Appendix I: Simulation

December 4, 2015

1 Simulation

In [1]:
```
import numpy as np
import scipy.stats
from scipy import stats
import matplotlib
matplotlib.use("Agg")
import matplotlib.animation as animation
import numpy as np
from pylab import *
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
%matplotlib inline
import networkx as nx
from mpl_toolkits.mplot3d import Axes3D

import matplotlib.cm as cmx
import matplotlib.colors as colors
```

1.1 Functions

1.1.1 Functions: Algorithm

In [2]:
```
def prior_energy(G,node,z):
    # needs state z,
    # counts number of non-matches.. non matches increase energy
    neighbor_voxel_matches = 0
    for edge in G.edge[node]:
        if z!=G.node[edge]['state']:
            neighbor_voxel_matches+=1
    return(neighbor_voxel_matches* G.graph['B'])
```

In [3]:
```
def liklihood_energy(G,node,z):
    k = G.graph['tp']
    Sigma = np.eye(k)*G.graph['sigma_true'] # use true sigma for now
    u = G.graph['u_est'][:,z][np.newaxis] # 1x time points
    x =  G.node[node]['x'] # 1 x time points
    lik_en = -0.5*np.log(np.linalg.det(Sigma)) - 0.5*np.dot(np.dot((u-x),np.linalg.inv(Sigma)),
    return(lik_en)
```

In [4]:
```
def node_conditional(G,node,prior_or_posterior):
    energy_per_state =  np.zeros(G.graph['num_states'])
    for s,state in enumerate(range(G.graph['num_states'])):
        if prior_or_posterior=='posterior':
```

```
            lik_energy=liklihood_energy(G,node,state)
        elif prior_or_posterior=='prior':
            lik_energy=0
        energy_per_state[s] = np.exp(-1.0*prior_energy(G,node,state)+lik_energy)
    return(energy_per_state/energy_per_state.sum())
```

```
In [5]: #### Gibbs sample from prior
        def gibbs_sample(G,num_gibbs_samples=500,prior_or_posterior='prior'):
            nn = G.graph['nn']
            posterior_samples = np.zeros((nn*nn,num_gibbs_samples))
            for samples in range(num_gibbs_samples):
                for n,node in enumerate(G.nodes()):
                    cond_probs = node_conditional(G,node,prior_or_posterior)
                    G.node[node]['state']=np.where(np.random.multinomial(1,cond_probs))[0][0]
                    posterior_samples[n,samples] = G.node[node]['state']
            return(G,posterior_samples)
```

```
In [6]: #### ICM
        def ICM(G,num_gibbs_samples=40,prior_or_posterior='posterior'):
            nn = G.graph['nn']
            for samples in range(num_gibbs_samples):
                for n,node in enumerate(G.nodes()):
                    cond_probs = node_conditional(G,node,prior_or_posterior)
                    G.node[node]['state_k+1']=np.argmax(cond_probs)
                for n,node in enumerate(G.nodes()):
                    G.node[node]['state']=G.node[node]['state_k+1']

            return(G)
```

1.1.2 Functions: Algorithm (2.0) Estimating B

```
In [7]: def prior_energy2(G,node,z,B):
            # needs state z,
            # counts number of non-matches.. non matches increase energy
            neighbor_voxel_matches = 0
            for edge in G.edge[node]:
                if z!=G.node[edge]['state']:
                    neighbor_voxel_matches+=1.0
            return(neighbor_voxel_matches* B)
```

```
In [8]: def node_conditional2(G,node,B):
            energy_per_state =  np.zeros(G.graph['num_states'])
            for s,state in enumerate(range(G.graph['num_states'])):
                energy_per_state[s] = np.exp(-1.0*prior_energy2(G,node,state,B))
            return(energy_per_state/energy_per_state.sum())
```

```
In [9]: def Z(G,node,B):
            energy_per_state =  np.zeros(G.graph['num_states'])
            for s,state in enumerate(range(G.graph['num_states'])):
                energy_per_state[s] = np.exp(-1.0*prior_energy2(G,node,state,B))
            return(energy_per_state.sum())
```

```
In [10]: def min_for_B(BB):
             total = 0
             for sample in range(num_samples-1):
                 for n,node in enumerate(G.node):
                     state = posterior_samples[n,sample]
                     total+=np.log(node_conditional2(G,node,BB))[state]
             total = total/num_samples
             return(-1.0*total)
```

```
In [11]: def min_for_B2(BB):
             total = 0
             for sample in range(num_samples-1):
                 for n,node in enumerate(G.node):
                     state = posterior_samples[n,sample]
                     total+=-1.0*prior_energy2(G,node,state,BB)-np.log(Z(G,node,BB))
             total = total/num_samples
             return(-1.0*total)
```

1.1.3 Functions: Simulations

```
In [12]: def generate_mean_time_series(timepoints):
             u =np.random.normal(size=timepoints)
             return(u)

         def generate_x_from_mean(u,sigma,samples):
             E = np.eye(len(u))*sigma
             x = np.random.multivariate_normal(u,E,size=samples)
             return(x)
```

```
In [13]: def generate_mean_time_series_normalized(n,rho,e,how_to_norm=1):
             u = np.zeros(n)
             u[0]=np.random.normal(1)
             for i in np.array(range(n-1))+1:
                 u[i] = u[i-1]*rho + np.random.normal(0,e)

             u = u-u.mean() # 0-mean
             if how_to_norm==1:
                 u = u/np.linalg.norm(u,2) # unit-norm
             elif how_to_norm==2:
                 u = u/u.std()
             else:
                 u = u
             return(u)
```

```
In [14]: def generate_x_from_mean_single_normalized(u,t,SNR,how_to_norm=1):
             x = np.zeros(t)
             x = u+np.random.normal(0,SNR,t)
             x = x-x.mean() # 0-mean
             if how_to_norm==1:
                 x = x/np.linalg.norm(x,2) # unit-norm
             elif how_to_norm==2:
                 x = x/x.std() # std
             else:
                 x = x
             return(x)
```

In [15]:
```
### Generate Data ###
def add_data_to_graph(G,tp=3,sigma=1,normalized=False):


    X = np.zeros((tp,G.graph['nn']**2))
    u = np.empty((tp,G.graph['num_states']))

    for ui in range(G.graph['num_states']):
        u[:,ui] =generate_mean_time_series(tp)
        #if normalized


    states = np.array([])
    for ni,n in enumerate(G.node):

        x = generate_x_from_mean(u[:,G.node[n]['state']],sigma,1)
        states = np.append(states,G.node[n]['state'])
        X[:,ni] = x
        G.node[n]['x'] = x


    G.graph['u_true'] = u
    G.graph['sigma_true'] = sigma
    G.graph['X'] = X

    # Generate Random Estimates
    u_est = np.empty((tp,G.graph['num_states']))
    for ui in range(G.graph['num_states']):
        u_est[:,ui] =generate_mean_time_series(tp)
    G.graph['u_est'] = u_est

    G.graph['tp'] = tp
    G.graph['states'] = states

    return(G)
```

In [16]:
```
def assess_perf(G):

    # for true states, find all matching nodes.
    for node1 in G.nodes():
        state1 = G.node[node1]['true_state']
        G.node[node1]['matching_nodes_true'] = []
        for node2 in G.nodes():
            state2 = G.node[node2]['true_state']
            if state1==state2:
                G.node[node1]['matching_nodes_true'].append(node2)

    for node1 in G.nodes():
        state1 = G.node[node1]['state']
        G.node[node1]['matching_nodes'] = []
        for node2 in G.nodes():
            state2 = G.node[node2]['state']
            if state1==state2:
                G.node[node1]['matching_nodes'].append(node2)
```

```
        # another graph
        matches = np.empty(G.graph['nn']**2)
        for ni,node1 in enumerate(G.nodes()):
            matches[ni]=len(set(G.node[node1]['matching_nodes']).intersection(set(G.node[node1]['ma


        return(matches.sum())
```

1.1.4 Functions: Plotting

```
In [17]: ### Change Color
         def change_node_color(G,changeto):

             for node in G.node:
                 G.node[node]['state'] = changeto[G.node[node]['state']]

             return(G)
```

```
In [18]: def plot_graph(G):
             im = np.empty((G.graph['nn'],G.graph['nn']))
             for node in G.node:
                 im[node]=G.node[node]['state']

             imm = plt.imshow(im,interpolation='None',origin='lower')
             return(imm)
```

```
In [19]: def get_cmap(N):
             '''Returns a function that maps each index in 0, 1, ... N-1 to a distinct
             RGB color.'''
             color_norm  = colors.Normalize(vmin=0, vmax=N-1)
             scalar_map = cmx.ScalarMappable(norm=color_norm, cmap='hsv')
             def map_index_to_rgb_color(index):
                 return scalar_map.to_rgba(index)
             return map_index_to_rgb_color
```

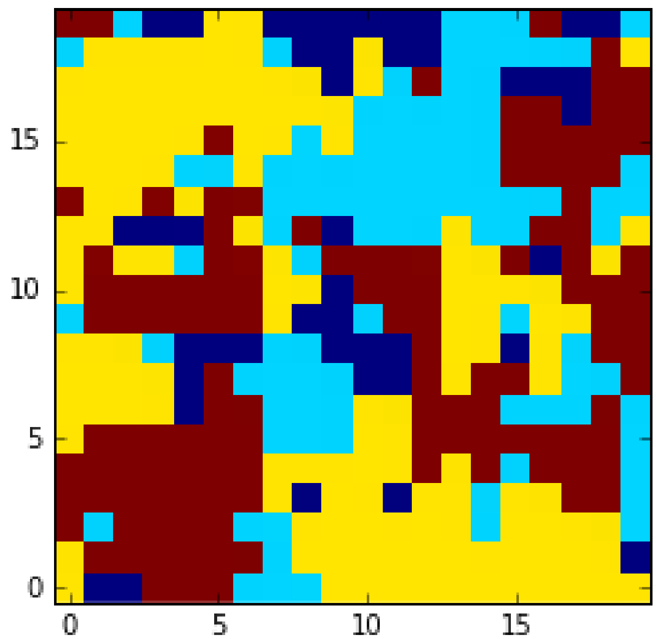
1.2 Simulation

1.2.1 Generate 'True' Network labels by sampling from the potts model (prior) with Gibbs Sampling

```
In [49]: ### Initialize Graph
         nn=20
         num_nodes = nn*nn
         G=nx.grid_2d_graph(nn,nn)  #4x4 grid
         G.graph['B'] = 1.1
         G.graph['num_states'] = 4
         G.graph['nn']=nn
         sigma = 1
         # randomize initialize states
         for n in G.node:
             state=np.random.randint(G.graph['num_states'])
             G.node[n]['state']=state
```

```
G,posterior_samples = gibbs_sample(G,100)

G = add_data_to_graph(G,3,sigma)
G.graph['X'].shape

for n in G.node:
    G.node[n]['true_state']=G.node[n]['state']
plt.figure()
plot_graph(G)
print(assess_perf(G))
```

