In [1]:

```
import numpy as np
import matplotlib.pyplot as plt
from astropy.io import fits
```

Part 1

Fit a Gaussian to the cluster in the map center assuming the map noise is uniform and uncorrelated between map pixels. What are best fit positions, sigma, and amplitude?

I will use Newton's method with numerical derivatives. The code is pulled from "lecture_7/mf_act_corrnoise_class.py" with some changes: I included a constant in the Gaussian model and roughly estimated the noise using the mean of the absolute value of the residuals. (This is just to help Newton converge, since we're doing proper correlated noise estimates later).

Looking at the cluster, it's less bright than its surroundings so the guess for the amplitude should be the minimum.

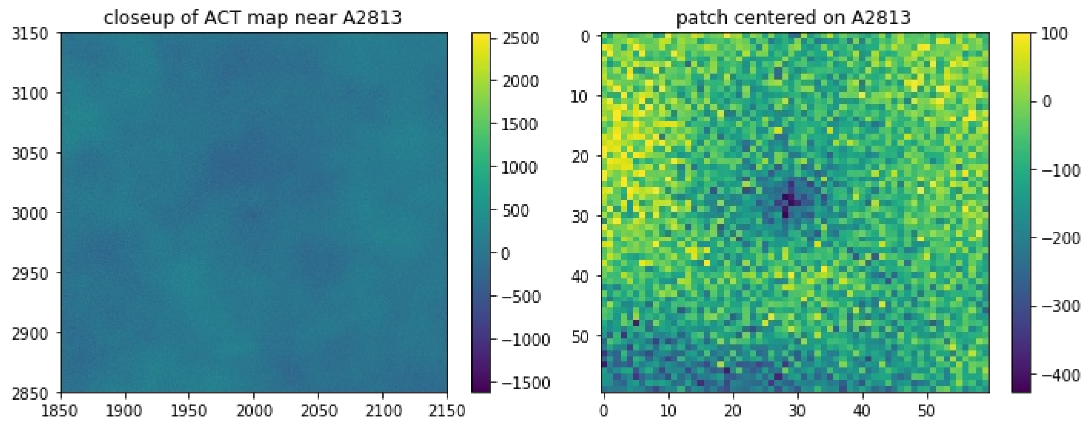
In [2]:

```
# look at the map

# load data
hdul=fits.open("advact_tt_patch.fits")
map=hdul[0].data
hdul.close()
map=np.asarray(map,dtype='float')
print('map shape = ', map.shape)

# saving data with a different name for later
act_map = map

# x and y midpoints
x0, y0 = 2000,3000
mwidth = 30

# make a patch of the map at the center
patch = map[x0-mwidth:x0+mwidth,y0-mwidth:y0+mwidth]

# new x-values for patch
x = np.arange(0,patch.shape[0])

# plot
fig, (ax1, ax2) = plt.subplots(1,2,figsize=(10,4),tight_layout=True)
im1 = ax1.imshow(map.T, aspect='auto')
ax1.set_xlim(1850,2150)
ax1.set_ylim(2850,3150)
ax1.set_title('closeup of ACT map near A2813')
fig.colorbar(im1, ax=ax1)
im2 = ax2.imshow(patch.T, vmax=100, aspect='auto')
ax2.set_title('patch centered on A2813')
fig.colorbar(im2, ax=ax2)
```

```
map shape =  (4000, 6000)
```

