Epidemics on Networks Documentation

Release 1.2rc1

Joel Miller, Istvan Kiss, Peter Simon

Jul 31, 2020

Contents

EoN (Epidemics on Networks) is a Python module that provides tools to study the spread of SIS and SIR diseases in networks.

Support EoN:

- The best way to support EoN is to cite EoN's publication
- The next best option is to let me know you're using it.
- Both of these will help my case when applying for grants & promotions and help me justify the time I spend on it.

MIT License: See `license.txt` for full details.

CHAPTER 1

Highlights

EoN is based on the book

Mathematics of Epidemics on Networks: from Exact to Approximate Models

EoN is built on top of NetworkX. Its repository is on github. EoN's tools fall into two broad categories:

- **Stochastic simulation of SIS and SIR disease**
  - Event-based simulation
    - much faster than traditional Gillespie simulation
    - allows weighted graphs
    - allows non-Markovian dynamics
  - Gillespie algorithms for Markovian dynamics
    - Through some careful optimization the unweighted SIS/SIR versions are comparable to the event-based simulation.
    - The weighted version is slower, but still reasonably fast.
    - There are methods for generic simple contagions and generic complex contagions.
  - discrete-time (synchronous update) models
  - tools for visualizing and animating simulated epidemics.
- **Numerical solvers for ODE models**
  - pair approximation models
  - effective degree models
  - edge-based compartmental models

CHAPTER 2

Table of Contents

2.1 Getting Started

2.1.1 Installation

You can **install EoN version 1.1** with pip

```
pip install EoN
```

If you have installed a previous version and want to reinstall with the most recent version available through pip (1.1). The easiest way is

```
pip install EoN --upgrade
```

If you are using Anaconda, go to the Anaconda command line and use *pip install EoN*

To **install EoN with a later (in development) version** You can clone or download the Github version at https://github.com/springer-math/Mathematics-of-Epidemics-on-Networks

Then just move into the main directory and run

```
python setup.py install
```

EoN requires `numpy`, `scipy`, and `matplotlib`. If you don't have them and you install through *pip*, these will automatically be added. Some of the visualization tools provide support for animations, but producing the animations will require installation of something like `ffmpeg`.

2.1.2 Current Version

The documentation provided here is for version 1.2rc1.

If you want to see changes from previous versions, please see Changes from v1.0.

2.1.3 Citing

If you use EoN, or publish anything based on it, please cite the Journal of Open Source Software publication

Also, please let me know so that I can use it for performance reviews and grant applications and generally because it makes me happy.

2.1.4 QuickStart Guide

The code here provides an example of creating a Barabasi-Albert network. Then it performs several simulations of an SIR epidemic starting with a fraction rho randomly infected initially. Finally it uses several analytic models to predict the spread of an epidemic in a random network with the given properties.

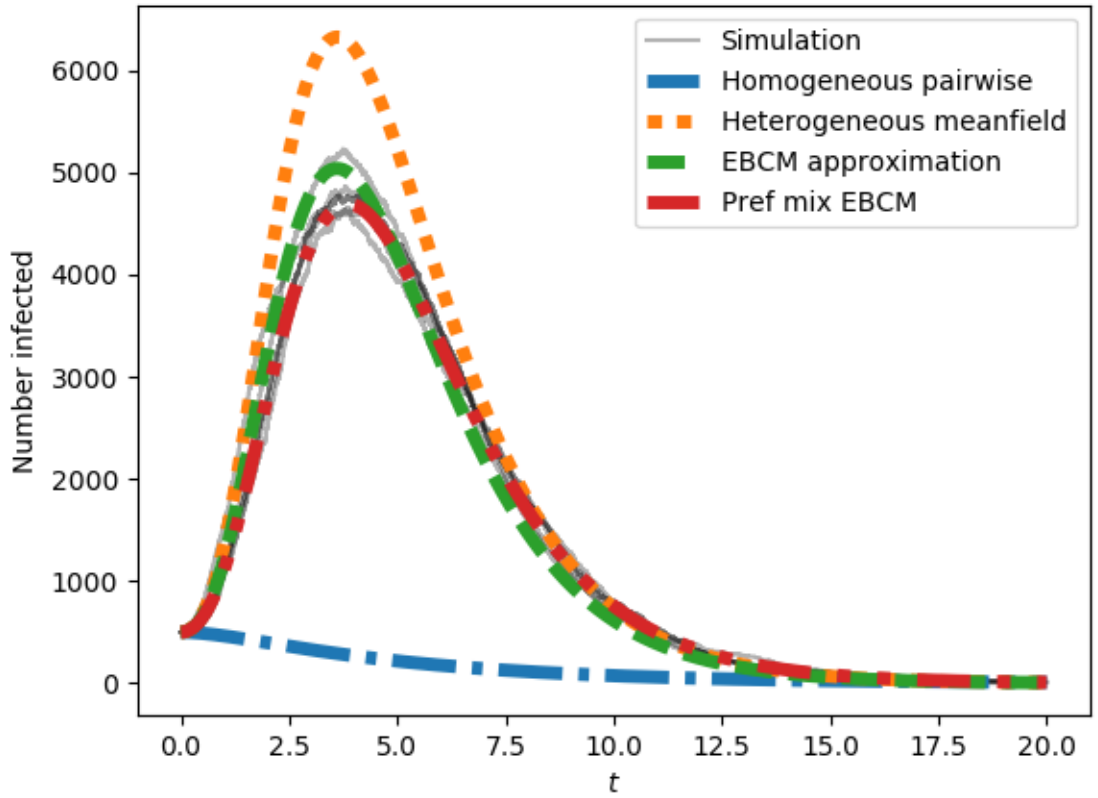
```
import networkx as nx
import matplotlib.pyplot as plt
import EoN

N=10**5
G=nx.barabasi_albert_graph(N, 5) #create a barabasi-albert graph

tmax = 20
iterations = 5  #run 5 simulations
tau = 0.1           #transmission rate
gamma = 1.0    #recovery rate
rho = 0.005      #random fraction initially infected

for counter in range(iterations): #run simulations
    t, S, I, R = EoN.fast_SIR(G, tau, gamma, rho=rho, tmax = tmax)
    if counter == 0:
        plt.plot(t, I, color = 'k', alpha=0.3, label='Simulation')
    plt.plot(t, I, color = 'k', alpha=0.3)

#Now compare with ODE predictions.  Read in the degree distribution of G
#and use rho to initialize the various model equations.
#There are versions of these functions that allow you to specify the
#initial conditions rather than starting from a graph.

#we expect a homogeneous model to perform poorly because the degree
#distribution is very heterogeneous
t, S, I, R = EoN.SIR_homogeneous_pairwise_from_graph(G, tau, gamma, rho=rho, tmax =␣
↪tmax)
plt.plot(t, I, '-.', label = 'Homogeneous pairwise', linewidth = 5)

#meanfield models will generally overestimate SIR growth because they
#treat partnerships as constantly changing.
t, S, I, R = EoN.SIR_heterogeneous_meanfield_from_graph(G, tau, gamma, rho=rho,␣
↪tmax=tmax)
plt.plot(t, I, ':', label = 'Heterogeneous meanfield', linewidth = 5)

#The EBCM model does not account for degree correlations or clustering
t, S, I, R = EoN.EBCM_from_graph(G, tau, gamma, rho=rho, tmax = tmax)
plt.plot(t, I, '--', label = 'EBCM approximation', linewidth = 5)

#the preferential mixing model captures degree correlations.
t, S, I, R = EoN.EBCM_pref_mix_from_graph(G, tau, gamma, rho=rho, tmax=tmax)
plt.plot(t, I, label = 'Pref mix EBCM', linewidth=5, dashes=[4, 2, 1, 2, 1, 2])
```

(continues on next page)

(continued from previous page)

```
plt.xlabel('$t$')
plt.ylabel('Number infected')

plt.legend()
plt.savefig('SIR_BA_model_vs_sim.png')
```

This produces



The preferential mixing version of the EBCM approach provides the best approximation to the (gray) simulated epidemics. We now move on to SIS epidemics:

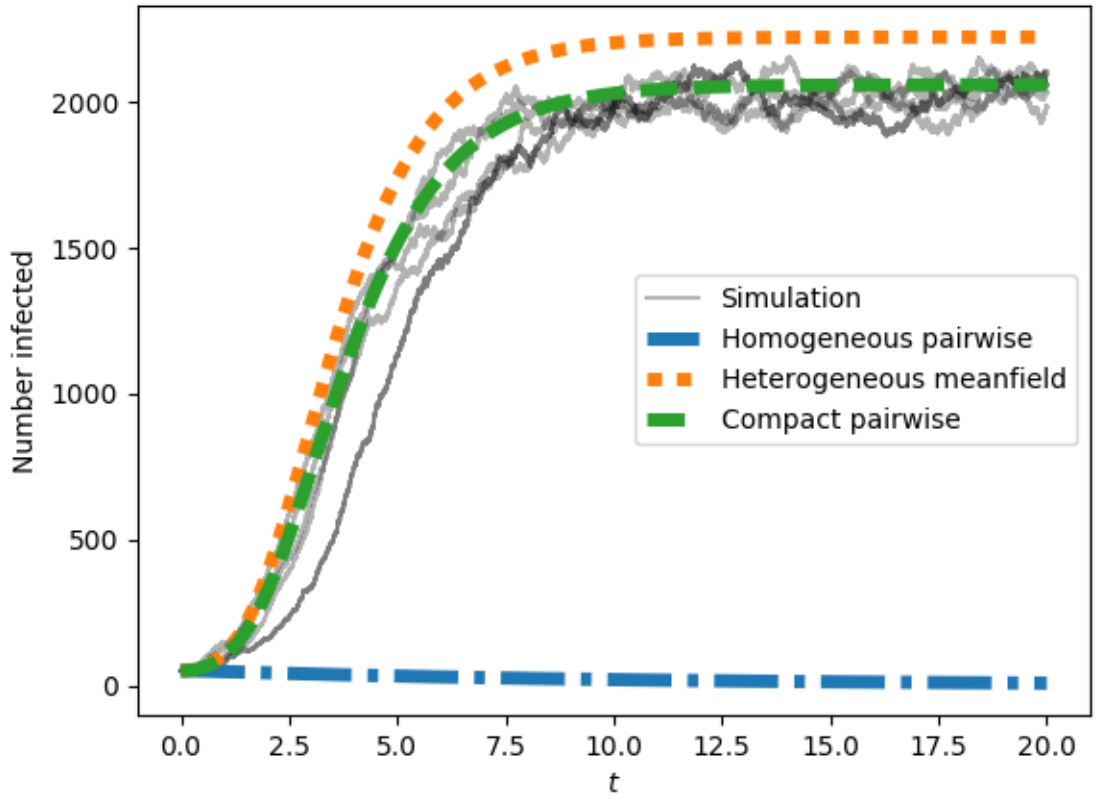
```
plt.clf()

#Now run for SIS.   Simulation is much slower so need smaller network
N=10**4
G=nx.barabasi_albert_graph(N, 5) #create a barabasi-albert graph
for counter in range(iterations):
    t, S, I = EoN.fast_SIS(G, tau, gamma, rho=rho, tmax = tmax)
    if counter == 0:
        plt.plot(t, I, color = 'k', alpha=0.3, label='Simulation')
    plt.plot(t, I, color = 'k', alpha=0.3)

#Now compare with ODE predictions.  Read in the degree distribution of G
#and use rho to initialize the various model equations.
```

(continues on next page)

(continued from previous page)

```
#There are versions of these functions that allow you to specify the
#initial conditions rather than starting from a graph.

#we expect a homogeneous model to perform poorly because the degree
#distribution is very heterogeneous
t, S, I = EoN.SIS_homogeneous_pairwise_from_graph(G, tau, gamma, rho=rho, tmax = tmax)
plt.plot(t, I, '-.', label = 'Homogeneous pairwise', linewidth = 5)

t, S, I = EoN.SIS_heterogeneous_meanfield_from_graph(G, tau, gamma, rho=rho,␣
↪tmax=tmax)
plt.plot(t, I, ':', label = 'Heterogeneous meanfield', linewidth = 5)

t, S, I = EoN.SIS_compact_pairwise_from_graph(G, tau, gamma, rho=rho, tmax=tmax)
plt.plot(t, I, '--', label = 'Compact pairwise', linewidth = 5)

plt.xlabel('$t$')
plt.ylabel('Number infected')
plt.legend()
plt.savefig('SIS_BA_model_vs_sim.png')
```

This produces



2.2 EoN Examples

We have collected a number of examples using **EoN** to generate figures. We start with examples from the book Mathematics of Epidemics on Networks: from Exact to Approximate Models. Then we give a few other examples. If none of these examples helps with the particular problem you are facing, Submit an issue or go to stackoverflow and use the 'eon' tag.

If you have an example you think would be useful here, please submit an issue or email me.

2.2.1 Reproducing figures from "Mathematics of Epidemics on Networks"

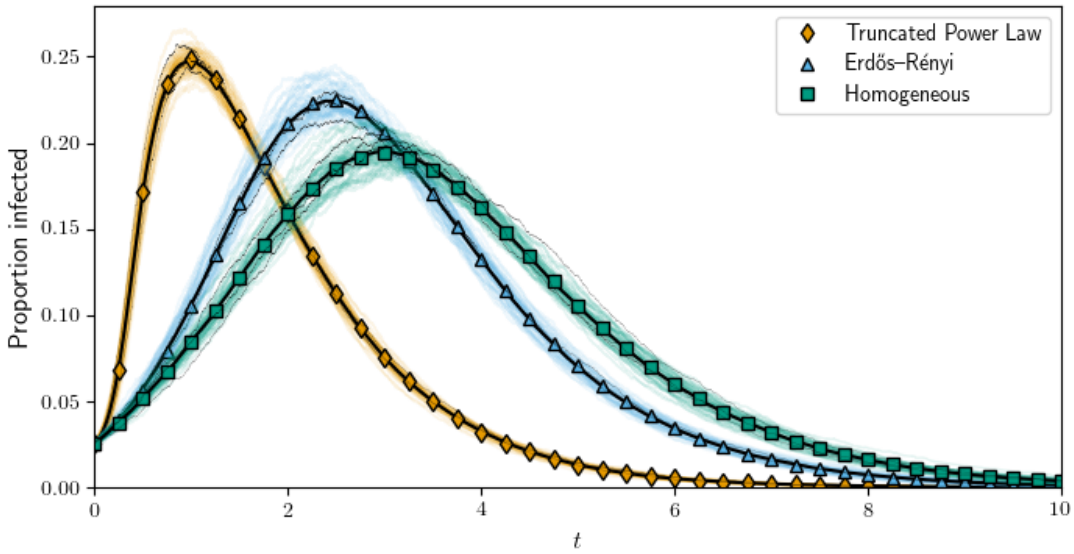
Here are examples to generate (close approximations to) many of the figures in Mathematics of Epidemics on Networks: from Exact to Approximate Models. Other examples are farther down.

Chapter 1

Introduction

Figure 1.2

Downloadable Source Code




```
import EoN
import networkx as nx
from matplotlib import rc
import matplotlib.pylab as plt


import scipy
import random
```

(continues on next page)

(continued from previous page)

```
colors = ['#5AB3E6','#FF2000','#009A80','#E69A00', '#CD9AB3', '#0073B3',
        '#F0E442']

#commands to make legend be in LaTeX font
#rc('font', **{'family': 'serif', 'serif': ['Computer Modern']})
rc('text', usetex=True)



rho = 0.025
target_k = 6
N=10000
tau = 0.5
gamma = 1.
ts = scipy.arange(0,40,0.05)
count = 50 #number of simulations to run for each



def generate_network(Pk, N, ntries = 100):
    r'''Generates an N-node random network whose degree distribution is given by Pk'''
    counter = 0
    while counter< ntries:
        counter += 1
        ks = []
        for ctr in range(N):
            ks.append(Pk())
        if sum(ks)%2 == 0:
            break
    if sum(ks)%2 ==1:
        raise EoN.EoNError("cannot generate even degree sum")
    G = nx.configuration_model(ks)
    return G



#An erdos-renyi network has a Poisson degree distribution.
def PkPoisson():
    return scipy.random.poisson(target_k)
def PsiPoisson(x):
    return scipy.exp(-target_k*(1-x))
def DPsiPoisson(x):
    return target_k*scipy.exp(-target_k*(1-x))



#a regular (homogeneous) network has a simple generating function.

def PkHomogeneous():
    return target_k
def PsiHomogeneous(x):
    return x**target_k
def DPsiHomogeneous(x):
    return target_k*x**(target_k-1)
```

(continues on next page)

(continued from previous page)

```
#The following 30 - 40 lines or so are devoted to defining the degree distribution
#and the generating function of the truncated power law network.

#defining the power law degree distribution here:
assert(target_k==6) #if you've changed target_k, then you'll
                #want to update the range 1..61 and/or
                #the exponent 1.5.

PlPk = {}
exponent = 1.5
kave = 0
for k in range(1,61):
    PlPk[k]=k**(-exponent)
    kave += k*PlPk[k]

normfactor= sum(PlPk.values())
for k in PlPk:
    PlPk[k] /= normfactor

def PkPowLaw():
    r = random.random()
    for k in PlPk:
        r -= PlPk[k]
        if r<0:
            return k

def PsiPowLaw(x):
    #print PlPk
    rval = 0
    for k in PlPk:
        rval += PlPk[k]*x**k
    return rval

def DPsiPowLaw(x):
    rval = 0
    for k in PlPk:
        rval += k*PlPk[k]*x**(k-1)
    return rval
#End of power law network properties.




def process_degree_distribution(N, Pk, color, Psi, DPsi, symbol, label, count):
    report_times = scipy.linspace(0,30,3000)
    sums = 0*report_times
    for cnt in range(count):
        G = generate_network(Pk, N)
        t, S, I, R = EoN.fast_SIR(G, tau, gamma, rho=rho)
        plt.plot(t, I*1./N, '-', color = color,
                                alpha = 0.1, linewidth=1)
        subsampled_I = EoN.subsample(report_times, t, I)
        sums += subsampled_I*1./N
```

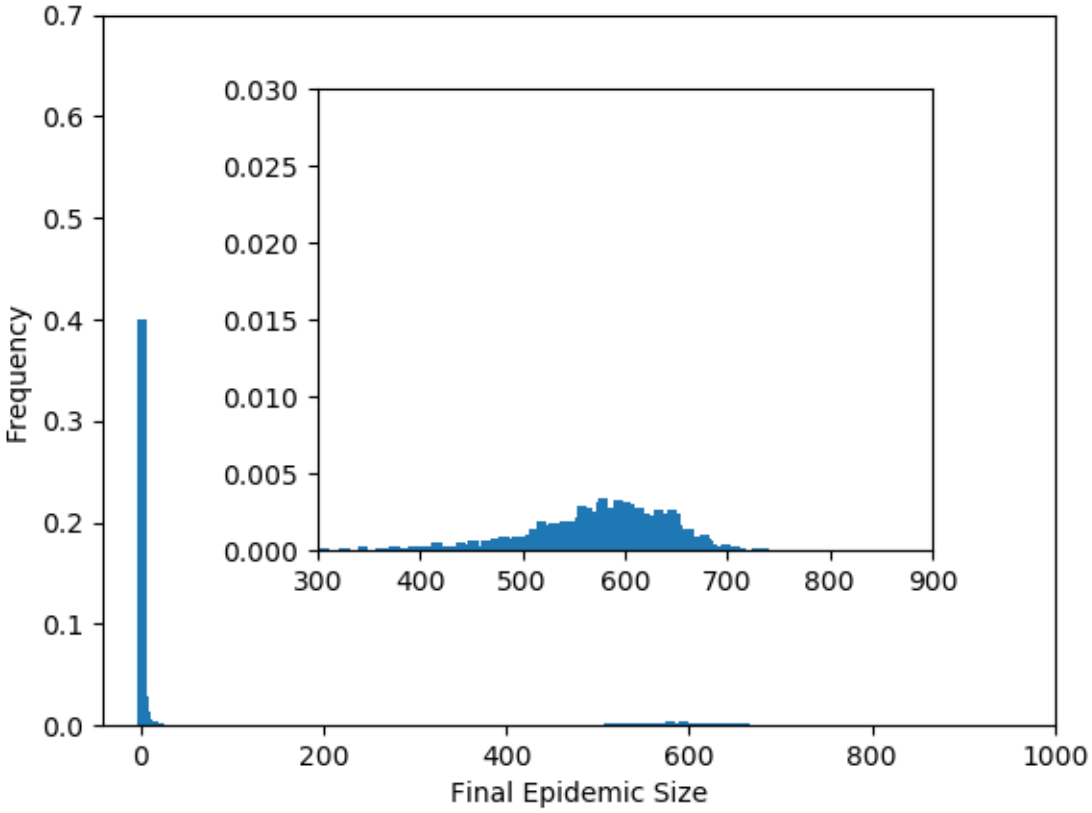
(continues on next page)

(continued from previous page)

```
    ave = sums/count
    plt.plot(report_times, ave, color = 'k')

    #Do EBCM
    N= G.order()#N is arbitrary, but included because our implementation of EBCM␣
↪assumes N is given.
    t, S, I, R = EoN.EBCM_uniform_introduction(N, Psi, DPsi, tau, gamma, rho, tmin=0,␣
↪tmax=10, tcount = 41)
    plt.plot(t, I/N, symbol, color = color, markeredgecolor='k', label=label)

    for cnt in range(3):  #do 3 highlighted simulations
        G = generate_network(Pk, N)
        t, S, I, R = EoN.fast_SIR(G, tau, gamma, rho=rho)
        plt.plot(t, I*1./N, '-', color = 'k', linewidth=0.1)




plt.figure(figsize=(8,4))



#Powerlaw
process_degree_distribution(N, PkPowLaw, colors[3], PsiPowLaw, DPsiPowLaw, 'd', r
↪'Truncated Power Law', count)

#Poisson
process_degree_distribution(N, PkPoisson, colors[0], PsiPoisson, DPsiPoisson, '^', r
↪'Erd\H{o}s--R\'{e}nyi', count)

#Homogeneous
process_degree_distribution(N, PkHomogeneous, colors[2], PsiHomogeneous,␣
↪DPsiHomogeneous, 's', r'Homogeneous', count)

plt.xlabel(r'$t$', fontsize=12)
plt.ylabel(r'Proportion infected', fontsize=12)
plt.legend(loc = 'upper right', numpoints = 1)

plt.axis(xmax=10, xmin=0, ymin=0)
plt.savefig('fig1p2.pdf')
```

Figure 1.5

Downloadable Source Code




```
import EoN
import networkx as nx
import matplotlib.pyplot as plt

from collections import defaultdict


N=1000
gamma = 1
tau = 1.5/N
G = nx.complete_graph(N)
iterations = 10000
binwidth = 10

H = defaultdict(int)
for counter in range(iterations):
    t, S, I, R = EoN.fast_SIR(G, tau, gamma)
    H[binwidth*(R[-1]/binwidth)] = H[binwidth*(R[-1]/binwidth)]+1./iterations


fig = plt.figure(1)
main = plt.axes()

main.bar(*zip(*H.items()), width = binwidth, linewidth=0)
main.axis(xmax=1000, ymax = 0.7)
```

(continues on next page)

(continued from previous page)

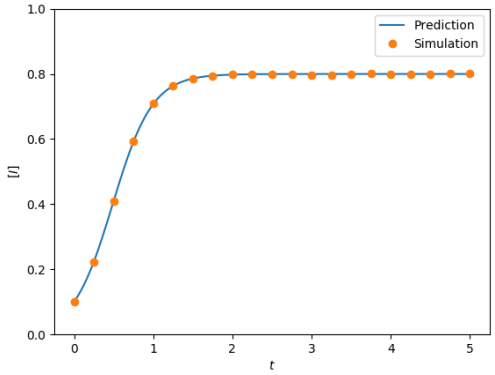
```
plt.xlabel('Final Epidemic Size')
plt.ylabel('Frequency')

inset = plt.axes([0.3,0.3,0.5,0.5])
inset.bar(*zip(*H.items()), width = binwidth, linewidth=0)
inset.axis(xmin = 300, xmax = 900, ymin=0, ymax = 0.03)

plt.savefig('fig1p5.png')
```

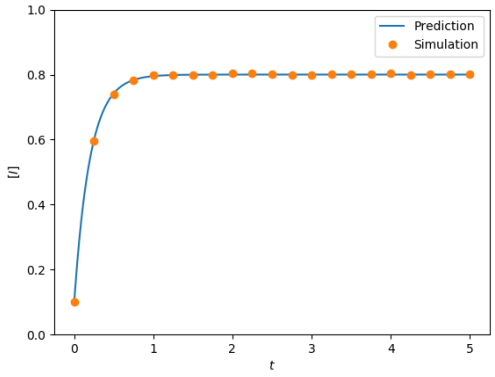
Chapter 2

Top down models

Figure 2.11 (a and b)

Downloadable Source Code






```
import EoN
import networkx as nx
import matplotlib.pyplot as plt
import scipy
import random
from scipy import integrate
```

(continues on next page)

(continued from previous page)

```
'''
Code to generate figure 2.11.  This is a bit messy because we have to
define the ODE models.  Since python deals with ODEs by taking 1D arrays
we have to set up all the variables into a single long vector.
'''

def star(N):
    G = nx.Graph()
    G.add_node(0)
    for node_id in range(1,N):
        G.add_edge(0,node_id)
    return G




def complete_graph_dX(X, t, tau, gamma, N):
    r'''This system is given in Proposition 2.3, taking Q=S, T=I
    f_{SI}(k) = f_{QT}= k*\tau
    f_{IS}(k) = f_{TQ} = \gamma

    \dot{Y}^0 = \gamma Y^1 - 0\\
    \dot{Y}^1 = 2\gamma Y^2  + 0Y^0 - (\gamma + (N-1)\tau)Y^1
    \dot{Y}^2 = 3\gamma Y^3 + (N-1)\tau Y^1 - (2\gamma+2(N-2))Y^2
    ...
    \dot{Y}^N = (N-1)\tau Y^{N-1} - N\gamma Y^N
    Note that X has length N+1
    '''
    #X[k] is probability of k infections.
    dX = []
    dX.append(gamma*X[1])
    for k in range(1,N):
        dX.append((k+1)*gamma*X[k+1]+ (N-k+1)*(k-1)*tau*X[k-1]
                  - ((N-k)*k*tau + k*gamma)*X[k])
    dX.append((N-1)*tau*X[N-1] - N*gamma*X[N])

    return scipy.array(dX)

def complete_graph_lumped(N, tau, gamma, I0, tmin, tmax, tcount):
    times = scipy.linspace(tmin, tmax, tcount)
    X0 = scipy.zeros(N+1)  #length N+1 of just 0 entries
    X0[I0]=1. #start with 100 infected.
    X = integrate.odeint(complete_graph_dX, X0, times, args = (tau, gamma, N))
    #X[t] is array whose kth entry is p(k infected| time=t).
    I = scipy.array([sum(k*Pkt[k] for k in range(len(Pkt))) for Pkt in X])
    S = N-I
    return times, S, I


def star_graph_dX(X, t, tau, gamma, N):
    '''this system is given in Proposition 2.4, taking Q=S, T=I
    so f_{SI}(k) = f_{QT}(k) = k*tau
    f_{IS}(k) = f_{TQ}(k) = gamma
    X has length 2*(N-1)+2 = 2N'''
```