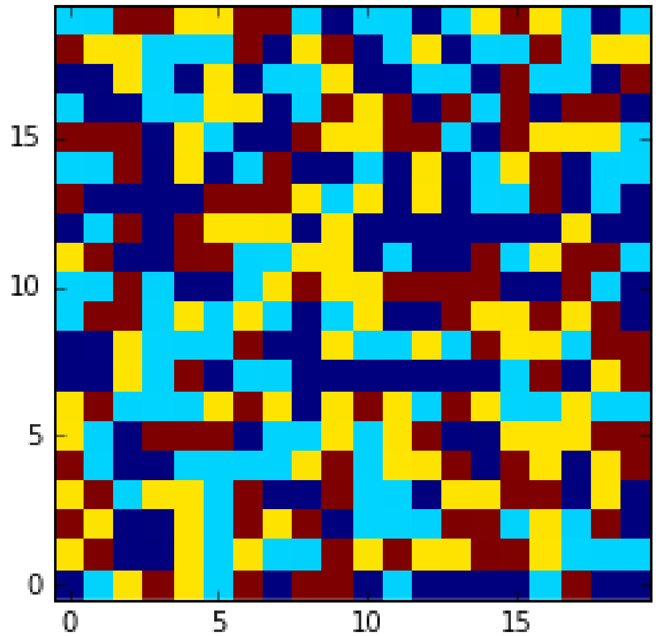
```
45294.0
```



1.2.2 Randomize Network Labels (to give to algorithm)

In [50]:
```
# randomize starting position
for n in G.node:
    state=np.random.randint(G.graph['num_states'])
    G.node[n]['state']=state
plt.figure()
plot_graph(G)

print(assess_perf(G))
```

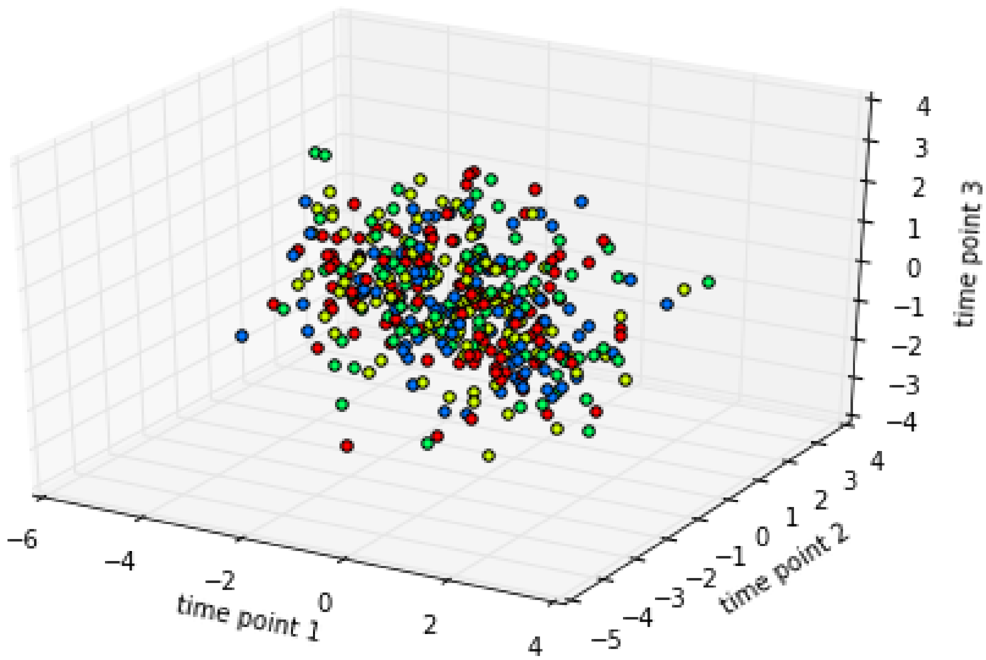
```
11490.0
```



1.2.3 Visualize Unnormalized X data

In [51]:
```
from mpl_toolkits.mplot3d import Axes3D
fig = plt.figure(figsize=(8,5))
ax = fig.add_subplot(111, projection='3d')
cmap = get_cmap(G.graph['num_states']+2)

for node in G.node:
    x = G.node[node]['x']
    state = G.node[node]['state']
    ax.scatter(x[0,0], x[0,1], x[0,2], c=cmap(state))

ax.set_xlabel('time point 1')
ax.set_ylabel('time point 2')
ax.set_zlabel('time point 3')
```

Out[51]: <matplotlib.text.Text at 0x106fb3c10>

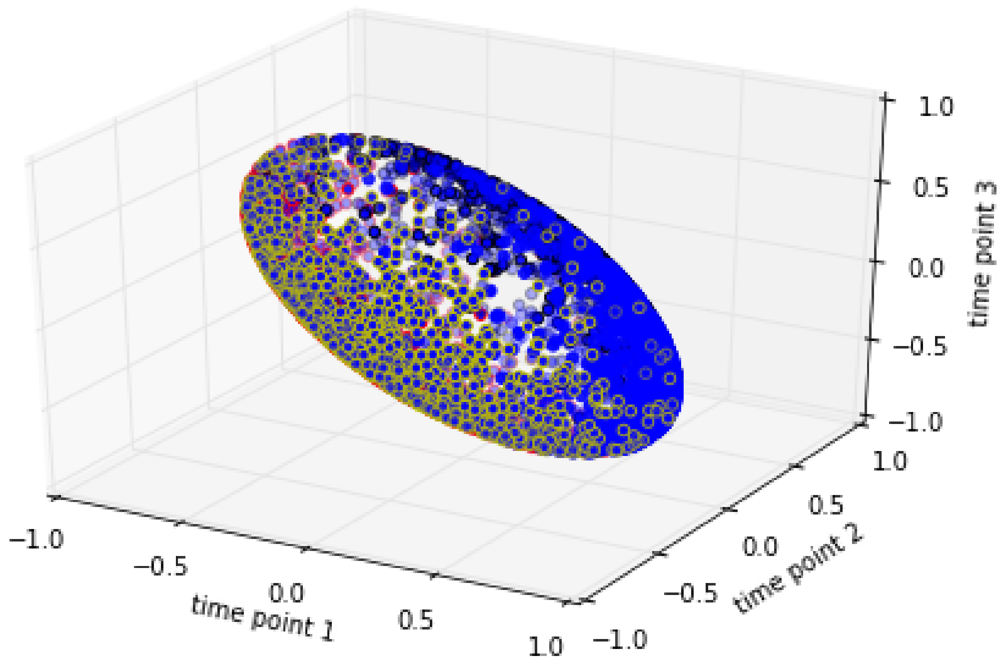


1.2.4 Visualize normalized X data (not used in this simulation)

In [52]:
```
#### Using Unit Normalization.

t = 4
n = 5000
clusters = 4
X = np.zeros((t,n))
u = np.empty((t,n))
l = np.empty((n))
corr = 0.6
e = 0.4
SNR = .6
for ui in range(clusters):
    u[:,ui] =generate_mean_time_series_normalized(t,corr,e)


for xi in range(n):
    clust = np.random.randint(clusters)
    x = generate_x_from_mean_single_normalized(u[:,clust],t,SNR)
    X[:,xi] = x
    l[xi] = clust

fig = plt.figure(figsize=(8,5))
ax = fig.add_subplot(111, projection='3d')
cols = ['b','y','r','k']
for c in range(clusters):
    ax.scatter(X[0,l==c],X[1,l==c], X[2,l==c],color=cols[c])
```

```
        ax.set_xlabel('time point 1')
        ax.set_ylabel('time point 2')
        ax.set_zlabel('time point 3')
```

Out[52]: <matplotlib.text.Text at 0x10747ba90>

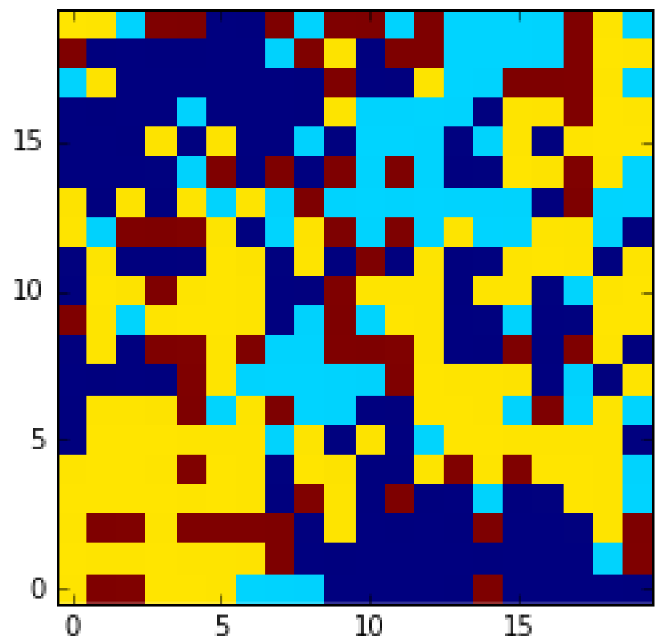


1.2.5 Cluster Data using K-means

In [ ]:

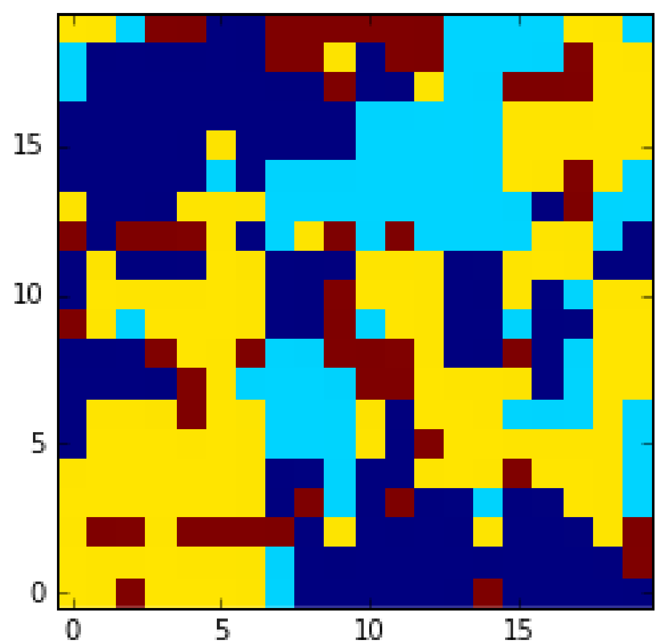
In [53]:
```
from sklearn.cluster import KMeans
kmeans = KMeans(G.graph['num_states'])
results = kmeans.fit(G.graph['X'].T)
labels = results.fit_predict(G.graph['X'].T)

# set states
for ni,n in enumerate(G.node):
    G.node[n]['state']=labels[ni]

# set centroids
G.graph['u_est'] = results.cluster_centers_.T
plot_graph(G)
print(assess_perf(G))
```

```
29010.0
```



```
In [54]: G,posterior_samples = gibbs_sample(G,100,prior_or_posterior='posterior')
         plot_graph(G)
```

```
Out[54]: <matplotlib.image.AxesImage at 0x106d81690>
```



```
In [55]: print(assess_perf(G))
```

