camb
    Cls_obs_bin = Cls_obs [np.where(np.logical_and(ls >= l_bin_edges[alpha], ls␣
↪< l_bin_edges[alpha+1]))]
    Cls_obs_bin = np.sum(Cls_obs_bin) / len(Cls_obs_bin)
    c_alpha_0s = np.append(c_alpha_0s , Cls_obs_bin)
```

```
C_alphas = np.reshape(C_alphas, (num_bins, nx*ny, nx*ny))
```

[36]:
```
l_bin_centers = np.array([(l_bin_edges[i] + l_bin_edges[i+1])/2 for i in␣
↪range(len(l_bin_edges)-1)])
plt.semilogy(Cls_obs, label='CAMB $C_l$');
plt.scatter(l_bin_centers, c_alpha_0s, color='orange', label=r'binned␣
↪$c^\alpha_0$');
plt.legend();
plt.ylabel('$C_l$');
plt.xlabel('$l$');
```



The binning looks good.

[65]:
```
# now we can calculate our estimates for c hat alpha:
C_test = CS_test + CN_test
C_test_inv = LA.inv(C_test)
F = (1/2)*np.array([np.trace(np.linalg.multi_dot([C_alphas[alpha], C_test_inv,␣
↪C_alphas[beta], C_test_inv])) for alpha in range(num_bins) for beta in␣
↪range(num_bins)])
F = np.reshape(F, (num_bins,num_bins))
```

[38]:
```
# finally calculate estimate for band powers
```

```
matrix1 = np.array([np.linalg.multi_dot([s_hat_test.flatten(), C_test_inv,␣
↪C_alphas[beta], s_hat_test.flatten()]) for beta in range(num_bins)])
matrix2 = np.array([np.trace(np.dot(C_test_inv,C_alphas[beta])) for beta in␣
↪range(num_bins)])
c_alpha_hat = c_alpha_0s + np.dot(LA.inv(F), matrix1 - matrix2)
```

5 Parameter inference

[39]:
```
# get test case from kiyo's estimator, with bins changed to 30 (instead of 50)␣
↪for the smaller test data
test_Cl, test_n_modes, test_l_bin_edges = cmb_utils.
↪naieve_PS_estimator(s_hat_test, pix_width)
# this doesn't seem to give us the fisher matrix, sadly
```

5.1 3.1 Least-squares

This is a non-linear least squares problem with

$$-2\ln\mathcal{L} + \text{constant} = \chi^2(p_\gamma) = \sum_{\alpha\beta}[\hat{c}_\alpha - C^{\text{obs}}_{l\alpha}(p_\gamma)]F_{\alpha\beta}[\hat{c}_\beta - C^{\text{obs}}_{l\beta}(p_\gamma)], \tag{9}$$

where $l_\alpha = (l^{\text{low}}_\alpha + l^{\text{low}}_{\alpha+1})/2$.

[40]:
```
def chisq(cosmo_params):
    """Chi squared to minimize.
    cosmo_params : list-like
        (omega_bh^2, omega_ch^2, tau, h, n_s, ln(A_s_))
    """
    # Get C_l from CAMB with these parameters
    pars = cmb_utils.fast_camb_settings()
    pars.set_cosmology(H0=100*cosmo_params[3], ombh2=cosmo_params[0],␣
↪omch2=cosmo_params[1],
                        omk=0, tau=cosmo_params[2])
    pars.InitPower.set_params(As=np.exp(cosmo_params[5]) / 1e10,␣
↪ns=cosmo_params[4])
    results = camb.get_results(pars)
    #get dictionary of CAMB power spectra
    powers = results.get_cmb_power_spectra(pars, CMB_unit='muK')
    totCL = powers['total']
    ls = np.arange(totCL.shape[0])[1:] # get rid of l = 0
    Cls_withfactors = totCL[:,0][1:] # this goes out to l = 1500 per the␣
↪settings, and we get rid of l=0
    Cls = Cls_withfactors / (ls * (ls + 1) * Tbar**2 / (2 * np.pi))
    Cls_obs = Cls * np.exp(-ls**2 * theta_beam**2)
```

```
    # bin these Cls_obs
    c_alpha_0s = np.array([])  # array to store fiducial c_alpha_0
    for alpha in range(num_bins):
        Cls_obs_bin = Cls_obs [np.where(np.logical_and(ls >=␣
↪l_bin_edges[alpha], ls < l_bin_edges[alpha+1]))]
        Cls_obs_bin = np.sum(Cls_obs_bin) / len(Cls_obs_bin)
        c_alpha_0s = np.append(c_alpha_0s, Cls_obs_bin)