Out[2]:

```
<matplotlib.colorbar.Colorbar at 0x7f2b951d5810>
```



In [3]:

```python
# Newton's method function to fit for amp, x0, y0, sig, c
# This uses code from "lecture_7/mf_act_corrnoise_class.py"

def gauss2d(pars,x):
    x0=pars[0]
    y0=pars[1]
    amp=pars[2]
    sig=pars[3]
    c=pars[4]

    dx=x-x0
    dy=x-y0

    dxmat=np.outer(dx,np.ones(len(dx)))
    dymat=np.outer(np.ones(len(dy)),dy)
    rsqr=dxmat**2+dymat**2
    Map=amp*np.exp(-.5*rsqr/sig**2)+c
    return Map

def get_derivs_ravel(fun,pars,dp,x):
    mymod=fun(pars,x)
    npar=len(pars)
    dplus=[None]*npar
    dminus=[None]*npar
    for i in range(npar):
        pp=pars.copy()
        pp[i]=pp[i]+dp[i]
        dplus[i]=fun(pp,x)
        pp=pars.copy()
        pp[i]=pp[i]-dp[i]
        dminus[i]=fun(pp,x)
    n=dplus[0].size
    A=np.empty([n,npar])
    #actually do the numerical derivatives
    for i in range(npar):
        dd=(dplus[i]-dminus[i])/(2*dp[i])
        A[:,i]=np.ravel(dd)

    return np.ravel(mymod),A

def newton(pars,fun,data,x,dp,niter=10):
    for i in range(niter):
        mod,A=get_derivs_ravel(fun,pars,dp,x)
        r=data-mod

        # i'm going to try to guess the noise...
        ninv = np.mean(np.abs(r))
        ninv = 1/ninv

        lhs=A.T@(ninv*A)
        rhs=A.T@(ninv*r)
        dp=np.linalg.inv(lhs)@rhs
        print('on interation ',i,' parameter shifts are ',dp)
        #print('on interation ',i,' parameterS are ',pars)
        pars=pars+dp


    return pars
```

Run Newton's method and plot the initial guess, real map, predicted map from fit parameters:

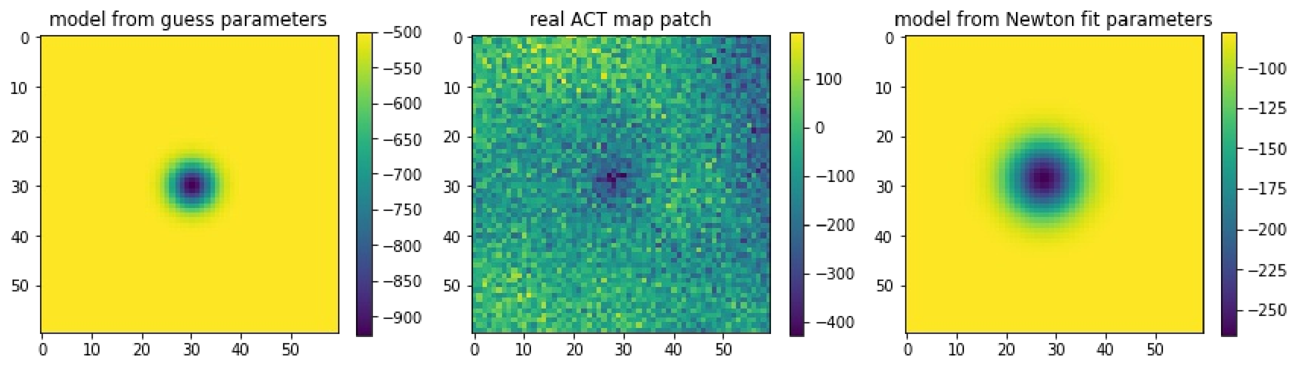
In [4]:

```
# starting guess for the parameters
guess = np.array([mwidth,mwidth,patch.min(),3,-500])

# starting guess for the model
model = gauss2d(guess,x)

# plot the guess next to the map to check
fig, (ax1, ax2, ax3) = plt.subplots(1,3,figsize=(12,4),tight_layout=True)
im1 = ax1.imshow(model)
im2 = ax2.imshow(patch)
fig.colorbar(im1, ax=ax1, shrink=0.75)
fig.colorbar(im2, ax=ax2, shrink=0.75)
ax1.set_title('model from guess parameters')
ax2.set_title('real ACT map patch')

# set the parameter-steps for Newtons method
dp = np.array([0.01,0.01,1.0,0.01,1.0])

# get model and derivatives
mod, derivs = get_derivs_ravel(gauss2d,guess,dp,x)

# get the fit parameters using Newton's method
fitp = newton(guess, gauss2d, np.ravel(patch), x, dp)
print('fitp = ', fitp)

# get the model using fitted parameters
modfit = gauss2d(fitp,x)

# plot the results
im3 = ax3.imshow(modfit)
fig.colorbar(im3, ax=ax3, shrink=0.75)
ax3.set_title('model from Newton fit parameters')
```

```
on interation  0  parameter shifts are  [ -0.51123307  -1.10703779 287.0221689    0.71500509 419.666
21364]
on interation  1  parameter shifts are  [ -1.05066506  -2.21752128 -51.4197877    1.07535206   1.166
38772]
on interation  2  parameter shifts are  [0.17729765 0.73820942 6.88723604 0.19758394 1.4598545 ]
on interation  3  parameter shifts are  [-0.03337075 -0.11528166 -4.06305029 -0.07682737 -0.08056221
]
on interation  4  parameter shifts are  [ 0.0111481   0.04510069 -0.26488534 -0.00462052 -0.00727867
]
on interation  5  parameter shifts are  [-3.00594785e-04 -4.19809764e-03 -3.37924472e-01 -9.53199511
e-03
 -1.66440485e-02]
on interation  6  parameter shifts are  [ 0.00127858  0.00326356 -0.09698151 -0.00272972 -0.00479329
]
on interation  7  parameter shifts are  [ 0.00031404  0.00035958 -0.05570574 -0.00158127 -0.00278632
]
on interation  8  parameter shifts are  [ 0.00020477  0.00040042 -0.02279344 -0.00064716 -0.00114095
]
on interation  9  parameter shifts are  [ 8.07776297e-05  1.31814972e-04 -1.09146239e-02 -3.10089639
e-04
 -5.46876755e-04]
fitp =  [  28.59475442   27.34342666 -189.36263818    4.89169295  -77.82129652]
```

