toy_model

December 17, 2014

```
In [117]: import numpy as np
          import matplotlib.pylab as pl
          from collections import Counter
```

```
In [62]: from scipy.linalg import block_diag
```

0.1 Consider a model has :

1. three agents a_1, a_2, a_3
2. one topic J_1
3. This topic has two initiatives (alternatives) $j_{1,1}$, $j_{1,2}$

0.2 Assumptions on input model

```
Assumption 1: initiatives and topics are all exclusive
Assumption 2: for actor i on one topic k the magnitude of preference vector of all initiatives under tha
Assumption 3: the preference on certain initiative is normal distributed ~ N(0,1) over all actors in the
```

now it is possible to surrogate preference values for our toy model

```
In [4]: # draw 6 random number for standard normal distribution
        import random
        preferences = [random.normalvariate(0,1) for i in range(6)]
        print preferences
```

```
[0.47708384379253094, 1.758049795259444, -1.3950766906120262, -2.417002575409951, 0.5711571712575537, -0
```

```
In [45]: # pick 2 from 6 without replacement assign to each one actor (each actor needs two number for
         a1 = np.array(preferences[:2])
         a2 = np.array(preferences[2:4])
         a3 = np.array(preferences[4:6])
         print a1, a2, a3
```

```
[ 0.47708384  1.7580498 ] [-1.39507669 -2.41700258] [ 0.57115717 -0.56681119]
```

```
In [48]: # normalize preference within actor, so that they sum up to one and all of them are positive
         def normalize_preference(input_list):
             """
             :param input_list: 1d numpy array
             :return output_list: 1d numpy array
             """
             magnitude = np.sqrt(sum(input_list**2))
             if magnitude == 0:
                 magnitude = 0.0001
             else:
                 output_list = input_list/magnitude
             return output_list
```

```
In [49]: a1 = normalize_preference(a1)
         a2 = normalize_preference(a2)
         a3 = normalize_preference(a3)
         print a1, a2, a3

[ 0.26189894  0.9650953 ] [-0.49989777 -0.86608442] [ 0.70980209 -0.70440115]
```

0.3 Assumption on Networks:

```
1. choose weight of node as #delegetions, everyone delegates only itself, and that invariant with time
2. choose weight of edge as distance of actor belief which can be purely deduced from beliefs
```

0.4 Assumption on Events:

```
1. preference vector of actor i towards various initiatives on a cerain topic rotate by a small amount 
2. delta is drawn from uniform distribution [-pi/100, pi/100]
3. ballot happens: each actor votes for the initiatives at the top of its preference list, and the init
4. the result of ballot doesn't have a feedback affect on actor beliefs
5 choose VAR lag =1, u=0, and rotation is encoded in the matrix D(t)
6 D(t) is time-invariant which can simply denote as D, but stochastic due to its including of delta
7 Belief of actors evolves independantly, so the matrix D is diagonal
```

0.5 System State:

The system state of the model (dynamic graph) above should be a vector of 9 positions

$(b(a_1), b(a_2), b(a_3), nodeweight(a_1), nodeweight(a_2), nodeweight(a_3), edgeweight(a_2a_3), edgeweight(a_2a_1), edgeweight(a_3$

because node_weight is just #delegation which do not variate with time, so need not be take as state and edge_weight can be computed directly from $b(a_1), b(a_2), b(a_3)$, they also can be dismissed

so remaining $b(a_1), b(a_2), b(a_3)$ is our toy system state

implement system state using numpy array

```
In [41]: s = np.array([a1, a2, a3])

In [42]: s_initial = s
         s_initial

Out[42]: array([[ 0.21344757,  0.78655243],
                [ 1.        ,  0.        ],
                [ 1.        ,  0.        ]])
```