    the_chisq = c_alpha_0s.dot(F.dot(c_alpha_0s))
    return the_chisq
```

[50]:
```
p0 = np.array([0.02, 0.1, 0.05, 0.67, 0.9, 3.0])
bnds = ((0, 1), (0, 1), (0, 1), (0, 1.5), (0, 2), (1, np.inf))
```

[42]:
```
res = minimize(chisq, p0, bounds=bnds)
res.x
```

[42]:
```
array([0.02, 0.1 , 0.05, 0.67, 0.9 , 3.  ])
```

Our likelihood estimation is just returning the initial guesses, which is a bit odd.

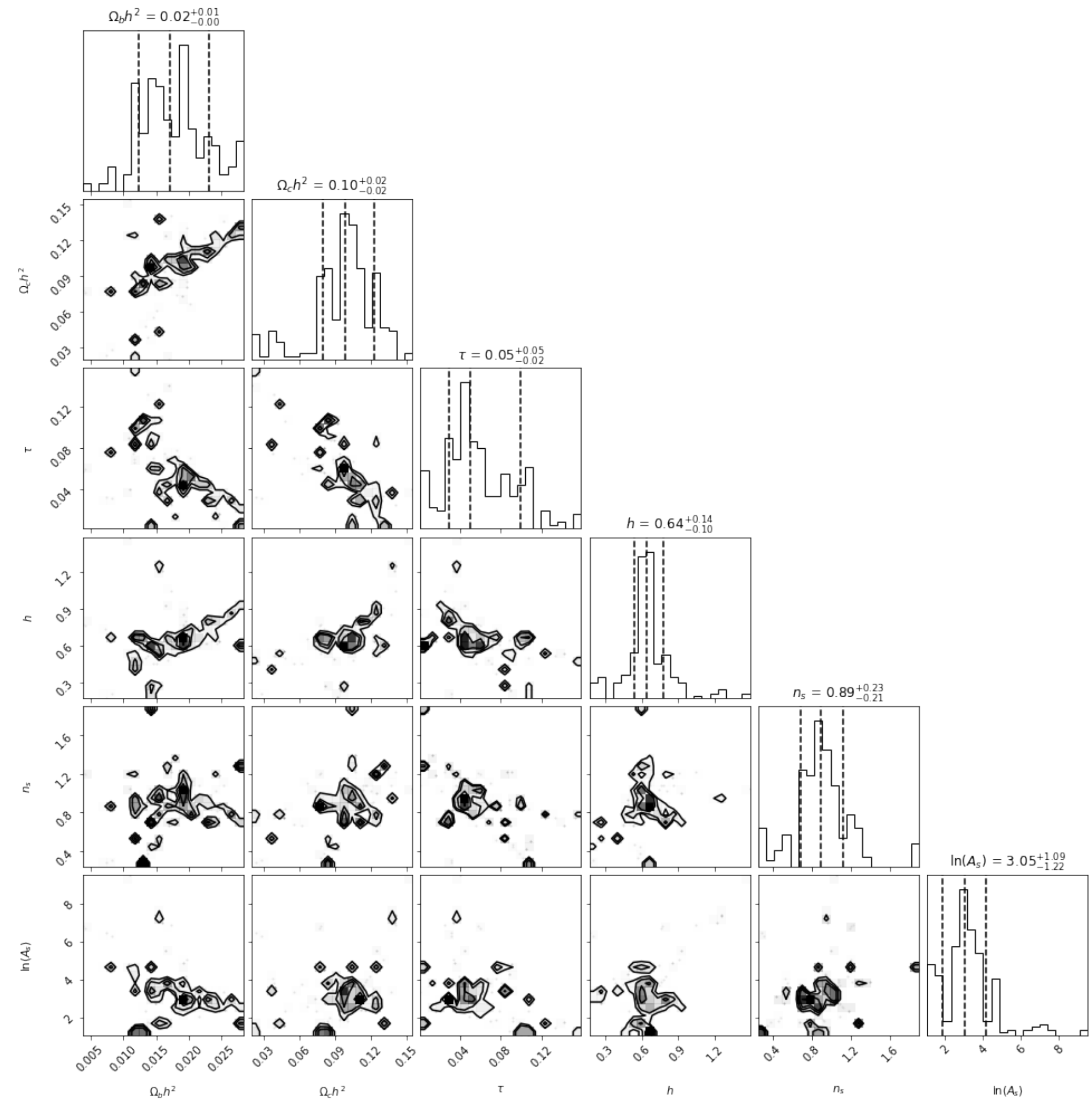
5.2 3.2 MCMC

[51]:
```
def logL(cosmo_params):
    for i, param in enumerate(cosmo_params):
        if not bnds[i][0] < param < bnds[i][1]:
            return -np.inf # enforce bounds
    return chisq(cosmo_params)/(-2)
```

[56]:
```
ndim, nwalkers = 6, 12
steps = 20
mcmc_p0 = np.array([np.random.uniform(0.9, 1.1, 6) * p0 for i in␣
↪range(nwalkers)]) # perturb around p0 by a bit
sampler = emcee.EnsembleSampler(nwalkers, ndim, logL)
state = sampler.run_mcmc(mcmc_p0, steps, progress=True)
```

```
100%|      | 20/20 [06:31<00:00, 19.57s/it]
```

[58]:
```
flat_samples = sampler.get_chain(flat=True)
labels = [r'$\Omega_bh^2$', r'$\Omega_ch^2$', r'$\tau$', r'$h$', r'$n_s$',␣
↪r'$\ln{(A_s)}$']
fig = corner.corner(flat_samples, labels=labels,
                    quantiles=[0.16, 0.5, 0.84],
                    show_titles=True, title_kwargs={"fontsize": 12})
```



These corner plots look kind of reasonable, though we don't have a ton of steps for each walker so it might be returning our prior.

```
[ ]:
```