Out[4]:

```
Text(0.5, 1.0, 'model from Newton fit parameters')
```



Part 2

Write a routine called estimate_ps to take the 2D power spectrum of the map. This should smooth the |2D FFT(map)|. Do this by convolving the squared FFT with a Gaussian (use get_gauss_kernel with norm=True). Make a plot of the log of the 2D power spectrum.

To convolve two functions, we multiply their DFTs then take the IFT. I decided to try using a window function and padding the map to take the Fourier transform.

In [5]:

```python
# from act_mf_tools.py:
def get_gauss_kernel(map,sig,norm=False):
    nx=map.shape[0]
    x=np.fft.fftfreq(map.shape[0])*map.shape[0]
    y=np.fft.fftfreq(map.shape[1])*map.shape[1]
    rsqr=np.outer(x**2,np.ones(map.shape[1]))+np.outer(np.ones(map.shape[0]),y**2)
    kernel=np.exp((-0.5/sig**2)*rsqr)
    if norm:
        kernel=kernel/kernel.sum()
    return kernel

# function to get the smoothed 2D power spectrum
def estimate_ps(Map, modfit, pad=True):
    """
    Parameters:
    Map = patch of the ACT map
    modfit = predicted model from Newton's method

    Returns:
    pspec = the noise spectrum
    """

    # Make the patch (data - predicted model)
    patch2 = Map-modfit

    # Get the Gaussian kernel
    kernel = get_gauss_kernel(patch2,sig=0.1,norm=True)

    if pad==True:
        # Pad the kernel
        kernel = np.hstack([kernel, np.fliplr(kernel)])
        kernel = np.vstack([kernel, np.flipud(kernel)])

        # Pad the map
        patch2 = np.hstack([patch2, np.fliplr(patch2)])
        patch2 = np.vstack([patch2, np.flipud(patch2)])

        # FFTs
        mapft = np.fft.fft2(patch2)
        kernelft = np.fft.fft2(kernel)

    if pad==False:
        # Make a window function - Blackman Harris taper?
        win = np.blackman(len(patch2))
        win2d = np.outer(win,win)

        # Normalize the window function
        normfac = np.sqrt(np.mean(win2d**2))

        # FFTs, with window normalization
        mapft = np.fft.fft2(patch2*win2d/normfac)
        kernelft = np.fft.fft2(kernel)

    # Convolve them
    conv = mapft*kernelft

    # Get the power spectrum
    pspec = np.abs(conv**2)

    # normalize...?
    #pspec = pspec/(patch2.shape[0]*patch2.shape[1])**0.5

    return pspec
```

Get the log spectrum using both methods (windowing and padding the map). Since we need to use the inverse power spectrum as the noise estimate, I'm going to proceed with using the window function because padding introduced zeros in the power spectrum.