(continues on next page)

(continued from previous page)

```
    #    [[central node infected] + [central node susceptible]]
    #X = [Y_1^1, Y_1^2, ..., Y_1^{N}, Y_2^0, Y_2^1, ..., Y_2^{N-1}]

    #Note that in proposition Y^0 is same as Y_2^0
    #and Y^N is same as Y_1^N

    #Y1[k]: central node infected, & k-1 peripheral nodes infected
    Y1vec = [0]+list(X[0:N])      #for Y_1^k, use Y1vec[k]
    #pad with 0 to make easier calculations Y_1^0=0
    #the probability of -1 nodes infected is 0

    #Y2[k]: central node susceptible & k peripheral nodes infected
    Y2vec = list(X[N:])+[0]   #for Y_2^k use Y2vec[k]
    #padded with 0 to make easier calculations. Y_2^N=0
    #the probability of N (of N-1) peripheral nodes infected is 0
    dY1vec = []
    dY2vec = []
    for k in range(1, N):
        #k-1 peripheral nodes infected, central infected
        dY1vec.append((N-k+1)*tau*Y1vec[k-1] + (k-1)*tau*Y2vec[k-1]
                    +k*gamma*Y1vec[k+1]
                    - ((N-k)*tau + (k-1)*gamma+gamma)*Y1vec[k])
    #now the Y^N equation
    dY1vec.append(tau*Y1vec[N-1] + (N-1)*tau*Y2vec[N-1] - N*gamma*Y1vec[N])

    #now the Y^0 equation
    dY2vec.append(gamma*(N-1)*Y1vec[1] + gamma*Y2vec[1]-0)

    for k in range(1,N):
        #k peripheral nodes infected, central susceptible
        dY2vec.append(0 + gamma*Y1vec[k+1] + gamma*(k+1)*Y2vec[k+1]
                    - (k*tau + 0 + k*gamma)*Y2vec[k])

    return scipy.array(dY1vec + dY2vec)

def star_graph_lumped(N, tau, gamma, I0, tmin, tmax, tcount):
    times = scipy.linspace(tmin, tmax, tcount)
    #    [[central node infected] + [central node susceptible]]
    #X = [Y_1^1, Y_1^2, ..., Y_1^{N}, Y_2^0, Y_2^1, ..., Y_2^{N-1}]
    X0 = scipy.zeros(2*N)  #length 2*N of just 0 entries
    X0[I0]=I0*1./N #central infected, + I0-1 periph infected prob
    X0[N+I0] = 1-I0*1./N #central suscept + I0 periph infected
    X = EoN.my_odeint(star_graph_dX, X0, times, args = (tau, gamma, N))
    #X looks like [[central susceptible,k periph] [ central inf, k-1 periph]] x T

    central_inf = X[:,:N]
    central_susc = X[:,N:]

    I = scipy.array([ sum(k*central_susc[t][k] for k in range(N))
            + sum((k+1)*central_inf[t][k] for k in range(N))
            for t in range(len(X))])
    S = N-I
    return times, S, I
```

(continues on next page)

(continued from previous page)

```
N=1000
I0=int(0.1*N)
iterations = 100 #number of simulations to compare
gamma = 1
tmin=0
tmax=5
tcount = 1001
report_times = scipy.linspace(tmin, tmax, 21) #for simulations


plt.figure(0)
tau = 0.005
G = nx.complete_graph(N)
t, S, I = complete_graph_lumped(N, tau, gamma, I0, tmin, tmax, tcount)
plt.plot(t, I/N, label = 'Prediction')

#now check with simulation
obs_I = 0*report_times
print("done with complete graph ODE.  Now simulating")
for counter in range(iterations):
    IC = random.sample(range(N),I0)
    t, S, I = EoN.fast_SIS(G, tau, gamma, initial_infecteds = IC, tmax = tmax)
    obs_I += EoN.subsample(report_times, t, I)
plt.plot(report_times, obs_I*1./(iterations*N), 'o', label='Simulation')
plt.axis(ymin=0, ymax=1)
plt.xlabel('$t$')
plt.ylabel('$[I]$')
plt.legend()
plt.savefig('fig2p11a.png')

print("done with complete graph.  Now star --- warning, this may be slow")









plt.clf()
#for star, if 100 nodes randomly start infected, 1/10 cases have
#central node infected and 99 peripheral.  9/10 have 100 peripheral.

tau = 4.
G = star(N)

t, S, I = star_graph_lumped(N, tau, gamma, I0, tmin, tmax, tcount)
plt.plot(t, I/N, label = 'Prediction')
print("done with star ODE, now simulating")

obs_I = 0*report_times
for counter in range(iterations):
    IC = random.sample(range(N),I0)
    t, S, I = EoN.fast_SIS(G, tau, gamma, initial_infecteds = IC, tmax = tmax)
    obs_I += EoN.subsample(report_times, t, I)
plt.plot(report_times, obs_I*1./(iterations*N), 'o', label='Simulation')
plt.axis(ymin=0, ymax=1)
```

(continues on next page)

(continued from previous page)

```
plt.xlabel('$t$')
plt.ylabel('$[I]$')
plt.legend()
plt.savefig('fig2p11b.png')
```

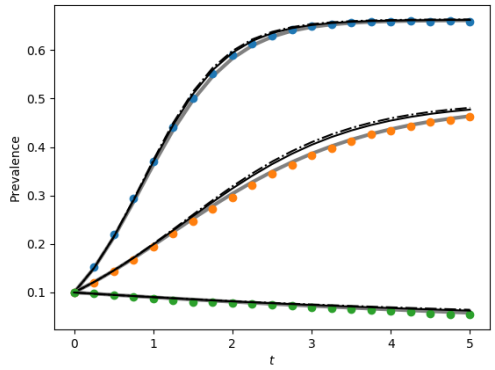
Chapter 3

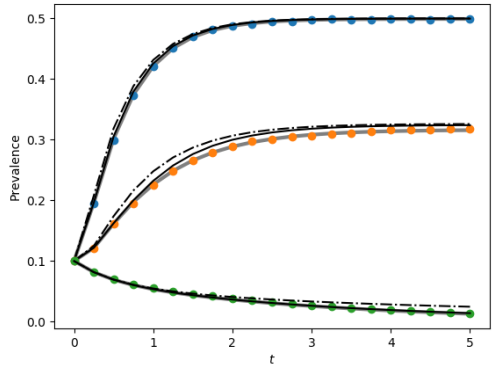
Bottom up models

Figure 3.2 (a and b)

`Downloadable Source Code`

- In addition to plots in the book's figure, this also plots the average of 1000 simulations.
- For the complete graph, the pair equations run quite slowly (there are N choose 2 edges, and we need equations for each).
- This code does not include the triangle corrections mentioned after system (3.26).






```
import EoN
import networkx as nx
import matplotlib.pyplot as plt
import scipy
```

(continues on next page)

(continued from previous page)

```
import random
from scipy import integrate



'''
Code to generate figure 3.2 from page 94.  This is a bit messy because
we have to define the lumped ODE models.  Since python deals with ODEs by
taking 1D arrays we have to set up all the variables into a single long vector.

This uses the same lumped model as for fig 2.11.

'''
print("There is a difference between the output here and figure3.2a for the\n",
    "book.  This code uses system 3.26, while the figure in the book\n"
    "includes the triangle correction mentioned following system 3.26.\n\n")

print("The code for the complete graph runs quite slowly.  There are very many␣
↪equations.")


def star(N):
    G = nx.Graph()
    G.add_node(0)
    for node_id in range(1,N):
        G.add_edge(0,node_id)
    return G




def complete_graph_dX(X, t, tau, gamma, N):
    r'''This system is given in Proposition 2.3, taking Q=S, T=I
    f_{SI}(k) = f_{QT}= k*\tau
    f_{IS}(k) = f_{TQ} = \gamma

    \dot{Y}^0 = \gamma Y^1 - 0\\
    \dot{Y}^1 = 2\gamma Y^2  + 0Y^0 - (\gamma + (N-1)\tau)Y^1
    \dot{Y}^2 = 3\gamma Y^3 + (N-1)\tau Y^1 - (2\gamma+2(N-2))Y^2
    ...
    \dot{Y}^N = (N-1)\tau Y^{N-1} - N\gamma Y^N
    Note that X has length N+1
    '''
    #X[k] is probability of k infections.
    dX = []
    dX.append(gamma*X[1])
    for k in range(1,N):
        dX.append((k+1)*gamma*X[k+1]+ (N-k+1)*(k-1)*tau*X[k-1]
                  - ((N-k)*k*tau + k*gamma)*X[k])
    dX.append((N-1)*tau*X[N-1] - N*gamma*X[N])

    return scipy.array(dX)

def complete_graph_lumped(N, tau, gamma, I0, tmin, tmax, tcount):
    times = scipy.linspace(tmin, tmax, tcount)
    X0 = scipy.zeros(N+1)  #length N+1 of just 0 entries
    X0[I0]=1. #start with 100 infected.
```

(continues on next page)

(continued from previous page)

```
    X = integrate.odeint(complete_graph_dX, X0, times, args = (tau, gamma, N))
    #X[t] is array whose kth entry is p(k infected| time=t).
    I = scipy.array([sum(k*Pkt[k] for k in range(len(Pkt))) for Pkt in X])
    S = N-I
    return times, S, I


def star_graph_dX(X, t, tau, gamma, N):
    '''this system is given in Proposition 2.4, taking Q=S, T=I
    so f_{SI}(k) = f_{QT}(k) = k*tau
    f_{IS}(k) = f_{TQ}(k) = gamma
    X has length 2*(N-1)+2 = 2N'''

    #    [[central node infected] + [central node susceptible]]
    #X = [Y_1^1, Y_1^2, ..., Y_1^{N}, Y_2^0, Y_2^1, ..., Y_2^{N-1}]

    #Note that in proposition Y^0 is same as Y_2^0
    #and Y^N is same as Y_1^N

    #Y1[k]: central node infected, & k-1 peripheral nodes infected
    Y1vec = [0]+list(X[0:N])      #for Y_1^k, use Y1vec[k]
    #pad with 0 to make easier calculations Y_1^0=0
    #the probability of -1 nodes infected is 0

    #Y2[k]: central node susceptible & k peripheral nodes infected
    Y2vec = list(X[N:])+[0]   #for Y_2^k use Y2vec[k]
    #padded with 0 to make easier calculations. Y_2^N=0
    #the probability of N (of N-1) peripheral nodes infected is 0
    dY1vec = []
    dY2vec = []
    for k in range(1, N):
        #k-1 peripheral nodes infected, central infected
        dY1vec.append((N-k+1)*tau*Y1vec[k-1] + (k-1)*tau*Y2vec[k-1]
                    +k*gamma*Y1vec[k+1]
                    - ((N-k)*tau + (k-1)*gamma+gamma)*Y1vec[k])
    #now the Y^N equation
    dY1vec.append(tau*Y1vec[N-1] + (N-1)*tau*Y2vec[N-1] - N*gamma*Y1vec[N])

    #now the Y^0 equation
    dY2vec.append(gamma*(N-1)*Y1vec[1] + gamma*Y2vec[1]-0)

    for k in range(1,N):
        #k peripheral nodes infected, central susceptible
        dY2vec.append(0 + gamma*Y1vec[k+1] + gamma*(k+1)*Y2vec[k+1]
                    - (k*tau + 0 + k*gamma)*Y2vec[k])

    return scipy.array(dY1vec + dY2vec)

def star_graph_lumped(N, tau, gamma, I0, tmin, tmax, tcount):
    times = scipy.linspace(tmin, tmax, tcount)
    #    [[central node infected] + [central node susceptible]]
    #X = [Y_1^1, Y_1^2, ..., Y_1^{N}, Y_2^0, Y_2^1, ..., Y_2^{N-1}]
    X0 = scipy.zeros(2*N)  #length 2*N of just 0 entries
    #X0[I0]=I0*1./N #central infected, + I0-1 periph infected prob
    X0[N+I0] = 1#-I0*1./N #central suscept + I0 periph infected
    X = EoN.my_odeint(star_graph_dX, X0, times, args = (tau, gamma, N))
    #X looks like [[central susceptible,k periph] [ central inf, k-1 periph]] x T
```

(continues on next page)

(continued from previous page)

```
    central_susc = X[:,N:]
    central_inf = X[:,:N]
    print(central_susc[-1][:])
    print(central_inf[-1][:])
    I = scipy.array([ sum(k*central_susc[t][k] for k in range(N))
            + sum((k+1)*central_inf[t][k] for k in range(N))
            for t in range(len(X))])
    S = N-I
    return times, S, I




N=100
I0 = 10
gamma = 1
tmin = 0
tmax = 5
tcount = 21
report_times = scipy.linspace(0,tmax, tcount) #for simulations

iterations = 1000


taus1 = [0.03, 0.02, 0.01]
taus2 = [1, 0.5, 0.1]

plt.figure(1)

G = nx.complete_graph(N)

for tau in taus1:

    print(tau)
    print('lumped')
    t, S, I = complete_graph_lumped(N, tau, gamma, I0, tmin, tmax, tcount)
    plt.plot(t, I/N, color = 'grey', linewidth = 3)

    print('I[-1]', I[-1])
    initial_infecteds=random.sample(range(N), I0)

    obs_I = 0*report_times
    for counter in range(iterations):
        if counter%100==0:
            print(counter)
        IC = random.sample(range(N),I0)
        t, S, I = EoN.fast_SIS(G, tau, gamma, initial_infecteds = initial_infecteds,␣
↪tmax = tmax)
        obs_I += EoN.subsample(report_times, t, I)
    plt.plot(report_times, obs_I*1./(iterations*N), 'o')
    #print(obs_I[-1]/iterations)
    print('individual based')
    t, S, I = EoN.SIS_individual_based_pure_IC(G, tau, gamma, initial_infecteds,␣
↪tmax=tmax, tcount = tcount)
    plt.plot(t, I/N, '-.', color = 'k')
```

(continues on next page)

(continued from previous page)

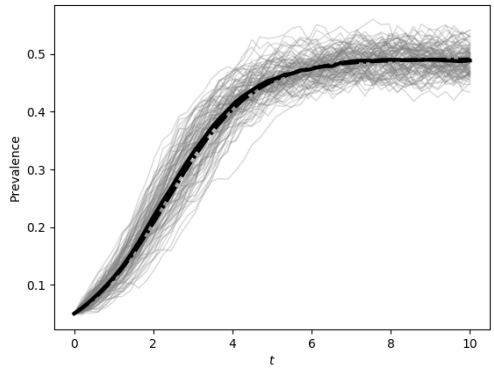
```
    print('pair based')
    t, S, I = EoN.SIS_pair_based_pure_IC(G, tau, gamma, initial_infecteds, tmax=tmax,␣
↪tcount = tcount)
    plt.plot(t, I/N, color = 'k')

plt.xlabel('$t$')
plt.ylabel('Prevalence')
plt.savefig('fig3p2a.png')

plt.clf()
G = star(N)
for tau in taus2:
    print(tau)
    print('lumped')
    t, S, I = star_graph_lumped(N, tau, gamma, I0, tmin, tmax, tcount)
    print(I)
    plt.plot(t, I/N, color = 'grey', linewidth = 3)

    initial_infecteds=random.sample(range(1,N), I0)#not central node 0

    obs_I = 0*report_times
    for counter in range(iterations):
        if counter%100==0:
            print(counter)
        IC = random.sample(range(N),I0)
        t, S, I = EoN.fast_SIS(G, tau, gamma, initial_infecteds = initial_infecteds,␣
↪tmax = tmax)
        obs_I += EoN.subsample(report_times, t, I)
    #print(obs_I/iterations)
    plt.plot(report_times, obs_I*1./(iterations*N), 'o')

    print('individual based')
    t, S, I = EoN.SIS_individual_based_pure_IC(G, tau, gamma, initial_infecteds,␣
↪tmax=tmax, tcount = tcount)
    plt.plot(t, I/N, '-.', color = 'k')

    print('pair based')
    t, S, I = EoN.SIS_pair_based_pure_IC(G, tau, gamma, initial_infecteds, tmax=tmax,␣
↪tcount = tcount)
    plt.plot(t, I/N, color = 'k')
plt.xlabel('$t$')
plt.ylabel('Prevalence')
plt.savefig('fig3p2b.png')
```

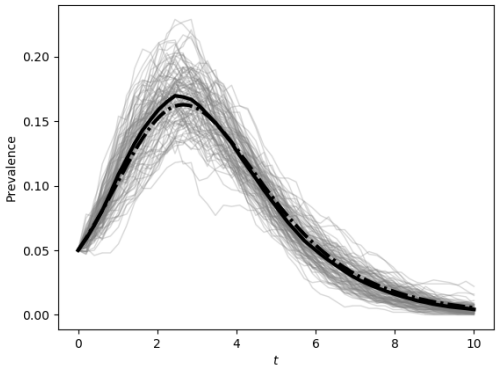
Chapter 4

Homogeneous meanfield

Figure 4.1 (a and b)

Downloadable Source Code






```
import EoN
import networkx as nx
import matplotlib.pyplot as plt
import scipy
import random

N = 1000
kave = 30.
gamma = 1.
tau = 1./15
tmax = 10

rho = 0.05
iterations = 100

tcount = 50

report_times = scipy.linspace(0,tmax,tcount)

SIS_Isum=scipy.zeros(tcount)
SIR_Isum=scipy.zeros(tcount)

for counter in range(iterations):
    #do SIS simulation and then SIR simulation.
    G = nx.fast_gnp_random_graph(N, kave/(N-1))
    initial_infecteds = random.sample(G.nodes(), int(rho*G.order()))
```

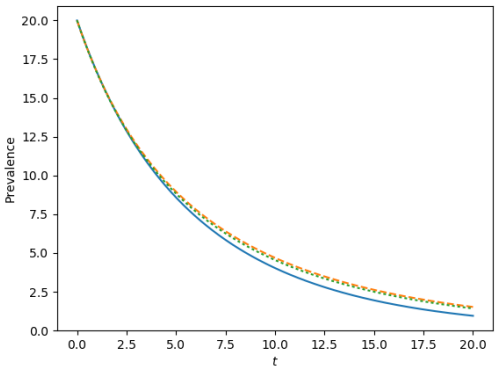
(continues on next page)

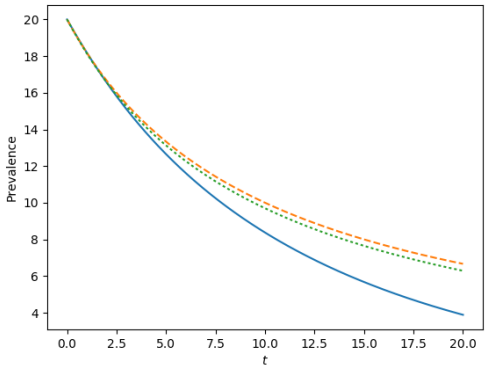
(continued from previous page)

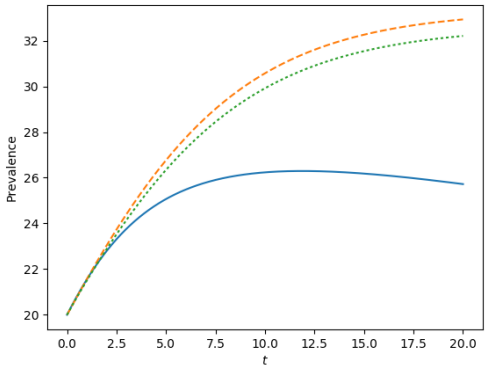
```
    t, S, I = EoN.fast_SIS(G, tau, gamma, initial_infecteds=initial_infecteds,␣
↪tmax=tmax)
    I = EoN.subsample(report_times, t, I)
    SIS_Isum += I
    plt.figure(0)
    plt.plot(report_times, I*1./N, linewidth=1, alpha=0.3, color='grey')
    t, S, I, R = EoN.fast_SIR(G, tau, gamma, initial_infecteds=initial_infecteds,␣
↪tmax=tmax)
    I = EoN.subsample(report_times, t, I)
    SIR_Isum += I
    plt.figure(1)
    plt.plot(report_times, I*1./N, linewidth=1, alpha=0.3, color='grey')

S0 = (1-rho)*N
I0 = rho*N
SI0 = (1-rho)*N*kave*rho
SS0 = (1-rho)*N*kave*(1-rho)
t, S, I = EoN.SIS_homogeneous_pairwise(S0, I0, SI0, SS0, kave, tau,
                                       gamma, tmax=tmax)
plt.figure(0)
plt.plot(report_times,SIS_Isum/(N*iterations), color='k', linewidth=3)
plt.plot(t, I/N, '-.', color='k', linewidth=3)
plt.xlabel('$t$')
plt.ylabel('Prevalence')
plt.savefig('fig4p1a.png')


t, S, I, R = EoN.SIR_homogeneous_pairwise(S0, I0, 0, SI0, SS0, kave, tau,
                                       gamma, tmax=tmax)
plt.figure(1)
plt.plot(report_times,SIR_Isum/(N*iterations), color='k', linewidth=3)
plt.plot(t, I/N, '-.', color='k', linewidth=3)
plt.xlabel('$t$')
plt.ylabel('Prevalence')
plt.savefig('fig4p1b.png')
```

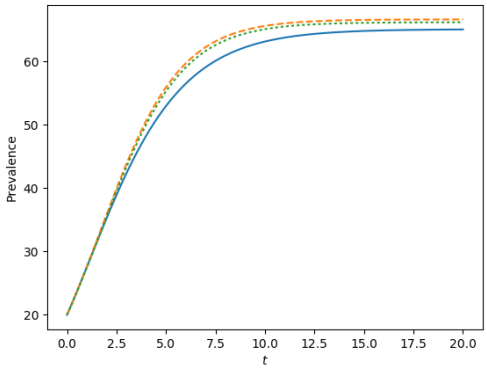
Figure 4.5 (a, b, c, and d)

Downloadable Source Code










```
import EoN
import networkx as nx
import matplotlib.pyplot as plt
import scipy
from scipy import integrate


def complete_graph_dX(X, t, tau, gamma, N):
    r'''This system is given in Proposition 2.3, taking Q=S, T=I
    f_{SI}(k) = f_{QT}= k*\tau
    f_{IS}(k) = f_{TQ} = \gamma
```

(continues on next page)

(continued from previous page)

```
    \dot{Y}^0 = \gamma Y^1 - 0\\
    \dot{Y}^1 = 2\gamma Y^2  + 0Y^0 - (\gamma + (N-1)\tau)Y^1
    \dot{Y}^2 = 3\gamma Y^3 + (N-1)\tau Y^1 - (2\gamma+2(N-2))Y^2
    ...
    \dot{Y}^N = (N-1)\tau Y^{N-1} - N\gamma Y^N
    Note that X has length N+1
    '''
    #X[k] is probability of k infections.
    dX = []
    dX.append(gamma*X[1])
    for k in range(1,N):
        dX.append((k+1)*gamma*X[k+1]+ (N-k+1)*(k-1)*tau*X[k-1]
                  - ((N-k)*k*tau + k*gamma)*X[k])
    dX.append((N-1)*tau*X[N-1] - N*gamma*X[N])

    return scipy.array(dX)

def complete_graph_lumped(N, I0, tmin, tmax, tcount):
    times = scipy.linspace(tmin, tmax, tcount)
    X0 = scipy.zeros(N+1)  #length N+1 of just 0 entries
    X0[I0]=1. #start with 100 infected.
    X = integrate.odeint(complete_graph_dX, X0, times, args = (tau, gamma, N))
    #X[t] is array whose kth entry is p(k infected| time=t).
    I = scipy.array([sum(k*Pkt[k] for k in range(len(Pkt))) for Pkt in X])
    S = N-I
    return times, S, I



N=200
gamma = 1


k = N-1.
tau_c = gamma/k
rho = 0.1

for tau, label in zip([0.9*tau_c, tau_c, 1.2*tau_c, 1.5*tau_c],['a', 'b', 'c', 'd']):
    plt.clf()
    t, S, I = complete_graph_lumped(N, int(N*rho), 0 , 20, 1001)
    plt.plot(t, I)

    S0 = (1-rho)*N
    I0 = rho*N

    t, S, I = EoN.SIS_homogeneous_meanfield(S0, I0, k, tau, gamma, tmin=0, tmax=20,
                                tcount=1001)
    plt.plot(t, I, '--')
    S0 = (1-rho)*N
    I0 = rho*N
    SI0 = (1-rho)*N*k*rho
    SS0 = (1-rho)*N*k*(1-rho)
    t, S, I = EoN.SIS_homogeneous_pairwise(S0, I0, SI0, SS0, k, tau, gamma, tmin = 0,
                                tmax=20, tcount=1001)
    plt.plot(t, I, ':')
    plt.xlabel('$t$')
    plt.ylabel('Prevalence')
```

(continues on next page)

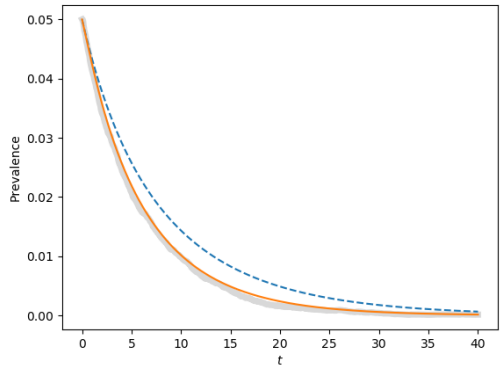
(continued from previous page)

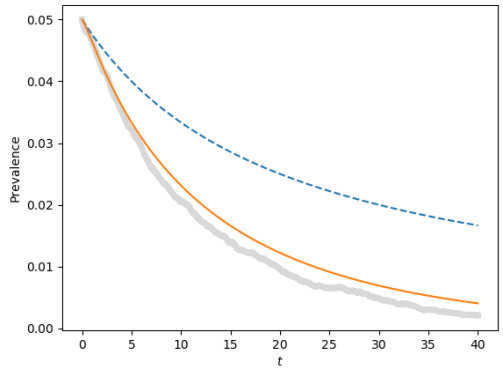
```
    plt.savefig('fig4p5{}.png'.format(label))
```

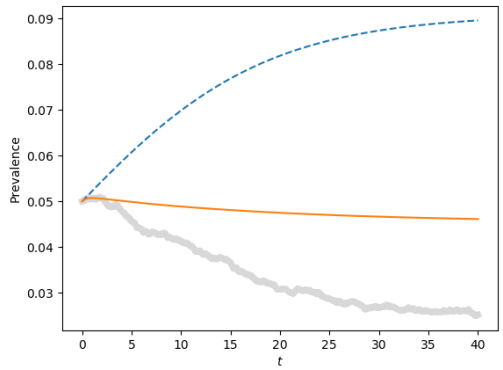
Figure 4.7 (a, b, c, and d)