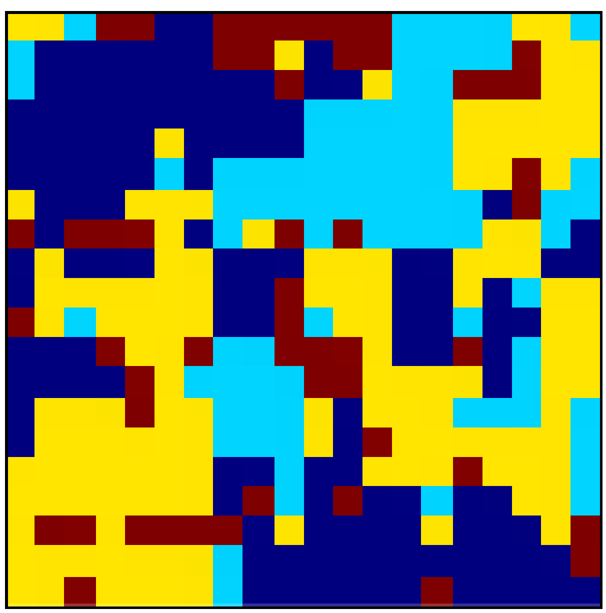
```
33742.0
```

1.2.6 Cluster Data using Current Model (Monte-Carlo EM)

In [56]:
```
### EM

#perf =
### Movie of Sampling from posterior
FFMpegWriter = animation.writers['ffmpeg']
metadata = dict(title='Movie Test', artist='Matplotlib',
                comment='Movie support!')
writer = FFMpegWriter(fps=10, metadata=metadata)


fig = plt.figure()
ax = fig.add_subplot(111)
ax.set_aspect('equal')
ax.get_xaxis().set_visible(False)
ax.get_yaxis().set_visible(False)
im = np.empty((G.graph['nn'],G.graph['nn']))
for node in G.node:
    im[node]=G.node[node]['state']

imm = plt.imshow(im,interpolation='None',origin='lower')

G.graph['Bstore'] = np.array([])

#with writer.saving(fig, "em_sampling.mp4", 100):
#    writer.grab_frame()

for em_step in range(10):
        ### E step -
        ### Get posterior P(Z|Z_n,X)
    print(em_step)
    num_samples=50
    G,posterior_samples = gibbs_sample(G,100,prior_or_posterior='posterior')
    posterior_samples = posterior_samples[:,-1*num_samples:-1]

        ### show samples from posterior
        #for i in range(10):

        #    G,posterior_samples2 = gibbs_sample(G,1,prior_or_posterior='posterior')
        #    temp = np.empty((G.graph['nn'],G.graph['nn']))
        #    for node in G.node:
        #        temp[node]=G.node[node]['state']
        #    imm.set_data(temp)
        #    writer.grab_frame()



        #### M step

        ### Estimate U
    u_est_new =G.graph['u_est']
    u_est_new_count = G.graph['u_est']*0.0+1.0
```

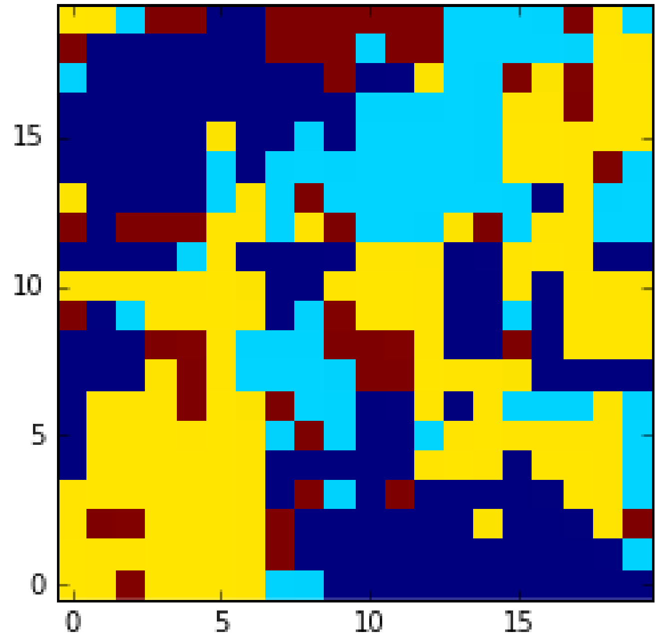
```
        for sample in range(num_samples-1):
            for n,node in enumerate(G.node):
                state = posterior_samples[n,sample] # node was in this state
                u_est_new[:,int(state)] += G.node[node]['x'][0,:] # add the x data to that time ve
                u_est_new_count[:,int(state)] += np.ones(G.graph['tp'])
        u_est_new = u_est_new/u_est_new_count
        G.graph['u_est'] = u_est_new

            ### Estimate Sigma (pooled across clusters)

            ### Estimate B
        val = np.array([])
        brange = np.arange(0.1,1.8,.1)
        for b in brange:
            val = np.append(val,min_for_B2(b))
            newb = brange[np.argmin(val)]
        G.graph['B'] = newb
        G.graph['Bstore'] = np.append(G.graph['Bstore'],newb)
        G.graph['u_est'] = u_est_new
```

```
0
1
2
3
4
5
6
7
8
9
```



```
In [57]: plot_graph(G)
```

Out[57]: <matplotlib.image.AxesImage at 0x106f11050>



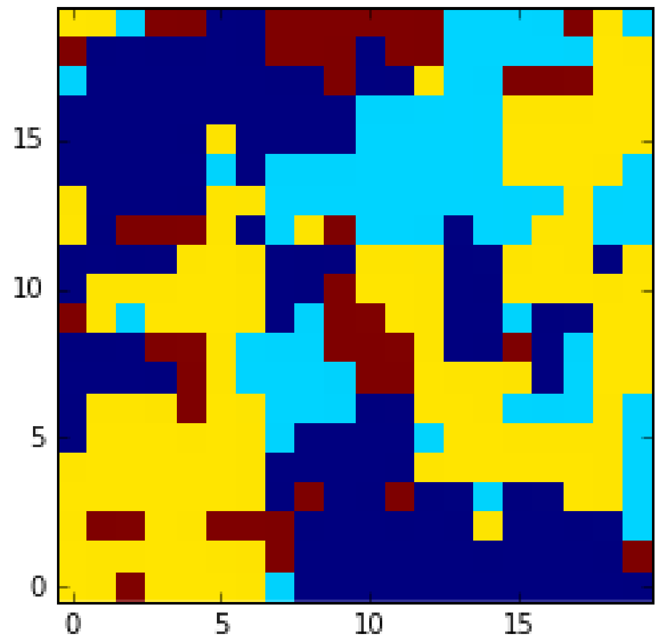
In [58]: G.graph['Bstore']

Out[58]: array([1.1, 1.1, 1. , 1.1, 1. , 1. , 1. , 1.1, 1. , 1.])

1.2.7 Perform Iterated Conditional Modes

```
In [59]: ICM(G,num_gibbs_samples=40,prior_or_posterior='posterior')
         plot_graph(G)
         print(assess_perf(G))
```

36184.0



1.2.8 Save a movie of sampling from the posteior

In [ ]:
```
### Movie of Sampling from posterior
FFMpegWriter = animation.writers['ffmpeg']
metadata = dict(title='Movie Test', artist='Matplotlib',
                comment='Movie support!')
writer = FFMpegWriter(fps=10, metadata=metadata)


fig = plt.figure()
ax = fig.add_subplot(111)
ax.set_aspect('equal')
ax.get_xaxis().set_visible(False)
ax.get_yaxis().set_visible(False)
im = np.empty((G.graph['nn'],G.graph['nn']))
for node in G.node:
    im[node]=G.node[node]['state']

imm = plt.imshow(im,interpolation='None',origin='lower')
#im.set_clim([0,1])


with writer.saving(fig, "posterior_sampling.mp4", 100):
    for i in range(100):
        writer.grab_frame()
        G,posterior_samples = gibbs_sample(G,1,prior_or_posterior='posterior')
        temp = np.empty((G.graph['nn'],G.graph['nn']))
        for node in G.node:
            temp[node]=G.node[node]['state']
        imm.set_data(temp)
```

1.2.9 Recovery of Hidden States is good.

1.3 Additional Stuff

1.3.1 Movie of Sampling from Potts Model

In [ ]:
```
### Movie of Sampling from Prior different B'
### Initialize Graph
nn=20
for BB in [0.2,0.5,1,1.2,1.5,1.7]:
    num_nodes = nn*nn
    G=nx.grid_2d_graph(nn,nn)  #4x4 grid
    G.graph['B'] = BB
    G.graph['num_states'] = 4
    G.graph['nn']=nn
    # randomize initialize states
    for n in G.node:
        state=np.random.randint(G.graph['num_states'])
        G.node[n]['state']=state



    FFMpegWriter = animation.writers['ffmpeg']
    metadata = dict(title='Movie Test', artist='Matplotlib',
                    comment='Movie support!')
    writer = FFMpegWriter(fps=10, metadata=metadata)


    fig = plt.figure()
    ax = fig.add_subplot(111)
    ax.set_aspect('equal')
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
    im = np.empty((G.graph['nn'],G.graph['nn']))
    for node in G.node:
        im[node]=G.node[node]['state']

    imm = plt.imshow(im,interpolation='None',origin='lower')
    #im.set_clim([0,1])


    with writer.saving(fig, "sample_prior"+str(BB)+".mp4", 100):
        for i in range(100):
            G,posterior_samples = gibbs_sample(G,1)
            temp = np.empty((G.graph['nn'],G.graph['nn']))
            for node in G.node:
                temp[node]=G.node[node]['state']
            imm.set_data(temp)
            writer.grab_frame()
```

1.4 Fixed my Estimation of B !

In [ ]:
```
### Initialize Graph

for BB in [0.2,0.5,1,1.2,1.5,1.7]:
```

```
nn=15
num_nodes = nn*nn
G=nx.grid_2d_graph(nn,nn)  #4x4 grid
G.graph['B'] = BB
G.graph['num_states'] = 4
G.graph['nn']=nn
sigma = 1
# randomize initialize states
for n in G.node:
    state=np.random.randint(G.graph['num_states'])
    G.node[n]['state']=state

num_samples=50
G,posterior_samples = gibbs_sample(G,100)
posterior_samples = posterior_samples[:,-1*num_samples:-1]


plt.figure()
plot_graph(G)
plt.figure()
BBs = np.arange(0,2,.1)
for b,bb in enumerate(BBs):
    plt.scatter(bb,min_for_B(bb))
plt.axvline(x=BB)
```