In [6]:

```
# plot with window vs. pad
fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2,2,figsize=(10,8))

# get the power spectrum for the patch using a window function
pspec = estimate_ps(patch,modfit, pad=False)
logspec = np.log(pspec)

# plot
im1 = ax1.imshow(logspec)
ax1.set_title('FFT using window function')
fig.colorbar(im1, ax=ax1)
im3 = ax3.imshow(np.fft.fftshift(logspec))
ax3.set_title('fft-shifted')
fig.colorbar(im3, ax=ax3)

# get the power spectrum for the patch using padded map
pspec = estimate_ps(patch,modfit)
logspec = np.log(pspec)

# plot
im2 = ax2.imshow(logspec, vmin=-20)
ax2.set_title('FFT using padded map')
fig.colorbar(im2, ax=ax2)
im4 = ax4.imshow(np.fft.fftshift(logspec), vmin=-20)
ax4.set_title('fft-shifted')
fig.colorbar(im4, ax=ax4)
```

```
/home/taylordb/anaconda3/envs/hera_env/lib/python3.7/site-packages/ipykernel_launcher.py:18: Runtime
Warning: divide by zero encountered in log
```

Out[6]:

```
<matplotlib.colorbar.Colorbar at 0x7f2b934e7a10>
```

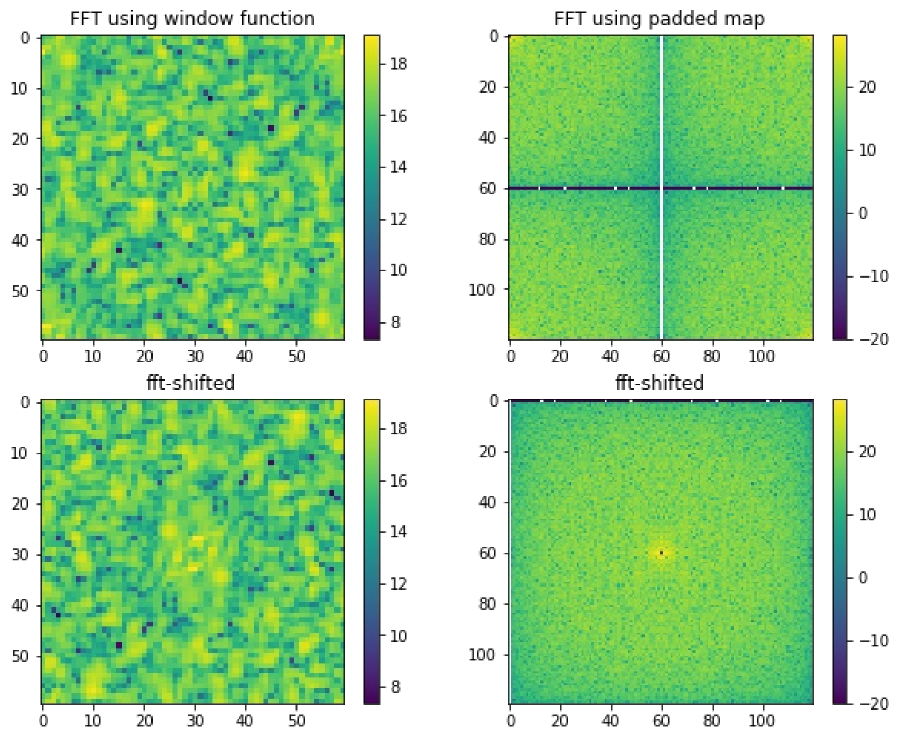


Fig: The shifted maps show a signal in the center.

Part 3

Write a routine called "filter_map" to apply N^{-1} to a map, using the output of the estimate_ps routine. It should accept a non-padded map and the power spectrum, and return a non-padded map. Plot the noise-filtered map, centered in a region around the center (where there ought to be a cluster).

Applying N^{-1} to the map means we take the windowed map, FFT it, divide by the smooth spectrum, then IFT.

The window function needs to be normalized to take the Fourier transform. I'm also normalizing by $1/\sqrt{n}$ (n is the number of data points).