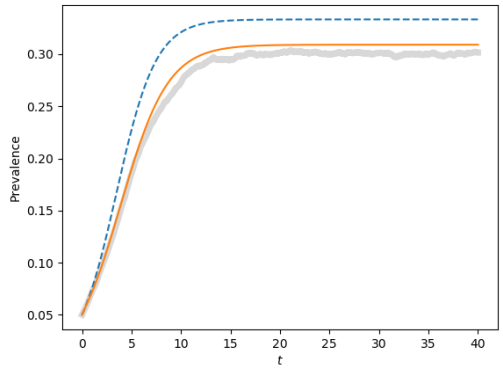
Downloadable Source Code

- Note that the book has a typo. For (c), $\tau = 1.1\tau_c$










```
import EoN
import networkx as nx
import matplotlib.pyplot as plt
import scipy


print(r"warning --- plot c in book is mislabeled.  It should be \tau = 1.1\tau_c, not␣
↪1.2\tau_c")
N=1000
kave = 20
gamma = 1.
iterations = 200
tmax = 40
tau_c = gamma/kave
rho = 0.05
tcount=1001

report_times = scipy.linspace(0,tmax,tcount)

for tau, label in zip([0.9*tau_c, tau_c, 1.1*tau_c, 1.5*tau_c],['a', 'b', 'c', 'd']):
    plt.clf()
    Isum = scipy.zeros(len(report_times))
    for counter in range(iterations):
        G = nx.configuration_model([kave]*N)
        t, S, I = EoN.fast_SIS(G, tau, gamma, tmax=tmax, rho=rho)
        I=I*1./N
        I = EoN.subsample(report_times, t, I)
        Isum += I
    plt.plot(report_times, Isum/iterations, color = 'grey', linewidth = 5, alpha=0.3)


    S0 = (1-rho)*N
    I0 = rho*N

    t, S, I = EoN.SIS_homogeneous_meanfield(S0, I0, kave, tau, gamma, tmin=0,␣
↪tmax=tmax,
                                tcount=tcount)
    plt.plot(t, I/N, '--')
    S0 = (1-rho)*N
    I0 = rho*N
    SI0 = (1-rho)*N*kave*rho
    SS0 = (1-rho)*N*kave*(1-rho)
    t, S, I = EoN.SIS_homogeneous_pairwise(S0, I0, SI0, SS0, kave, tau, gamma, tmin =␣
↪0,
```

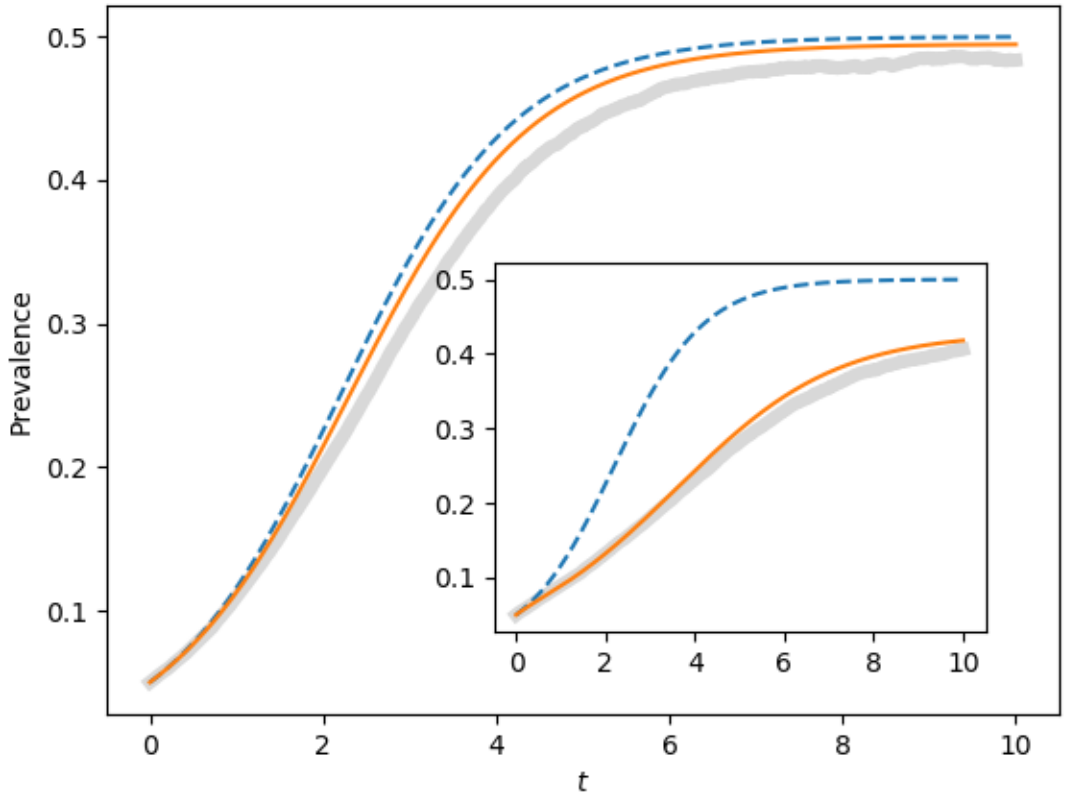
(continues on next page)

(continued from previous page)

```
                                    tmax=tmax, tcount=tcount)
    plt.plot(t, I/N)
    plt.xlabel('$t$')
    plt.ylabel('Prevalence')
    plt.savefig('fig4p7{}.png'.format(label))
```

Figure 4.8

Downloadable Source Code




```
import EoN
import networkx as nx
import matplotlib.pyplot as plt
import scipy


N=1000
gamma = 1.
iterations = 200
rho = 0.05
tmax = 10
tcount = 1001
```

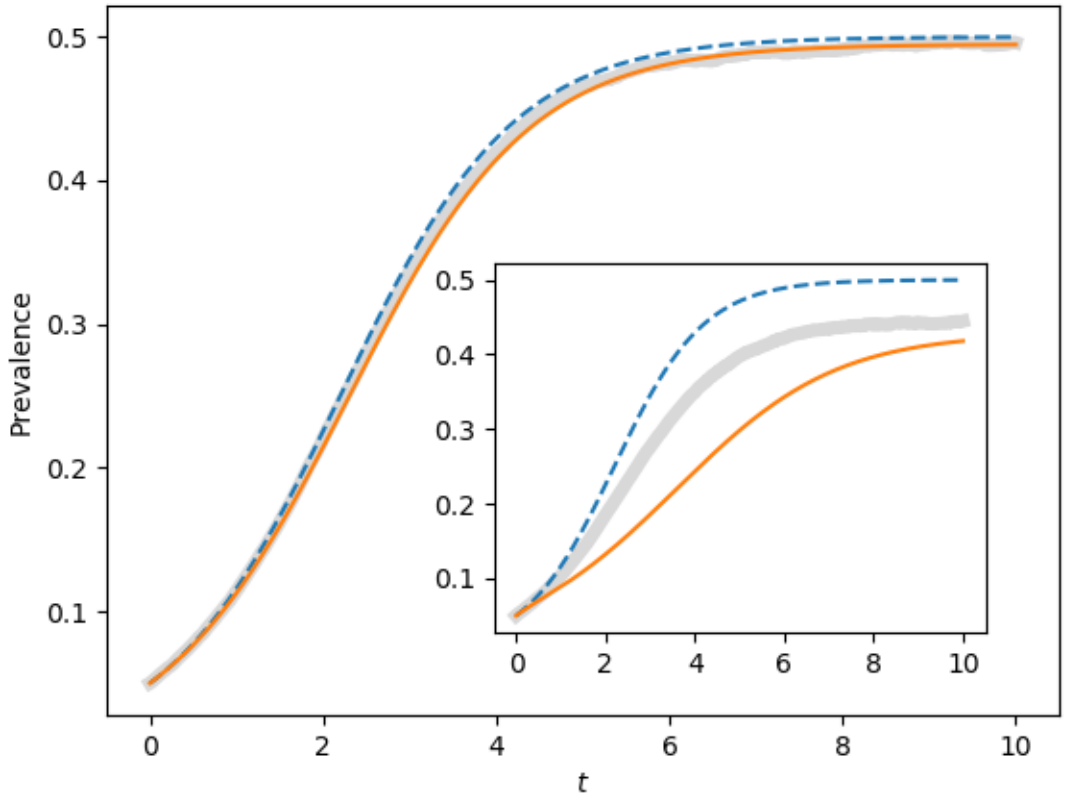
(continues on next page)

(continued from previous page)

```
report_times = scipy.linspace(0,tmax,tcount)

#plt.plot([],[])
ax1 = plt.gca()#axes([0.1,0.1,0.9,0.9])
ax2 = plt.axes([0.44,0.2,0.4,0.4])
for kave, ax in zip([50, 5], [ax1, ax2]):
    tau = 2*gamma/kave
    Isum = scipy.zeros(tcount)

    for counter in range(iterations):
        G = nx.configuration_model(N*[kave])
        t, S, I = EoN.fast_SIS(G, tau, gamma, tmax=tmax, rho=rho)
        I = I*1./N
        I = EoN.subsample(report_times, t, I)
        Isum += I
    ax.plot(report_times, Isum/iterations, color='grey', linewidth=5, alpha=0.3)

    S0 = (1-rho)*N
    I0 = rho*N

    t, S, I = EoN.SIS_homogeneous_meanfield(S0, I0, kave, tau, gamma, tmin=0,␣
↪tmax=tmax,
                                tcount=tcount)
    ax.plot(t, I/N, '--')
    S0 = (1-rho)*N
    I0 = rho*N
    SI0 = (1-rho)*N*kave*rho
    SS0 = (1-rho)*N*kave*(1-rho)
    t, S, I = EoN.SIS_homogeneous_pairwise(S0, I0, SI0, SS0, kave, tau, gamma, tmin =␣
↪0,
                                tmax=tmax, tcount=tcount)
    ax.plot(t, I/N)

ax1.xlabel('$t$')
ax1.ylabel('Prevalence')
plt.savefig('fig4p8.png')
```

Figure 4.9

Downloadable Source Code




```
import EoN
import networkx as nx
import matplotlib.pyplot as plt
import scipy


N=1000
gamma = 1.
iterations = 200
rho = 0.05
tmax = 10
tcount = 1001

report_times = scipy.linspace(0,tmax,tcount)

ax1 = plt.gca()#axes([0.1,0.1,0.9,0.9])
ax2 = plt.axes([0.44,0.2,0.4,0.4])
for kave, ax in zip([50, 5], [ax1, ax2]):
    tau = 2*gamma/kave
    Isum = scipy.zeros(tcount)

    for counter in range(iterations):
        G = nx.fast_gnp_random_graph(N, kave/(N-1.))
        t, S, I = EoN.fast_SIS(G, tau, gamma, tmax=tmax, rho=rho)
        I = I*1./N
```

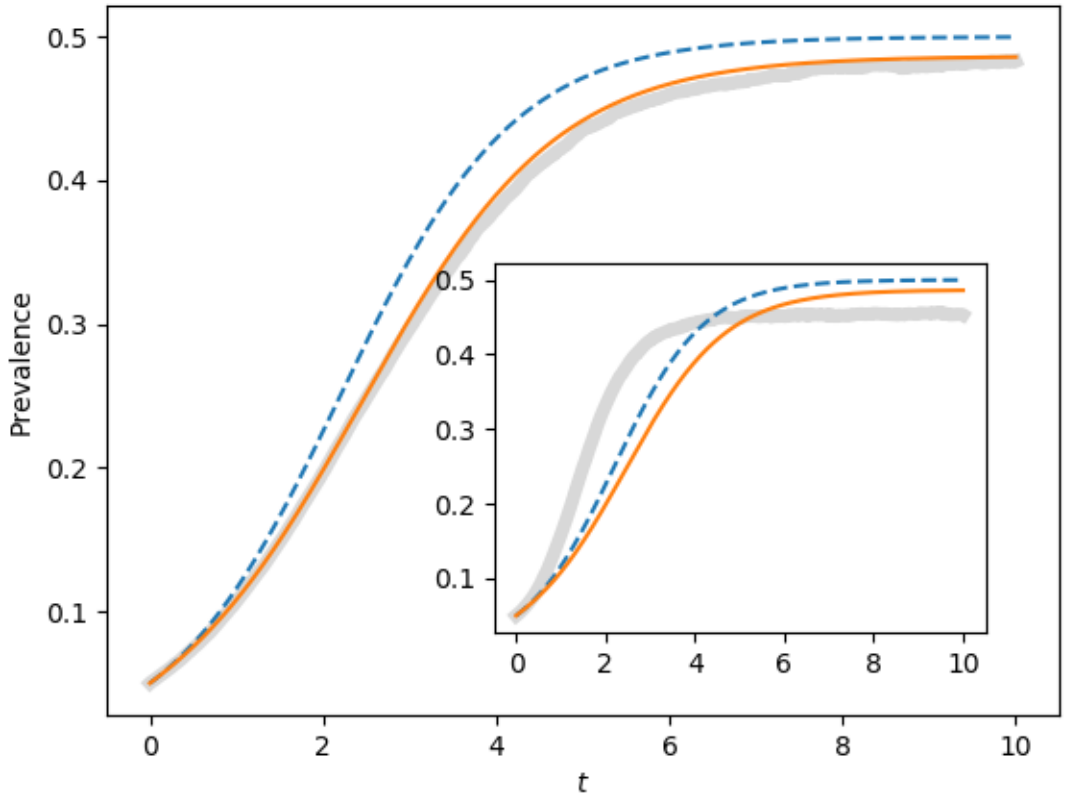
(continues on next page)

(continued from previous page)

```
        I = EoN.subsample(report_times, t, I)
        Isum += I
    ax.plot(report_times, Isum/iterations, color='grey', linewidth=5, alpha=0.3)

    S0 = (1-rho)*N
    I0 = rho*N

    t, S, I = EoN.SIS_homogeneous_meanfield(S0, I0, kave, tau, gamma, tmin=0,␣
↪tmax=tmax,
                                tcount=tcount)
    ax.plot(t, I/N, '--')
    S0 = (1-rho)*N
    I0 = rho*N
    SI0 = (1-rho)*N*kave*rho
    SS0 = (1-rho)*N*kave*(1-rho)
    t, S, I = EoN.SIS_homogeneous_pairwise(S0, I0, SI0, SS0, kave, tau, gamma, tmin =␣
↪0,
                                tmax=tmax, tcount=tcount)
    ax.plot(t, I/N)

ax1.set_xlabel('$t$')
ax1.set_ylabel('Prevalence')
plt.savefig('fig4p9.png')
```

Figure 4.10

Downloadable Source Code




```
import EoN
import networkx as nx
import matplotlib.pyplot as plt
import scipy


N=1000
gamma = 1.
iterations = 200
rho = 0.05
tmax = 10
tcount = 1001

report_times = scipy.linspace(0,tmax,tcount)

deg_dist1 = [18,22]*int((N/2)+0.01)
deg_dist2 = [5,35]*int((N/2)+0.01)
ax1 = plt.gca()#axes([0.1,0.1,0.9,0.9])
ax2 = plt.axes([0.44,0.2,0.4,0.4])
for deg_dist, ax in zip([deg_dist1, deg_dist2], [ax1, ax2]):
    kave = sum(deg_dist1)*1./N
    tau = 2*gamma/kave
    Isum = scipy.zeros(tcount)

    for counter in range(iterations):
```

(continues on next page)

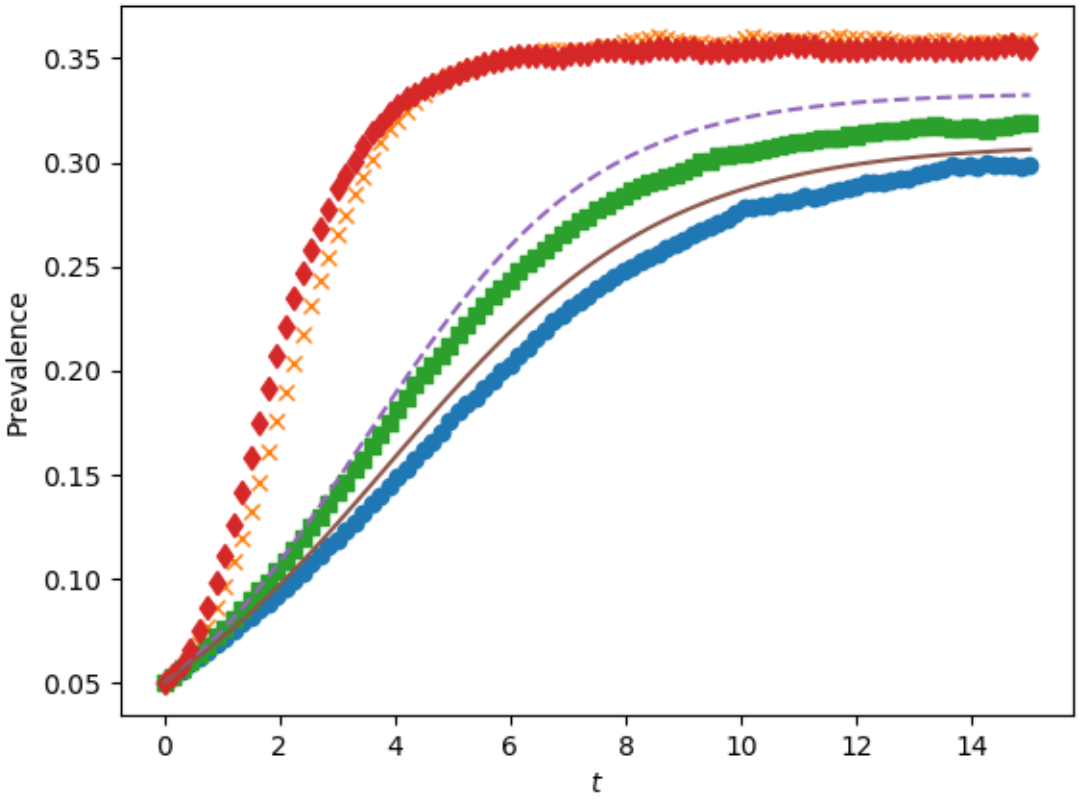
(continued from previous page)

```
        G = nx.configuration_model(deg_dist)
        t, S, I = EoN.fast_SIS(G, tau, gamma, tmax=tmax, rho=rho)
        I = I*1./N
        I = EoN.subsample(report_times, t, I)
        Isum += I
    ax.plot(report_times, Isum/iterations, color='grey', linewidth=5, alpha=0.3)

    S0 = (1-rho)*N
    I0 = rho*N

    t, S, I = EoN.SIS_homogeneous_meanfield(S0, I0, kave, tau, gamma, tmin=0,␣
↪tmax=tmax,
                                tcount=tcount)
    ax.plot(t, I/N, '--')

    SI0 = (1-rho)*N*kave*rho
    SS0 = (1-rho)*N*kave*(1-rho)
    t, S, I = EoN.SIS_homogeneous_pairwise(S0, I0, SI0, SS0, kave, tau, gamma, tmin =␣
↪0,
                                tmax=tmax, tcount=tcount)
    ax.plot(t, I/N)

ax1.set_xlabel('$t$')
ax1.set_ylabel('Prevalence')

plt.savefig('fig4p10.png')
```

Figure 4.11

Downloadable Source Code

- Note that the book has a typo. In fact $\tau = 1.5\gamma/<K>$




```
import EoN
import networkx as nx
import matplotlib.pyplot as plt
import scipy
import random

print(r"Warning, book says \tau=2\gamma/<K>, but it's really 1.5\gamma/<K>")
print(r"Warning - for the power law graph the text says k_{max}=110, but I believe it␣
↪is 118.")

N=1000
gamma = 1.
iterations = 200
rho = 0.05
tmax = 15
tcount = 101

kave = 20


tau = 1.5*gamma/kave


def simulate_process(graph_function, iterations, tmax, tcount, rho, kave, tau, gamma,␣
↪symbol):
```

(continues on next page)

(continued from previous page)

```
    Isum = scipy.zeros(tcount)
    report_times = scipy.linspace(0,tmax,tcount)
    for counter in range(iterations):
        G = graph_function()
        t, S, I = EoN.fast_SIS(G, tau, gamma, rho=rho, tmax=tmax)
        I = EoN.subsample(report_times, t, I)
        Isum += I
    plt.plot(report_times, Isum*1./(N*iterations), symbol)

#regular
symbol = 'o'
graph_function = lambda : nx.configuration_model(N*[kave])
simulate_process(graph_function, iterations, tmax, tcount, rho, kave, tau, gamma,␣
↪symbol)

#bimodal
symbol='x'
graph_function = lambda: nx.configuration_model([5,35]*int(N/2+0.01))
simulate_process(graph_function, iterations, tmax, tcount, rho, kave, tau, gamma,␣
↪symbol)

#erdos-renyi
symbol = 's'
graph_function = lambda : nx.fast_gnp_random_graph(N, kave/(N-1.))
simulate_process(graph_function, iterations, tmax, tcount, rho, kave, tau, gamma,␣
↪symbol)

symbol = 'd'
pl_kmax = 118
pl_kmin = 7
pl_alpha = 2.
Pk={}
for k in range(pl_kmin, pl_kmax+1):
    Pk[k] = k**(-pl_alpha)
valsum = sum(Pk.values())
for k in Pk.keys():
    Pk[k] /= valsum

#print sum(k*Pk[k] for k in Pk.keys())
def generate_sequence(Pk, N):
    while True:
        sequence = []
        for counter in range(N):
            r = random.random()
            for k in Pk.keys():
                if r< Pk[k]:
                    break
                else:
                    r-=Pk[k]
            sequence.append(k)
        if sum(sequence)%2==0:
            break
    return sequence

graph_function = lambda : nx.configuration_model(generate_sequence(Pk,N))
simulate_process(graph_function, iterations, tmax, tcount, rho, kave, tau, gamma,␣
↪symbol)
```

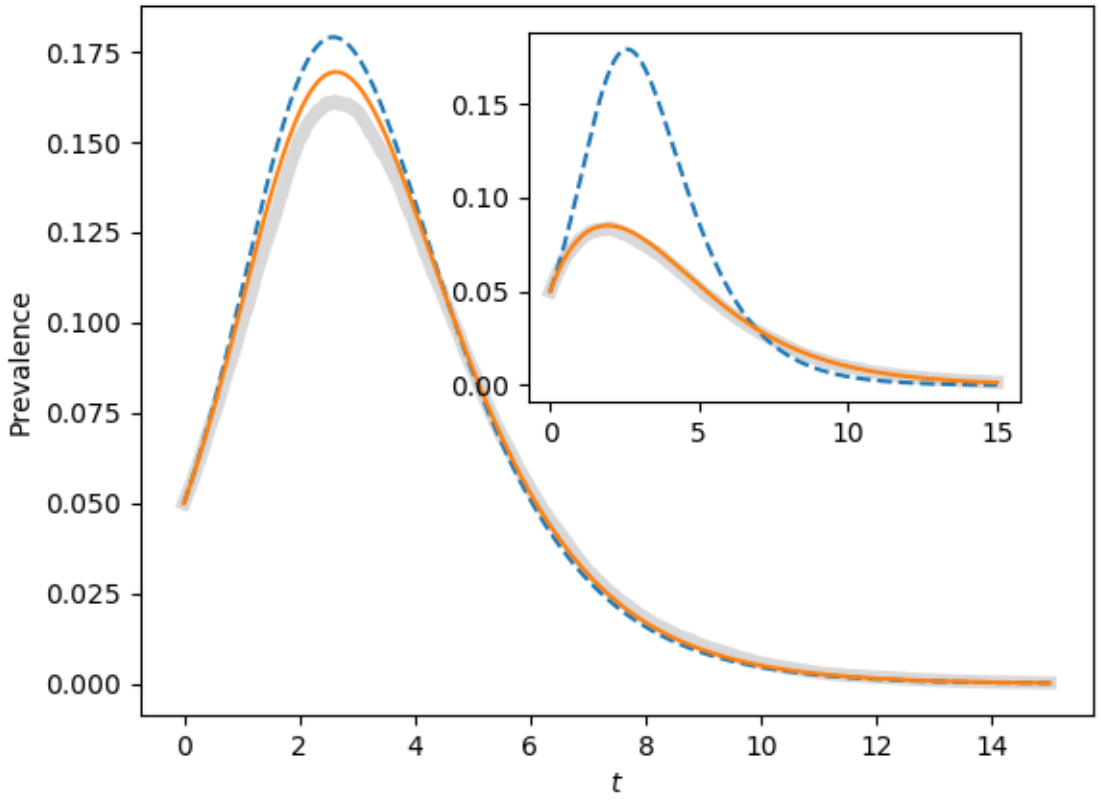
(continues on next page)

(continued from previous page)

```
symbol = '--'
S0 = (1-rho)*N
I0 = rho*N
t, S, I = EoN.SIS_homogeneous_meanfield(S0, I0, kave, tau, gamma, tmax=tmax,␣
↪tcount=tcount)
plt.plot(t, I/N, symbol)

symbol = '-'
S0 = (1-rho)*N
I0 = rho*N
SI0 = (1-rho)*N*kave*rho
SS0 = (1-rho)*N*kave*(1-rho)
t, S, I = EoN.SIS_homogeneous_pairwise(S0, I0, SI0, SS0, kave, tau, gamma, tmax=tmax,␣
↪tcount=tcount)
plt.plot(t, I/N, symbol)

plt.xlabel('$t$')
plt.ylavel('Prevalence')
plt.savefig('fig4p11.png')
```

Figure 4.12

Downloadable Source Code




```
import EoN
import networkx as nx
import matplotlib.pyplot as plt
import scipy

r'''
Reproduces figure 4.12

With N=1000, there is still significant stochasticity.  Some epidemics have earlier
peaks than others.  When many of these are averaged together, the final outcome
is that the average has a lower, broader peak than a typical epidemic.

Increasing N to 10000 will eliminate this.
'''

print("Often stochastic effects cause the peak for <K>=50 to be lower than predicted.
↪")
print("See comments in code for explanation")


N=1000
gamma = 1.
iterations = 200
rho = 0.05
tmax = 15
```