In [ ]:
```
nn=20
num_nodes = nn*nn
G=nx.grid_2d_graph(nn,nn)  #4x4 grid
G.graph['B'] = 1
G.graph['num_states'] = 4
G.graph['nn']=nn
sigma = 1
# randomize initialize states
for n in G.node:
    state=np.random.randint(G.graph['num_states'])
    G.node[n]['state']=state

num_samples=100
G,posterior_samples = gibbs_sample(G,100)
posterior_samples = posterior_samples[:,-1*num_samples:-1]

BBs = np.arange(0,3,.01)
for b,bb in enumerate(BBs):
    plt.scatter(bb,min_for_B2(bb))
```

In [ ]:
```
len(G.node)
```

In [ ]:
```
from scipy.optimize import minimize
x0=1
#res = minimize(min_for_B, x0, method='nelder-mead', options={'xtol': 1e-8, 'disp': True})
```

In [ ]:
```
res.values()
```

In [ ]:
```
val = np.array([])
brange = np.arange(0.1,1.8,.1)
```

```python
for b in brange:
    val = np.append(val,min_for_B2(b))
newb = brange[np.argmin(val)]
print(newb)
```

Appendix II: fMRI Implementation

December 4, 2015

1 Plan

1.1 fMRI data extraction portion

1. get gray matter.
2. normalize data.
3. grab stable block data.
4. estimate variance roughly (as if it was from normal).
5. given image, get coordinates and transform into nodes/edges
6. run prior simulation (save movie?). See if beta=1 still reasonable given 3D? E.g. looks like brain data.
7. K-means
8. MEMC-HMRF

   [‘n8’ ‘n10’ ‘n13’ ‘n14’ ‘n20’ ‘n21’ ‘n22’ ‘n23’ ‘n24’ ‘n25’ ‘n28’ ‘n32’ ‘n33’ ‘n34’ ‘n35’ ‘n41’ ‘n42’]

2 Imports

```
In [1]: import nibabel as nib
        import nipy as nipy
        import os
        import glob
        import numpy as np
        import nilearn as nil

        from nilearn.input_data import NiftiMasker
        import scipy
        import pickle
        from scipy import stats
        from sklearn import linear_model
        import re
        import pandas as pd
        import os
        from nipy.modalities.fmri.utils import events, Symbol, lambdify_t
        from nipy.modalities.fmri.hrf import glover
        import time
        import matplotlib.pyplot as plt
        import matplotlib.gridspec as gridspec
        from nilearn import image
```

```
import copy
from matplotlib.backends.backend_pdf import PdfPages
import scipy.linalg as spl
from nipy.modalities.fmri.hemodynamic_models import compute_regressor, _orthogonalize
from nipy.modalities.fmri.design_matrix import _make_drift

from nilearn.image.image import mean_img
### Show results ###############################################################
from nilearn.plotting import plot_stat_map
from nilearn.image import index_img

from nitime.analysis import SpectralAnalyzer, FilterAnalyzer
from nitime.timeseries import TimeSeries
import h5py

import nilearn.plotting.img_plotting
import nilearn.plotting.displays
reload(nilearn.plotting.displays)
from nilearn.plotting.img_plotting import plot_epi, plot_roi,plot_stat_map
%matplotlib inline

import networkx as nx
```

In [ ]:
```
def plot_img_4_rows(img,threshold=0,title=None,vmax=None):

    slices = np.linspace(-44,55,32)

    f = plt.figure(figsize=(14,10))
    gs = gridspec.GridSpec(8, 2)
    #print('here')

    ax = plt.subplot(gs[0:2, :])
    display = plot_stat_map(img,threshold=threshold,display_mode='z', cut_coords=slices[0:8],ax

    ax = plt.subplot(gs[2:4, :])
    display = plot_stat_map(img,threshold=threshold,display_mode='z', cut_coords=slices[8:16],a

    ax = plt.subplot(gs[4:6, :])
    display = plot_stat_map(img,threshold=threshold,display_mode='z', cut_coords=slices[16:24],

    ax = plt.subplot(gs[6:8, :])
    display = plot_stat_map(img,threshold=threshold,display_mode='z', cut_coords=slices[24:32],
```

2.0.1 Get data / Mask / Normalize

In [6]:
```
sub = 'n10'
analysisdir = '/home/bishop/studies/AnxLearn2/anxlearn_fmri/input_epi/'
filename = analysisdir+sub+'/pain/preprocessed_nounwarp.feat/prefiltered_func_data_smooth.nii.g
filenames=glob.glob(filename)
nifti_masker = NiftiMasker( mask_strategy='epi',standardize=True,detrend=True)
nifti_masker.fit(filenames[0])
```

Out[6]:
```
NiftiMasker(detrend=True, high_pass=None, low_pass=None, mask_args=None,
      mask_img=None, mask_strategy='epi', memory=Memory(cachedir=None),
      memory_level=1, sample_mask=None, sessions=None, smoothing_fwhm=None,
```

```
                standardize=True, t_r=None, target_affine=None, target_shape=None,
                verbose=0)
```

```
In [7]: fmri_masked_flat = nifti_masker.transform(filenames[0])
```

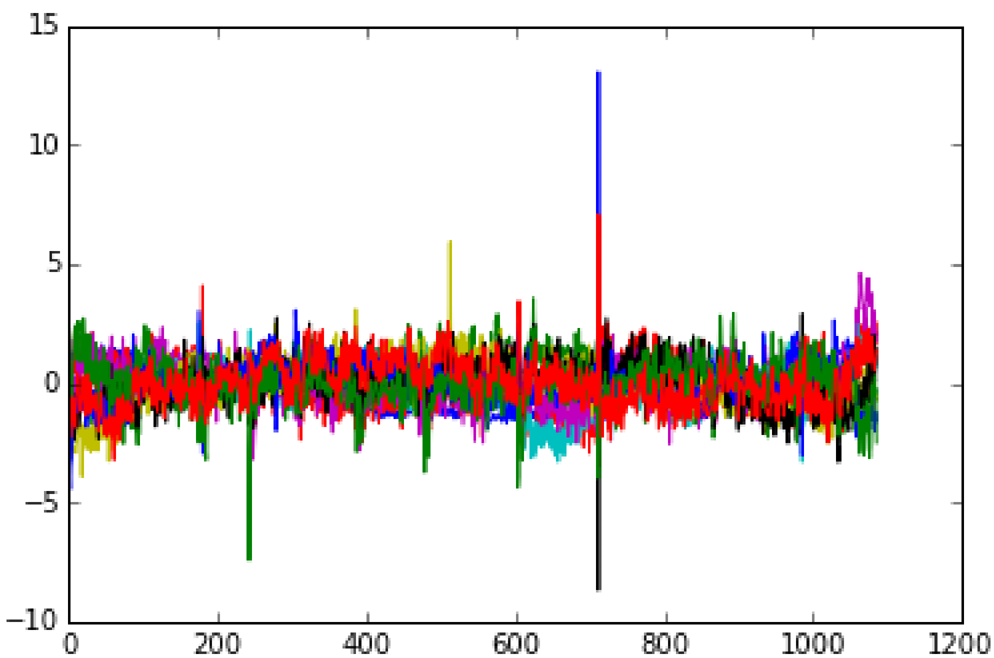
2.0.2 Take A look at Data

```
In [8]: fmri_masked_flat.shape
```

```
Out[8]: (1086, 61398)
```

```
In [9]: plt.plot(fmri_masked_flat[:,np.random.randint(fmri_masked_flat.shape[1],size=10)])
```

```
Out[9]: [<matplotlib.lines.Line2D at 0x7fa3f7b92e10>,
         <matplotlib.lines.Line2D at 0x7fa3f7ba70d0>,
         <matplotlib.lines.Line2D at 0x7fa3f7ba7310>,
         <matplotlib.lines.Line2D at 0x7fa3f7ba74d0>,
         <matplotlib.lines.Line2D at 0x7fa3f7ba7690>,
         <matplotlib.lines.Line2D at 0x7fa3f7ba7850>,
         <matplotlib.lines.Line2D at 0x7fa3f7ba7a10>,
         <matplotlib.lines.Line2D at 0x7fa3f7b52c50>,
         <matplotlib.lines.Line2D at 0x7fa3f7ba7d90>,
         <matplotlib.lines.Line2D at 0x7fa3f7ba7f50>]
```




```
In [58]: #fmri_masked_image2 = nifti_masker.inverse_transform(fmri_masked_flat*2)
```

```
In [61]: #fmri_masked_image2.to_filename('test.nii')
```

2.0.3 Index Image

```
In [10]: index_image = nifti_masker.inverse_transform(np.arange(fmri_masked_flat.shape[1]))
```

```
In [11]: index_image_data= index_image.get_data()
         index_image_data.shape
```

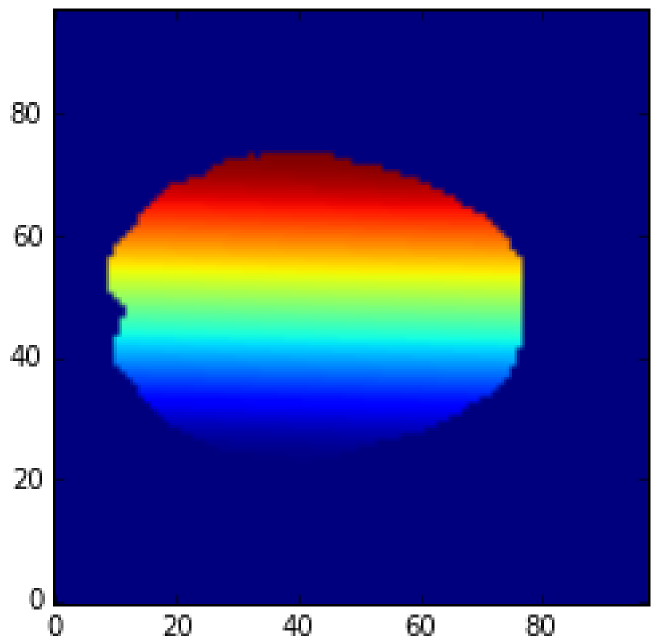
```
Out[11]: (98, 98, 31)
```

```
In [12]: np.where(index_image_data==100)
```

```
Out[12]: (array([22]), array([42]), array([12]))
```

```
In [13]: plt.imshow(index_image_data[:,:,20],origin='lower') # cmap='gray'
```

```
Out[13]: <matplotlib.image.AxesImage at 0x7fa3f59f7ad0>
```



2.0.4 Graph

```
In [57]: s = np.shape(index_image_data)
         G = nx.grid_graph(dim=[s[0],s[1],s[2]])
```

2.0.5 Prune Graph

```
In [60]: for d1 in range(s[0]):
             for d2 in range(s[1]):
                 for d3 in range(s[2]):
                     if index_image_data[d1,d2,d3]==0:
                         G.remove_node((d1,d2,d3))
```

```
In [61]: G.edge[(70,34,14)]
```

```
Out[61]: {(69, 34, 14): {},
          (70, 33, 14): {},
          (70, 34, 13): {},
          (70, 34, 15): {},
          (70, 35, 14): {},
          (71, 34, 14): {}}
```

```
In [62]: len(G.node)
```

```
Out[62]: 61397
```