In [7]:

```
def filter_map(Map,modfit,pspec,return_fft=False):
    """
    Parameters:
    Map = patch of ACT map
    modfit = model from Newton
    pspec = noise spectrum
    return_fft = returns the real or FFT filtered map

    Return:
    dat_filt = noise-filtered map
    """

    # Patch of map to be filtered
    patch2 = Map

    # Make a window function - Blackman Harris taper
    win = np.blackman(len(patch2))
    win2d = np.outer(win,win)

    # Normalize the window function
    normfac = np.sqrt(np.mean(win2d**2))

    # FFTs, with normalizationS
    mapft = np.fft.fft2(patch2*win2d/normfac)/(patch2.shape[0]*patch2.shape[1])**0.5

    # Set N^-1 = 1/pspec
    Ninv = 1/pspec

    if return_fft==False:
        # Noise filtered map in real space (normalized?)
        dat_filt = np.fft.ifft2(mapft*Ninv)*(patch2.shape[0]*patch2.shape[1])
    elif return_fft==True:
        # Noise filtered map in Fourier space
        dat_filt = mapft*Ninv

    return dat_filt
```

In [8]:

```
# get the power spectrum (windowing)
pspec = estimate_ps(patch,modfit,pad=False)

# get filtered data
dat_filt = filter_map(patch,modfit,pspec)

# plot
fig, (ax1, ax2) = plt.subplots(1,2,figsize=(10,4),tight_layout=True)
im1 = ax1.imshow(np.real(dat_filt))
ax1.set_title('noise-filtered map')
fig.colorbar(im1, ax=ax1)
im2 = ax2.imshow(patch)
ax2.set_title('patch of ACT map')
fig.colorbar(im2, ax=ax2)
```

Out[8]:

```
<matplotlib.colorbar.Colorbar at 0x7f2b932f0ed0>
```

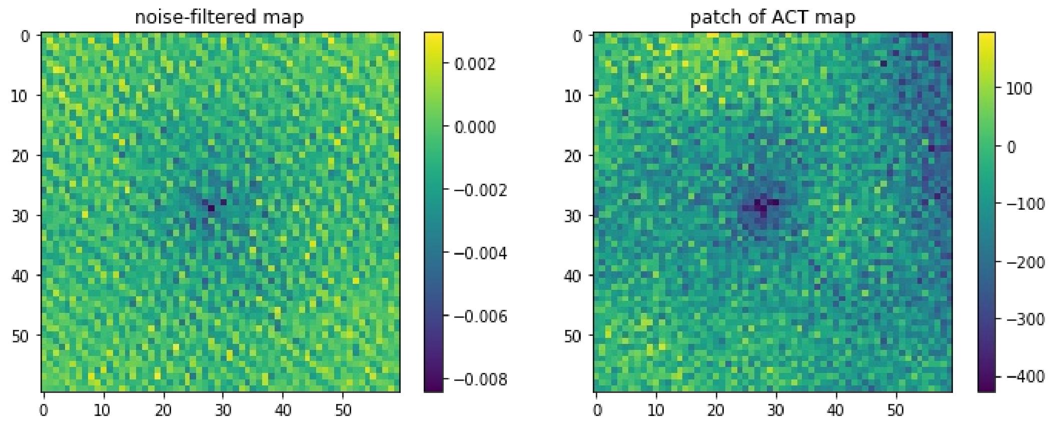


Fig: It looks like there's a signal in the middle but the blob-like noise that we had previously is now looking more like white noise. Which is what we want (right?)

Part 4

Make sure noise filtering is properly normalized. Show that if you start with a map of white noise, run estimate_ps, call filter_map, that the variance of the filtered map agrees with what you expect.

The power spectrum of white noise is constant, and is equal to the variance of the white noise (sigma).

So the variance of the filtered map should be the same as the variance of our model.

In [9]:

```
# generate white noise map with sigma=0.5
sigma = 0.5
wn_map = np.random.normal(0,sigma,size=patch.shape)

# get the power spectrum (windowing)
pspec = estimate_ps(wn_map,modfit, pad=False)

# get the filtered map
map_filt = filter_map(wn_map,modfit,pspec)

# get the variance
np.mean(wn_map**2), np.real(np.mean(map_filt*np.conj(map_filt)))
```

Out[9]:

```
(0.25174424803734924, 0.09378078347269893)
```

Ok so these are off, meaning the filtered map is not properly normalized.

I'm using the "noise_normalization.py" code from the 2018 641 Jon repo to check the variances.