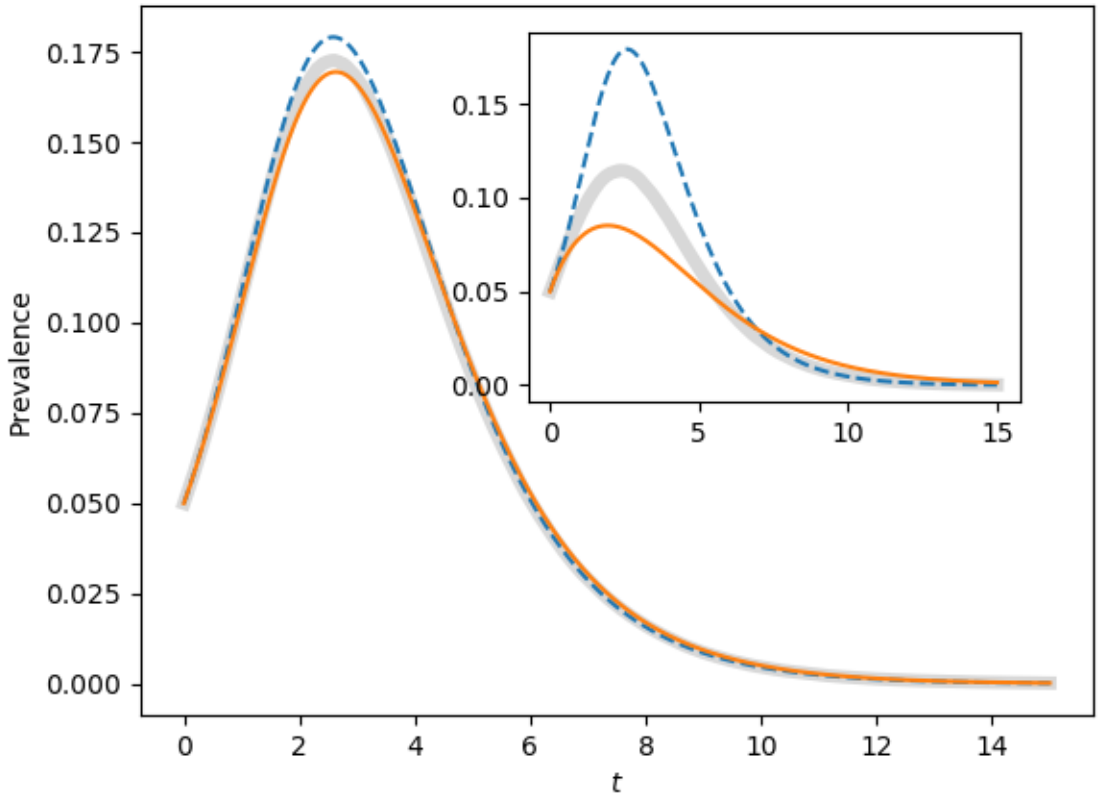
(continues on next page)

(continued from previous page)

```
tcount = 1001

report_times = scipy.linspace(0,tmax,tcount)
ax1 = plt.gca()#axes([0.1,0.1,0.9,0.9])
ax2 = plt.axes([0.44,0.45,0.4,0.4])

for kave, ax in zip((50, 5), (ax1, ax2)):
    tau = 2*gamma/kave
    Isum = scipy.zeros(tcount)

    for counter in range(iterations):
        G = nx.configuration_model(N*[kave])
        t, S, I, R = EoN.fast_SIR(G, tau, gamma, tmax=tmax, rho=rho)
        I = I*1./N
        I = EoN.subsample(report_times, t, I)
        Isum += I
    ax.plot(report_times, Isum/iterations, color='grey', linewidth=5, alpha=0.3)

    S0 = (1-rho)*N
    I0 = rho*N
    R0=0

    t, S, I, R = EoN.SIR_homogeneous_meanfield(S0, I0, R0, kave, tau, gamma,
                                               tmax=tmax, tcount=tcount)
    ax.plot(t, I/N, '--')

    SI0 = (1-rho)*N*kave*rho
    SS0 = (1-rho)*N*kave*(1-rho)
    t, S, I, R = EoN.SIR_homogeneous_pairwise(S0, I0, R0, SI0, SS0, kave, tau, gamma,
                                tmax=tmax, tcount=tcount)
    ax.plot(t, I/N)

ax1.set_xlabel('$t$')
ax1.set_ylabel('Prevalence')
plt.savefig('fig4p12.png')
```

Figure 4.13

Downloadable Source Code




```
import EoN
import networkx as nx
import matplotlib.pyplot as plt
import scipy

r'''
Reproduces figure 4.13

With N=1000, there is still significant stochasticity.  Some epidemics have earlier
peaks than others.  When many of these are averaged together, the final outcome
is that the average has a lower, broader peak than a typical epidemic.  In
this case it tends to make the Erdos-Renyi network for <K>=50 look like a better
fit for the homogeneous_pairwise model.

Increasing N to 10000 will eliminate this.
'''

print("Often stochastic effects cause the peak for <K>=50 to be lower than predicted.
↪")
print("See comments in code for explanation")



N=10000
gamma = 1.
```

(continues on next page)

(continued from previous page)

```
iterations = 200
rho = 0.05
tmax = 15
tcount = 1001

report_times = scipy.linspace(0,tmax,tcount)
ax1 = plt.gca()#axes([0.1,0.1,0.9,0.9])
ax2 = plt.axes([0.44,0.45,0.4,0.4])

for kave, ax in zip((50, 5), (ax1, ax2)):
    tau = 2*gamma/kave
    Isum = scipy.zeros(tcount)

    for counter in range(iterations):
        G = nx.fast_gnp_random_graph(N,kave/(N-1.))
        t, S, I, R = EoN.fast_SIR(G, tau, gamma, tmax=tmax, rho=rho)
        I = I*1./N
        I = EoN.subsample(report_times, t, I)
        Isum += I
    ax.plot(report_times, Isum/iterations, color='grey', linewidth=5, alpha=0.3)

    S0 = (1-rho)*N
    I0 = rho*N
    R0=0

    t, S, I, R = EoN.SIR_homogeneous_meanfield(S0, I0, R0, kave, tau, gamma,
                                               tmax=tmax, tcount=tcount)
    ax.plot(t, I/N, '--')

    SI0 = (1-rho)*N*kave*rho
    SS0 = (1-rho)*N*kave*(1-rho)
    t, S, I, R = EoN.SIR_homogeneous_pairwise(S0, I0, R0, SI0, SS0, kave, tau, gamma,
                                tmax=tmax, tcount=tcount)
    ax.plot(t, I/N)


ax1.set_xlabel('$t$')
ax1.set_ylabel('Prevalence')
plt.savefig('fig4p13.png')
```

Chapter 5

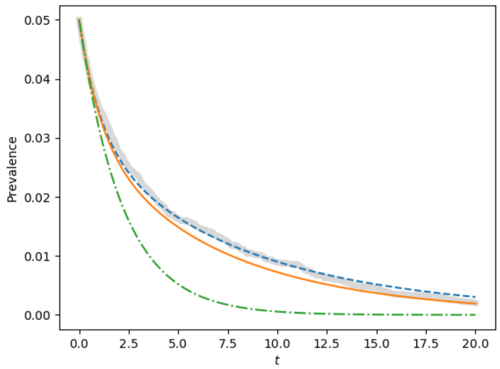
Heterogeneous meanfield

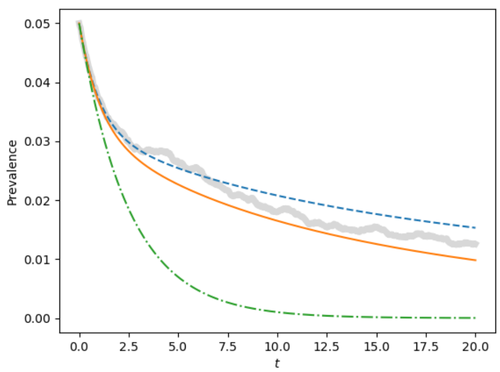
For Chapter 5 figures, these examples use larger populations than the figures in the text.

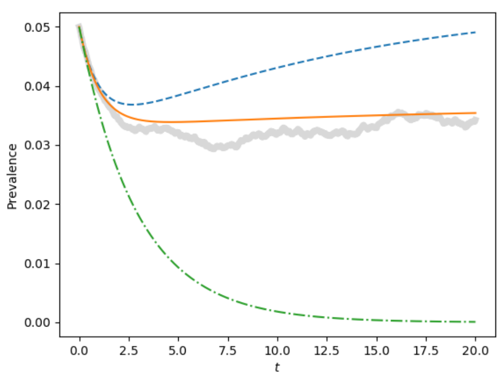
Figures 5.3, 5.4, and 5.5 demonstrate the ease of the X_from_graph versions of the analytic equations

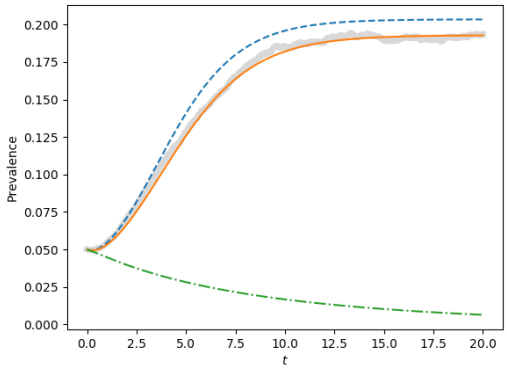
Figure 5.2 (a, b, c, and d)

- Note that the book has a typo. As with fig 4.7, for (c), $\tau = 1.1\tau_c$.
- It's worth looking at $1.2\tau_c$ as well. It's interesting.


Prevalence
0.05
0.04
0.03
0.02
0.01
0.00
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
t


Prevalence
0.05
0.04
0.03
0.02
0.01
0.00
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
t


Prevalence
0.05
0.04
0.03
0.02
0.01
0.00
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
t



Downloadable Source Code

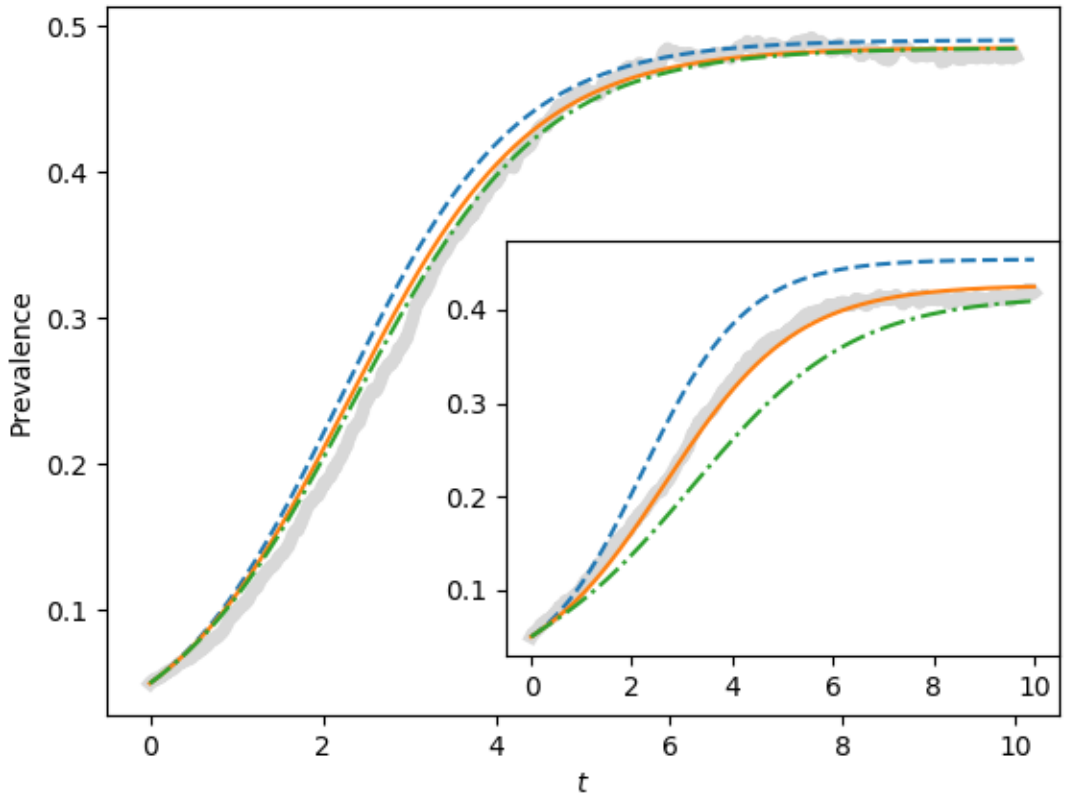
```
import EoN
import networkx as nx
import matplotlib.pyplot as plt
import scipy



N=100000  #100 times as large as the value given in the text
gamma = 1.
iterations = 1
rho = 0.05
tmax = 20
tcount = 1001
kave = 20.
ksqave = (5**2 + 35**2)/2.

tau_c = gamma*kave/ksqave

ksmall = 5
kbig = 35
deg_dist = [ksmall, kbig]*int(N/2)

report_times = scipy.linspace(0, tmax, tcount)

for tau, label in zip([0.9*tau_c, tau_c, 1.1*tau_c, 1.5*tau_c],['a', 'b', 'c', 'd']):
    print(str(tau_c)+" "+str(tau))
    plt.clf()

    Isum = scipy.zeros(tcount)
    for counter in range(iterations):
        G = nx.configuration_model(deg_dist)
        t, S, I = EoN.fast_SIS(G, tau, gamma, tmax=tmax, rho=rho)
        I = I*1./N
        I = EoN.subsample(report_times, t, I)
        Isum += I
    plt.plot(report_times, Isum/iterations, color='grey', linewidth=5, alpha=0.3)

    degree_array = scipy.zeros(kbig+1)
    degree_array[kbig]=N/2
    degree_array[ksmall]=N/2
    Sk0 = degree_array*(1-rho)
```

(continues on next page)

(continued from previous page)

```
    Ik0 = degree_array*rho

    t, S, I = EoN.SIS_heterogeneous_meanfield(Sk0, Ik0, tau, gamma, tmax=tmax,
                                                  tcount=tcount)
    plt.plot(t, I/N, '--')


    SI0 = ((kbig + ksmall)*N/2.)*(1-rho)*rho
    SS0 = ((kbig+ksmall)*N/2.)*(1-rho)*(1-rho)
    II0 = ((kbig+ksmall)*N/2.)*rho*rho

    t, S, I = EoN.SIS_compact_pairwise(Sk0, Ik0, SI0, SS0, II0, tau, gamma,
                                tmax=tmax, tcount=tcount)
    plt.plot(t, I/N)
    #t, S, I = EoN.SIS_compact_pairwise(Sk0, I0, SI0, SS0, II0, tau, gamma, tmax=tmax,
↪ tcount=tcount)
    #plt.plot(t, I/N)

    I0 = N*rho
    S0 = N*(1-rho)
    kave = (kbig+ksmall)/2.

    t, S, I = EoN.SIS_homogeneous_pairwise(S0, I0, SI0, SS0, kave, tau, gamma,␣
↪tmax=tmax, tcount=tcount)
    plt.plot(t, I/N, '-.')
    plt.xlabel('$t$')
    plt.ylabel('Prevalence')
    plt.savefig('fig5p2{}.png'.format(label))
```

Figure 5.3



Downloadable Source Code

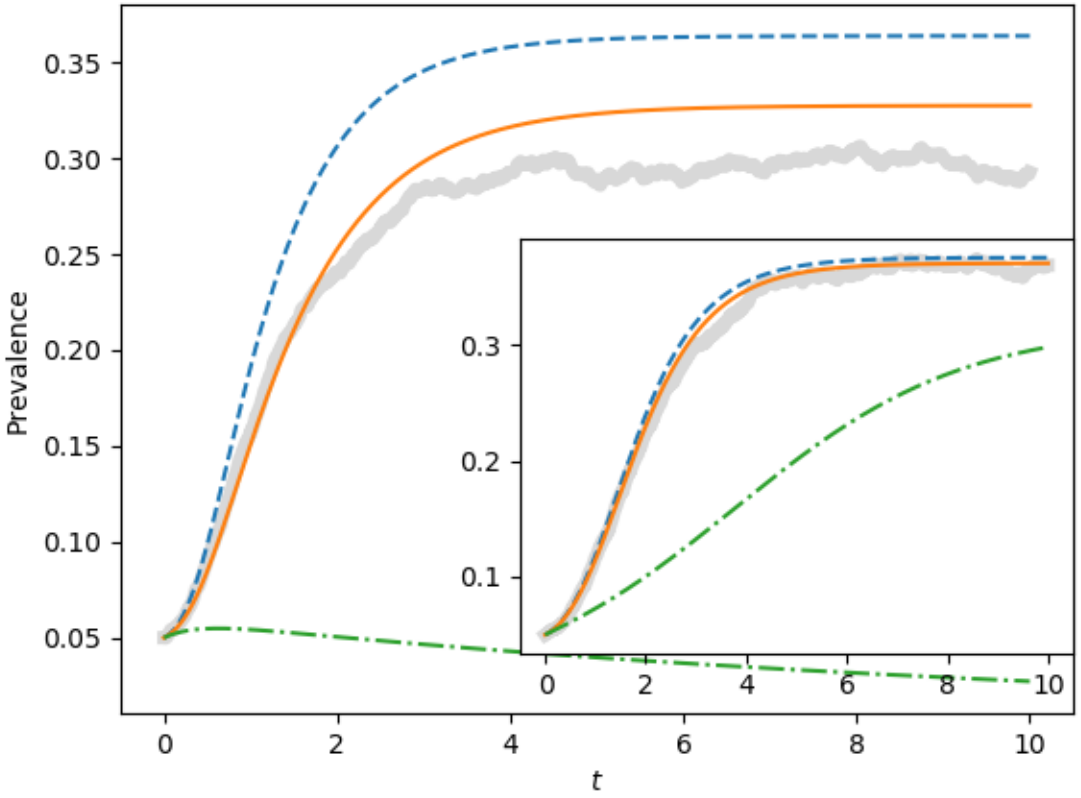
```
import EoN
import networkx as nx
import matplotlib.pyplot as plt
import scipy



def sim_and_plot(G, tau, gamma, rho, tmax, tcount, ax):
    t, S, I = EoN.fast_SIS(G, tau, gamma, rho=rho, tmax = tmax)
    report_times = scipy.linspace(0, tmax, tcount)
    I = EoN.subsample(report_times, t, I)
    ax.plot(report_times, I/N, color='grey', linewidth=5, alpha=0.3)

    t, S, I, = EoN.SIS_heterogeneous_meanfield_from_graph(G, tau, gamma, rho=rho,
                                                          tmax=tmax, tcount=tcount)
    ax.plot(t, I/N, '--')
    t, S, I = EoN.SIS_compact_pairwise_from_graph(G, tau, gamma, rho=rho,
                                                  tmax=tmax, tcount=tcount)
    ax.plot(t, I/N)

    t, S, I = EoN.SIS_homogeneous_pairwise_from_graph(G, tau, gamma, rho=rho,
```

(continues on next page)

(continued from previous page)

```
                                                    tmax=tmax, tcount=tcount)
    ax.plot(t, I/N, '-.')


tcount = 1001
tmax = 10.
gamma = 1.
N=10000
rho = 0.05

fig = plt.figure(1)
main = plt.axes()


target_kave = 50.
G = nx.fast_gnp_random_graph(N, target_kave/(N-1.))
kave = sum(G.degree(node) for node in G.nodes())
ksqave =sum(G.degree(node)**2 for node in G.nodes())
tau_c = gamma*kave/ksqave
tau = 2*tau_c

sim_and_plot(G, tau, gamma, rho, tmax, tcount, main)


inset = plt.axes([0.45,0.175,0.45,0.45])
target_kave = 10.

G = nx.fast_gnp_random_graph(N, target_kave/(N-1.))
kave = sum(G.degree(node) for node in G.nodes())
ksqave =sum(G.degree(node)**2 for node in G.nodes())
tau_c = gamma*kave/ksqave
tau = 2*tau_c

sim_and_plot(G, tau, gamma, rho, tmax, tcount, inset)

main.set_xlabel('$t$')
main.set_ylabel('Prevalence')
plt.savefig('fig5p3.png')
```

Figure 5.4



Downloadable Source Code

```
import EoN
import networkx as nx
import matplotlib.pyplot as plt
import scipy
import random


def get_deg_seq(N, Pk):
    while True:  #run until degree sequence has even sum of N entries
        deg_seq = []
        for counter in range(N):
            r = random.random()
            for k in Pk:
                if Pk[k]>r:
                    break
                else:
                    r-= Pk[k]
            deg_seq.append(k)
        if sum(deg_seq)%2 ==0:
            break
    return deg_seq
```

(continues on next page)

(continued from previous page)

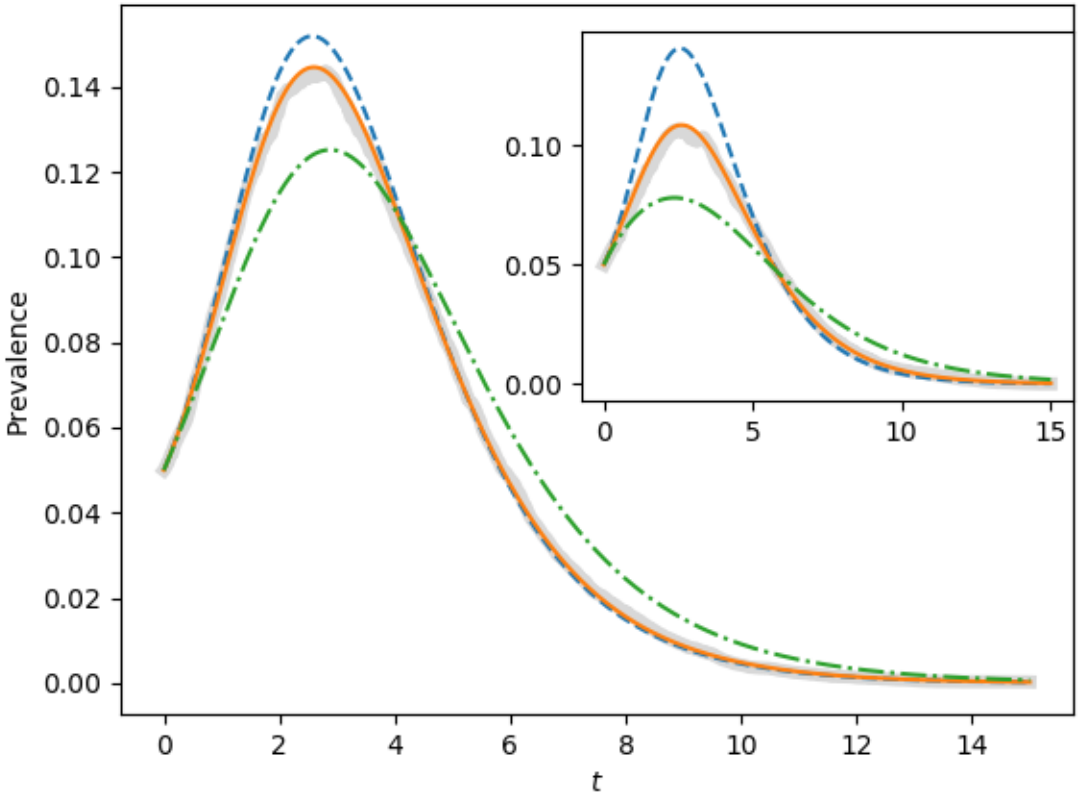
```
def sim_and_plot(G, tau, gamma, rho, tmax, tcount, ax):
    t, S, I = EoN.fast_SIS(G, tau, gamma, rho = rho, tmax = tmax)
    report_times = scipy.linspace(0, tmax, tcount)
    I = EoN.subsample(report_times, t, I)
    ax.plot(report_times, I/N, color='grey', linewidth=5, alpha=0.3)

    t, S, I, = EoN.SIS_heterogeneous_meanfield_from_graph(G, tau, gamma, rho=rho,
                                                          tmax=tmax, tcount=tcount)
    ax.plot(t, I/N, '--')
    t, S, I = EoN.SIS_compact_pairwise_from_graph(G, tau, gamma, rho=rho,
                                                  tmax=tmax, tcount=tcount)
    ax.plot(t, I/N)

    t, S, I = EoN.SIS_homogeneous_pairwise_from_graph(G, tau, gamma, rho=rho,
                                                      tmax=tmax, tcount=tcount)
    ax.plot(t, I/N, '-.')

N=10000
gamma = 1
rho = 0.05
tmax = 10
tcount = 1001

kmin = 1
kmax = 40
Pk = {}
for k in range(kmin, kmax+1):
    Pk[k] = k**(-2.)
norm_factor = sum(Pk.values())
for k in Pk:
    Pk[k] /= norm_factor

deg_seq = get_deg_seq(N, Pk)
G = nx.configuration_model(deg_seq)
kave = sum(deg_seq)/N

tau = 1.5*gamma/kave

fig = plt.figure(1)
main = plt.axes()
sim_and_plot(G, tau, gamma, rho, tmax, tcount, main)


kmin = 10
kmax = 150
Pk = {}
for k in range(kmin, kmax+1):
    Pk[k] = k**(-2.)
norm_factor = sum(Pk.values())
for k in Pk:
    Pk[k] /= norm_factor


deg_seq = get_deg_seq(N, Pk)
```

(continues on next page)

(continued from previous page)

```
G = nx.configuration_model(deg_seq)
kave = (sum(deg_seq)/N)

tau = 1.5*gamma/kave

fig = plt.figure(1)
ax1 = plt.gca()
inset = plt.axes([0.45,0.175,0.45,0.45])
sim_and_plot(G, tau, gamma, rho, tmax, tcount, inset)

ax1.set_xlabel('$t$')
ax1.set_ylabel('Prevalence')
plt.savefig('fig5p4.png')
```

Figure 5.5



Downloadable Source Code

```
import EoN
import networkx as nx
import matplotlib.pyplot as plt
import scipy
```

(continues on next page)

(continued from previous page)

```
import random



def sim_and_plot(G, tau, gamma, rho, tmax, tcount, ax):
    t, S, I, R= EoN.fast_SIR(G, tau, gamma, rho = rho, tmax = tmax)
    report_times = scipy.linspace(0, tmax, tcount)
    I = EoN.subsample(report_times, t, I)
    ax.plot(report_times, I/N, color='grey', linewidth=5, alpha=0.3)

    t, S, I, R = EoN.SIR_heterogeneous_meanfield_from_graph(G, tau, gamma, rho=rho,
                                                            tmax=tmax, tcount=tcount)
    ax.plot(t, I/N, '--')
    t, S, I, R = EoN.SIR_compact_pairwise_from_graph(G, tau, gamma, rho=rho,
                                                            tmax=tmax, tcount=tcount)
    ax.plot(t, I/N)

    t, S, I, R = EoN.SIR_homogeneous_pairwise_from_graph(G, tau, gamma, rho=rho,
                                                            tmax=tmax, tcount=tcount)
    ax.plot(t, I/N, '-.')

N=50000
gamma = 1
rho = 0.05
tmax = 15
tcount = 1001

deg_seq = [30]*int(N/2) + [70]*int(N/2)
G = nx.configuration_model(deg_seq)
kave = sum(deg_seq)/N
ksqave = sum(k*k for k in deg_seq)/N

tau= 2*gamma*kave/ksqave

fig = plt.figure(1)
main = plt.axes()
sim_and_plot(G, tau, gamma, rho, tmax, tcount, main)
plt.ylabel('Prevalence')


deg_seq = [5]*int(N/2) + [15]*int(N/2)
G = nx.configuration_model(deg_seq)

kave = (sum(deg_seq)/N)
ksqave = sum(k*k for k in deg_seq)/N

tau= 2*gamma*kave/ksqave

fig = plt.figure(1)
ax1 = plt.gca()
inset = plt.axes([0.5,0.45,0.4,0.4])
sim_and_plot(G, tau, gamma, rho, tmax, tcount, inset)

ax1.set_xlabel('$t$')
ax1.set_ylabel('Prevalence')

plt.savefig('fig5p5.png')
```

Chapter 6

Percolation and EBCM

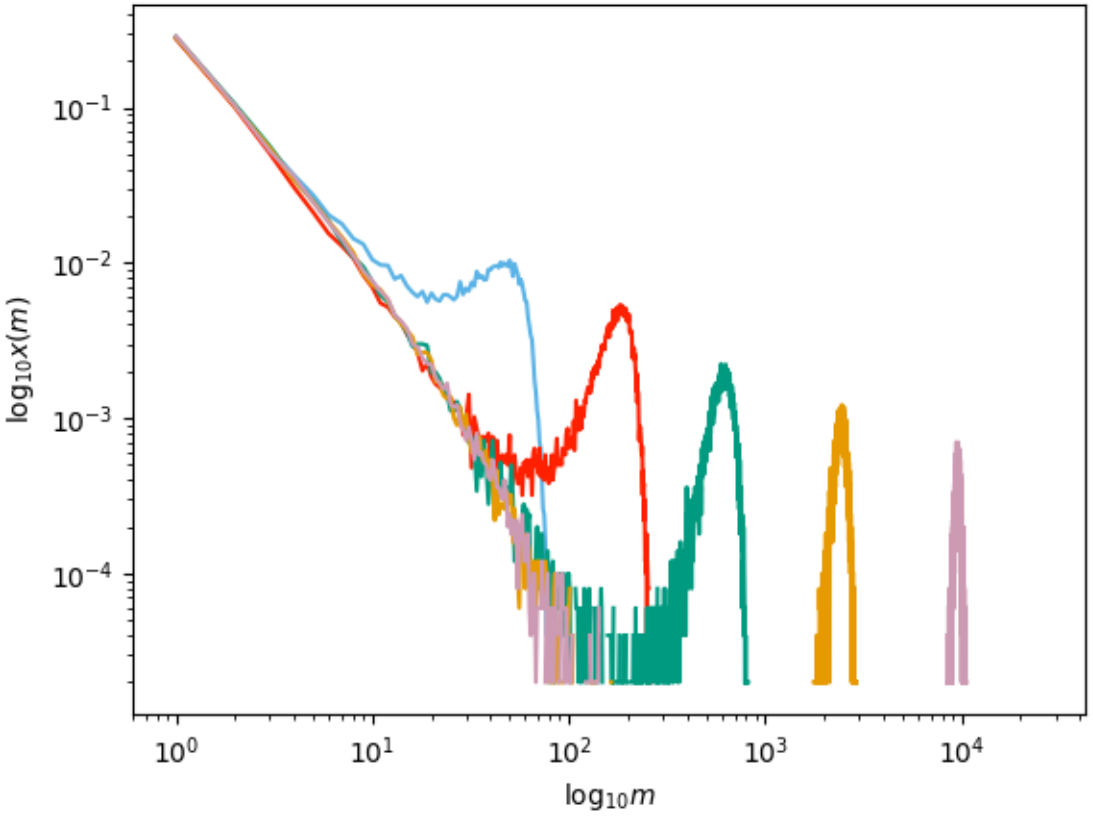
The remainder of these simulations use reduced sizes or numbers of iterations compared to the published figure. This is to save time.