construct D dynamically each time step because its non-deterministic property

```
In [63]: # generate delta
         delta = random.uniform(-np.pi/100, np.pi/100)

         cos = np.cos(delta)
         sin = np.sin(delta)
         print cos, sin

         d = np.array([
             [cos,-sin],
             [sin,cos]
         ])
         print d

         D = block_diag(d,d,d)
```

```
        print delta,D.shape
```

```
0.999608949996 0.0279633168129
[[ 0.99960895 -0.02796332]
 [ 0.02796332  0.99960895]]
0.0279669624016 (6, 6)
```

wrap one step simulation

```
In [67]: def step(s):
             """
             :param s: state of system
             """
             delta = random.uniform(-np.pi/100, np.pi/100)
             cos = np.cos(delta)
             sin = np.sin(delta)

             d = np.array([
             [cos,-sin],
             [sin,cos]
             ])

             D = block_diag(d,d,d)
             return np.dot(D, s)
```

```
In [99]: def simul(initial_s, nsteps):
             history=[]
             history.append(initial_s)
             s = initial_s
             for i in range(nsteps):
                 s = step(s)
                 history.append(s)
             history = np.array(history)
             print 'successfully finished'
             return history, s
```

introduce ballot

```
In [135]: # stupid implementation
          def ballot(s):
              """
              :param s:
              """
              vote_a1= np.argmax(s[0:2])
              vote_a2= np.argmax(s[2:4])
              vote_a3= np.argmax(s[4:6])
              cnt = Counter()
              for vote in [vote_a1, vote_a2, vote_a3]:
                  cnt[vote]+=1
              #result of vote
              return cnt.most_common(1)[0][0]
```

integrate ballot into simulation

```
In [140]: def simul(initial_s, nsteps, periodofballot):
              history=[]
              ballot_history=Counter()
              history.append(initial_s)
              s = initial_s
              for i in range(nsteps):
                  s = step(s)
                  history.append(s)
                  if i%periodofballot:
                      # do ballot
                      res = ballot(s)
                      ballot_history[res]+=1

              history = np.array(history)
              print 'successfully finished'
              return history, s, ballot_history
```

```
In [123]: s_initial[0:2]
```

```
Out[123]: array([[ 0.21344757,  0.78655243],
                 [ 1.        ,  0.        ]])
```

```
In [114]: history, final_s = simul(s_initial.flatten(),100000)
          print final_s
```

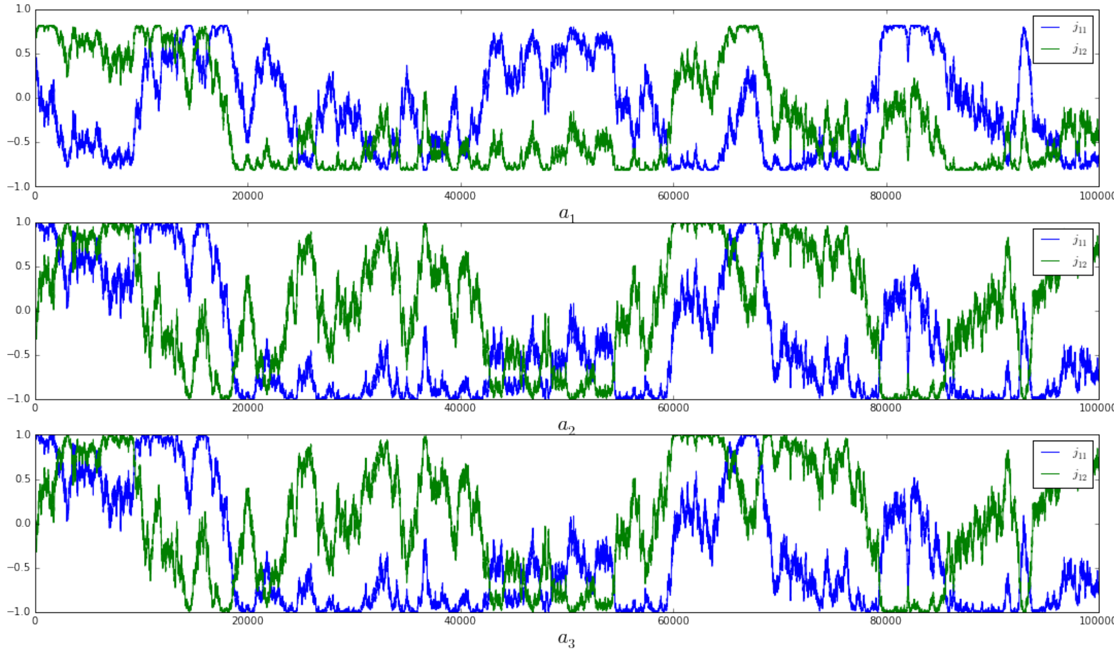
```
successfully finished
[-0.64138598 -0.50284055 -0.80155529  0.59792067 -0.80155529  0.59792067]
```