In [10]:

```
# variance in real space
var_real = np.mean(wn_map**2)
print('real variance = {}, expected variance = {}'.format(var_real, sigma**2))

# variance in fourier space
datft = np.fft.fft2(wn_map)
var_fourier = np.real(np.mean(datft*np.conj(datft)))
print('fourier variance = {}'.format(var_fourier))

# normalize by 1/sqrt(number of data points)
datft_normed = datft/(wn_map.shape[0]*wn_map.shape[1])**0.5
var_fourier_normed = np.real(np.mean(datft_normed*np.conj(datft_normed)))
print('fourier variance with normalization = {}'.format(var_fourier_normed))

# inverse fourier transform and check variance
datift = np.fft.ifft2(datft_normed)
var_ift = np.real(np.mean(datift*np.conj(datift)))
print('real (via IFT) variance with normalization = {}'.format(var_ift*(wn_map.shape[0]*wn_map.shape[1])))
```

```
real variance = 0.25174424803734924, expected variance = 0.25
fourier variance = 906.2792929344572
fourier variance with normalization = 0.25174424803734924
real (via IFT) variance with normalization = 0.2517442480373492
```

So I need to normalize the FFT'd map by $1/\sqrt{n}$, but if I inverse FFT then I need to multiply by $\sqrt{n}$? I'm very confused about the normalization, sorry!

Part 5

Make a properly normalized matched-filter map of the ACT data, using the width from part 1. What does the matched filter say the amplitude and uncertainty on the amplitude of the cluster is in μK? Look at the standard deviation of the matched filter output, do you buy this error bar?

The best-fit value for the amplitude is $(A^TN^{-1}A)^{-1}(A^TN^{-1}d)$ for a template A. The error on the amplitude m is $\sqrt{(A^TN^{-1}A)^{-1}}$.

The matched filter code makes the model in Fourier space (converting x to frequency), where the template is a Gaussian signal with amplitude of 1, sigma from part 1, and the added constant from part 1. The template (or model, same thing) is filtered using the power spectrum code from part 3. I use a window function to FFT the map, then normalize by $\sqrt{n}$.

The right-hand side of the LLS equation comes from inverse FFT-ing the filtered model (in Fourier space) times the filtered template. The left hand side comes from inverse FFT-ing the filtered model (in Fourier space) times the FFT of the map. The best fit amplitude is rhs, the error is $1\sqrt{lhs}$.

In [15]:

```
# normalized matched filter function
def matched_filter(Map, sigma, c, show_map=False):
    """
    Parameters:
    Map = map to be matched filtered
    sigma = width of Gaussian model
    c = constant in the Gaussian model
    show_map = choose whether to plot the matched filtered map

    Returns:
    amp = amplitude of the cluster in the map
    err = uncertainty on the amplitude
    """

    patch2 = Map

    # make the frequencies
    x = np.arange(0,Map.shape[0])
    y = np.arange(0,Map.shape[1])
    xmat = np.outer(x,np.ones(len(x)))
    ymat = np.outer(np.ones(len(y)),y)
    rsqr = xmat**2 + ymat**2

    # get the template signal
    template=np.exp(-0.5*rsqr/sigma**2) + c

    # FFT template
    tft=np.fft.fft2(template)

    # get the power spectrum (windowing)
    pspec = estimate_ps(patch2,template, pad=False)

    # get the noise-filtered model
    model_filt = filter_map(patch2,template,pspec) /(patch2.shape[0]*patch2.shape[1])**0.5 # norm bcs of problem
4?
    modelft_filt = np.fft.fft2(model_filt)

    # make a window function
    win = np.blackman(len(patch2))
    win2d = np.outer(win,win)
    normfac = np.sqrt(np.mean(win2d**2))

    # FFT the map and normalize it
    mapft = np.fft.fft2(patch2*win2d/normfac)/(patch2.shape[0]*patch2.shape[1])**0.5

    # get amplitude and its uncertainty
    rhs = np.sum(np.real(np.fft.irfft2(modelft_filt*tft)))
    lhs = np.sum(np.real(np.fft.irfft2(modelft_filt*mapft)))
    amp = rhs/lhs
    err = 1/np.sqrt(lhs)

    # show the matched filter map
    if show_map:
        # get the map
        mf_map = np.real(np.fft.irfft2(modelft_filt*mapft))

        # plot it
        plt.figure()
        plt.imshow(mf_map)
        plt.colorbar()

    return amp, err
```

