In [10]:

```
import igraph
import random
import numpy as np
from matplotlib import pylab as plt
```

In [11]:

```
def make_edge_list(df_v1, df_v2, df_weight):
    '''Input must be list of node which has asset, list of nodes in which asset is
    assets. This function returns an edge list and a weight list'''
    edge_list = []
    weight_list = []
    if len(df_v1) != len(df_v2):
        print "warning"
    if len(df_v1) != len(df_weight):
        print "warning"

    for i in df_v1.index:
        c1 = df_v1[i]
        c2 = df_v2[i]
        edge_list.append([c1, c2])
        weight_list.append(df_weight[i])
    return edge_list, weight_list
```

In [12]:

```
def make_graph_from_edge(edge_list, weight_list):
    '''takes edge list and weight list and returns a weighted directed graph'''
    vertices = set()
    for line in edge_list:
        vertices.update(line)
    vertices = sorted(vertices)
    g = igraph.Graph(directed = True)

    g.add_vertices(vertices)
    g.add_edges(edge_list)
    g.vs["name"] = vertices
    g.es["weight"] = weight_list
    g.es["width"] = weight_list
    g.simplify(combine_edges={"weight": "sum"})
    return g
```

In [13]:

```
def SI_contagion_unweighted(G, lam = 1, chosen_one = -1, count_break = 1e6):
    '''Susceptible and Infected compartment model. Takes a graph and a contagion ra
    contagion mechanism. Does no take into account weights of nodes. Starts with ra
    specified otherwise.'''
    AM_nonweight = G.get_adjacency()
    n_countries = len(AM_nonweight[0])
    susceptible = [i for i in range(n_countries)] #initially all nodes are suscepti

    if chosen_one == -1:
        chosen_one = random.choice(susceptible)
    infected = [[chosen_one]]    #initialize infection
    susceptible.remove(chosen_one)

    count = 0
    while len(susceptible) > 0:
        infected.append([])
        for i in range(n_countries):
            n_infected_neigh = 0
            if i in infected[count]: #if node is infected it will keep being infect
                infected[count + 1].append(i)
                pass
            else: #if it is not infected the probability of becoming infected depen
                in_neigh_list = G.vs[i].neighbors(mode = "IN")
                for j in range(len(in_neigh_list)):
                    graph_index = in_neigh_list[j].index

                    if graph_index in infected[count]:
                        n_infected_neigh += 1

                if random.random() < 1 - (1 - lam)**n_infected_neigh:
                    infected[count + 1].append(i)
                    susceptible.remove(i)

        count += 1

        print count
        if count > count_break:
            print "warning"
            break

    return infected
```

In [6]:

```
def SI_contagion_weighted(G, lam = 1, chosen_one = -1, count_break = 1e6):
    '''Susceptible and Infected compartment model. Takes a graph and a contagion ra
    contagion mechanism but instead of using the number of infected neighbors uses
    degree which comes from infected neighbors. Starts with random node, unless spe
    AM_weight = G.get_adjacency(attribute="weight")
    in_strength_weight = G.strength(weights=G.es["weight"], mode = "IN") #remember
    n_countries = len(AM_weight[0])
    susceptible = [i for i in range(n_countries)]

    if chosen_one == -1:
        chosen_one = random.choice(susceptible)
    infected = [[chosen_one]]
    susceptible.remove(chosen_one)

    count = 0
    while len(susceptible) > 0:
        infected.append([])
        for i in range(n_countries):
            weights_infected_neigh = 0
            if i in infected[count]: #if node is infected it will keep being infect
                infected[count + 1].append(i)
                pass
            else: #if it is not infected the probability of becoming infected depen
                in_neigh_list = G.vs[i].neighbors(mode = "IN")
                for j in range(len(in_neigh_list)):
                    graph_index = in_neigh_list[j].index

                    if graph_index in infected[count]:
                        weights_infected_neigh +=AM_weight[graph_index][i]

                frac_weight_inf_neigh = float(weights_infected_neigh)/in_strength_w


                if random.random() < 1 - (1 - lam)**frac_weight_inf_neigh:
                    infected[count + 1].append(i)
                    susceptible.remove(i)

        count += 1


        if count > count_break:
            print "warning"
            break

    return infected
```