0.6 Visualization of simulation result

100000 steps

```
In [105]: %matplotlib inline
```

```
In [115]: history_plot = history.T
          fig, axes = pl.subplots(ncols=1, nrows=3)
          fig.set_size_inches(18.5,10.5)
          axes[0].plot(history_plot[0,:])
          axes[0].plot(history_plot[1,:])
          axes[0].legend(['$j_{11}$','$j_{12}$'])
          axes[0].set_xlabel('$a_{1}$',fontsize=20)
          axes[1].plot(history_plot[2,:])
          axes[1].plot(history_plot[3,:])
          axes[1].legend(['$j_{11}$','$j_{12}$'])
          axes[1].set_xlabel('$a_{2}$',fontsize=20)
          axes[2].plot(history_plot[4,:])
          axes[2].plot(history_plot[5,:])
          axes[2].legend(['$j_{11}$','$j_{12}$'])
          axes[2].set_xlabel('$a_{3}$',fontsize=20)
          pl.show()
```

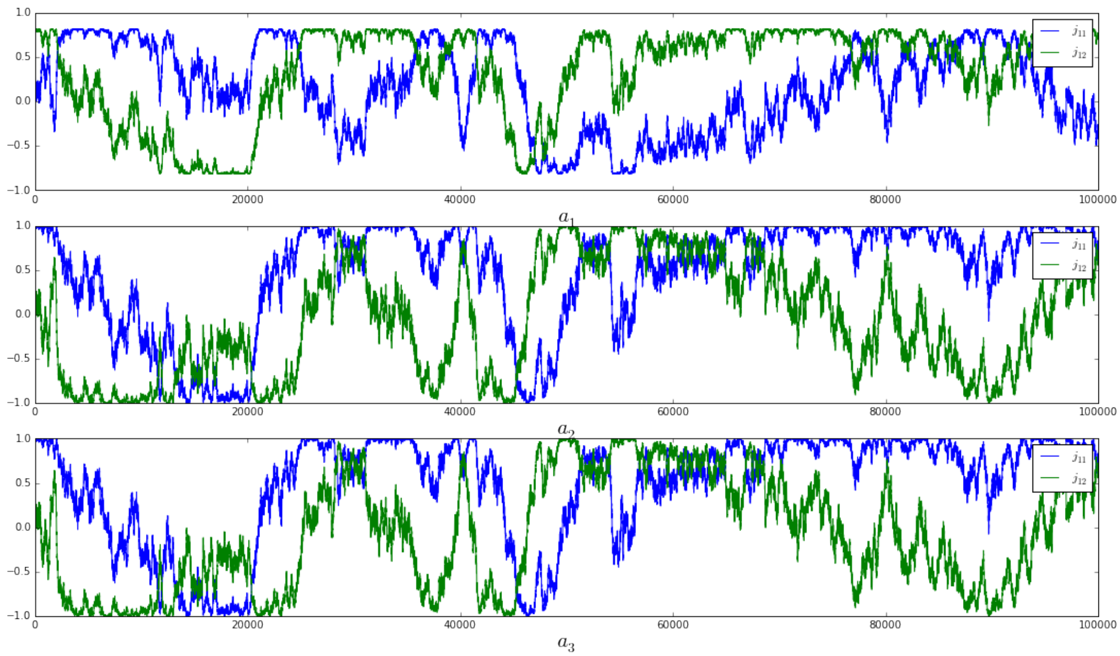


0.7 Ballot System

```
In [141]: history, final_s, ballot_result = simul(s_initial.flatten(),100000,365)
```

```
successfully finished
```

```
In [142]: history_plot = history.T
          fig, axes = pl.subplots(ncols=1, nrows=3)
          fig.set_size_inches(18.5,10.5)