2.0.6 Get Mean Image

```
In [16]: mean_func_img = mean_img(filenames[0])
```

2.0.7 ICA

```
In [17]: #from sklearn.decomposition import FastICA
         #n_components = 7
         #ica = FastICA(n_components=n_components, random_state=42)
         #components_masked = ica.fit_transform(fmri_masked_flat.T).T
```

```
In [18]: #components_masked.shape
```

```
Out[18]: (7, 61398)
```

```
In [22]: #components = nifti_masker.inverse_transform(components_masked)
```

```
In [21]: #for comp in range(7):
         #    index_img(components, comp).to_filename('ICA'+str(comp)+'.nii')
```

```
In [89]: #plot_img_4_rows(index_img(components, 1),threshold=0.001,vmax=0.02)
```

2.0.8 Choose 7 Clusters

```
In [90]: num_states = 7
```

2.0.9 K-Means

```
In [ ]: #from sklearn.cluster import KMeans
        #kmeans = KMeans(num_states)
```

```
In [31]: #results = kmeans.fit(fmri_masked_flat.T) #time second argument
```

```
In [32]: #labels = results.fit_predict(fmri_masked_flat.T)
```

```
In [33]: #labels_image = nifti_masker.inverse_transform(labels)
```

```
In [34]: #labels.shape
```

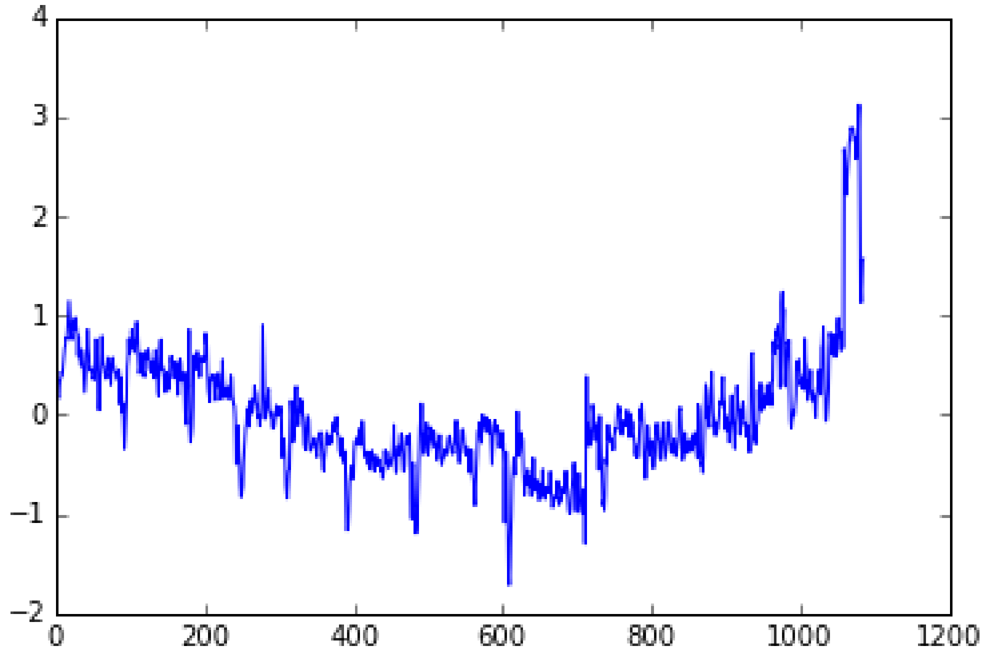
```
Out[34]: (61398,)
```

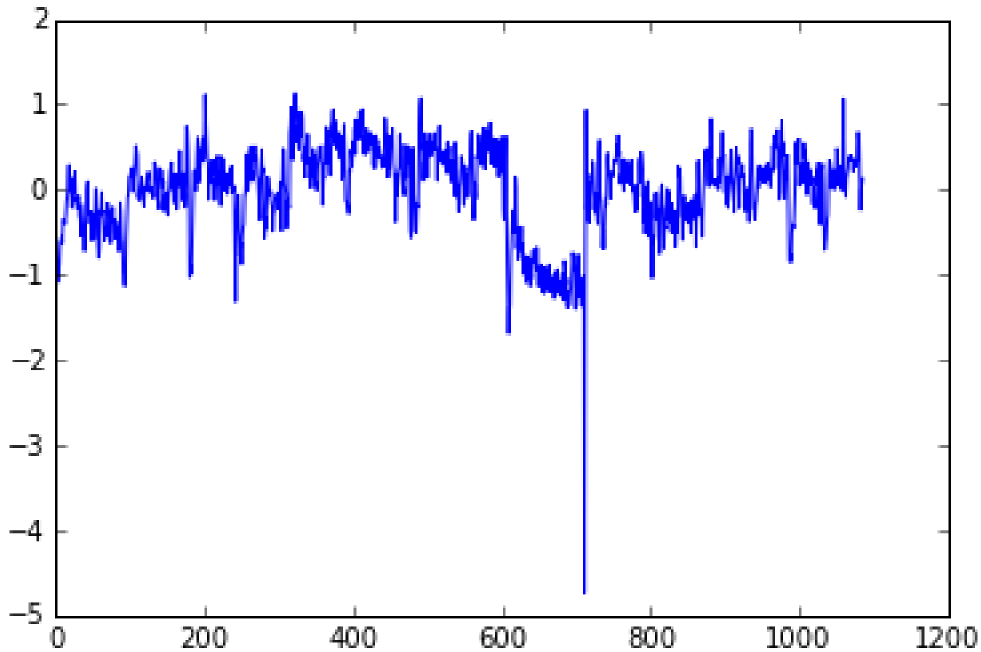
```
In [58]: labels_image.to_filename('Kclust.nii')
```

```
In [91]: #plot_img_4_rows(labels_image,threshold=0,vmax=10)
```

```
In [38]: cluster_time_series = results.cluster_centers_.T
```

In [92]:
```
for i in range(2):
    plt.figure()
    plt.plot(cluster_time_series[:,i])
```





2.0.10 Add Data to Graph

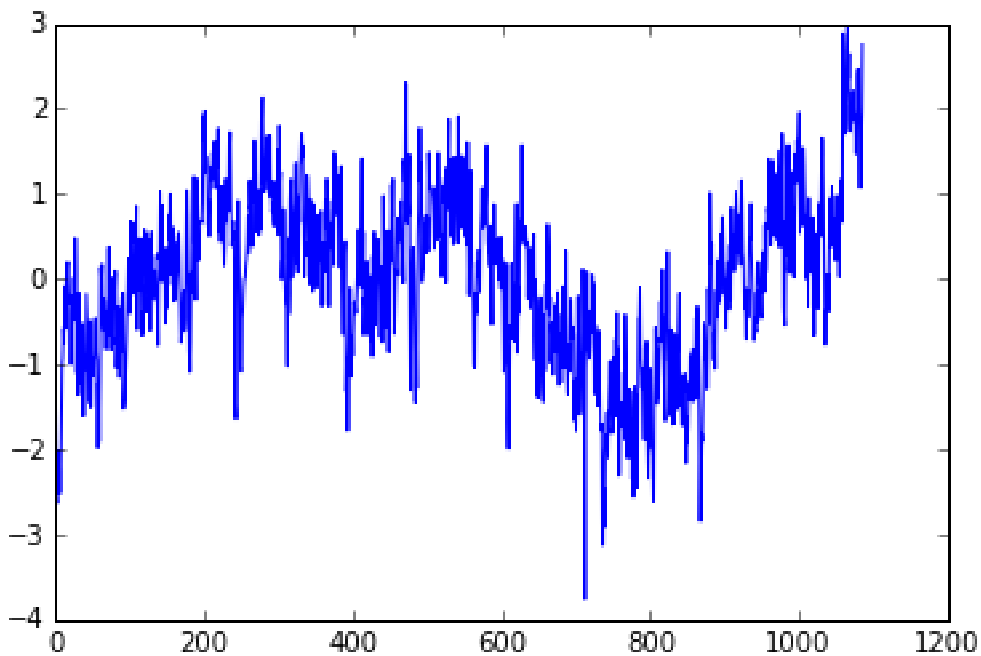
```
In [65]: fmri_masked_flat.shape
```

```
Out[65]: (1086, 61398)
```

```
In [66]: for node in G.nodes():
             G.node[node]['x'] = fmri_masked_flat[:,index_image_data[node[0],node[1],node[2]]]
```

```
In [68]: plt.plot(G.node[node]['x'])
```

```
Out[68]: [<matplotlib.lines.Line2D at 0x7fa37a0a2dd0>]
```




```
In [81]: G.graph['img_shape']= index_image_data.shape
```

```
In [93]: G.graph['num_states'] = num_states
```

```
In [143]: G.graph['tp']= fmri_masked_flat.shape[0]
```

```
In [116]: G.graph['num_nodes'] = fmri_masked_flat.shape[1]
```

2.0.11 Add K-means states

```
In [78]: labels.shape
```

```
Out[78]: (61398,)
```

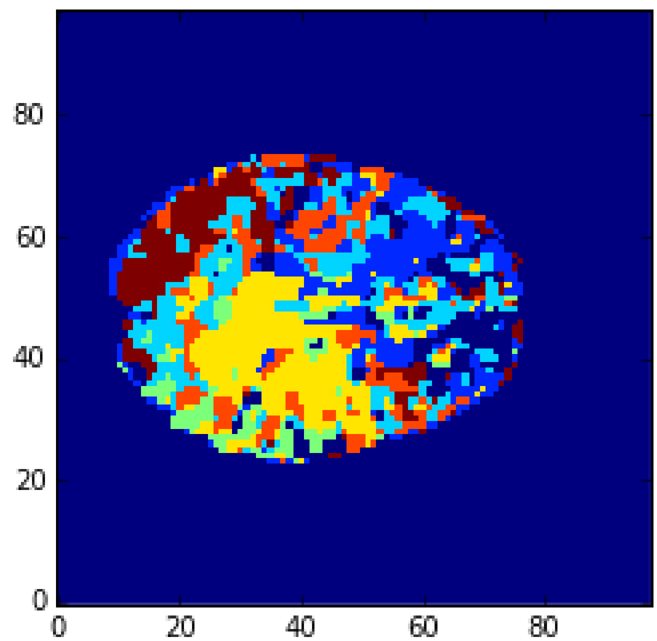
```
In [80]: for node in G.nodes():
             G.node[node]['state'] = labels[index_image_data[node[0],node[1],node[2]]]
             G.node[node]['kstate'] = labels[index_image_data[node[0],node[1],node[2]]]
```