In [7]:

```
def LTM_contagion_weighted(G, lam = 1 ,chosen_one = -1, count_break = 1e6):
    '''Linear Threshold Model for weighted network. The probability of being infect
    weighted degree which comes from infected neighbors. Starts with random node, u
    damp = lam #damping or expanding factor of probability of contagion.
    AM_weight = G.get_adjacency(attribute="weight")
    in_strength_weight = G.strength(weights=G.es["weight"], mode = "IN")
    n_countries = len(AM_weight[0])
    susceptible = [i for i in range(n_countries)]

    if chosen_one == -1:
        chosen_one = random.choice(susceptible)
    infected = [[chosen_one]]
    susceptible.remove(chosen_one)

    count = 0
    while len(susceptible) > 0:
        infected.append([])
        for i in range(n_countries):
            weights_infected_neigh = 0
            if i in infected[count]: #if node is infected it will keep being infect
                infected[count + 1].append(i)
                pass
            else: #if it is not infected the probability of becoming infected depen
                in_neigh_list = G.vs[i].neighbors(mode = "IN")
                for j in range(len(in_neigh_list)):
                    graph_index = in_neigh_list[j].index

                    if graph_index in infected[count]:
                        weights_infected_neigh +=AM_weight[graph_index][i]
                #probabilty of infection if fraction of weights coming from neighbo
                frac_weight_inf_neigh = float(weights_infected_neigh)/in_strength_w


                if random.random() < damp*frac_weight_inf_neigh:
                    infected[count + 1].append(i)
                    susceptible.remove(i)

        count += 1


        if count > count_break:
            print "warning"
            break

    return infected
```

In [8]:

```
def SI_contagion_time(G, contagion_function = SI_contagion_weighted, lam = 1, chose
    '''Computes the average time and standard deviation of time steps it takes to r
    contagion_time_list = []

    for t in range(iterations):
        T = len(contagion_function(G, lam = lam, chosen_one = chosen_one))
        contagion_time_list.append(T)

    mean = np.mean(contagion_time_list)
    std = np.std(contagion_time_list)
    return mean, std
```

In [9]:

```
def LTM_contagion_time(G, contagion_function = LTM_contagion_weighted, lam = 1, cho
    '''Computes the average time and standard deviation of time steps it takes to r
    contagion_time_list = []

    for t in range(iterations):
        T = len(contagion_function(G, lam = lam, chosen_one = chosen_one))
        contagion_time_list.append(T)

    mean = np.mean(contagion_time_list)
    std = np.std(contagion_time_list)
    return mean, std
```

In [ ]:

```
def SI_multi_contagion_weighted(G_list, lam = 1, chosen_one = -1, count_break = 1e3
    '''Susceptible and Infected compartment model. Takes a list of graphs and a con
    contagion mechanism on the multiplex network but instead of using the number of
    the fraction of weighted degree which comes from infected neighbors. If a node
    is infected in all layers in the next step. Seeds node is random, unless specif
    #remember to check link direction, it may be other way around
    AM_weight = [G.get_adjacency(attribute="weight") for G in G_list]
    in_strength_weight = [G.strength(weights=G.es["weight"], mode = "IN") for G in

    n_countries = len(AM_weight[0][0]) #I am assuming all layers have same number o
    n_layers = len(G_list)


    susceptible = [i for i in range(n_countries)]
    if chosen_one == -1:
        chosen_one = random.choice(susceptible)

    susceptible.remove(chosen_one)
    infected = [[[chosen_one]] for i in range(n_layers)] #Assume node in contagion

    count = 0
    while len(susceptible) > 0:

        for l in range(n_layers):
            #print" layer =", l, "time = ", count, infected[l][count]
            #print"sus = ", count, susceptible[l]
            infected[l].append([])

            for i in range(n_countries):
                weights_infected_neigh = 0
                already_infected = False

                for inf in infected:
                    if i in inf[count]:#if node is infected in at least one layer i
                        already_infected = True
                        break

                if already_infected:
                    if i not in infected[l][count + 1]:
                        infected[l][count + 1].append(i)

                    try:
                        susceptible.remove(i)
                    except:
                        pass

                else: #if it is not infected the probability of becoming infected d
                    in_neigh_list = G_list[l].vs[i].neighbors(mode = "IN")
                    for j in range(len(in_neigh_list)):
                        graph_index = in_neigh_list[j].index

                        if graph_index in infected[l][count]:
                            weights_infected_neigh +=AM_weight[l][graph_index][i]

                    frac_weight_inf_neigh = float(weights_infected_neigh)/in_streng


                    if random.random() < 1 - (1 - lam)**frac_weight_inf_neigh and i
                        infected[l][count + 1].append(i)
```

```
        count += 1

        if count > count_break:
            print "warning"
            break

    return infected
```