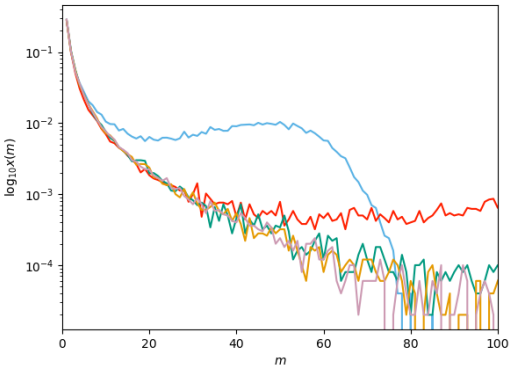
Figure 6.1 (a, b, and c) and 6.3 (a, b, c, d, and e)

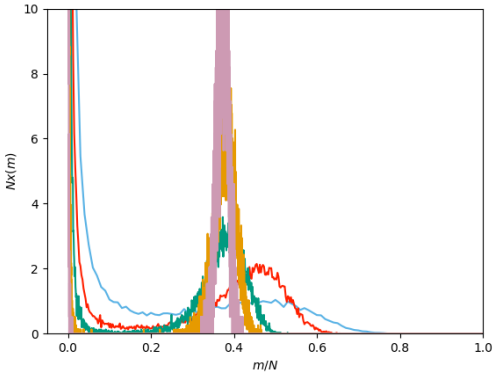
`Downloadable Source Code`

This code does both 6.1 and 6.3 since they use the same data. It produces an additional figure not included in the text for 6.3.

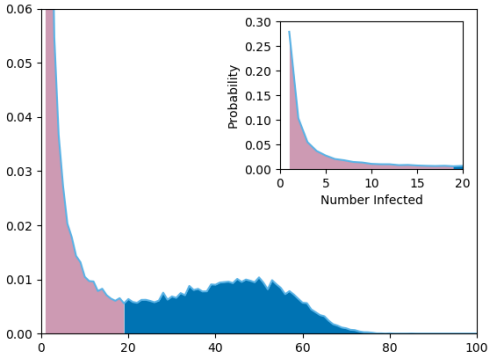
6.1

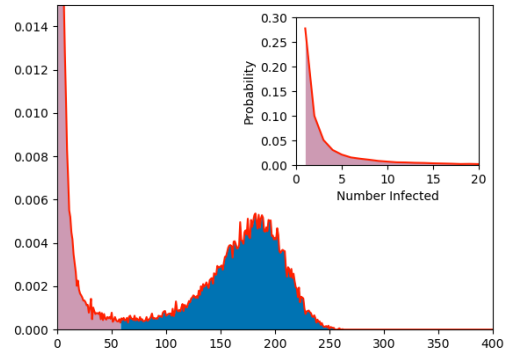


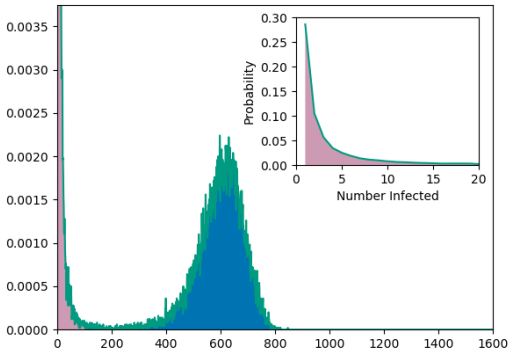

10^{-1}
10^{-2}
10^{-3}
10^{-4}
$\log_{10}x(m)$
0
20
40
60
80
100
m

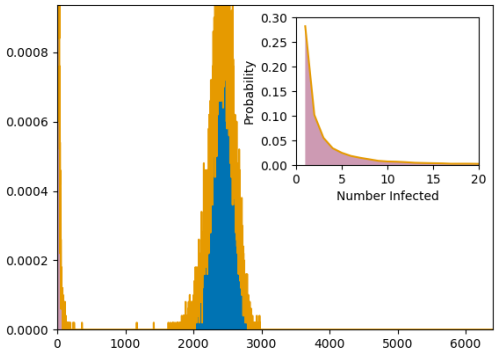

10
8
6
4
2
0
$Nx(m)$
0.0
0.2
0.4
0.6
0.8
1.0
m/N

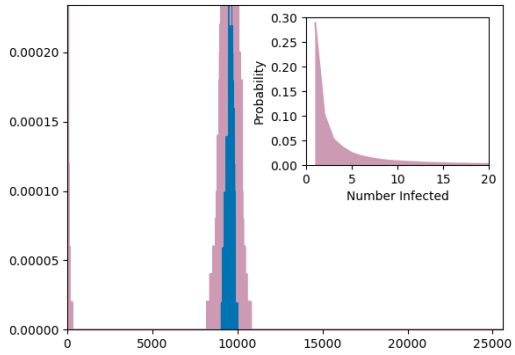
6.3


0.06
0.05
0.04
0.03
0.02
0.01
0.00
0
20
40
60
80
100
0.30
0.25
0.20
0.15
0.10
0.05
0.00
Probability
0
5
10
15
20
Number Infected


Probability
Number Infected


Probability
Number Infected


Probability
Number Infected




```
import networkx as nx
import EoN
from collections import defaultdict
import matplotlib.pyplot as plt
import scipy

colors = ['#5AB3E6','#FF2000','#009A80','#E69A00', '#CD9AB3', '#0073B3','#F0E442']



def getMs(counts):
    r'''used for figure 6.3 to get the values of M1, Mstar, and M2'''
    N=len(counts)
    M1 = 0
    val1 = 0
    M2 = 0
    val2=0
    Mstar = 0
    valstar = 1
    for index, val in enumerate(counts):
        if index<2:
            continue
        if val < valstar:
            Mstar = index
            valstar = val
        elif index - Mstar > 0.1*N:
            break
    for index, val in enumerate(counts):
        if index>Mstar:
            break
        elif val>val1:
            val1=val
            M1 = index
    for index, val in enumerate(counts):
        if index < Mstar:
            continue
        elif val > val2:
            val2 = val
            M2 = index
    return M1, Mstar, M2


iterations = 5*10**4
```

(continues on next page)

(continued from previous page)

```
p=0.25
kave = 5.
labels=['a', 'b', 'c', 'd', 'e']

for N, color, label in zip([100, 400, 1600, 6400, 25600], colors, labels):
    print(N)
    xm = {m:0 for m in range(1,N+1)}
    G = nx.fast_gnp_random_graph(N, kave/(N-1.))
    for counter in range(iterations):
        t, S, I, R = EoN.basic_discrete_SIR_epidemic(G, p)
        xm[R[-1]] += 1./iterations
    items = sorted(xm.items())
    m, freq = zip(*items)

    plt.figure(1)
    plt.loglog(m, freq, color = color)

    plt.figure(2)
    plt.plot(m, freq, color=color)
    plt.yscale('log')

    freq = scipy.array(freq)
    m= scipy.array(m)
    plt.figure(3)
    plt.plot(m/float(N), N*freq, color = color)  #float is required in case python 2.X

    M1, Mstar, M2 = getMs(freq)
    plt.figure(4)
    plt.clf()
    plt.axis(xmin = 0,xmax = N, ymax=6./(N), ymin = 0)
    plt.plot(m, freq, color= color)
    plt.fill_between(range(1,Mstar+2), 0, freq[0:Mstar+1], linewidth=0, color =␣
↪colors[4])
    plt.fill_between(range(Mstar+1,len(freq)+1), 0, freq[Mstar:], linewidth=0, color␣
↪= colors[5])
    inset = plt.axes([0.55,0.5,0.325,0.35])
    inset.plot(m, freq, color= color)
    inset.fill_between(range(1,Mstar+2), 0, freq[0:Mstar+1], linewidth=0, color =␣
↪colors[4])
    inset.fill_between(range(Mstar+1,len(freq)), 0, freq[Mstar+1:], linewidth=0,␣
↪color = colors[5])
    inset.axis(xmin=0., xmax=20, ymin=0, ymax = 0.3)#, ymin=-counts[0]*iterations/100)
    inset.set_xticks([0,5,10,15,20])
    plt.xlabel('Number Infected')
    plt.ylabel('Probability')
    plt.savefig('fig6p3{}.png'.format(label))



plt.figure(1)
plt.ylabel(r'$\log_{10} x(m)$')
plt.xlabel(r'$\log_{10} m$')
plt.savefig('fig6p1a.png')

plt.figure(2)
plt.xlabel('$m$')
plt.ylabel('$\log_{10} x(m)$')
```

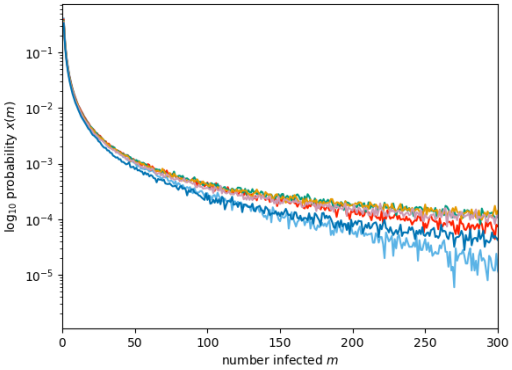
(continues on next page)

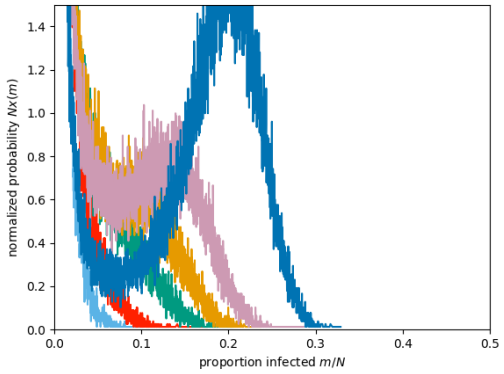
(continued from previous page)

```
plt.axis(xmin = 0, xmax = 100)
plt.savefig('fig6p1b.png')

plt.figure(3)
plt.xlabel('$m/N$')
plt.ylabel('$Nx(m)$')
plt.axis(ymax=10, xmax=1, ymin=0)
plt.savefig('fig6p1c.png')
```

Figure 6.2 (a and b)

Downloadable Source Code






```
import networkx as nx
import EoN
from collections import defaultdict
import matplotlib.pyplot as plt
import scipy

colors = ['#5AB3E6','#FF2000','#009A80','#E69A00', '#CD9AB3', '#0073B3','#F0E442']

iterations = 5*10**5
```

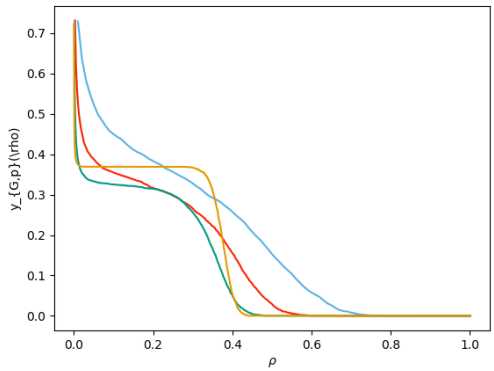
(continues on next page)

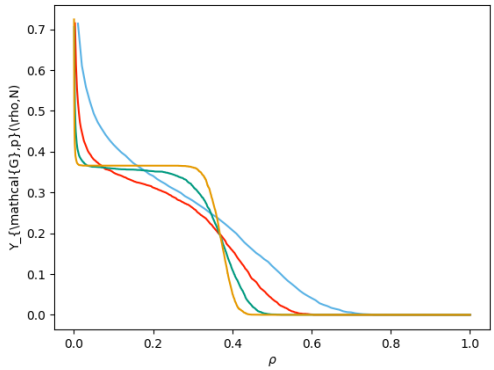
(continued from previous page)

```
N=6400
kave = 5
G = nx.fast_gnp_random_graph(N, kave/(N-1.))

for index, p in enumerate([0.18, 0.19, 0.2, 0.205, 0.21, 0.22]):
    print(p)
    xm = defaultdict(int)
    for counter in range(iterations):
        t, S, I, R = EoN.basic_discrete_SIR_epidemic(G, p)
        xm[R[-1]] += 1./iterations
    items = sorted(xm.items())
    m, freq = zip(*items)


    plt.figure(1)
    plt.plot(m, freq, color=colors[index])
    plt.yscale('log')

    freq = scipy.array(freq)
    m= scipy.array(m)
    plt.figure(2)
    plt.plot()
    plt.plot(m/N, N*freq, color = colors[index])

plt.figure(1)
plt.axis(xmin=0, xmax=300)
plt.xlabel('number infected $m$')
plt.ylabel(r'$\log_{10}$ probability $x(m)$')
plt.savefig('fig6p2a.png')

plt.figure(2)
plt.axis(xmin=0,xmax=0.5, ymax = 1.5, ymin=0)
plt.xlabel('proportion infected $m/N$')
plt.ylabel(r'normalized probability $Nx(m)$')
plt.savefig('fig6p2b.png')
```

Figure 6.4 (a and b)

Downloadable Source Code






```
import networkx as nx
import EoN
from collections import defaultdict
import matplotlib.pyplot as plt
import scipy

colors = ['#5AB3E6','#FF2000','#009A80','#E69A00', '#CD9AB3', '#0073B3','#F0E442']


iterations = 5*10**3
p=0.25
kave = 5.
Ns = [100, 400, 1600, 6400]#, 25600]
for index, N in enumerate(Ns):
    r'''First we do it with the same network for each iteration'''
    print(N)
    xm = {m:0 for m in range(1,N+1)}
    G = nx.fast_gnp_random_graph(N, kave/(N-1.))
    for counter in range(iterations):
        t, S, I, R = EoN.basic_discrete_SIR_epidemic(G, p)
        xm[R[-1]] += 1./iterations
    items = sorted(xm.items())
    m, freq = zip(*items)

    freq = scipy.array(freq)
    m= scipy.array(m)

    cum_freq = scipy.cumsum(freq)
    plt.figure(1)
    plt.plot(m/N, 1-cum_freq, color = colors[index])

plt.figure(1)
plt.xlabel(r'$\rho$')
plt.ylabel(r'y_{G,p}(\rho)')
plt.savefig('fig6p4a.png')

for index, N in enumerate(Ns):
    '''Now we generate a new network for each iteration'''
    print(N)
    xm = {m:0 for m in range(1,N+1)}
    for counter in range(iterations):
        G = nx.fast_gnp_random_graph(N, kave/(N-1.))
        t, S, I, R = EoN.basic_discrete_SIR_epidemic(G, p)
```

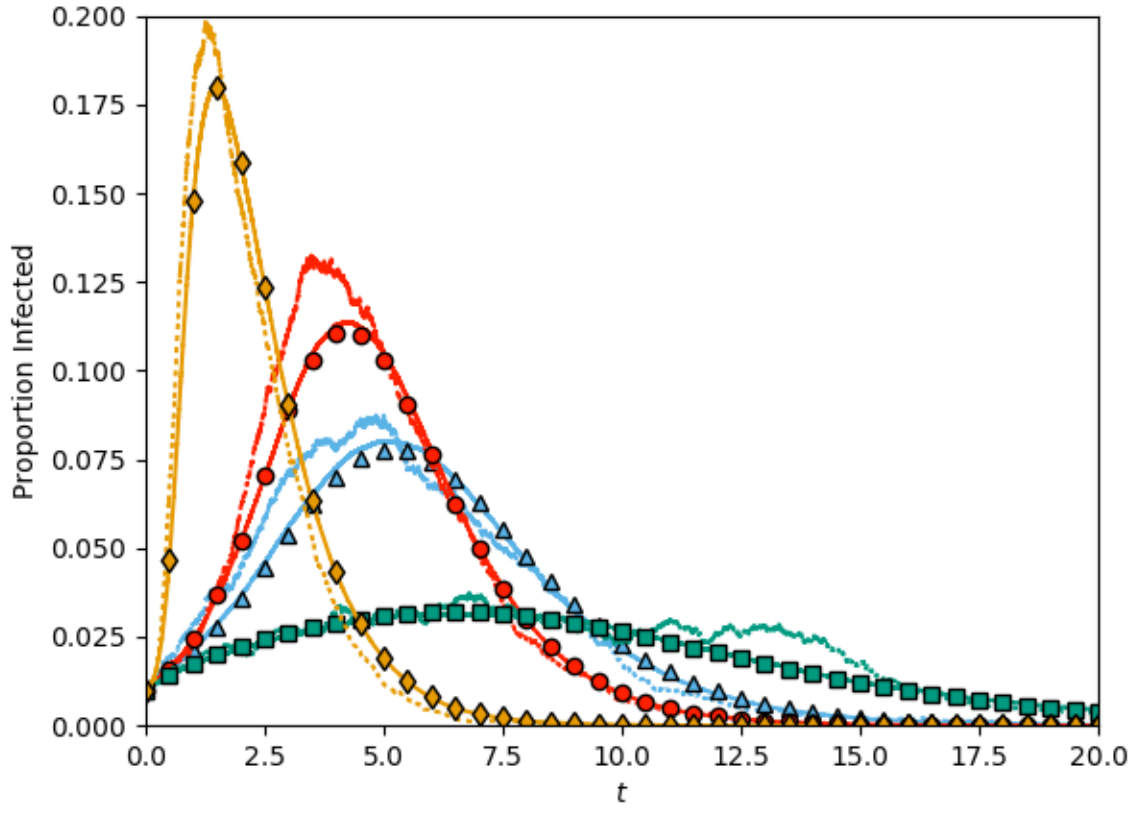
(continues on next page)

(continued from previous page)

```
        xm[R[-1]] += 1./iterations
    items = sorted(xm.items())
    m, freq = zip(*items)

    freq = scipy.array(freq)
    m= scipy.array(m)

    cum_freq = scipy.cumsum(freq)
    plt.figure(2)
    plt.plot(m/N, 1-cum_freq, color = colors[index])

plt.figure(2)
plt.xlabel(r'$\rho$')
plt.ylabel(r'Y_{\mathcal{G},p}(\rho,N)')
plt.savefig('fig6p4b.png')
```

Figure 6.24

Downloadable Source Code




```
import networkx as nx
import EoN
from collections import defaultdict
```

(continues on next page)

(continued from previous page)

```
import matplotlib.pyplot as plt
import scipy
import random

colors = ['#5AB3E6','#FF2000','#009A80','#E69A00', '#CD9AB3', '#0073B3','#F0E442']
rho = 0.01
Nbig=500000
Nsmall = 5000
tau =0.4
gamma = 1.

def poisson():
    return scipy.random.poisson(5)
def PsiPoisson(x):
    return scipy.exp(-5*(1-x))
def DPsiPoisson(x):
    return 5*scipy.exp(-5*(1-x))

bimodalPk = {8:0.5, 2:0.5}
def PsiBimodal(x):
    return (x**8 +x**2)/2.
def DPsiBimodal(x):
    return(8*x**7 + 2*x)/2.

def homogeneous():
    return 5
def PsiHomogeneous(x):
    return x**5
def DPsiHomogeneous(x):
    return 5*x**4


PlPk = {}
exponent = 1.418184432
kave = 0
for k in range(1,81):
    PlPk[k]=k**(-exponent)*scipy.exp(-k*1./40)
    kave += k*PlPk[k]

normfact= sum(PlPk.values())
for k in PlPk:
    PlPk[k] /= normfact

#def trunc_pow_law():
#    r = random.random()
#    for k in PlPk:
#        r -= PlPk[k]
#        if r<0:
#            return k


def PsiPowLaw(x):
    #print PlPk
    rval = 0
    for k in PlPk:
        rval += PlPk[k]*x**k
    return rval
```

(continues on next page)

(continued from previous page)

```
def DPsiPowLaw(x):
    rval = 0
    for k in PlPk:
        rval += k*PlPk[k]*x**(k-1)
    return rval


def get_G(N, Pk):
    while True:
        ks = []
        for ctr in range(N):
            r = random.random()
            for k in Pk:
                if r<Pk[k]:
                    break
                else:
                    r-= Pk[k]
            ks.append(k)
        if sum(ks)%2==0:
            break
    G = nx.configuration_model(ks)
    return G


report_times = scipy.linspace(0,20,41)


def process_degree_distribution(Gbig, Gsmall, color, Psi, DPsi, symbol):
    t, S, I, R = EoN.fast_SIR(Gsmall, tau, gamma, rho=rho)
    plt.plot(t, I*1./Gsmall.order(), ':', color = color)
    t, S, I, R = EoN.fast_SIR(Gbig, tau, gamma, rho=rho)
    plt.plot(t, I*1./Gbig.order(), color = color)
    N= Gbig.order()#N is arbitrary, but included because our implementation of EBCM␣
↪assumes N is given.
    t, S, I, R = EoN.EBCM(N, lambda x: (1-rho)*Psi(x), lambda x: (1-rho)*DPsi(x), tau,
↪ gamma, 1-rho)
    I = EoN.subsample(report_times, t, I)
    plt.plot(report_times, I/N, symbol, color = color, markeredgecolor='k')

#Erdos Renyi
Gsmall = nx.fast_gnp_random_graph(Nsmall, 5./(Nsmall-1))
Gbig = nx.fast_gnp_random_graph(Nbig, 5./(Nbig-1))
process_degree_distribution(Gbig, Gsmall, colors[0], PsiPoisson, DPsiPoisson, '^')

#Bimodal
Gsmall = get_G(Nsmall, bimodalPk)
Gbig = get_G(Nbig, bimodalPk)
process_degree_distribution(Gbig, Gsmall, colors[1], PsiBimodal, DPsiBimodal, 'o')

#Homogeneous
Gsmall = get_G(Nsmall, {5:1.})
Gbig = get_G(Nbig, {5:1.})
process_degree_distribution(Gbig, Gsmall, colors[2], PsiHomogeneous, DPsiHomogeneous,
↪'s')

#Powerlaw
```

(continues on next page)

(continued from previous page)

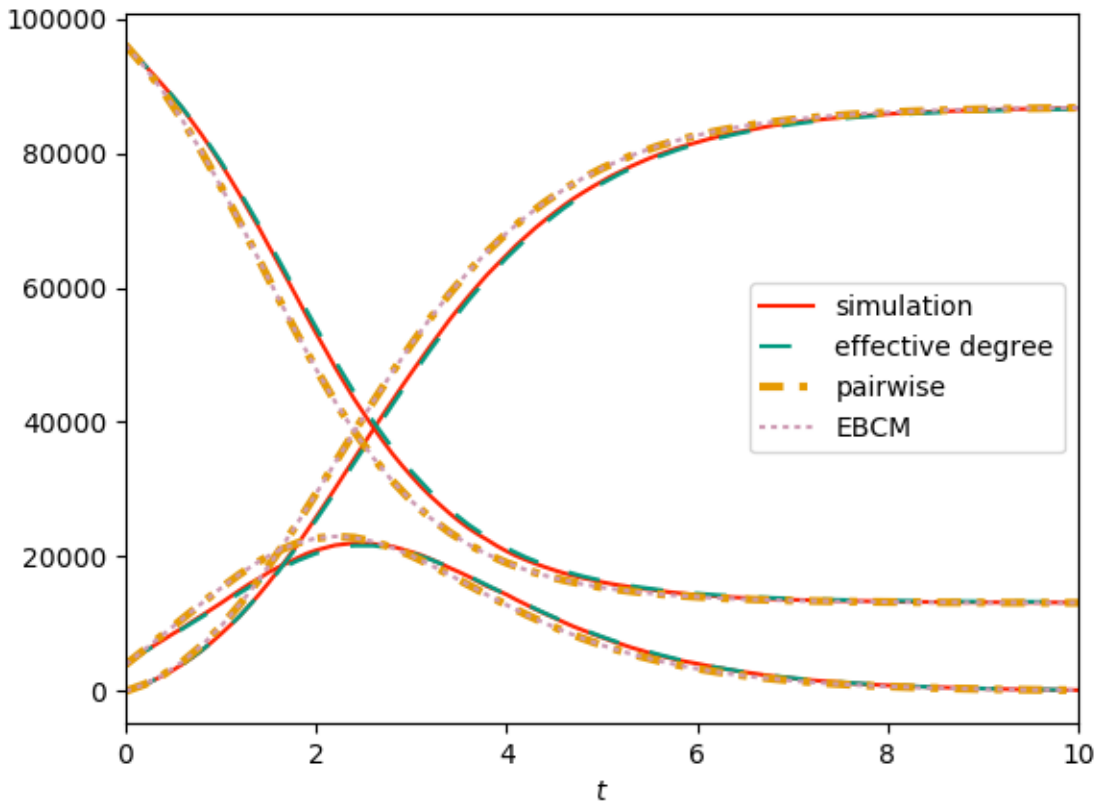
```
Gsmall = get_G(Nsmall, PlPk)
Gbig = get_G(Nbig, PlPk)
process_degree_distribution(Gbig, Gsmall, colors[3], PsiPowLaw, DPsiPowLaw, 'd')


plt.axis(xmin=0, ymin=0, xmax = 20, ymax = 0.2)
plt.xlabel('$t$')
plt.ylabel('Proportion Infected')
plt.savefig('fig6p24.png')
```

