Uploading the maps from the cluster

In [8]:

```
filename1 = "/moto/hill/projects/actpol/SO_Sims_LAMBDA/d56/SO_skymaps_deep5
6/so_skymap_deep56_30GHz.fits"
filename2 = "/moto/hill/projects/actpol/SO_Sims_LAMBDA/d56/SO_skymaps_deep5
6/so_skymap_deep56_44GHz.fits"
filename3 = "/moto/hill/projects/actpol/SO_Sims_LAMBDA/d56/SO_skymaps_deep5
6/so_skymap_deep56_70GHz.fits"
filename4 = "/moto/hill/projects/actpol/SO_Sims_LAMBDA/d56/SO_skymaps_deep5
6/so_skymap_deep56_100GHz.fits"
filename5 = "/moto/hill/projects/actpol/SO_Sims_LAMBDA/d56/SO_skymaps_deep5
6/so_skymap_deep56_143GHz.fits"
filename6 = "/moto/hill/projects/actpol/SO_Sims_LAMBDA/d56/SO_skymaps_deep5
6/so_skymap_deep56_145GHz.fits"
filename7 = "/moto/hill/projects/actpol/SO_Sims_LAMBDA/d56/SO_skymaps_deep5
6/so_skymap_deep56_217GHz.fits"
filename8 = "/moto/hill/projects/actpol/SO_Sims_LAMBDA/d56/SO_skymaps_deep5
6/so_skymap_deep56_353GHz.fits"

imap30 = enmap.read_map(filename1)
imap44 = enmap.read_map(filename2)
imap70 = enmap.read_map(filename3)
imap100 = enmap.read_map(filename4)
imap143 = enmap.read_map(filename5)
imap145 = enmap.read_map(filename6)
imap217 = enmap.read_map(filename7)
imap353 = enmap.read_map(filename8)

print("Map shape and dtype (same for all the maps):")
print(imap44.shape, imap44.dtype)

fmap30 = np.ndarray.flatten(imap30)
fmap44 = np.ndarray.flatten(imap44)
fmap70 = np.ndarray.flatten(imap70)
fmap100 = np.ndarray.flatten(imap100)
fmap143 = np.ndarray.flatten(imap143)
fmap145 = np.ndarray.flatten(imap145)
fmap217 = np.ndarray.flatten(imap217)
fmap353 = np.ndarray.flatten(imap353)
```

Defining the R Matrix

The r matrix is a matrix of size (n_map X n_map), where each entry is the sum of the multiplied entries of the specified indices in map_array, which is then divided by n_pix. Note that each entry in map_array is the corresponding flattened map.

In [ ]:

```
r = np.zeros([n_map,n_map])

for i in range(n_map):
    for j in range(n_map):
        r[i][j] = np.sum((map_array[i] * map_array[j])) / n_pix
```

Defining the B Matrix

The b matrix here is a 3D matrix of size (n_map X n_map X n_map), where each entry is the sum of the product of the corresponding flattened map, now with another dimension represented by k, and is still divided by n_pix.

In [5]:

```
b = np.zeros([n_map,n_map,n_map])

for i in range(n_map):
    for j in range(n_map):
        for k in range(n_map):
            b[i][j][k] = np.sum(map_array[i] * map_array[j] * map_array[k])
/ n_pix
```

Saving Arrays as Files to Reduce Memory Consumption

In [6]:

```
numpy.save('r_arr',r)
numpy.save('b_arr',b)
```

In [9]:

```
np.save('imap30',imap30)
np.save('imap44',imap44)
np.save('imap70',imap70)
np.save('imap100',imap100)
np.save('imap143',imap143)
np.save('imap145',imap145)
np.save('imap217',imap217)
np.save('imap353',imap353)
```

In [ ]:

```
np.save('map30',fmap30)
np.save('map44',fmap44)
np.save('map70',fmap70)
np.save('map100',fmap100)
np.save('map143',fmap143)
np.save('map145',fmap145)
np.save('map217',fmap217)
np.save('map353',fmap353)
```

DO NOT RUN CODE ABOVE THIS CELL

All the code in the above cells is for informational purposes i.e. displaying how the R and B matrices were initially constructed, and method for saving files so that may be loaded in using the given file name (all files use the .npy extension)

Packages for performing the optimization

In [2]:

```
import numpy as np, numpy.random
import statistics
from statistics import mean, pvariance
import scipy
from scipy.optimize import minimize,NonlinearConstraint
```

Packages for performing map manipulations

In [2]:

```
from __future__ import print_function
from pixell import enmap,utils
import matplotlib.pyplot as plt
from pixell import enplot
import os,sys
import urllib.request
```

Loading in Unflattened Map Data

The unflattened maps are used later on to show CMB maps. For now, the data works with flattened maps, particularly to build the R and B matrices displayed in the cells above. They have alerady been built, saved, and are loaded in 'map_array' below.