In [17]:

```python
# first, make the model, but set the amplitude to 1
mf_params = np.array([1] + list(fitp[1:]))
print('amp from Newton (fitp[0]) = {}'.format(fitp[0]))
sigma, c = fitp[-2], fitp[-1]

# find amplitude and error with the matched filter
amp, err = matched_filter(patch, sigma, c,show_map=True)
print('amp = {}, err = {}'.format(amp, err))

# look at the standard deviation of the residuals
resid = patch - model*amp
std = np.std(resid)
print('standard deviation of residuals = ', std)
```

```
amp from Newton (fitp[0]) = 28.59475442381758
amp = 70.74056465732926, err = 2.5444475931350343
standard deviation of residuals =  1565.3462665939824
```

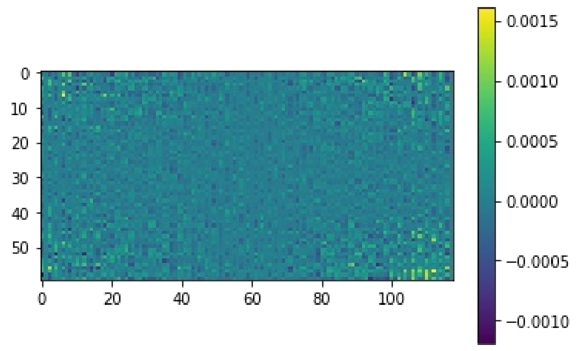


Fig: This is definitely not what we're supposed to be looking at, right? This is just noise?

Previously, the fitted amplitude from Newton was ~28. The matched filter says it's ~70. These numbers should be close, so I've made one or more mistakes. I'm almost certain that I messed up the normalization, I'm not sure if I've made mistakes elsewhere too.

Also, the standard deviation of the residuals is massive so I definitely don't buy this error bar.

Part 6

Use matched filter to fit amplitude for another region. Each pixel on the map is 30 arcseconds. There is one cluster 200 pixels to the right and up. RA decreases to the right.

According to https://web.pa.msu.edu/astro/MC2/accept/clusters/9409.html (https://web.pa.msu.edu/astro/MC2/accept/clusters/9409.html), Abell 2813 is centered at RA (hr:min:sec) = 00:43:24.881, and decl (hr:min:sec) = -20:37:25.08. So the fit parameters from Newton's method would give this position, and shifting 200 pixels to the right gives a shift in RA of -6000 arcseconds or 100 arcminutes (30 arcseconds per pixel).

My idea is to then shift the declination (y-values) and use the matched filter to find the amplitude of a cluster at the center. I think this would work if my matched filter was working properly.