2.0.12 Plot Slice of Labels

In [86]:
```
def plot_graph(G,slicee):
    im = np.empty((G.graph['img_shape'][0],G.graph['img_shape'][1]))
    for node in G.node:
        if node[2]==slicee:
            im[node[0],node[1]]=G.node[node]['state']

    imm = plt.imshow(im,interpolation='None',origin='lower')
    return(imm)
```

In [87]:
```
plot_graph(G,20)
```

Out[87]: <matplotlib.image.AxesImage at 0x7fa39fc1da50>



In [ ]:

2.0.13 MRF Functions

In [100]:
```
def prior_energy(G,node,z):
    # needs state z,
    # counts number of non-matches.. non matches increase energy
    neighbor_voxel_matches = 0
    for edge in G.edge[node]:
        if z!=G.node[edge]['state']:
            neighbor_voxel_matches+=1
    return(neighbor_voxel_matches* G.graph['B'])
```

In [101]:
```
def node_conditional(G,node,prior_or_posterior):
    energy_per_state =  np.zeros(G.graph['num_states'])
    for s,state in enumerate(range(G.graph['num_states'])):
```

```
            if prior_or_posterior=='posterior':
                lik_energy=liklihood_energy(G,node,state)
            elif prior_or_posterior=='prior':
                lik_energy=0
            energy_per_state[s] = np.exp(-1.0*prior_energy(G,node,state)+lik_energy)
        return(energy_per_state/energy_per_state.sum())
```

```
In [170]: def liklihood_energy(G,node,z):
              k = G.graph['tp']
              #Sigma = np.eye(k)*G.graph['sigma_est'] # use true sigma for now
              u = G.graph['u_est'][:,z] # 1x time points
              x =  G.node[node]['x'] # 1 x time points
              #lik_en = -0.5*np.log(1.0) - 0.5*np.dot(np.dot((u-x),np.eye(k)),(u-x).T) - k/2.0*np.log(2
              lik_en = -0.5*np.log(1.0) - 0.5*np.sum((u-x)**2) - k/2.0*np.log(2.0*np.pi)
              return(lik_en)
```

```
In [136]: #### Gibbs sample from prior
          def gibbs_sample(G,num_gibbs_samples=1,prior_or_posterior='prior'):

              posterior_samples = np.zeros((G.graph['num_nodes'],num_gibbs_samples))
              for samples in range(num_gibbs_samples):
                  for n,node in enumerate(G.nodes()):
                      cond_probs = node_conditional(G,node,prior_or_posterior)
                      G.node[node]['state']=np.where(np.random.multinomial(1,cond_probs))[0][0]
                      posterior_samples[n,samples] = G.node[node]['state']
              return(G,posterior_samples)
```

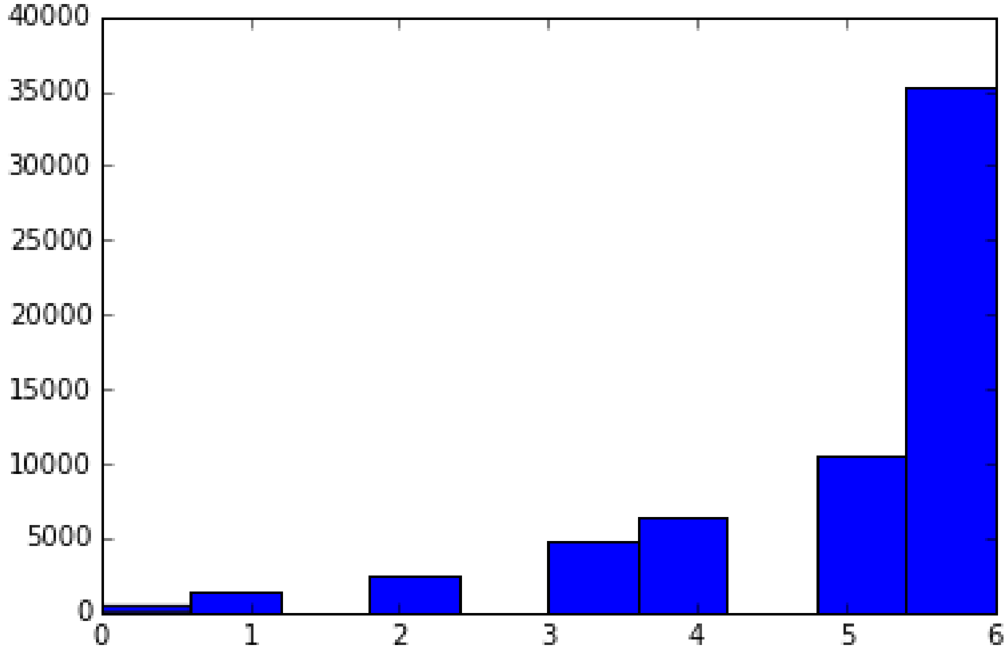
2.0.14 Set B and Sample from Potts

```
In [110]: G.graph['B'] = 1.0
```

```
In [113]: energies = np.array([])
          for node in G.nodes():
              energies = np.append(energies,prior_energy(G,node,1))
```

```
In [114]: plt.hist(energies)
```

```
Out[114]: (array([   435.,   1331.,      0.,   2469.,      0.,   4771.,   6456.,
                     0.,  10620.,  35315.]),
           array([ 0. ,  0.6,  1.2,  1.8,  2.4,  3. ,  3.6,  4.2,  4.8,  5.4,  6. ]),
           <a list of 10 Patch objects>)
```

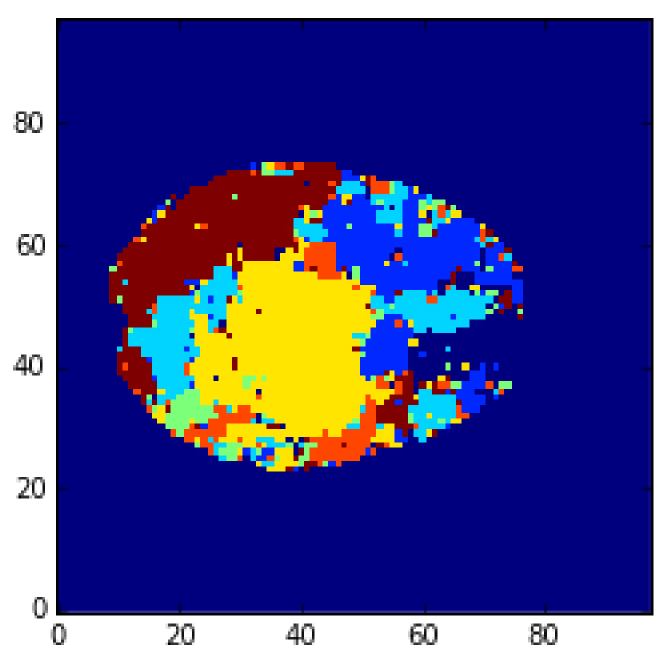



```
In [164]: %time G,posterior_samples = gibbs_sample(G,num_gibbs_samples=2,prior_or_posterior='posterior')
```

```
CPU times: user 13.4 s, sys: 14 ms, total: 13.4 s
Wall time: 13.5 s
```

```
In [158]: plot_graph(G,20)
```

```
Out[158]: <matplotlib.image.AxesImage at 0x7fa3bb5a1c90>
```



```
In [127]: nx.write_gpickle(G, "G.gpickle")
```

```
In [141]: (20*4*5)/60
```

```
Out[141]: 6
```

2.1 Use K-means for u-est

```
In [145]: cluster_time_series.shape
```

```
Out[145]: (1086, 7)
```

```
In [153]: G.graph['u_est'] = cluster_time_series
          G.graph['u_est'].shape
```

```
Out[153]: (1086, 7)
```

```
In [99]: G.graph['sigma_est'] = 1.0
```

```
In [ ]: (30*5*4)/60
```

2.1.1 EM

```
In [171]: %time G,posterior_samples = gibbs_sample(G,1,prior_or_posterior='posterior')
```

```
CPU times: user 18.1 s, sys: 4 ms, total: 18.1 s
Wall time: 18.1 s
```

```
In [172]: for em_step in range(3):
              print('em: '+str(em_step))


              u_est_new =G.graph['u_est']
              u_est_new_count = G.graph['u_est']*0.0+1

              for gs in range(5):
                  print('gibbs: '+str(gs))
                  G,posterior_samples = gibbs_sample(G,1,prior_or_posterior='posterior')

                  for n,node in enumerate(G.node):
                      state = posterior_samples[n] # node was in this state
                      u_est_new[:,int(state)] += G.node[node]['x'] # add the x data to that time vector
                      u_est_new_count[:,int(state)] += np.ones(G.graph['tp'])

              u_est_new = u_est_new/u_est_new_count
              G.graph['u_est'] = u_est_new
```

```
em: 0
gibbs: 0
gibbs: 1
gibbs: 2
gibbs: 3
gibbs: 4
em: 1
```

```
gibbs: 0
gibbs: 1
gibbs: 2
gibbs: 3
gibbs: 4
em: 2
gibbs: 0
gibbs: 1
gibbs: 2
gibbs: 3
gibbs: 4
```

2.1.2 Save Brain

```
In [161]: states = np.zeros(np.shape(labels))
          for node in G.nodes():
              states[index_image_data[node[0],node[1],node[2]]] = G.node[node]['state']
```

```
In [162]: states_image = nifti_masker.inverse_transform(states)
```

```
In [163]: states_image.to_filename('MRFclust'+sub+'.nii')
```

Appendix III: Multi-Subject Hierarchical HMM

December 4, 2015

0.0.1 Multi-Subject Potts Model

The states of the system z, are assumed be distributed according to a Gibbs distribution.

$$p(Z) = 1/Pexp(-U(Z))$$

$$p(Z) = 1/Pexp(-U(Z))$$

There are nodes in group and nodes in each subjects map.
How to sample from the Potts model.

$$p(y_s|y_{-s}, x) = p(y)/p(x|y_s)p(y_{-s})$$

Initialize Random States for All Nodes in both group and subject 1. sample from group using Gibbs Sampling -

$$p(y_s|y_{-s}, x) = 1/Pexp(-U(Z))$$

- where U(Z) in this case is

$$U(Z) = \alpha \sum_j \delta(y_j, y'_j) + \beta \sum_n \delta(y_s, y_n) +$$

2. sample from each subject using Gibbs Sampling -

$$U(Z) = \alpha\delta(y_j, y'_j) + \beta \sum_n \delta(y_s, y_n) - kux - logCp$$

- Only use the liklihood components

In [192]:
```
import numpy as np
import scipy.stats
from scipy import stats
import matplotlib
print('imported')
```

```
imported
```