Chapter 7

Model hierarchies

Figure 7.2

Downloadable Source Code




```
import networkx as nx
import EoN
from collections import defaultdict
```

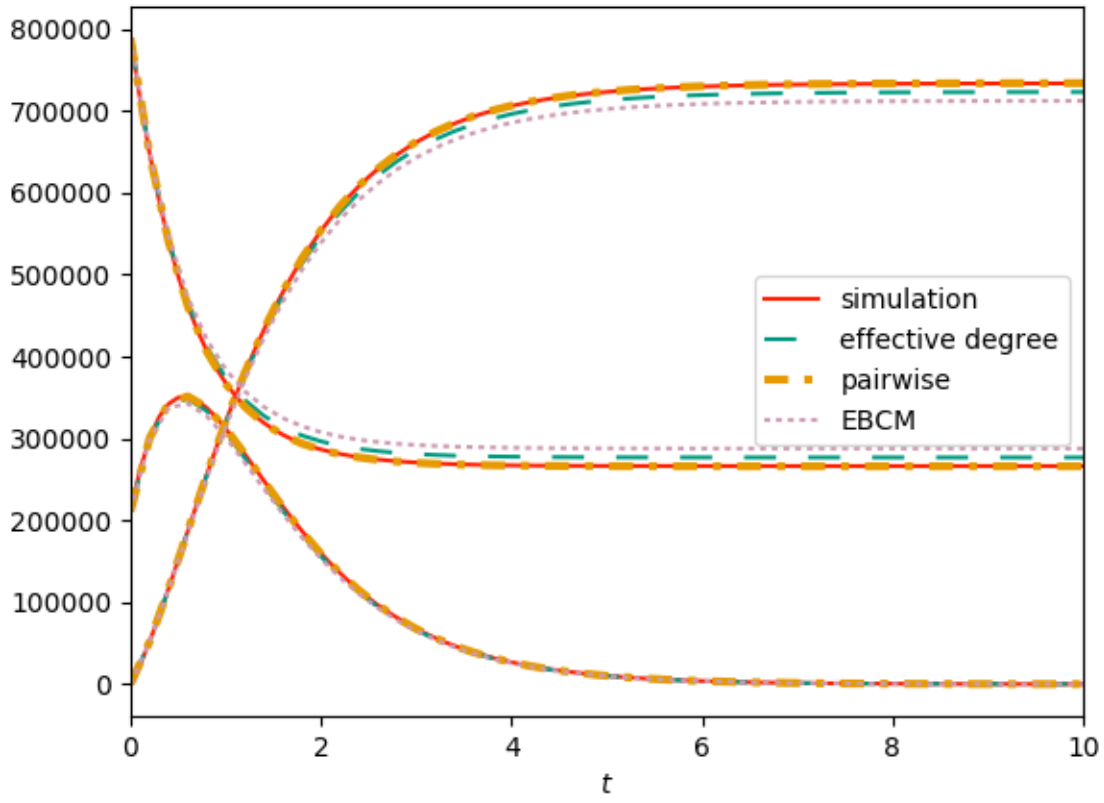
(continues on next page)

(continued from previous page)

```
import matplotlib.pyplot as plt
import scipy
import random

tau = 1.
gamma = 1.
N=10**5
colors = ['#5AB3E6','#FF2000','#009A80','#E69A00', '#CD9AB3', '#0073B3','#F0E442']

print('setting up')
G = nx.configuration_model([4]*N)

chosen = random.sample(range(N),int(0.01*N))

initial_infecteds = set()

for node in chosen:
    for nbr in G.neighbors(node):
        initial_infecteds.add(nbr)

print('simulating')
t, S, I, R = EoN.fast_SIR(G, tau, gamma, initial_infecteds = initial_infecteds)
report_times = scipy.linspace(0,10,101)

S, I, R = EoN.subsample(report_times, t, S, I, R)

plt.plot(report_times, S, color =colors[1], label = 'simulation')
plt.plot(report_times, I, color = colors[1])
plt.plot(report_times, R, color = colors[1])

print('doing ODE models')
t, S, I, R = EoN.SIR_effective_degree_from_graph(G, tau, gamma, initial_
↪infecteds=initial_infecteds, tmax = 10, tcount = 51)
plt.plot(t,S, color = colors[2], dashes = [6,6], label = 'effective degree')
plt.plot(t,I, color = colors[2], dashes = [6,6])
plt.plot(t,R, color = colors[2], dashes = [6,6])

t, S, I, R = EoN.SIR_heterogeneous_pairwise_from_graph(G, tau, gamma, initial_
↪infecteds=initial_infecteds, tmax = 10, tcount = 51)
plt.plot(t, S, color = colors[3],  dashes = [3,2,1,2], linewidth=3, label = 'pairwise
↪')
plt.plot(t, I, color = colors[3],  dashes = [3,2,1,2], linewidth=3)
plt.plot(t, R, color = colors[3],  dashes = [3,2,1,2], linewidth=3)  #, dashes = [6,3,
↪2,3]


t, S, I, R = EoN.EBCM_from_graph(G, tau, gamma, initial_infecteds=initial_infecteds,␣
↪tmax = 10, tcount =51)
plt.plot(t, S, ':', color = colors[4], label = 'EBCM')
plt.plot(t, I, ':', color = colors[4])
plt.plot(t, R, ':', color = colors[4])

plt.axis(xmax=10, xmin=0)
plt.xlabel('$t$')
plt.legend(loc = 'center right')
plt.savefig('fig7p2.png')
```

Figure 7.3

Downloadable Source Code




```
import networkx as nx
import EoN
from collections import defaultdict
import matplotlib.pyplot as plt
import scipy
import random

N=10**6
tau = 1.
gamma = 1.
colors = ['#5AB3E6','#FF2000','#009A80','#E69A00', '#CD9AB3', '#0073B3','#F0E442']

print('setting up --- requires a large network to make clear that effect is real, not␣
↪noise.')
deg_seq = []
for counter in range(N):
    if random.random()< 1./6:
        deg_seq.append(10)
    else:
        deg_seq.append(2)
```

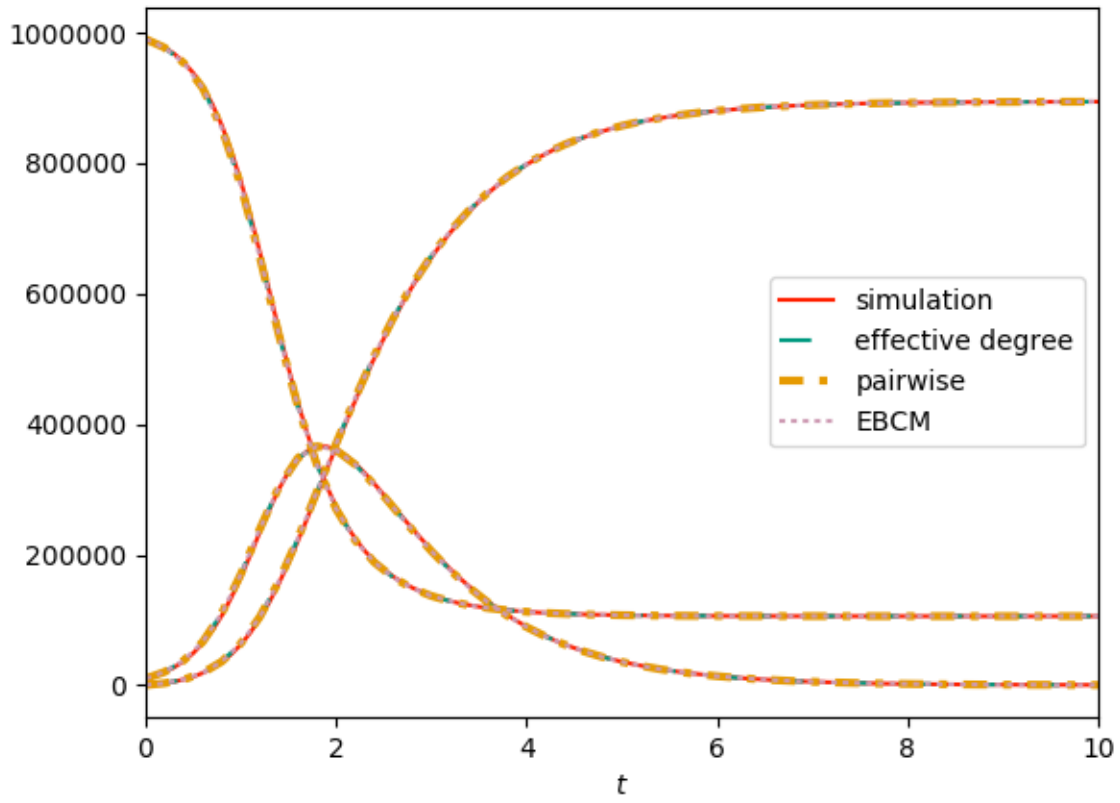
(continues on next page)

(continued from previous page)

```
G = nx.configuration_model(deg_seq)

initial_infecteds = set()
for u,v in G.edges():
    if G.degree(u) == G.degree(v):
        if random.random()<0.3:
            infectee = random.choice((u,v))
            initial_infecteds.add(infectee)

print('simulating')
t, S, I, R = EoN.fast_SIR(G, tau, gamma, initial_infecteds = initial_infecteds)
report_times = scipy.linspace(0,10,101)

S, I, R = EoN.subsample(report_times, t, S, I, R)

plt.plot(report_times, S, color =colors[1], label = 'simulation')
plt.plot(report_times, I, color = colors[1])
plt.plot(report_times, R, color = colors[1])

print('doing ODE models')
t, S, I, R = EoN.SIR_effective_degree_from_graph(G, tau, gamma, initial_
↪infecteds=initial_infecteds, tmax = 10, tcount = 51)
plt.plot(t,S, color = colors[2], dashes = [6,6], label = 'effective degree')
plt.plot(t,I, color = colors[2], dashes = [6,6])
plt.plot(t,R, color = colors[2], dashes = [6,6])

t, S, I, R = EoN.SIR_heterogeneous_pairwise_from_graph(G, tau, gamma, initial_
↪infecteds=initial_infecteds, tmax = 10, tcount = 51)
plt.plot(t, S, color = colors[3],  dashes = [3,2,1,2], linewidth=3, label = 'pairwise
↪')
plt.plot(t, I, color = colors[3],  dashes = [3,2,1,2], linewidth=3)
plt.plot(t, R, color = colors[3],  dashes = [3,2,1,2], linewidth=3)  #, dashes = [6,3,
↪2,3]


t, S, I, R = EoN.EBCM_from_graph(G, tau, gamma, initial_infecteds=initial_infecteds,␣
↪tmax = 10, tcount =51)
plt.plot(t, S, ':', color = colors[4], label = 'EBCM')
plt.plot(t, I, ':', color = colors[4])
plt.plot(t, R, ':', color = colors[4])

plt.axis(xmax=10, xmin=0)
plt.legend(loc = 'center right')
plt.xlabel('$t$')
plt.savefig('fig7p3.png')
```

Figure 7.4

Downloadable Source Code




```
import networkx as nx
import EoN
from collections import defaultdict
import matplotlib.pyplot as plt
import scipy
import random

N=10**6
tau = 1.
gamma = 1.
colors = ['#5AB3E6','#FF2000','#009A80','#E69A00', '#CD9AB3', '#0073B3','#F0E442']
kave = 5

G = nx.fast_gnp_random_graph(N, kave/(N-1.))

initial_infecteds = random.sample(range(N),int(0.01*N))

print('simulating')
t, S, I, R = EoN.fast_SIR(G, tau, gamma, initial_infecteds = initial_infecteds)
report_times = scipy.linspace(0,10,101)

S, I, R = EoN.subsample(report_times, t, S, I, R)

plt.plot(report_times, S, color =colors[1], label = 'simulation')
plt.plot(report_times, I, color = colors[1])
```

(continues on next page)

(continued from previous page)

```
plt.plot(report_times, R, color = colors[1])

print('doing ODE models')
t, S, I, R = EoN.SIR_effective_degree_from_graph(G, tau, gamma, initial_
↪infecteds=initial_infecteds, tmax = 10, tcount = 51)
plt.plot(t,S, color = colors[2], dashes = [6,6], label = 'effective degree')
plt.plot(t,I, color = colors[2], dashes = [6,6])
plt.plot(t,R, color = colors[2], dashes = [6,6])

t, S, I, R = EoN.SIR_heterogeneous_pairwise_from_graph(G, tau, gamma, initial_
↪infecteds=initial_infecteds, tmax = 10, tcount = 51)
plt.plot(t, S, color = colors[3],  dashes = [3,2,1,2], linewidth=3, label = 'pairwise
↪')
plt.plot(t, I, color = colors[3],  dashes = [3,2,1,2], linewidth=3)
plt.plot(t, R, color = colors[3],  dashes = [3,2,1,2], linewidth=3)  #, dashes = [6,3,
↪2,3]


t, S, I, R = EoN.EBCM_from_graph(G, tau, gamma, initial_infecteds=initial_infecteds,␣
↪tmax = 10, tcount =51)
plt.plot(t, S, ':', color = colors[4], label = 'EBCM')
plt.plot(t, I, ':', color = colors[4])
plt.plot(t, R, ':', color = colors[4])

plt.axis(xmax=10, xmin=0)
plt.xlabel('$t$')
plt.legend(loc = 'center right')
plt.savefig('fig7p4.png')
```

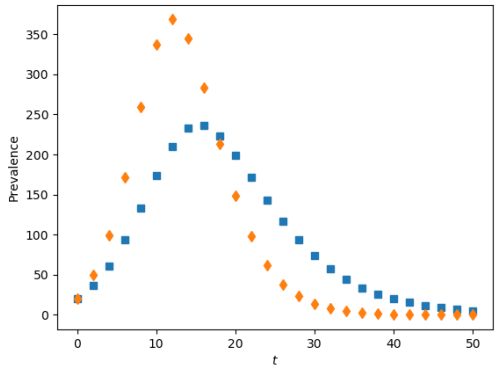
Chapter 9

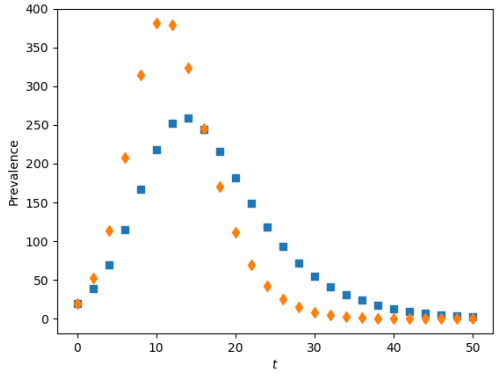
Non-Markovian processes

For Chapter 9 (nonMarkovian) figures, we have not implemented code that solves the dynamic equations but we do have code that will do the simulations. These are given here.

Figure 9.2 (a and b)

Downloadable Source Code






```
import EoN
import networkx as nx
import matplotlib.pyplot as plt
import random
import scipy

print("for figure 9.2, we have not yet coded up the system of equations (9.5), so␣
↪this just gives simulations")
N = 1000
n = 10
gamma = 1./5.5
tau = 0.545/n
iterations = 250
rho = 0.02

ER = nx.fast_gnp_random_graph(N, n/(N-1.)) #erdos-renyi graph
regular = nx.configuration_model([n]*N) # [n]*N is [n,n, ..., n]

def rec_time_fxn(u, K, gamma):
    duration = 0
    for counter in range(K):
        duration += random.expovariate(K*gamma)
    return duration

def trans_time_fxn(u, v, tau):
    return random.expovariate(tau)

display_ts = scipy.linspace(0, 50, 26)
for G, filename in ([regular, 'fig9p2a.png'], [ER, 'fig9p2b.png']):
    plt.clf()
    Isum = scipy.zeros(len(display_ts))
    for K, symbol in ([1, 's'], [3, 'd']):
        for counter in range(iterations):
            t, S, I, R = EoN.fast_nonMarkov_SIR(G,
                                                trans_time_fxn=trans_time_fxn,
                                                trans_time_args=(tau,),
                                                rec_time_fxn=rec_time_fxn,
                                                rec_time_args=(K, gamma),
                                                rho=rho)
            newI = EoN.subsample(display_ts, t, I)
            Isum += newI
        Isum /= iterations
        plt.plot(display_ts, Isum, symbol)
```

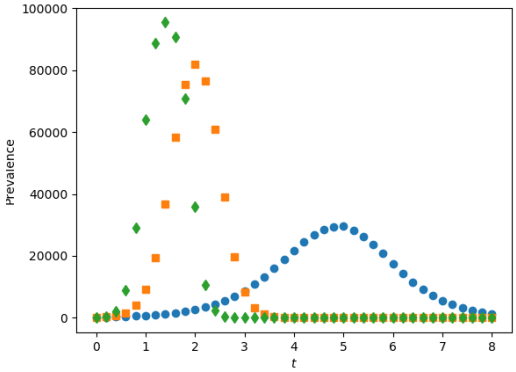
(continues on next page)

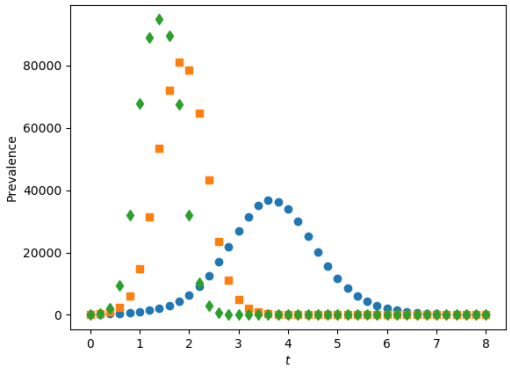
(continued from previous page)

```
    plt.xlabel('$t$')
    plt.ylabel('Prevalence')
    plt.savefig(filename)
```

Figure 9.4 (a and b)

Downloadable Source Code






```
import EoN
import networkx as nx
import matplotlib.pyplot as plt
import random
import scipy

print("for figure 9.4, we have not yet coded up the system of equations, so this just␣
↪gives simulations")

r'''We use a large value of N and only a single iteration.  The code is
similar to fig 9.2's code, but the loops are structured a little differently.

We loop over graph type first.
then we loop over kave.
'''

N = 100000
```

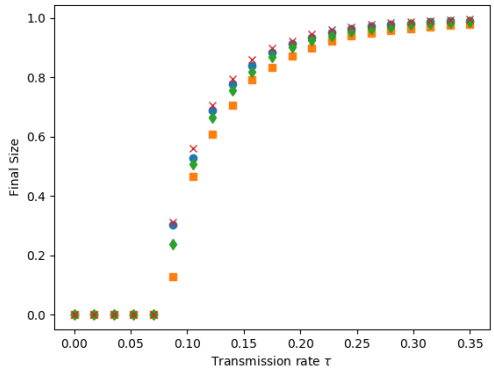
(continues on next page)

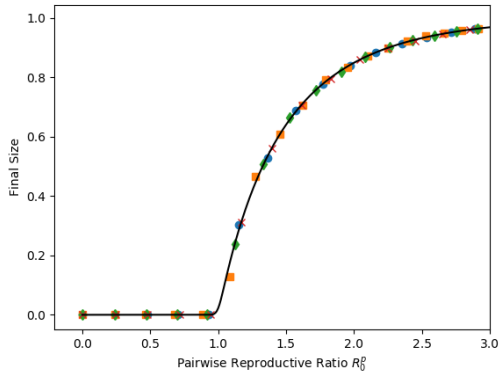
(continued from previous page)

```
n = 10
gamma = 1./5.5
tau = 0.55
iterations = 1
rho = 0.001


def rec_time_fxn(u):
    return 1

def trans_time_fxn(u, v, tau):
    return random.expovariate(tau)

def ER_graph_generation(N, kave):
    return nx.fast_gnp_random_graph(N, kave/(N-1.))
def regular_graph_generation(N, kave):
    return nx.configuration_model([kave]*N)

display_ts = scipy.linspace(0, 8, 41) #[0, 0.2, 0.4, ..., 7.8, 8]
for graph_algorithm, filename in ([regular_graph_generation, 'fig9p4a.png'], [ER_
↪graph_generation, 'fig9p4b.png']):
    plt.clf()
    for kave, symbol in ([5, 'o'], [10, 's'], [15, 'd']):
        print(kave)
        G = graph_algorithm(N, kave)
        t, S, I, R = EoN.fast_nonMarkov_SIR(G,
                                            trans_time_fxn=trans_time_fxn,
                                            trans_time_args=(tau,),
                                            rec_time_fxn=rec_time_fxn,
                                            rec_time_args=(),
                                            rho=rho)
        newI = EoN.subsample(display_ts, t, I)
        plt.plot(display_ts, newI, symbol)
    plt.xlabel('$t$')
    plt.ylabel('Prevalence')
    plt.savefig(filename)
```

Figure 9.5 (a and b)

Downloadable Source Code






```
import EoN
import networkx as nx
import matplotlib.pyplot as plt
import random
import scipy

print("for figure 9.5, we have not coded up the equations to calculate size as a␣
↪function of tau (fig a), so this just gives simulations.  It does calculate the␣
↪predicted size as a function of R_0^p. (fig b)")

r'''
Rather than doing the dynamic simulations, this uses the directed percolation approach
described in chapter 6.
'''

N = 100000
gamma = 1./5.5
tau = 0.55
iterations = 1
rho = 0.001
kave=15

def rec_time_fxn_gamma(u, alpha, beta):
    return scipy.random.gamma(alpha,beta)

def rec_time_fxn_fixed(u):
    return 1

def rec_time_fxn_exp(u):
    return random.expovariate(1)

def trans_time_fxn(u, v, tau):
    if tau >0:
        return random.expovariate(tau)
    else:
        return float('Inf')


def R0first(tau):
    return (kave-1) * (1- 4/(2+tau)**2)
def R0second(tau):
    return (kave-1) * (1- 1/scipy.sqrt(1+2*tau))
def R0third(tau):
```

(continues on next page)

(continued from previous page)

```
    return (kave-1)*tau/(tau+1)
def R0fourth(tau):
    return (kave-1)*(1-scipy.exp(-tau))


G = nx.configuration_model([kave]*N)

taus = scipy.linspace(0,0.35,21)

def do_calcs_and_plot(G, trans_time_fxn, rec_time_fxn, trans_time_args, rec_time_args,
↪ R0fxn, symbol):
    As = []
    for tau in taus:
        P, A = EoN.estimate_nonMarkov_SIR_prob_size_with_timing(G,trans_time_
↪fxn=trans_time_fxn,
                                                                rec_time_fxn = rec_time_fxn,
                                                                trans_time_args = (tau,),
                                                                rec_time_args=rec_time_args)
        As.append(A)
    plt.figure(1)
    plt.plot(taus, As, symbol)
    plt.figure(2)
    plt.plot( R0fxn(taus), As, symbol)

print("first distribution")
do_calcs_and_plot(G, trans_time_fxn, rec_time_fxn_gamma, (tau,), (2,0.5), R0first, 'o
↪')
print("second distribution")
do_calcs_and_plot(G, trans_time_fxn, rec_time_fxn_gamma, (tau,), (0.5,2), R0second, 's
↪')
print("fourth distribution")
do_calcs_and_plot(G, trans_time_fxn, rec_time_fxn_exp, (tau,), (), R0third, 'd')
print("fifth distribution")
do_calcs_and_plot(G, trans_time_fxn, rec_time_fxn_fixed, (tau,), (), R0fourth, 'x')


plt.figure(1)
plt.xlabel(r'Transmission rate $\tau$')
plt.ylabel('Final Size')
plt.savefig('fig9p5a.png')

R0s = scipy.linspace(0,3,301)
ps = R0s/(kave-1)
Apred = [EoN.Attack_rate_discrete({kave:1}, p) for p in ps]
plt.figure(2)
plt.plot(R0s, Apred, '-', color = 'k')
plt.axis(xmax = 3)
plt.xlabel('Pairwise Reproductive Ratio $R_0^p$')
plt.ylabel('Final Size')
plt.savefig('fig9p5b.png')
```

2.2.2 Additional Examples

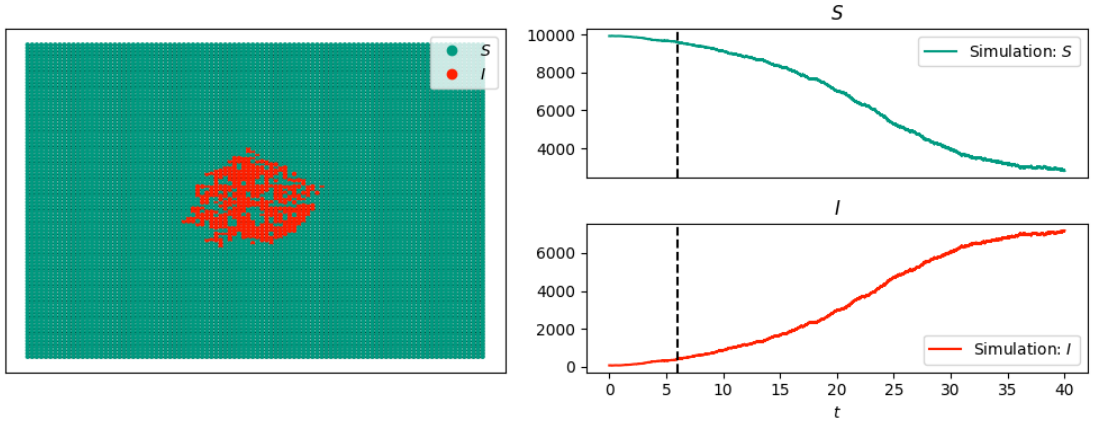
Visualizing or animating disease spread

We can visualize snapshots or animations of disease spread in a network. For these examples, we'll take a 100x100 grid of nodes [each node is (i,j)] connected to their 4 nearest neighbors (except the nodes on the edges). This isn't the most realistic network, but it is a good example for showing the automatic plotting tools.