In [19]:

```python
# x and y midpoints from above
x0, y0 = 2000,3000
mwidth = 50

# shift in x to the right
x0 = x0 + 200

# shift y by 1 pixel in a loop for a total of 50 pixels (?)
yshift = np.arange(50)

# sweep through new patches and use the matched filter to find the amplitude
for i, y_ in enumerate(yshift):
    y0 = 3000
    y0 = y0 + y_
    patch = act_map[x0-mwidth:x0+mwidth,y0-mwidth:y0+mwidth]

    # new x-values for patch (this actually doesn't change anything...)
    x = np.arange(0,patch.shape[0])

    # make the model again
    model = gauss2d(mf_params,x)

    # find amp, error
    amp, err = matched_filter(patch, sigma, c)

    print('center at (x0, y0) = ({},{}): amp = {}, err = {}'.format(x0,y0,amp,err))
```

```
center at (x0, y0) = (2200,3000): amp = 895.9739538059723, err = 41.37168228375002
center at (x0, y0) = (2200,3001): amp = 796.3216607427125, err = 35.6580050988829
center at (x0, y0) = (2200,3002): amp = 715.3663476727774, err = 31.016340456122396
center at (x0, y0) = (2200,3003): amp = 648.4932273878425, err = 27.182094299745636
center at (x0, y0) = (2200,3004): amp = 592.4794340822497, err = 23.970479965448234
center at (x0, y0) = (2200,3005): amp = 545.0026907701068, err = 21.248347069800012
center at (x0, y0) = (2200,3006): amp = 504.3584336788979, err = 18.917962516678923
center at (x0, y0) = (2200,3007): amp = 469.26385827017117, err = 16.905775253733317
center at (x0, y0) = (2200,3008): amp = 438.7345102868481, err = 15.155340480190787
center at (x0, y0) = (2200,3009): amp = 412.00446172660287, err = 13.622742845813915
center at (x0, y0) = (2200,3010): amp = 388.46960884928006, err = 12.273345377870669
center at (x0, y0) = (2200,3011): amp = 367.64785680195826, err = 11.079506621485276
center at (x0, y0) = (2200,3012): amp = 349.1483513667326, err = 10.018816507014629
center at (x0, y0) = (2200,3013): amp = 332.6515212910859, err = 9.07295205444329
center at (x0, y0) = (2200,3014): amp = 317.8950338212101, err = 8.226872117960225
center at (x0, y0) = (2200,3015): amp = 304.661301862208, err = 7.468101094831495
center at (x0, y0) = (2200,3016): amp = 292.76859503041106, err = 6.786219267506391
center at (x0, y0) = (2200,3017): amp = 282.06487750157453, err = 6.17250948966004
center at (x0, y0) = (2200,3018): amp = 272.42341067894836, err = 5.6197050601815794
center at (x0, y0) = (2200,3019): amp = 263.7374130781915, err = 5.121683541622902
center at (x0, y0) = (2200,3020): amp = 255.91530356233395, err = 4.67319402568289
center at (x0, y0) = (2200,3021): amp = 248.8795263880319, err = 4.269789765902612
center at (x0, y0) = (2200,3022): amp = 242.5642001811818, err = 3.907693379017256
center at (x0, y0) = (2200,3023): amp = 236.91364703746598, err = 3.5837125092683717
center at (x0, y0) = (2200,3024): amp = 231.88049835448754, err = 3.295131224142272
center at (x0, y0) = (2200,3025): amp = 227.4240824724909, err = 3.0396175679230164
center at (x0, y0) = (2200,3026): amp = 223.5104644378945, err = 2.8152258415796347
center at (x0, y0) = (2200,3027): amp = 220.11189508730078, err = 2.6203650153733116
center at (x0, y0) = (2200,3028): amp = 217.20565916349693, err = 2.453732684282576
center at (x0, y0) = (2200,3029): amp = 214.7733111161659, err = 2.314271251770835
center at (x0, y0) = (2200,3030): amp = 212.80091043013746, err = 2.2011814225983577
center at (x0, y0) = (2200,3031): amp = 211.27830713182422, err = 2.113881236523895
center at (x0, y0) = (2200,3032): amp = 210.1992183459536, err = 2.0520104577716523
center at (x0, y0) = (2200,3033): amp = 209.56139648116715, err = 2.0154402181433153
center at (x0, y0) = (2200,3034): amp = 209.3656696754079, err = 2.004217999866582
center at (x0, y0) = (2200,3035): amp = 209.61656174449266, err = 2.0186031810493463
center at (x0, y0) = (2200,3036): amp = 210.32319192093124, err = 2.0591186232247716
center at (x0, y0) = (2200,3037): amp = 211.4988390043158, err = 2.1265256814198246
center at (x0, y0) = (2200,3038): amp = 213.16061045001132, err = 2.2218052309802063
center at (x0, y0) = (2200,3039): amp = 215.33029058230903, err = 2.346206300833072
center at (x0, y0) = (2200,3040): amp = 218.03507489410808, err = 2.501288175391549
center at (x0, y0) = (2200,3041): amp = 221.30885106307989, err = 2.6889938430687486
center at (x0, y0) = (2200,3042): amp = 225.19294296972478, err = 2.9116926553963376
center at (x0, y0) = (2200,3043): amp = 229.7356027854138, err = 3.172151205316707
center at (x0, y0) = (2200,3044): amp = 234.99401971725484, err = 3.473648500846925
center at (x0, y0) = (2200,3045): amp = 241.03606599726635, err = 3.8200760733471797
center at (x0, y0) = (2200,3046): amp = 247.9422235344951, err = 4.2160484443039286
center at (x0, y0) = (2200,3047): amp = 255.80751416138222, err = 4.667013798193305
center at (x0, y0) = (2200,3048): amp = 264.74520144293047, err = 5.179466229906104
center at (x0, y0) = (2200,3049): amp = 274.89142918723167, err = 5.761211699006685
```

I was hoping that for one of these choices of patches, one amplitude would stand out as being close to the amplitude from using the matched filter on the original patch containing Abell 2813. Then I would use A2813's position and the Newton fit parameters to find the position of this new cluster in hr:min:sec.