In [193]:
```
matplotlib.use("Agg")
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.animation as manimation
import matplotlib.pyplot as plt
%matplotlib inline
import networkx as nx
print('imported')
```

```
imported
```

```
/System/Library/Frameworks/Python.framework/Versions/2.7/Extras/lib/python/matplotlib/__init__.py:921: Us
because the the backend has already been chosen;
matplotlib.use() must be called *before* pylab, matplotlib.pyplot,
or matplotlib.backends is imported for the first time.

  if warn: warnings.warn(_use_error_msg)
```

In [194]:
```
#FFMpegWriter  = manimation.writers['ffmpeg']
```

In [195]:
```
matplotlib.__version__
```

Out[195]:
```
'1.1.1'
```

In [196]:
```
matplotlib.__file__
```

Out[196]:
```
'/System/Library/Frameworks/Python.framework/Versions/2.7/Extras/lib/python/matplotlib/__init_
```

In [197]:
```
# subject Graphs
# group Graph

# initialize all to random state

num_states = 5
num_subs = 3 # CONFUSING INCLUDES GROUP
nn = 6
B = 1
A = .5
    ### Initialize Empty Graph (Random States)

Group=nx.grid_2d_graph(nn,nn)  #4x4 grid
for n in Group.node:
    state=np.random.randint(num_states)
    Group.node[n]['state']=state

Group.graph['B'] = B
Group.graph['A'] = A
Group.graph['num_states']=num_states
Group.graph['nn']=nn
Group.graph['sub_num']='group'
Group.graph['num_subs']=num_subs

Graphs = []
Graphs.append(Group)

for sub_num in range(num_subs-1):
    Sub=nx.grid_2d_graph(nn,nn)  #4x4 grid
    for n in Sub.node:
        state=np.random.randint(num_states)
        Sub.node[n]['state']=state

    Sub.graph['B'] = B
    Sub.graph['A'] = A
    Sub.graph['sub_num']= sub_num
    Sub.graph['num_states']=num_states
    Sub.graph['nn']=nn
```

```
            Graphs.append(Sub.copy())
```

In [198]:
```
Graphs
```

Out[198]:
```
[<networkx.classes.graph.Graph at 0x111ee9a90>,
 <networkx.classes.graph.Graph at 0x10a4b5190>,
 <networkx.classes.graph.Graph at 0x1084caad0>]
```

In [199]:
```
Graphs[0].graph
```

Out[199]:
```
{'A': 0.5,
 'B': 1,
 'name': 'grid_2d_graph',
 'nn': 6,
 'num_states': 5,
 'num_subs': 3,
 'sub_num': 'group'}
```

In [200]:
```
Graphs[1].graph
```

Out[200]:
```
{'A': 0.5,
 'B': 1,
 'name': 'grid_2d_graph',
 'nn': 6,
 'num_states': 5,
 'sub_num': 0}
```

In [201]:
```
Graphs[0].edge[(2,2)]
```

Out[201]:
```
{(1, 2): {}, (2, 1): {}, (2, 3): {}, (3, 2): {}}
```

In [202]:
```
len(Graphs)
```

Out[202]:
```
3
```

In [203]:
```
def to_str(state):
    if state==0:
        return('b')
    elif state ==1:
        return('r')
    elif state ==2:
        return('g')
    elif state ==3:
        return('y')
    elif state ==4:
        return('w')
```

Group Energy Function

$$U(Z) = \alpha \sum_j \delta(y_j, y_j') + \beta \sum_n \delta(y_s, y_n) +$$

In [204]:
```
np.arange(Graphs[0].graph['num_subs'])+1
```

Out[204]:
```
array([1, 2, 3])
```

In [205]:
```
def Ug(Graphs,node,norm=False,z=None):

    Group = Graphs[0]

    # if calculating the numerator, use the actual state of the node
    if not norm:
        z = Group.node[node]['state']

    # else use one that is put in

    # sum up alpha priors. (based on same nodes in each subject graph
    same_voxel_matches = 0
    for sub in np.arange(1,Group.graph['num_subs']):
        Sub = Graphs[sub]
        if z!=Sub.node[node]['state']:
            same_voxel_matches+=1
    same_voxel_matches = same_voxel_matches*Group.graph['A']

    # sum up beta priors
    neighbor_voxel_matches = 0
    for edge in Group.edge[node]:
        if z!=Group.node[edge]['state']:
            neighbor_voxel_matches+=1

    neighbor_voxel_matches = neighbor_voxel_matches* Group.graph['B']

    U = neighbor_voxel_matches+same_voxel_matches

    return(U)
```

Subject Energy Function

$$U(Z) = \alpha\delta(y_j, y_j') + \beta\sum_n \delta(y_s, y_n) - kux - logCp$$

In [206]:
```
def Us(Graphs,node,sub,norm=False,z=None,generative=True):

    Group = Graphs[0]
    Sub = Graphs[sub]


    # if calculating the numerator, use the actual state of the node
    if not norm:
        z = Sub.node[node]['state']



    # alpha prior
    if z!=Group.node[node]['state']:
        same_voxel_matches=1
    else:
        same_voxel_matches=0

    # beta prior
    neighbor_voxel_matches = 0
```

```
            for edge in Sub.edge[node]:

                if z!=Sub.node[edge]['state']:
                    neighbor_voxel_matches+=1

            neighbor_voxel_matches = neighbor_voxel_matches* Graphs[0].graph['B']

            U = neighbor_voxel_matches+same_voxel_matches

            # liklihood component
            if generative==False:
                U = U + 1 ### INCOMPLETE

            return(U)
```

```
In [207]: Sub = Graphs[1]
          node = (1,3)
          for edge in Sub.edge[node]:
              print(edge)
```

```
(1, 2)
(0, 3)
(2, 3)
(1, 4)
```

Conditional Probability of Hidden States

$$p(y_s|y_{-s},x) = \frac{1}{Z}exp(-U(y))$$

- calculating normalization factor.
- Sum over exp(-U(y)) for each possible

$$y \in 1, 2, 3, 4..l$$

```
In [208]: def conditional_prob_hidden(Graphs,node,sub,generative=True,norm=False,z=None):

              num_states = Graphs[0].graph['num_states']

              # if you do norm, Ug and Us expects a z value (state value) for the numerator
              # otherwise it grabs the state of the node

              if sub == 0: # group graph
                  numerator = np.exp(-1.0*Ug(Graphs,node,norm=norm,z=z))
                  denominator = 0
                  for state in range(num_states):
                      denominator+= np.exp(-1.0*Ug(Graphs,node,norm=True,z=state))
                  prob = numerator/denominator

              else: # subject graph
                  numerator = np.exp(-1.0*Us(Graphs,node,sub,norm=norm,z=z,generative=generative))
                  denominator = 0
                  for state in range(num_states):
                      denominator+= np.exp(-1.0*Us(Graphs,node,sub,norm=True,z=state,generative=generati
```

```
            prob = numerator/denominator

        return(prob)
```

0.0.2 Calculate Graph Energy

$$\sum Ug(z) + \sum Us(z)$$

```
In [209]: def calc_energy(Graphs):
              energy = 0
              for sub in range(Graphs[0].graph['num_subs']):
                  if sub==0:
                      for node in Graphs[sub].nodes():
                          energy+=Ug(Graphs,node,norm=False)
                  else:
                      for node in Graphs[0].nodes():
                          energy+=Us(Graphs,node,sub,norm=False,z=None,generative=True)


              return(energy)
```

```
In [210]: calc_energy(Graphs)
```

```
Out[210]: 385.0
```

```
In [211]: ### Plotting The Graphs

          def plot_graphs(Graphs,true_also=True,plot_conditional=False):

              fig, axes = plt.subplots(nrows=2, ncols=len(Graphs),figsize=(16,6))
              for g,G in enumerate(Graphs):
                  plt.sca(axes[0,g])
                  if g ==0:
                      title='group'
                  else:
                      title='subject'
                  axes[0,g].set_title(title)
                  pos = dict( (n, n) for n in G.nodes() )
                  labels = dict( ((i, j), i * 10 + j) for i, j in G.nodes() )
                  if plot_conditional:
                      #labels = dict( ((i, j), i * 10 + j) for i, j in G.nodes() )
                      labels = dict( ((i,j), np.round(conditional_prob_hidden(Graphs,(i,j),g,generative

                  colors =  [to_str(G.node[i,j]['state']) for i, j in G.nodes() ]
                  nx.draw_networkx(G, pos=pos, labels=labels,node_color=colors)
```

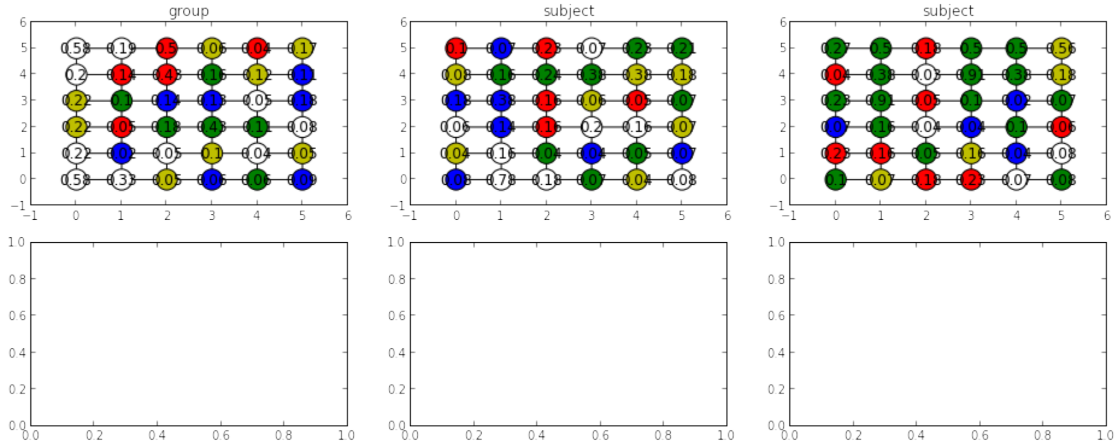
```
In [212]: conditional_prob_hidden(Graphs,(0,1),1,generative=True)
```

```
Out[212]: 0.038753953809111424
```

```
In [213]: conditional_prob_hidden(Graphs,(0,1),1,generative=True,norm=True,z=0)
```

```
Out[213]: 0.10534416842024878
```