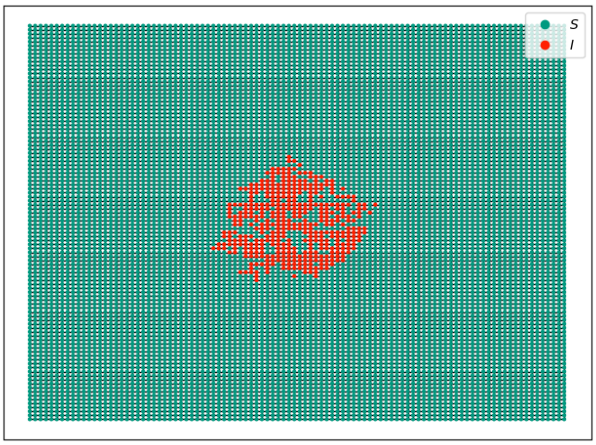
SIS display example

`Downloadable Source Code`

The code below produces a snapshot of an SIS epidemic with the timeseries:



and without the timeseries:



We start with an SIS epidemic in a 100x100 grid with a few nodes infected in the middle. We will display the epidemic at time 6, with and without the time series included.

```
import networkx as nx
import EoN
import matplotlib.pyplot as plt
G = nx.grid_2d_graph(100,100) #each node is (u,v) where 0<=u,v<=99
#we'll initially infect those near the middle
initial_infections = [(u,v) for (u,v) in G if 45<u<55 and 45<v<55]
sim = EoN.fast_SIS(G, 1.0, 1.0, initial_infecteds = initial_infections,
```

(continues on next page)

(continued from previous page)

```
                return_full_data=True, tmax = 40)
pos = {node:node for node in G}
sim.set_pos(pos)
sim.display(6, node_size = 4) #display time 6
plt.savefig('SIS_2dgrid.png')
```

If we changed the `display` command to have `ts_plots=False` or `ts_plots = []` we get just the network.

```
plt.clf()
sim.display(6, node_size = 4, ts_plots=[]) #display time 6
plt.savefig('SIS_2dgrid_no_time_series.png')
```

Animations are shown in the next example.

SIR Animation Example

`Downloadable Source Code`

The code below produces an animation of an SIR epidemic:

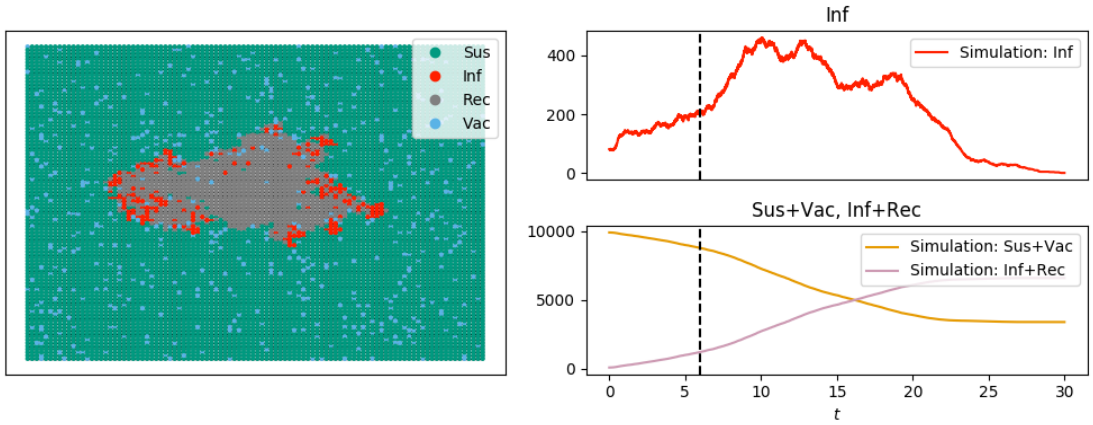
You may need to install additional software for this to work and modify *extra_args* appropriately. The commands below work on a mac with ffmpeg installed. The commands below also show an alternate way to specify the position of nodes passing *pos* through *fast_SIR* to be used when generating the simulation_investigation object.

```
import networkx as nx
import EoN
import matplotlib.pyplot as plt
G = nx.grid_2d_graph(100,100) #each node is (u,v) where 0<=u,v<=99
#we'll initially infect those near the middle
initial_infections = [(u,v) for (u,v) in G if 45<u<55 and 45<v<55]
pos = {node:node for node in G}
sim_kwargs = {'pos': pos}
sim = EoN.fast_SIR(G, 2.0, 1.0, initial_infecteds = initial_infections,
                tmax = 40, return_full_data=True, sim_kwargs = sim_kwargs)

ani=sim.animate(ts_plots=['I', 'SIR'], node_size = 4)
ani.save('SIR_2dgrid.mp4', fps=5, extra_args=['-vcodec', 'libx264'])
```

SIRV display and animation example

`Downloadable Source Code`

The code below looks at an SIRV epidemic (SIR + vaccination) and produces the image



and the animation

Note that the labels are in plain text rather than math mode (since `tex=False`). Note also that we can plot `'Sus'+'Vac'` and similar time series using the commands here.

If we use a model with other states than `'S'`, `'I'`, and `'R'`, the default colors aren't specified. In this case we need to do a little bit more.

Consider a model where the states are `'Sus'`, `'Inf'`, `'Rec'`, or `'Vac'`. That is, an SIR model with vaccination. We will use `Gillespie_simple_contagion` for this. I'm choosing the status names to be longer than one character to show changes in the argument `ts_plots` stating what the time-series plots should show.

In this model, susceptible people have a rate of becoming vaccinated which is independent of the disease status. Otherwise, it is just like the SIR disease in the previous example. So the "spontaneous transitions" are `'Suc'` to `'Vac'` with rate `0.01` and `'Inf'` to `'Rec'` with rate `1.0`. The "induced transitions" are `('Inf', 'Sus')` to `('Inf', 'Inf')` with rate `2.0`.

The method is built on Gillespie_simple_contagion

```
import networkx as nx
import EoN
import matplotlib.pyplot as plt
from collections import defaultdict

G = nx.grid_2d_graph(100,100) #each node is (u,v) where 0<=u,v<=99
#we'll initially infect those near the middle
initial_infections = [(u,v) for (u,v) in G if 45<u<55 and 45<v<55]

H = nx.DiGraph()  #the spontaneous transitions
H.add_edge('Sus', 'Vac', rate = 0.01)
H.add_edge('Inf', 'Rec', rate = 1.0)

J = nx.DiGraph()  #the induced transitions
J.add_edge(('Inf', 'Sus'), ('Inf', 'Inf'), rate = 2.0)

IC = defaultdict(lambda:'Sus')
for node in initial_infections:
    IC[node] = 'Inf'

return_statuses = ['Sus', 'Inf', 'Rec', 'Vac']

color_dict = {'Sus': '#009a80','Inf':'#ff2000', 'Rec':'gray','Vac': '#5AB3E6'}
pos = {node:node for node in G}
```

(continues on next page)

(continued from previous page)

```
tex = False
sim_kwargs = {'color_dict':color_dict, 'pos':pos, 'tex':tex}

sim = EoN.Gillespie_simple_contagion(G, H, J, IC, return_statuses, tmax=30, return_
↪full_data=True, sim_kwargs=sim_kwargs)

times, D = sim.summary()
#
#imes is a numpy array of times.  D is a dict, whose keys are the entries in
#return_statuses.  The values are numpy arrays giving the number in that
#status at the corresponding time.

newD = {'Sus+Vac':D['Sus']+D['Vac'], 'Inf+Rec' : D['Inf'] + D['Rec']}
#
#newD is a new dict giving number not yet infected or the number ever infected
#Let's add this timeseries to the simulation.
#
new_timeseries = (times, newD)
sim.add_timeseries(new_timeseries, label = 'Simulation', color_dict={'Sus+Vac':'
↪#E69A00', 'Inf+Rec':'#CD9AB3'})

sim.display(6, node_size = 4, ts_plots=[['Inf'], ['Sus+Vac', 'Inf+Rec']])
plt.savefig('SIRV_display.png')

ani=sim.animate(ts_plots=[['Inf'], ['Sus+Vac', 'Inf+Rec']], node_size = 4)
ani.save('SIRV_animate.mp4', fps=5, extra_args=['-vcodec', 'libx264'])
```

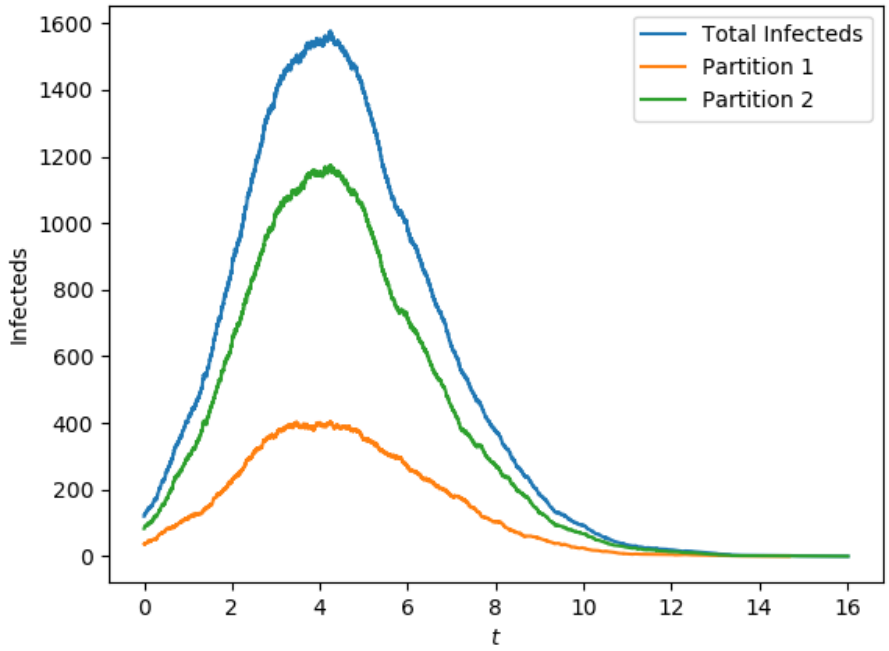
The last example shows how we might plot things like `'I'+'R'` or other combinations of the data.

Non-uniform infectiousness

Perhaps the population is made up of different types of individuals with different infectiousness. This example considers a heterosexual population where one group is more infectious than the other. This example includes a user-defined transmission function and uses the simulation investigation tools to look at the dynamics in each population.

Bipartite example

Downloadable Source Code




```
import EoN
import networkx as nx
import random
import matplotlib.pyplot as plt


G= nx.bipartite.configuration_model([1,11]*2000, [3]*8000)
#the graph now consists of two parts.  The first part has 2000 degree 1 nodes
#and 2000 degree 11 nodes.  The second has 8000 degree 3 nodes.
#there are 24000 edges in the network.
#
#  We assume the first ones are twice as infectious as the second ones.
#

for node in G:
    if G.degree(node) in [1,11]:
        G.node[node]['type'] = 'A'
    else:
        G.node[node]['type'] = 'B'

#We have defined the two types of nodes.

#now define the transmission and recovery functions:
def trans_time_function(source, target, tau):
    if G.node[source]['type'] is 'A':
        return random.expovariate(2*tau)
    else:
        return random.expovariate(tau)

def rec_time_function(node, gamma):
    return random.expovariate(gamma)
```

(continues on next page)

(continued from previous page)

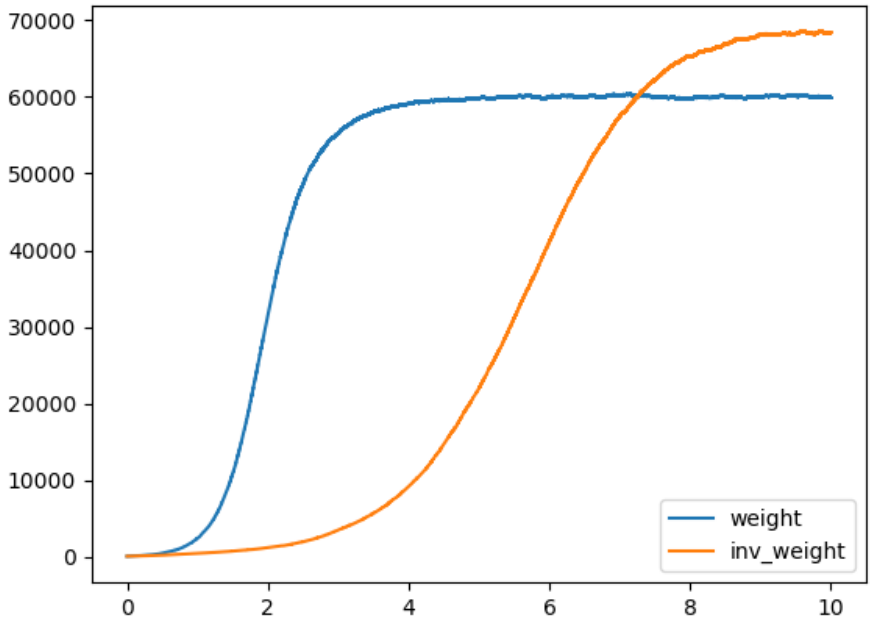
```
tau = 0.4
gamma = 1.
sim = EoN.fast_nonMarkov_SIR(G, trans_time_function, rec_time_function,
                             trans_time_args=(tau,), rec_time_args=(gamma,),
                             rho = 0.01, return_full_data=True)

t, S, I, R = sim.summary()
plt.plot(t, I, label='Total Infecteds')

t1, S1, I1, R1 = sim.summary(nodelist = [node for node in G if G.node[node]['type']==
↪'A'])
plt.plot(t1, I1, label = 'Partition 1')

t2, S2, I2, R2 = sim.summary(nodelist = [node for node in G if G.node[node]['type']==
↪'B'])
plt.plot(t2, I2, label = 'Partition 2')

plt.legend()
plt.xlabel('$t$')
plt.ylabel('Infecteds')
plt.savefig('bipartite.png')
```

Weighted networks

You may have edges (or nodes) with weights affecting transmission or recovery rates. (for this be aware that the syntax of edge/node attributes is different for networkx 2.x and 1.x).

Weighted network example

Downloadable Source Code




```
import networkx as nx
import EoN
import matplotlib.pyplot as plt

r'''This code simulates an SIS epidemic in a graph.  The edges are weighted by
two methods: the product of the degrees, or the inverse of that product.

I then run simulations with the transmission rates scaled by the edge weights
such that a random edge would have expected transmission weight 1 (though since
there will be biases in which nodes are most likely to be infected, the
random edge that has an infected node will have higher transmission weight).
'''

N= 100000
rho = 0.001
gamma = 1

G = nx.configuration_model([2,6]*int(N/2)) #N nodes, half have degree 6 and half␣
↪degree 2
G=nx.Graph(G)

#assign edge weights to be product of degree.  Also give another weight to be inverse␣
↪of product of degrees
weight_sum = 0
inv_weight_sum = 0

for edge in G.edges():
    G.edges[edge[0],edge[1]]['weight'] = G.degree(edge[0])*G.degree(edge[1])
    G.edges[edge[0],edge[1]]['inv_weight'] = 1./(G.degree(edge[0])*G.degree(edge[1]))
    #If networkx is older, use G.edge[edge[0]][edge[1]][...
```

(continues on next page)

(continued from previous page)

```
    weight_sum += G.degree(edge[0])*G.degree(edge[1])
    inv_weight_sum += 1./(G.degree(edge[0])*G.degree(edge[1]))

#first do it with weight, scaled so that average weight is 1.
t, S, I = EoN.fast_SIS(G, G.number_of_edges()/weight_sum, gamma, rho = rho,␣
↪transmission_weight= 'weight', tmax = 10)
plt.plot(t, I, label = 'weight')


t, S, I = EoN.fast_SIS(G, G.number_of_edges()/inv_weight_sum, gamma, rho = rho,␣
↪transmission_weight= 'inv_weight', tmax = 10)
plt.plot(t, I, label = 'inv_weight')

plt.legend(loc = 'lower right')
plt.savefig('SIS_weighted.png')
```

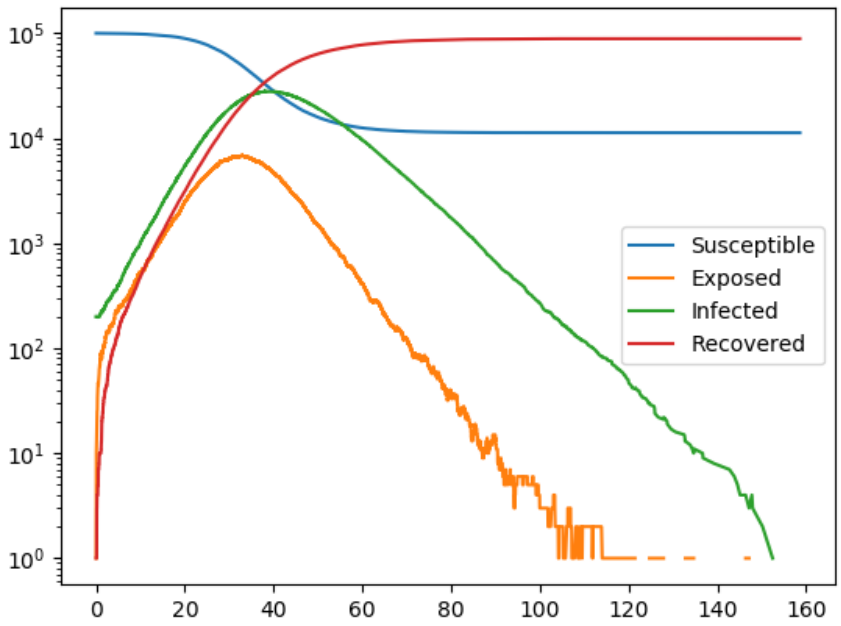
Non-SIS/SIR processes with Gillespie_simple_contagion

The default Gillespie and Event-driven algorithms assume SIS or SIR dynamics. However, you may want something else, such as SEIR or SIRS or maybe more exotic things. If you are willing to assume that events all happen as Poisson processes (that is, an event happens at a rate), then it's possible to do this with a Gillespie approach.

Note that right now if a node has an edge to itself, this can cause the code to crash. I will try to fix this soon.

SEIR

Downloadable Source Code



This has a variable transmission rate across different edges and a variable rate of transitioning from exposed to infected across different nodes.

These heterogeneities are introduced by assigning attributes to the individuals and partnerships in the contact network. The transition rates are simply multiplied by those attributes. More complex methods are provided in *SIRS with Heterogeneity* which allow us to scale the transitions by some function of the nodes [this may be particularly useful when disease is more infectious in one direction, as in many sexually transmitted diseases].

The method is built on Gillespie_simple_contagion

```
import EoN
import networkx as nx
from collections import defaultdict
import matplotlib.pyplot as plt
import random

N = 100000
G = nx.fast_gnp_random_graph(N, 5./(N-1))

#they will vary in the rate of leaving exposed class.
#and edges will vary in transition rate.
#there is no variation in recovery rate.

node_attribute_dict = {node: 0.5+random.random() for node in G.nodes()}
edge_attribute_dict = {edge: 0.5+random.random() for edge in G.edges()}

nx.set_node_attributes(G, values=node_attribute_dict, name='expose2infect_weight')
nx.set_edge_attributes(G, values=edge_attribute_dict, name='transmission_weight')
#These individual and partnership attributes will be used to scale
#the transition rates.  When we define `H` and `J`, we provide the name
#of these attributes.


#We show how node and edge attributes in the contact network 'G' can be used
#to scale the transmission rates.  More advanced techniques are shown in
#other examples.

H = nx.DiGraph()
H.add_node('S') #This line is actually unnecessary since 'S' does not change status␣
↪intrinsically
#
H.add_edge('E', 'I', rate = 0.6, weight_label='expose2infect_weight')
#  The line above states that the transition from 'E' to 'I' occurs with rate
#  0.6  times  whatever value is in the individual's attribute 'expose2infect_weight'
#
H.add_edge('I', 'R', rate = 0.1)
#  The line above states that the I to 'R' transition occurs with rate 0.1
#  and does not depend on any attribute

J = nx.DiGraph()
J.add_edge(('I', 'S'), ('I', 'E'), rate = 0.1, weight_label='transmission_weight')
#  The line above states that an 'I' individual will cause an 'S' individual
#  to transition to 'E' with rate equal to 0.1 times the partnership's attribute
#  'transmission_weight'.


IC = defaultdict(lambda: 'S')
for node in range(200):
```

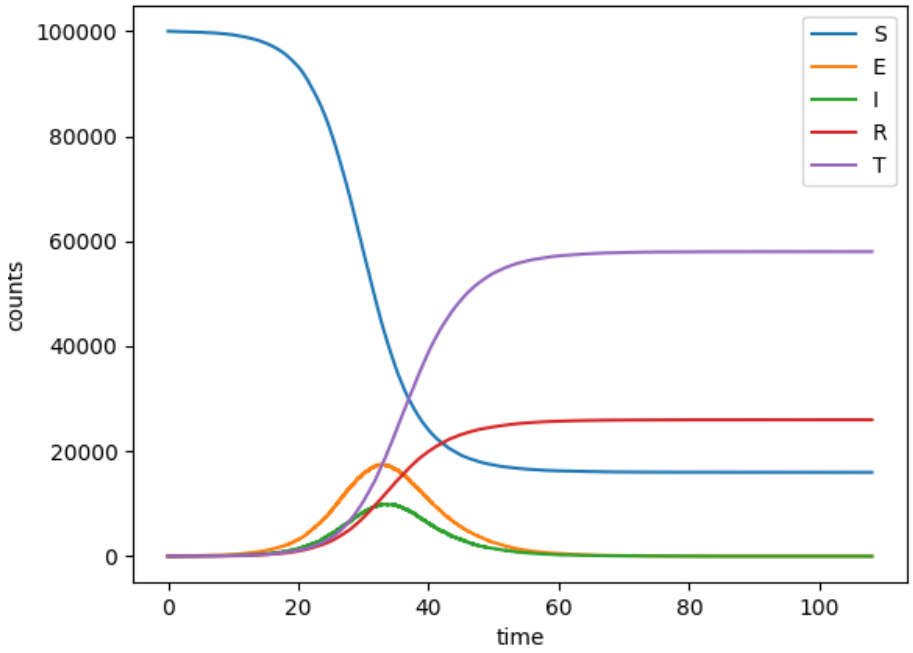
(continues on next page)

(continued from previous page)

```
    IC[node] = 'I'

return_statuses = ('S', 'E', 'I', 'R')

t, S, E, I, R = EoN.Gillespie_simple_contagion(G, H, J, IC, return_statuses,
                                               tmax = float('Inf'))

plt.semilogy(t, S, label = 'Susceptible')
plt.semilogy(t, E, label = 'Exposed')
plt.semilogy(t, I, label = 'Infected')
plt.semilogy(t, R, label = 'Recovered')
plt.legend()

plt.savefig('SEIR.png')
```

SEIRT

Downloadable Source Code



We consider an SEIR style model, but with contact tracing of identified contacts. An infected person may be identified, in which case his/her contacts are traced with some rate.

We consider a simple model, a separate example shows an SEIR model with weighted edges. The method is built on Gillespie_simple_contagion

```
import EoN
import networkx as nx
from collections import defaultdict
import matplotlib.pyplot as plt
```

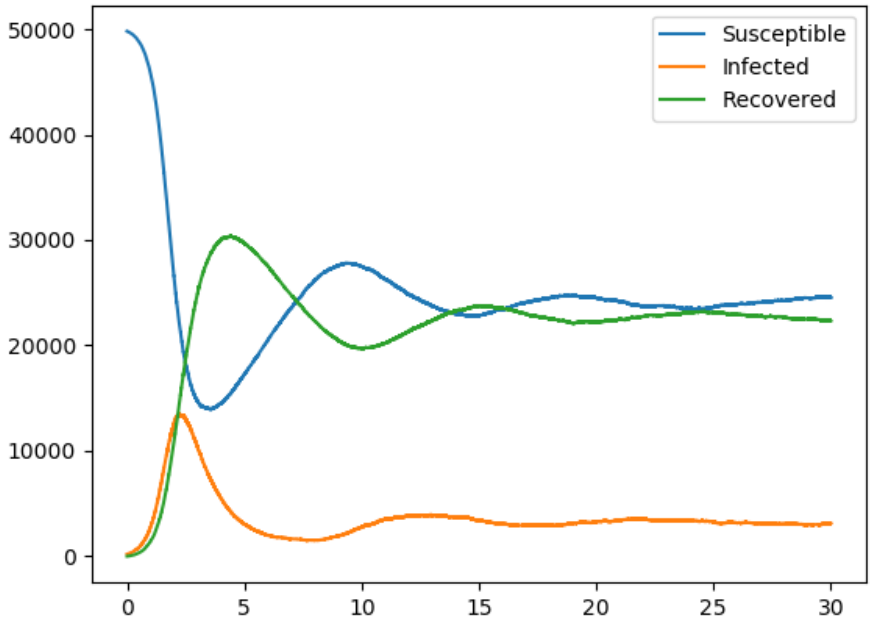
(continues on next page)

(continued from previous page)

```
import random

N = 100000
G = nx.fast_gnp_random_graph(N, 5./(N-1))


#we must define two graphs, one of which has the internal transitions
H = nx.DiGraph()
H.add_node('S')  #This line is unnecessary.
H.add_edge('E', 'I', rate = 1./4)
H.add_edge('I', 'R', rate = 1./7)
H.add_edge('I', 'T', rate = 1./10)

#and the other graph has transitions caused by a neighbor.
J = nx.DiGraph()
J.add_edge(('I', 'S'), ('I', 'E'), rate = 2.5/7)
J.add_edge(('T', 'I'), ('T', 'T'), rate = 0.2)
IC = defaultdict(lambda: 'S')
for node in range(20):
    IC[node] = 'I'

return_statuses = ('S', 'E', 'I', 'R', 'T')

t, S, E, I, R, T = EoN.Gillespie_simple_contagion(G, H, J, IC, return_statuses,
                                          tmax = float('Inf'))

plt.plot(t, S, label = 'S')
plt.plot(t, E, label = 'E')
plt.plot(t, I, label = 'I')
plt.plot(t, R, label = 'R')
plt.plot(t, T, label = 'T')
plt.clf()
plt.legend()
plt.xlabel('time')
plt.ylabel('counts')
plt.savefig('SEIRT.png')
plt.show()
```

SIRS

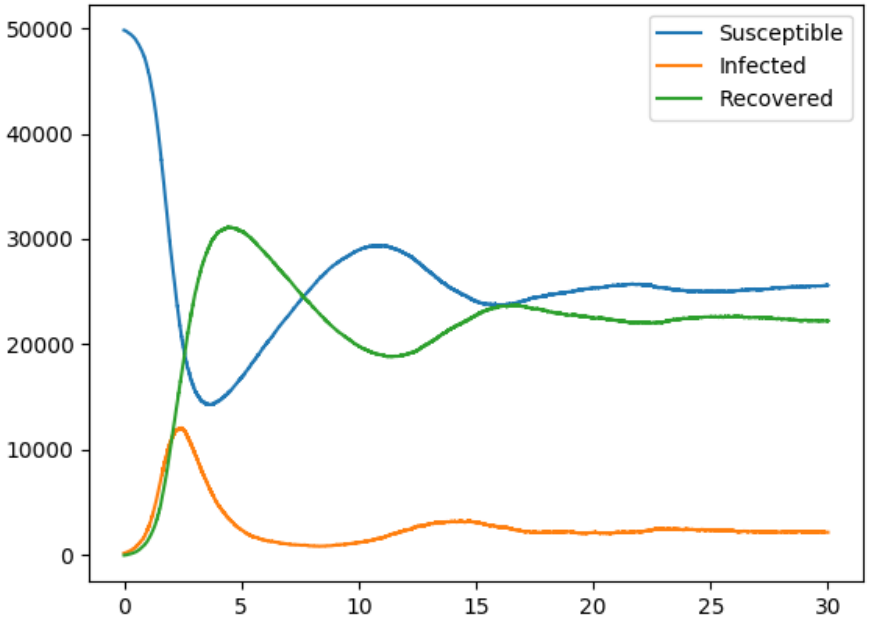
Downloadable Source Code




```
import EoN
import networkx as nx
from collections import defaultdict
import matplotlib.pyplot as plt

N = 50000
G = nx.fast_gnp_random_graph(N, 5./(N-1))

H = nx.DiGraph()  #DiGraph showing possible transitions that don't require an␣
↪interaction
H.add_edge('I', 'R', rate = 1.4)    #I->R
H.add_edge('R', 'S', rate = 0.2)    #R->S

J = nx.DiGraph()    #DiGraph showing transition that does require an interaction.
J.add_edge(('I', 'S'), ('I', 'I'), rate = 1)  #IS->II

IC = defaultdict(lambda: 'S')
for node in range(200):
    IC[node] = 'I'

return_statuses = ('S', 'I', 'R')

t, S, I, R = EoN.Gillespie_simple_contagion(G, H, J, IC, return_statuses, tmax = 30)

plt.plot(t, S, label = 'Susceptible')
plt.plot(t, I, label = 'Infected')
plt.plot(t, R, label = 'Recovered')
plt.legend()
plt.savefig('SIRS.png')
```

SIRS with Heterogeneity

Downloadable Source Code



We consider a model comparable to our basic SIRS example, except that some of the contacts are more infectious than before. However, some individuals recover faster, while others take longer to return to the susceptible state after recovery. So the difference from our basic SIRS outcomes are not very large.