Each map shape represents the number of total pixels, which is of the form (2182,9455,1) and dim=3.

In [3]:

```
imap30 = np.load('imap30.npy')
imap44 = np.load('imap44.npy')
imap70 = np.load('imap70.npy')
imap100 = np.load('imap100.npy')
imap143 = np.load('imap143.npy')
imap145 =  np.load('imap145.npy')
imap217 = np.load('imap217.npy')
imap353 = np.load('imap353.npy')
```

Loading in Matrix Data

In [3]:

```
r = np.load('r_arr.npy')
b = np.load('b_arr.npy')
```

Defining ΔT

I define the number of maps, number of total pixels, and flattened maps (which are now 1D numpy arrays), as this was necessary for building the matrices and other computations.

In [4]:

```
n_map = 8
n_pix = 2182*9455
map_array = [np.load('map30.npy'),np.load('map44.npy'),np.load('map70.npy'
),np.load('map100.npy'),np.load('map143.npy'),np.load('map145.npy'),np.load
('map217.npy'),np.load('map353.npy')]
```

Optimizing the variance

In [6]:

```
w0 = np.array([.125,.125,.125,.125,.125,.125,.125,.125])
a = np.ones(len(w0))

#defines a lambda function that returns dot product of weights and array of ones
fun = lambda w: np.dot(w,a)

print("The sum of all the weights is:")
print(fun(w0))

#nonlinear constraint that implements the function and sets the constraint bounds
con = NonlinearConstraint(fun,1,1)

#defines a lambda function that implements the variance equation
var = lambda w: np.dot(np.dot(w,r),w)

print("The variance is:")
print(var(w0))

#minimization method ()
mini = minimize(var,w0,method='SLSQP',constraints=con)
weights_var = mini.x
print(mini.x)

print()
print(var(mini.x))

print()
print(fun(mini.x))
```

```
The sum of all the weights is:
1.0
The variance is:
30332.48984382074
[ 0.16548252  0.10405147 -2.95107189  1.93440481  0.79667526
0.5442029
  0.52879756 -0.12254262]

12885.913669980198

0.9999999999999998
```

Optimizing the skewness

In [8]:

```
a = np.ones(len(w0))

#defines a lambda function that returns dot product of weights and array of
ones
fun = lambda w: np.dot(w,a)

print("The sum of all the weights is:")
print(sum(w0))

#nonlinear constraint that implements the function and sets the constraint
 bounds
con = NonlinearConstraint(fun,1,1)

#defines a lambda function that implements the variance equation
skew = lambda w: abs(np.dot(np.dot(np.dot(w,b),w), w))

#minimization method
n_iter = 100
min_skew_arr = np.zeros(n_iter)
min_skew_weights_arr = np.zeros((n_iter,8))
for i in range(n_iter):
    w0 = np.random.dirichlet(np.ones(8),size=1)[0]
    mini = minimize(skew,w0,method='SLSQP',constraints=con)

    weights_skew = mini.x
    skewness_val = skew(mini.x)

    min_skew_weights_arr[i] = weights_skew
    min_skew_arr[i] = skewness_val

min_skew_val = np.argmin(min_skew_arr) #can plot array values

print()
print("The smallest skewness value for 100 iterations is:")
print(min_skew_arr[min_skew_val])
print()
print("The solution array at the smallest skewness value is:")
print(min_skew_weights_arr[min_skew_val])
```

```
The sum of all the weights is:
1.0

The smallest skewness value for 100 iterations is:
8.102038436174245e-05

The solution array at the smallest skewness value is:
[ -1.93716447   6.68487575  -6.55649399   4.98213832 -42.59911
142
  54.78961549 -16.02020995   1.65635026]
```

In [9]:

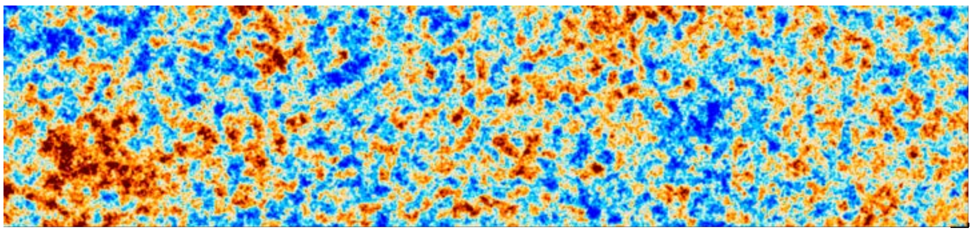
```
imap_array = np.array([imap30,imap44,imap70,imap100,imap143,imap145,imap217
,imap353])

yp_var = imap30.copy()
yp_var *= 0.

for i in range(n_map):
    yp_var += imap_array[i]*weights_var[i]

print(yp_var)

yp_skew = imap30.copy()
yp_skew *= 0

for i in range(n_map):
    yp_skew += imap_array[i]*weights_skew[i]

print(yp_skew)
```

```
[[[  57.07162856   58.04260984   59.62865946 ...   -0.29197655
     -4.50279207   -8.39625092]
  [  53.02689343   53.98265651   56.50107275 ...    6.02794916
      4.644423      2.18457073]
  [  50.19762036   52.38265102   56.01501181 ...   14.66181657
     16.49981471   15.47660926]
  ...
  [  57.48272055   47.8738915    40.25440711 ... -170.60003797
   -162.75597507 -152.56591185]
  [  65.20591467   57.81174522   52.25058953 ... -170.00285968
   -162.82751191 -152.99239267]
  [  70.30875788   65.36743423   62.45252181 ... -171.31875561
   -164.47482965 -154.42585164]]]
[[[ 188.16077581  197.89948735  209.89117016 ... -108.54371071
    -99.7150596   -85.79694524]
  [ 200.76516787  214.43740782  228.00239974 ... -153.03997308
   -156.95568315 -139.30556019]
  [ 214.61025162  222.85007831  230.70171599 ... -170.16868434
   -183.29477468 -164.55036843]
  ...
  [ 516.01050234  527.76378     513.38613503 ...    9.60099222
     33.51908031   44.78903342]
  [ 531.3874654   538.3030439   514.55090705 ...   -1.62182862
     21.59958397   32.86402372]
  [ 546.37968514  550.27509581  522.32979225 ...   -2.34326107
     13.86332205   16.29494863]]]
```

Map of the Variance

In [10]:

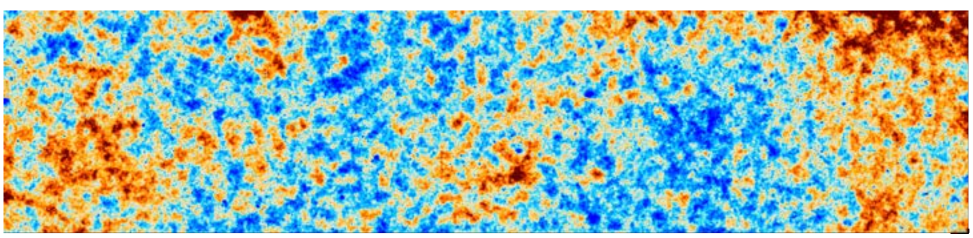
```
plots = enplot.plot(yp_var, range=300,mask=0)

def eshow(x,**kwargs): enplot.show(enplot.plot(x,**kwargs))
eshow(yp_var)
```



Map of the Skewness

In [11]:

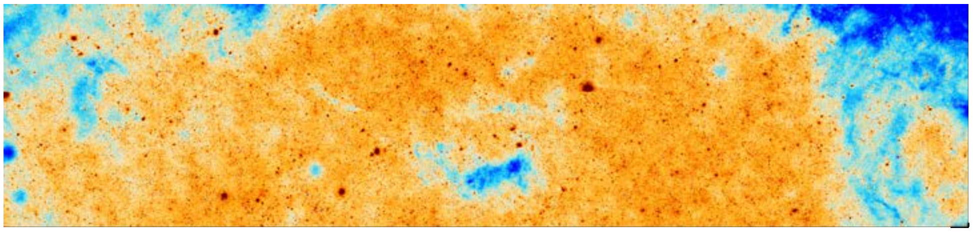
```
plots = enplot.plot(yp_skew, range=300,mask=0)

def eshow(x,**kwargs): enplot.show(enplot.plot(x,**kwargs))
eshow(yp_skew)
```



Map of the (Variance - Skewness)

In [12]:

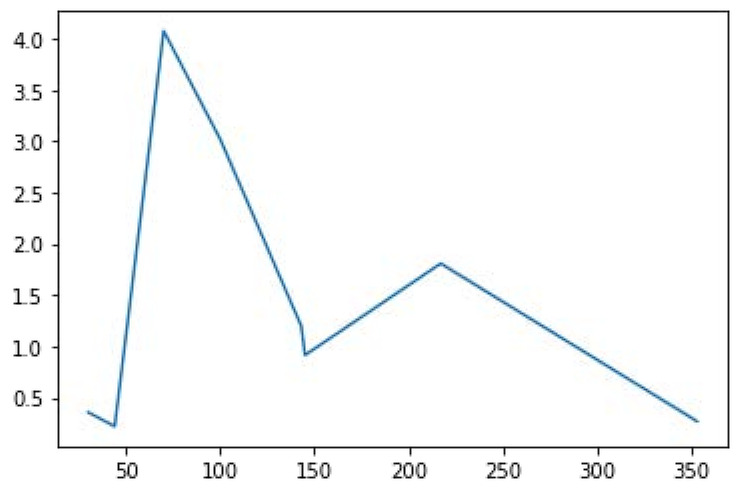
```
plots = enplot.plot(yp_var-yp_skew, range=300,mask=0)

def eshow(x,**kwargs): enplot.show(enplot.plot(x,**kwargs))
eshow(yp_var-yp_skew)
```



Comparison of the Variance and Skewness

In [13]:

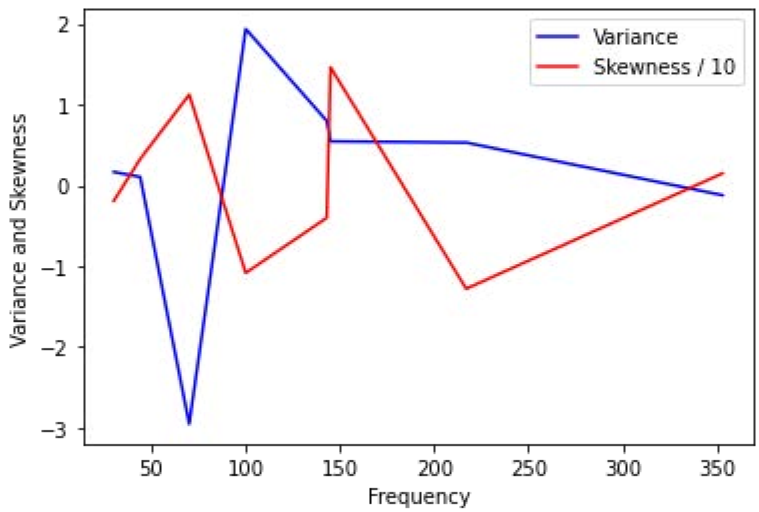
```
freq = np.array([30,44,70,100,143,145,217,353])

diff = weights_var - (weights_skew / 10)
plt.plot(freq, abs(diff))
plt.show()
```



Plot of the Sets of Weights

In [14]:

```
freq = np.array([30,44,70,100,143,145,217,353])

plot = plt.plot(freq, weights_var,'b-',freq,weights_skew/10,'r')
plt.xlabel('Frequency')
plt.ylabel('Variance and Skewness')
plt.legend(['Variance','Skewness / 10'])
plt.show()
print("Note that the skewness weights are reduced to a tenth of their origi
nal value to better compare to the variance weights.")
```




```
Note that the skewness weights are reduced to a tenth of their
original value to better compare to the variance weights.
```

Optimizing for Both Variance and Skewness

In [15]:

```
n_iter = 100
a = np.ones(len(w0))

#defines a lambda function that returns dot product of weights and array of
ones
fun = lambda w: np.dot(w,a)

#nonlinear constraint that implements the function and sets the constraint
 bounds
con = NonlinearConstraint(fun,1,1)

x_arr = []
vy_arr = []
sy_arr = []
#minimization method
def multi_mini(x,y):
    min_combo_weights_arr = np.zeros((n_iter,8))
    min_combo_arr = np.zeros(n_iter)

    for i in range(n_iter):
        w0 = np.random.dirichlet(np.ones(8),size=1)[0]

        #defines a lambda function that implements the variance equation
        var = lambda w: np.dot(np.dot(w,r),w)
        skew = lambda w: abs(np.dot(np.dot(np.dot(w,b),w), w))
        combo_func = lambda w: x*var(w) + y*skew(w)

        minimum = minimize(combo_func,w0,method='SLSQP',constraints=con)

        weights_combo = mini.x
        combo_val = combo_func(mini.x)

        min_combo_weights_arr[i] = weights_combo
        min_combo_arr[i] = combo_val

    #appends to x and var/skew arrays after finding smallest var/skew
    x_arr.append(x)
    vy_arr.append(var(minimum.x))
    sy_arr.append(skew(minimum.x))

    min_combo_val = np.argmin(min_skew_arr)

    #checking that the var and skew arrays are the same as the combined arr
ays when the weights eliminate one statistic
    if (x == 1 and y == 0):
        if (weights_var.all() == minimum.x.all()):
            print("Variance end-checker: passed")
        else:
```

```
            print("Variance end-checker: failed")
    if (x == 0 and y == 1):
        if (weights_skew.all() == minimum.x.all()):
            print("Skewness end-checker: passed")
        else:
            print("Skewness end-checker: failed")

    print()
    print("Variance weight:")
    print(x)
    print("Skewness weight:")
    print(y)
    #print("The solution array for the given linear combination of var and
 skew:")
    #print(minimum.x)
    print("Variance:")
    print(var(minimum.x))
    print("Skewness:")
    print(skew(minimum.x))
    print()
    print()

multi_mini(0,1)
multi_mini(.5,.5)
multi_mini(.7,.3)
multi_mini(.8,.2)
multi_mini(.85,.15)
multi_mini(.9,.1)
multi_mini(.95,.05)
multi_mini(.96,.04)
multi_mini(.969,.031)
multi_mini(.97,.03)
multi_mini(.971,.029)
multi_mini(1,0)
#x-axis = x
#y-axis = var/skew
```