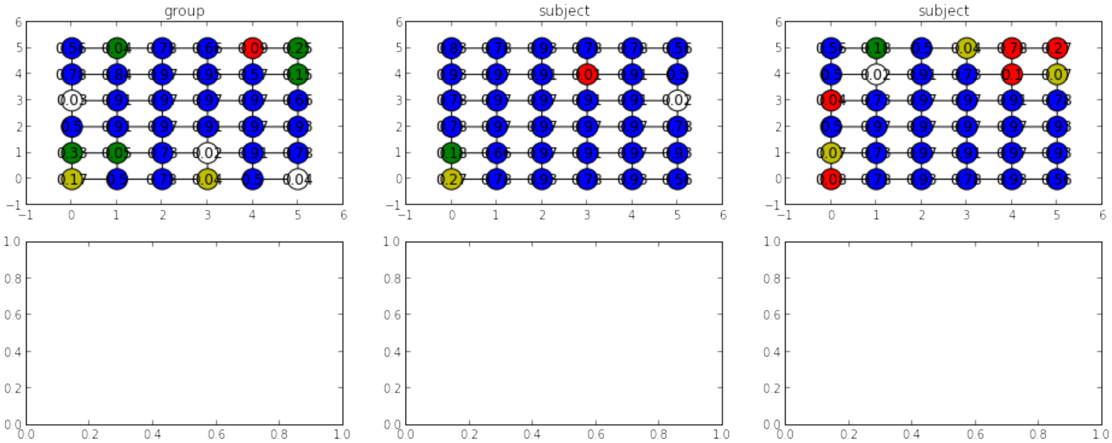
```
In [214]: plot_graphs(Graphs,plot_conditional=True)
```



```
In [215]: ### Gibbs Sample from Potts ###

          num_gibbs_samples=200

          num_states = Graphs[0].graph['num_states']
          num_subs = Graphs[0].graph['num_subs']

          # stores
          energies = np.array([])

          # sample loop
          for samples in range(num_gibbs_samples):

              ### Group Graph ####
              # start with group graph
              Group = Graphs[0]

              # loop through each node,
              for node in Group.nodes():
                  conditional_probs = np.zeros(num_states)
                  # calculate conditional probabilty for each possible state (given current states of a
                  for possible_state in range(num_states):
                      conditional_probs[possible_state] = conditional_prob_hidden(Graphs,node,0,generat

                  # draw new state based on conditional distribution.
                  Group.node[node]['state']=np.where(np.random.multinomial(1,conditional_probs))[0][0]

              ### Subject Graphs ####
              # loop through each subject graph
              for sub in np.arange(1,num_subs):
                  for node in Graphs[sub].nodes():
                      conditional_probs = np.zeros(num_states)
                      for possible_state in range(num_states):
                          conditional_probs[possible_state] = conditional_prob_hidden(Graphs,node,sub,g
                      Graphs[sub].node[node]['state']=np.where(np.random.multinomial(1,conditional_prob
```

```
    ### Calcualte Total Energy to Keep Track of Convergence ###
    energies = np.append(energies,calc_energy(Graphs))
```

In [216]:
```
plot_graphs(Graphs,plot_conditional=True)
```

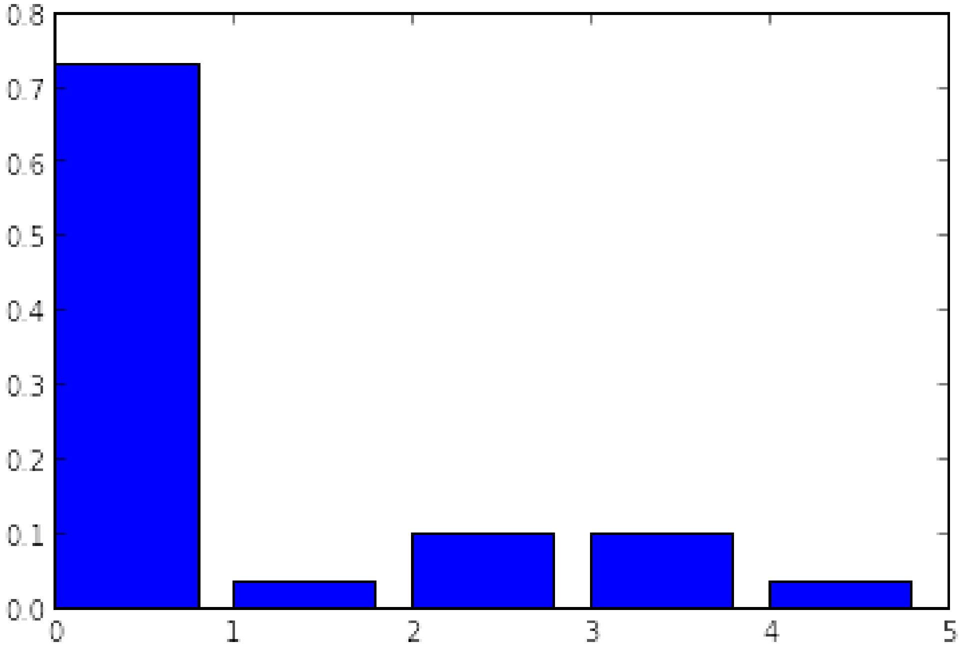


In [217]:
```
conditional_prob_hidden(Graphs,(1,1),1,generative=True,norm=True,z=0)
```

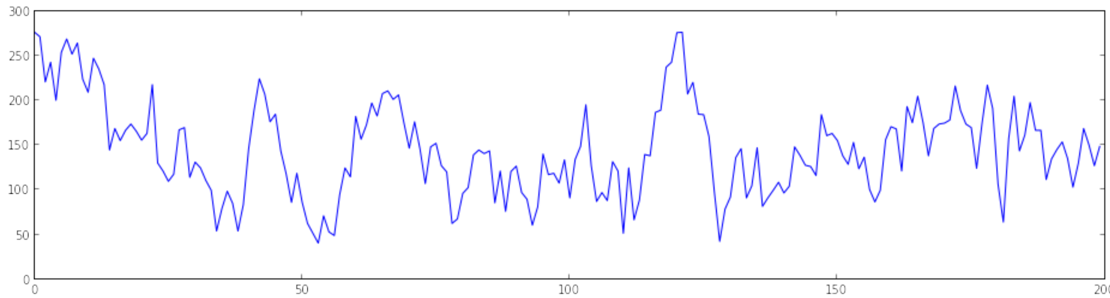
Out[217]: 0.65909122752220761

In [218]:
```
node = (1,1)
conditional_probs = np.zeros(num_states)
for possible_state in range(num_states):
    conditional_probs[possible_state] = conditional_prob_hidden(Graphs,node,sub,generative=Tr
plt.bar(np.arange(num_states),conditional_probs)
```

Out[218]: <Container object of 5 artists>



```
In [219]: plt.figure(figsize=(16,4))
          plt.plot(energies)

Out[219]: [<matplotlib.lines.Line2D at 0x104684950>]
```




```
In [220]: manimation.

          File "<ipython-input-220-8c5bdff29611>", line 1
        manimation.
                  ^
    SyntaxError: invalid syntax
```

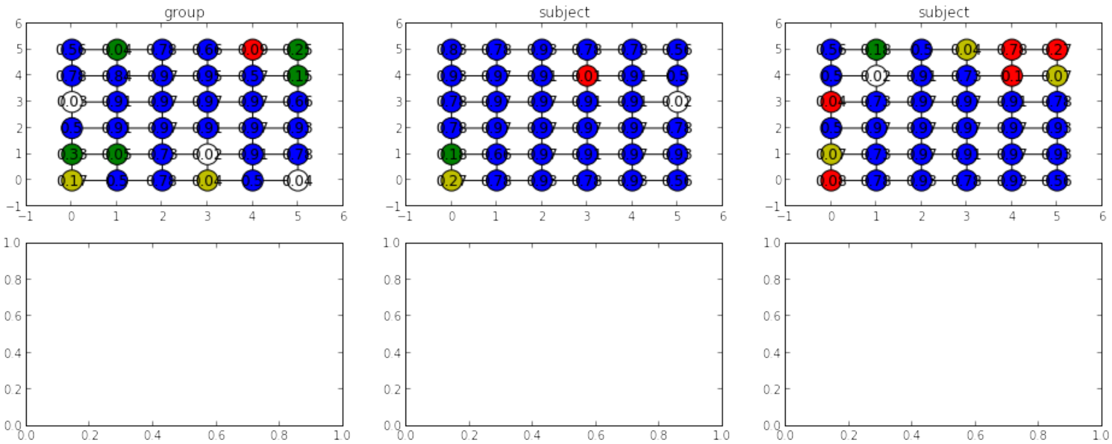
```
In [ ]: FFMpegWriter  = manimation.writers['ffmpeg']
        metadata = dict(title='test',artist='Matplotlib',comment='asdf')
        writer = FFMpegWriter(fps=15,metadata=metadata)
        fig = plt.figure()
        l, = plt.plot([],[],'k-o')
        x0, y0 =0,0
        with writer.saving(fig,"writer_test.mp4",100):
            for i in range(100):
                x0+= 0.1 * np.random.randn()
                y0+= 0.1 * np.random.randn()
                l.set_data(x0,y0)
                writer.grab_frame()
```

```
In [ ]: G = Graphs[0]
        g = 0
        fig, axes = plt.subplots(nrows=1, ncols=1,figsize=(16,6))

        for i in range(10):
            pos = dict( (n, n) for n in G.nodes() )
            labels = dict( ((i,j), np.round(conditional_prob_hidden(Graphs,(i,j),g,generative=True),2))

            colors =  [to_str(G.node[i,j]['state']) for i, j in G.nodes() ]
            nx.draw_networkx(G, pos=pos, labels=labels,node_color=colors)
```

0.0.3 Recover the Graphs?

```
In [ ]: np.argmax(conditional_probs)
```

```
In [221]: plot_graphs(Graphs,plot_conditional=True)
```




```
In [224]: ### iterative conditional modes

          ### start on last sample from MCMC
          ### start at voxel..
          ### calcualte energies for each possible states.
          ### choose max ..

          num_gibbs_samples=20

          num_states = Graphs[0].graph['num_states']
          num_subs = Graphs[0].graph['num_subs']

          # stores
          energies = np.array([])

          # sample loop
          for samples in range(num_gibbs_samples):

              ### Group Graph ####
              # start with group graph
              Group = Graphs[0]

              # loop through each node,
              for node in Group.nodes():
                  conditional_probs = np.zeros(num_states)
                  # calculate conditional probabilty for each possible state (given current states of a
                  for possible_state in range(num_states):
                      conditional_probs[possible_state] = conditional_prob_hidden(Graphs,node,0,generat

                  # draw new state based on conditional distribution.
                  Group.node[node]['state_k+1']=np.argmax(conditional_probs)
```

```
### Subject Graphs ####
# loop through each subject graph
for sub in np.arange(1,num_subs):
    for node in Graphs[sub].nodes():
        conditional_probs = np.zeros(num_states)
        for possible_state in range(num_states):
            conditional_probs[possible_state] = conditional_prob_hidden(Graphs,node,sub,g
        Graphs[sub].node[node]['state_k+1']=np.argmax(conditional_probs)




### Now actually go through and change it?
for node in Group.nodes():
    Group.node[node]['state']=Group.node[node]['state_k+1']
for sub in np.arange(1,num_subs):
    for node in Graphs[sub].nodes():
        Graphs[sub].node[node]['state']=Graphs[sub].node[node]['state_k+1']

### Calcualte Total Energy to Keep Track of Convergence ###
energies = np.append(energies,calc_energy(Graphs))
```

```
In [225]: plot_graphs(Graphs,plot_conditional=True)
```