In [ ]:

```
def LTM_multi_contagion_weighted(G_list, lam = 1, chosen_one = -1, count_break = 1e
    '''Susceptible and Infected compartment model. Takes a list of graphs and a con
    contagion mechanism on the multiplex network but instead of using the number of
    the fraction of weighted degree which comes from infected neighbors. If a node
    is infected in all layers in the next step. Seeds node is random, unless specif
    #remember to check link direction, it may be other way around
    AM_weight = [G.get_adjacency(attribute="weight") for G in G_list]
    in_strength_weight = [G.strength(weights=G.es["weight"], mode = "IN") for G in

    n_countries = len(AM_weight[0][0]) #I am assuming all layers have same number o
    n_layers = len(G_list)


    susceptible = [i for i in range(n_countries)]
    if chosen_one == -1:
        chosen_one = random.choice(susceptible)

    susceptible.remove(chosen_one)
    infected = [[[chosen_one]] for i in range(n_layers)] #Assume node in contagion

    count = 0
    while len(susceptible) > 0:

        for l in range(n_layers):

            infected[l].append([])

            for i in range(n_countries):
                weights_infected_neigh = 0
                already_infected = False

                for inf in infected:
                    if i in inf[count]:#if node is infected in at least one layer i
                        if i not in infected[l][count + 1]:
                            infected[l][count + 1].append(i)

                            try:
                                susceptible.remove(i)
                            except:
                                pass

                    else: #if it is not infected the probability of becoming infected d
                        in_neigh_list = G_list[l].vs[i].neighbors(mode = "IN")
                        for j in range(len(in_neigh_list)):
                            graph_index = in_neigh_list[j].index

                            if graph_index in infected[l][count]:
                                weights_infected_neigh +=AM_weight[l][graph_index][i]

                        frac_weight_inf_neigh = float(weights_infected_neigh)/in_streng

                        if random.random() < lam*frac_weight_inf_neigh and i not in inf
                            infected[l][count + 1].append(i)


        count += 1

        if count > count_break:
            print "warning"
            break
```

```
    return infected
```

In [ ]:

```
def multi_contagion_time(G, contagion_function = SI_multi_contagion_weighted, lam =
    '''Computes the average time and standard deviation of time steps it takes to r
    contagion_time_list = []

    for t in range(iterations):
        T = len(contagion_function(G, lam = lam, chosen_one = chosen_one)[0])
        contagion_time_list.append(T)


    mean = np.mean(contagion_time_list)
    std = np.std(contagion_time_list)
    return mean, std
```

In [ ]:

```
def make_several_multi_contagion_list(G_list, contagion_function= LTM_multi_contagi
    cont_time_mean_list = []
    cont_time_std_list = []
    for i in range(len(parameters)):
            cont_time_mean_list.append([])
            cont_time_std_list.append([])
            lam = parameters[i]
            for node in range(n_countries):
                m, s = multi_contagion_time(G_list, contagion_function = contagion_
                cont_time_mean_list[i].append(m)
                cont_time_std_list[i].append(s)
    return cont_time_mean_list, cont_time_std_list
```