Skewness end-checker: passed

Variance weight:
0
Skewness weight:
1
Variance:
57836.64283599049
Skewness:
0.3852223605033438

Variance weight:
0.5
Skewness weight:
0.5
Variance:
41715.79041558348
Skewness:
1.7455476616205907

Variance weight:
0.7
Skewness weight:
0.3
Variance:
251469.1530190134
Skewness:
48.70291581094653

Variance weight:
0.8
Skewness weight:
0.2
Variance:
13751.827863396591
Skewness:
0.1602044224174267

Variance weight:
0.85
Skewness weight:
0.15

```
Variance:
15365.257682387759
Skewness:
0.13841367847255956


Variance weight:
0.9
Skewness weight:
0.1
Variance:
13106.314745245772
Skewness:
0.2660811406662528


Variance weight:
0.95
Skewness weight:
0.05
Variance:
13021.042637229444
Skewness:
0.003929609624402662


Variance weight:
0.96
Skewness weight:
0.04
Variance:
12997.941075352011
Skewness:
0.0002748141465079666


Variance weight:
0.969
Skewness weight:
0.031
Variance:
12953.872739668457
Skewness:
0.01039761776564922
```

```
Variance weight:
0.97
Skewness weight:
0.03
Variance:
13088.190329160256
Skewness:
0.001081158529326577
```

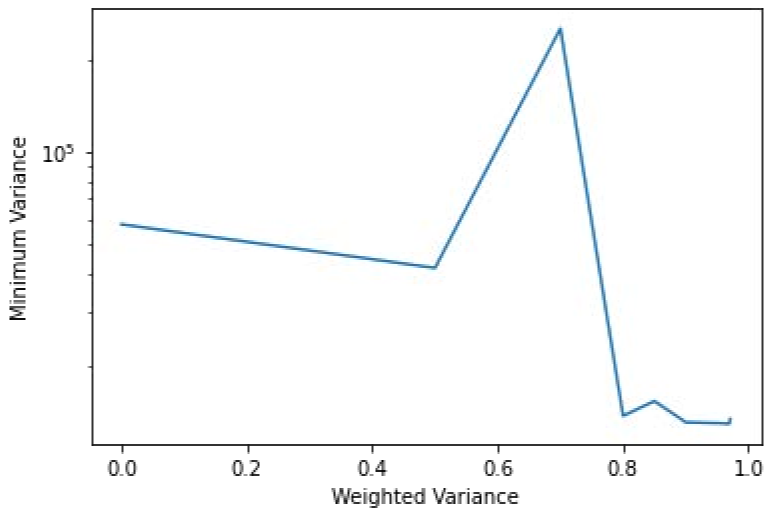
```
Variance weight:
0.971
Skewness weight:
0.029
Variance:
13439.310181988509
Skewness:
0.775646545489677
```

```
Variance end-checker: passed

Variance weight:
1
Skewness weight:
0
Variance:
12885.91368755922
Skewness:
80891.409531262
```

Plot of the Minimum Variance Values

In [22]:

```
plt.plot(x_arr, vy_arr)
plt.yscale("log")

plt.xlabel("Weighted Variance")
plt.ylabel("Minimum Variance")
plt.show()
```



Plot of the Minimum Skewness Values

In [23]:

```
plt.plot(x_arr, sy_arr)
plt.yscale("log")

plt.xlabel('Weighted Variance')
plt.ylabel('Minimum Skewness')
plt.show()
```