          axes[0].plot(history_plot[0,:])
          axes[0].plot(history_plot[1,:])
          axes[0].legend(['$j_{11}$','$j_{12}$'])
          axes[0].set_xlabel('$a_{1}$',fontsize=20)
          axes[1].plot(history_plot[2,:])
          axes[1].plot(history_plot[3,:])
          axes[1].legend(['$j_{11}$','$j_{12}$'])
          axes[1].set_xlabel('$a_{2}$',fontsize=20)
          axes[2].plot(history_plot[4,:])
          axes[2].plot(history_plot[5,:])
          axes[2].legend(['$j_{11}$','$j_{12}$'])
          axes[2].set_xlabel('$a_{3}$',fontsize=20)
          pl.show()
```

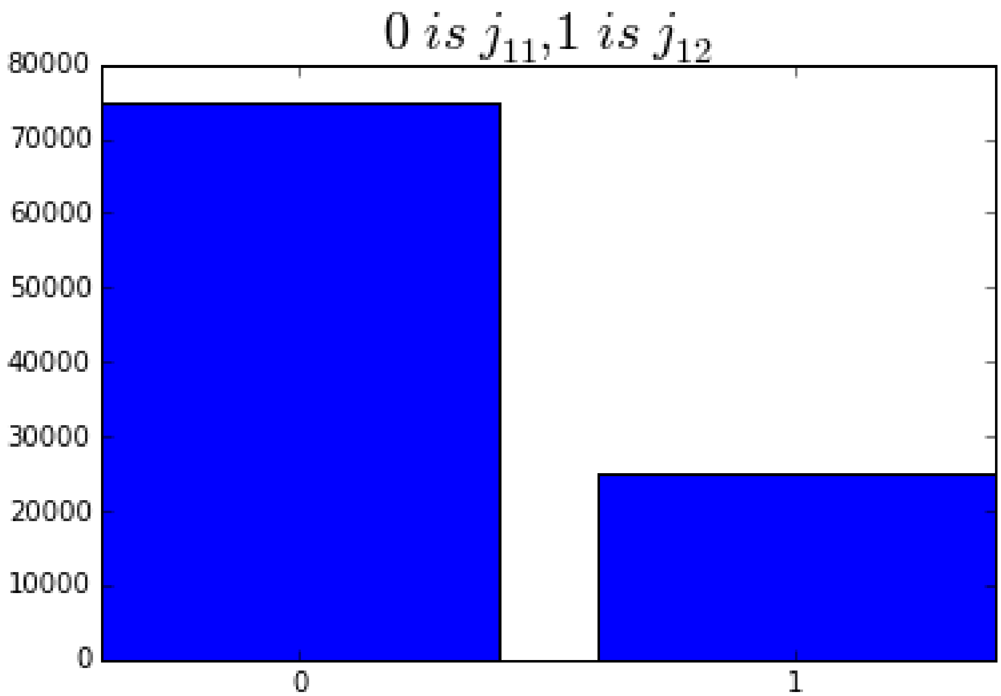


ballot result

```
In [143]: ballot_result

Out[143]: Counter({0: 74688, 1: 25038})

In [148]: pl.bar(range(len(ballot_result)), ballot_result.values(), align='center')
          pl.xticks(range(len(ballot_result)), ballot_result.keys())
          pl.title('$0\ is\ j_{11}, 1\ is\ j_{12}$',fontsize=20)
          pl.show()
```



Thinking:

By increasing number of actors to larger number (1000 , for example) and keep other things unchanged. I expect a 1:1 ratio on $j_{1,1}$ and $j_{1,2}$