Specifically we have transmission and recovery rates depend on age and gender. Transmission rates are not always symmetric, so it is not as simple as introducing a weight to scale the partnerships. So we introduce functions to scale the transition rates.

The method is built on Gillespie_simple_contagion

```
import EoN
import networkx as nx
from collections import defaultdict
import matplotlib.pyplot as plt
import random

N = 50000
G = nx.fast_gnp_random_graph(N, 5./(N-1))

#Let's consider a disease like that in the basic SIRS example, except:
#   children are more susceptible
#   males are more infectious if the partner is female
#   children recover faster.
#   females return to susceptibility slower.
#   and let's say that we want the cutoff age for a child to be a parameter

#So first we define the node attributes:
ages = {node: random.random()*100 for node in G}
```

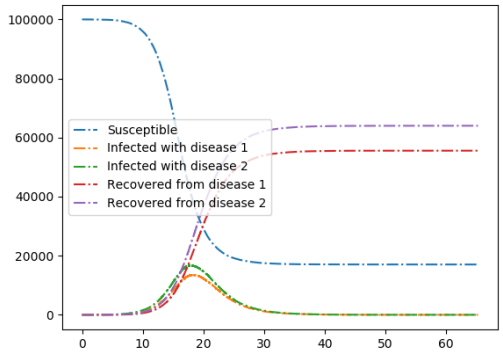
(continues on next page)

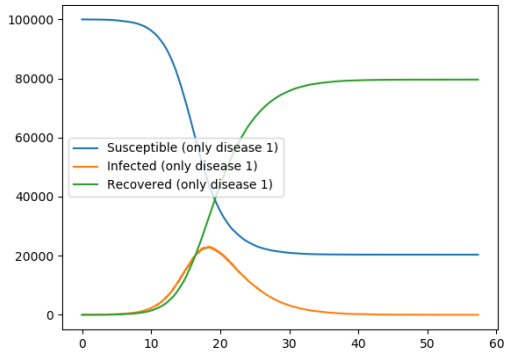
(continued from previous page)

```
genders = {node: 'M' if random.random()<0.5 else 'F' for node in G}
nx.set_node_attributes(G, values=ages, name = 'age')
nx.set_node_attributes(G, values = genders, name = 'gender')

#Now we define functions which will be used to scale the transition rates
def transmission_weighting(G, source, target, **kwargs):
    scale = 1
    if G.node[target]['age']<kwargs['age_cutoff']:
        scale *= 1.5
    if G.node[target]['gender'] is 'F' and G.node[source]['gender'] is 'M':
        scale *= 1.5
    return scale

def recovery_weighting(G, node, **kwargs):
    scale = 1
    if G.node[node]['age']<kwargs['age_cutoff']:
        scale *= 1.5
    return scale

def return_to_susceptibility_weighting(G, node, **kwargs):
    scale = 1
    if G.node[node]['gender'] is 'F':
        scale *= 0.5
    return scale

H = nx.DiGraph()  #DiGraph showing possible transitions that don't require an␣
↪interaction
H.add_edge('I', 'R', rate = 1.4, rate_function=recovery_weighting)   #I->R
H.add_edge('R', 'S', rate = 0.2, rate_function = return_to_susceptibility_weighting) ␣
↪ #R->S

J = nx.DiGraph()    #DiGraph showing transition that does require an interaction.
J.add_edge(('I', 'S'), ('I', 'I'), rate = 1, rate_fuction = transmission_weighting)
↪#IS->II

IC = defaultdict(lambda: 'S')
for node in range(200):
    IC[node] = 'I'

return_statuses = ('S', 'I', 'R')

age_cutoff = 18
t, S, I, R = EoN.Gillespie_simple_contagion(G, H, J, IC, return_statuses, tmax = 30,
                            spont_kwargs = {'age_cutoff':age_cutoff},
                            nbr_kwargs = {'age_cutoff':age_cutoff})

plt.plot(t, S, label = 'Susceptible')
plt.plot(t, I, label = 'Infected')
plt.plot(t, R, label = 'Recovered')
plt.legend()
plt.savefig('SIRS_heterogeneous.png')
```

Competing Diseases

Downloadable Source Code





We consider two diseases that compete in the sense that if an individual has recovered from one disease, then if it is infected with the other disease it transmits with lower rate and it recovers faster.

Because our initial condition is quite small, the final outcome is somewhat stochastic, even though the curves look quite smooth. The stochasticity is manifested when the populations are still small.

Our first plot shows the two diseases competing. The second plot shows what happens if there is just one disease present.

The method is built on Gillespie_simple_contagion

```
r'''We consider two diseases that compete in the sense that if an individual has
recovered from one disease, then if it is infected with the other disease it
transmits with lower rate and it recovers faster.'''
import EoN
import networkx as nx
from collections import defaultdict
import matplotlib.pyplot as plt

N = 100000
G = nx.fast_gnp_random_graph(N, 5./(N-1))

H = nx.DiGraph()  #DiGraph showing possible transitions that don't require an␣
↪interaction
H.add_node('SS')
H.add_edge('SI', 'SR', rate = 0.3)
H.add_edge('IS', 'RS', rate = 0.3)
H.add_edge('II', 'IR', rate = 0.3)
H.add_edge('II', 'RI', rate = 0.3)
```

(continues on next page)

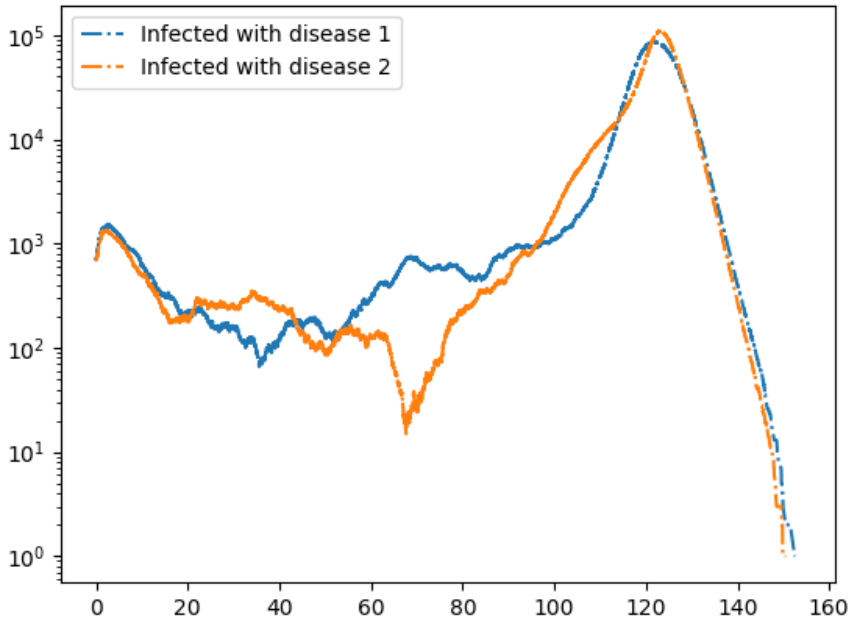
(continued from previous page)

```
H.add_edge('IR', 'RR', rate = 1)
H.add_edge('RI', 'RR', rate = 1)

J = nx.DiGraph()    #DiGraph showing transitions that do require an interaction.
J.add_edge(('SI', 'SS'), ('SI', 'SI'), rate = 0.2)
J.add_edge(('SI', 'IS'), ('SI', 'II'), rate = 0.2)
J.add_edge(('SI', 'RS'), ('SI', 'RI'), rate = 0.2)
J.add_edge(('II', 'SS'), ('II', 'SI'), rate = 0.2)
J.add_edge(('II', 'IS'), ('II', 'II'), rate = 0.2)
J.add_edge(('II', 'RS'), ('II', 'RI'), rate = 0.2)
J.add_edge(('RI', 'SS'), ('RI', 'SI'), rate = 0.1)
J.add_edge(('RI', 'IS'), ('RI', 'II'), rate = 0.1)
J.add_edge(('RI', 'RS'), ('RI', 'RI'), rate = 0.1)
J.add_edge(('IS', 'SS'), ('IS', 'IS'), rate = 0.2)
J.add_edge(('IS', 'SI'), ('IS', 'II'), rate = 0.2)
J.add_edge(('IS', 'SR'), ('IS', 'IR'), rate = 0.2)
J.add_edge(('II', 'SS'), ('II', 'IS'), rate = 0.2)
J.add_edge(('II', 'SI'), ('II', 'II'), rate = 0.2)
J.add_edge(('II', 'SR'), ('II', 'IR'), rate = 0.2)
J.add_edge(('IR', 'SS'), ('IR', 'IS'), rate = 0.1)
J.add_edge(('IR', 'SI'), ('IR', 'II'), rate = 0.1)
J.add_edge(('IR', 'SR'), ('IR', 'IR'), rate = 0.1)

IC = defaultdict(lambda: 'SS')
for node in range(5):
    IC[node] = 'II'

return_statuses = ('SS', 'SI', 'SR', 'IS', 'II', 'IR', 'RS', 'RI', 'RR')

t, SS, SI, SR, IS, II, IR, RS, RI, RR = EoN.Gillespie_simple_contagion(G, H, J, IC,␣
↪return_statuses,
                                        tmax = float('Inf'))

plt.plot(t, SS, '-.', label = 'Susceptible')
plt.plot(t, IS+II+IR, '-.', label = 'Infected with disease 1')
plt.plot(t, SI+II+RI, '-.', label = 'Infected with disease 2')
plt.plot(t, RS+IR+RR, '-.', label = 'Recovered from disease 1')
plt.plot(t, SR+RI+RR, '-.', label = 'Recovered from disease 2')
plt.legend(loc = 'center left')
plt.savefig('Compete_both.png')

IC = defaultdict(lambda: 'SS')
for node in range(5):
    IC[node] = 'IS'
t, SS, SI, SR, IS, II, IR, RS, RI, RR = EoN.Gillespie_simple_contagion(G, H, J, IC,␣
↪return_statuses,
                                        tmax = float('Inf'))

plt.clf()
plt.plot(t, SS, '-', label = 'Susceptible (only disease 1)')
plt.plot(t, IS+II+IR, '-', label = 'Infected (only disease 1)')
plt.plot(t, RS+IR+RR, '-', label = 'Recovered (only disease 1)')
plt.legend(loc = 'center left')
plt.savefig('Compete_just1disease.png')
```

Cooperative diseases

`Downloadable Source Code`

Note that the dynamics of this case are sensitive to noise, so rerunning this code will not yield an exact match to this figure.



We consider two diseases that cooperate in the sense that if an individual has recovered from one disease, then if it is infected with the other disease it transmits with higher rate and it remains infectious longer. This is a bit unrealistic, but could correspond to an individual having a more mild infection resulting in the individual remaining active in the population rather than self-isolating at home.

The method is built on Gillespie_simple_contagion

```
import EoN
import networkx as nx
from collections import defaultdict
import matplotlib.pyplot as plt

N = 2000000
G = nx.fast_gnp_random_graph(N, 5./(N-1))

#In the below:
#'SS' means a node susceptible to both diseases
#'SI' means susceptible to disease 1 and infected with disease 2
#'RS' means recovered from disease 1 and susceptible to disease 2.
#etc.

H = nx.DiGraph()  #DiGraph showing possible transitions that don't require an␣
↪interaction
H.add_node('SS')
H.add_edge('SI', 'SR', rate = 1)
```

(continues on next page)

(continued from previous page)

```
H.add_edge('IS', 'RS', rate = 1)
H.add_edge('II', 'IR', rate = 1)
H.add_edge('II', 'RI', rate = 1)
H.add_edge('IR', 'RR', rate = 0.5)
H.add_edge('RI', 'RR', rate = 0.5)

#In the below the edge (('SI', 'SS'), ('SI', 'SI')) means an
#'SI' node connected to an 'SS' node can lead to a transition in which
#the 'SS' node becomes 'SI'.  The rate of this transition is 0.2.
#
#Note that `IR` and `RI` nodes are more infectious than other nodes.
#
J = nx.DiGraph()    #DiGraph showing transitiona that do require an interaction.
J.add_edge(('SI', 'SS'), ('SI', 'SI'), rate = 0.2)
J.add_edge(('SI', 'IS'), ('SI', 'II'), rate = 0.2)
J.add_edge(('SI', 'RS'), ('SI', 'RI'), rate = 0.2)
J.add_edge(('II', 'SS'), ('II', 'SI'), rate = 0.2)
J.add_edge(('II', 'IS'), ('II', 'II'), rate = 0.2)
J.add_edge(('II', 'RS'), ('II', 'RI'), rate = 0.2)
J.add_edge(('RI', 'SS'), ('RI', 'SI'), rate = 1)
J.add_edge(('RI', 'IS'), ('RI', 'II'), rate = 1)
J.add_edge(('RI', 'RS'), ('RI', 'RI'), rate = 1)
J.add_edge(('IS', 'SS'), ('IS', 'IS'), rate = 0.2)
J.add_edge(('IS', 'SI'), ('IS', 'II'), rate = 0.2)
J.add_edge(('IS', 'SR'), ('IS', 'IR'), rate = 0.2)
J.add_edge(('II', 'SS'), ('II', 'IS'), rate = 0.2)
J.add_edge(('II', 'SI'), ('II', 'II'), rate = 0.2)
J.add_edge(('II', 'SR'), ('II', 'IR'), rate = 0.2)
J.add_edge(('IR', 'SS'), ('IR', 'IS'), rate = 1)
J.add_edge(('IR', 'SI'), ('IR', 'II'), rate = 1)
J.add_edge(('IR', 'SR'), ('IR', 'IR'), rate = 1)


return_statuses = ('SS', 'SI', 'SR', 'IS', 'II', 'IR', 'RS', 'RI', 'RR')

initial_size = 700
IC = defaultdict(lambda: 'SS')
for node in range(initial_size):
    IC[node] = 'II'


t, SS, SI, SR, IS, II, IR, RS, RI, RR = EoN.Gillespie_simple_contagion(G, H, J, IC,␣
↪return_statuses,
                                        tmax = float('Inf'))

plt.semilogy(t, IS+II+IR, '-.', label = 'Infected with disease 1')
plt.semilogy(t, SI+II+RI, '-.', label = 'Infected with disease 2')

plt.legend()
plt.savefig('Cooperate.png')
```

Other

Are you trying to do something but can't figure it out and would like an example?

Submit an issue or go to stackoverflow and use the 'eon' tag. I'll try to help.

If you have developed something that you think would make a good example and you'd like to share it, please let me know.

2.3 EoN module

2.3.1 Introduction

EoN (Epidemics on Networks) is a Python package for the simulation of epidemics on networks and solving ODE models of disease spread.

The algorithms are based on the book

Mathematics of Epidemics on Networks: from Exact to Approximate Models by Kiss, Miller & Simon (possibly freely available for download here depending on your institutional subscription).

Please cite the book if using these algorithms.

If you use EoN, please - cite the Journal of Open Source Software publication - leave a note here

This helps me **get promoted / get funding**, and it makes me happy when people use the software. A happy developer whose job prospects are improving because of all the people using the software will work to improve the software.

For simulations, we assume that input networks are **NetworkX** graphs; see https://networkx.github.io/

EoN consists of several sets of algorithms.

- The first deals with **stochastic simulation of epidemics on networks**. The most significant of these are `fast_SIS` and `fast_SIR` which usually outperform Gillespie algorithms (also included). These algorithms are discussed in more detail in the appendix of the book.
- A significant extension of these simulations is a set of tools designed to **visualize and animate simulated epidemics**, and generally help investigate a given stochastic simulation.
- Another set deals with **numerical solution of systems of analytic equations** derived in the book. For these it is possible to either provide the degree distribution, or simply use a network and let the code determine the degree distribution.
- There are a few additional algorithms which are not described in the book, but which we believe will be useful. Most notably, related to visualization and generation of animations.

Distributed under MIT license. See `license.txt` for full details.

2.3.2 Simulation Toolkit

This submodule deals with epidemic simulation. We start with a quick list of the functions with links to the individual functions. A brief description is below.

Quick list

fast_SIR(G, tau, gamma[, initial_infecteds, ...])	fast SIR simulation for exponentially distributed infection and recovery times
fast_nonMarkov_SIR(G[, trans_time_fxn, ...])	A modification of the algorithm in figure A.3 of Kiss, Miller, & Simon to allow for user-defined rules governing time of transmission.

Continued on next page

Table 1 – continued from previous page

fast_SIS(G, tau, gamma[, initial_infecteds, ...])	Fast SIS simulations for epidemics on weighted or unweighted networks, allowing edge and node weights to scale the transmission and recovery rates.
fast_nonMarkov_SIS(G[, trans_time_fxn, ...])	Similar to fast_nonMarkov_SIR.
Gillespie_SIR(G, tau, gamma[, ...])	Performs SIR simulations for epidemics.
Gillespie_SIS(G, tau, gamma[, ...])	Performs SIS simulations for epidemics on networks with or without weighted edges.
Gillespie_Arbitrary(G, ...[, tmin, tmax, ...])	Calls Gillespie_simple_contagion.
Gillespie_simple_contagion(G, ...[, tmin, ...])	Performs simulations for epidemics, allowing more flexibility than SIR/SIS.
Gillespie_complex_contagion(G, ...[, tmin, ...])	Initially intended for a complex contagion.
basic_discrete_SIR(G, p[, ...])	Performs simple discrete SIR simulation assuming constant transmission probability p.
basic_discrete_SIS(G, p[, ...])	Does a simulation of the simple case of all nodes transmitting with probability p independently to each susceptible neighbor and then recovering.
discrete_SIR(G[, test_transmission, args, ...])	Simulates an SIR epidemic on G in discrete time, allowing user-specified transmission rules
percolate_network(G, p)	Performs percolation on a network G with each edge persisting with probability p
directed_percolate_network(G, tau, gamma[, ...])	performs directed percolation, assuming that transmission and recovery are Markovian
nonMarkov_directed_percolate_network_with_transmission(G, ...)	Performs directed percolation on G for user-specified transmission time and recovery time distributions.
nonMarkov_directed_percolate_network(G, xi, ...)	performs directed percolation on a network following user-specified rules.
estimate_SIR_prob_size(G, p)	Uses percolation to estimate the probability and size of epidemics assuming constant transmission probability p
estimate_SIR_prob_size_from_dir_perc(H)	Estimates probability and size of SIR epidemics for an input network after directed percolation
estimate_directed_SIR_prob_size(G, tau, gamma)	Predicts probability and attack rate assuming continuous-time Markovian SIR disease on network G
estimate_nonMarkov_SIR_prob_size_with_transmission(G, ...)	estimate probability and size for user-input transmission and recovery time functions.
estimate_nonMarkov_SIR_prob_size(G, xi, ...)	Predicts epidemic probability and size using nonMarkov_directed_percolate_network.
get_infected_nodes(G, tau, gamma[, ...])	Finds all eventually infected nodes in an SIR simulation, through a percolation approach
percolation_based_discrete_SIR(G, p[, ...])	perfoms a simple SIR epidemic but using percolation as the underlying method.

EoN.fast_SIR

EoN.**fast_SIR**(*G*, *tau*, *gamma*, *initial_infecteds=None*, *initial_recovereds=None*, *rho=None*, *tmin=0*, *tmax=inf*, *transmission_weight=None*, *recovery_weight=None*, *return_full_data=False*, *sim_kwargs=None*)

fast SIR simulation for exponentially distributed infection and recovery times

From figure A.3 of Kiss, Miller, & Simon. Please cite the book if using this algorithm.

Arguments

G networkx Graph The underlying network

tau number transmission rate per edge

gamma number recovery rate per node

initial_infecteds node or iterable of nodes if a single node, then this node is initially infected

if an iterable, then whole set is initially infected

if None, then choose randomly based on rho.

If rho is also None, a random single node is chosen.

If both initial_infecteds and rho are assigned, then there is an error.

initial_recovereds iterable of nodes (default None) this whole collection is made recovered. Currently there is no test for consistency with initial_infecteds. Understood that everyone who isn't infected or recovered initially is initially susceptible.

rho number initial fraction infected. number is int(round(G.order()*rho))

tmin number (default 0) starting time

tmax number (default Infinity) maximum time after which simulation will stop. the default of running to infinity is okay for SIR, but not for SIS.

transmission_weight string (default None) the label for a weight given to the edges. transmission rate is G.adj[i][j][transmission_weight]*tau

recovery_weight string (default None)) a label for a weight given to the nodes to scale their recovery rates gamma_i = G.nodes[i][recovery_weight]*gamma

return_full_data boolean (default False) Tells whether a Simulation_Investigation object should be returned.

sim_kwargs keyword arguments Any keyword arguments to be sent to the Simulation_Investigation object Only relevant if `return_full_data=True`

Returns

times, S, I, R numpy arrays

Or if `return_full_data is True`

full_data Simulation_Investigation object from this we can extract the status history of all nodes. We can also plot the network at given times and create animations using class methods.

SAMPLE USE

```
import networkx as nx
import EoN
import matplotlib.pyplot as plt

G = nx.configuration_model([1,5,10]*100000)
initial_size = 10000
gamma = 1.
tau = 0.3
t, S, I, R = EoN.fast_SIR(G, tau, gamma,
                          initial_infecteds = range(initial_size))

plt.plot(t, I)
```

EoN.fast_nonMarkov_SIR

EoN.**fast_nonMarkov_SIR**(*G*, *trans_time_fxn=None*, *rec_time_fxn=None*, *trans_and_rec_time_fxn=None*, *trans_time_args=()*, *rec_time_args=()*, *trans_and_rec_time_args=()*, *initial_infecteds=None*, *initial_recovereds=None*, *rho=None*, *tmin=0*, *tmax=inf*, *return_full_data=False*, *sim_kwargs=None*)

A modification of the algorithm in figure A.3 of Kiss, Miller, & Simon to allow for user-defined rules governing time of transmission.

Please cite the book if using this algorithm.

This is useful if the transmission rule is non-Markovian in time, or for more elaborate models.

Allows the user to define functions (details below) to determine the rules of transmission times and recovery times. There are two ways to do this. The user can define a function that calculates the recovery time and another function that calculates the transmission time. If recovery is after transmission, then transmission occurs. We do this if the time to transmission is independent of the time to recovery.

Alternately, the user may want to model a situation where time to transmission and time to recovery are not independent. Then the user can define a single function (details below) that would determine both recovery and transmission times.

Arguments

G Networkx Graph

trans_time_fxn a user-defined function returns the delay until transmission for an edge. May depend on various arguments and need not be Markovian.

Returns float

Will be called using the form

trans_delay = trans_time_fxn(source_node, target_node, *trans_time_args)
Here trans_time_args is a tuple of the additional arguments the functions needs.

the source_node is the infected node the target_node is the node that may receive transmission rec_delay is the duration of source_node's infection, calculated by rec_time_fxn.

rec_time_fxn a user-defined function returns the delay until recovery for a node. May depend on various arguments and need not be Markovian.

Returns float.

Called using the form

rec_delay = rec_time_fxn(node, *rec_time_args) Here rec_time_args is a uple of additional arguments the function needs.

trans_and_rec_time_fxn a user-defined function returns both a dict giving delay until transmissions for all edges from source to susceptible neighbors and a float giving delay until recovery of the source.

Can only be used **INSTEAD OF** `trans_time_fxn` AND `rec_time_fxn`.

Gives an **ERROR** if these are also defined

Called using the form ‘‘trans_delay_dict, rec_delay = trans_and_rec_time_fxn(

node, susceptible_neighbors, *trans_and_rec_time_args)‘‘

here trans_delay_dict is a dict whose keys are those neighbors who receive a transmission and rec_delay is a float.

trans_time_args tuple see trans_time_fxn

rec_time_args tuple see rec_time_fxn

trans_and_rec_time_args tuple see trans_and_rec_time_fxn

initial_infecteds node or iterable of nodes if a single node, then this node is initially infected

if an iterable, then whole set is initially infected

if None, then choose randomly based on rho. If rho is also None, a random single node is chosen.

If both initial_infecteds and rho are assigned, then there is an error.

initial_recovereds iterable of nodes (default None) this whole collection is made recovered.

Currently there is no test for consistency with initial_infecteds.

Understood that everyone who isn't infected or recovered initially is initially susceptible.

rho number initial fraction infected. number is int(round(G.order()*rho))

tmin number (default 0) starting time

tmax number (default infinity) final time

return_full_data boolean (default False) Tells whether a Simulation_Investigation object should be returned.

sim_kwargs keyword arguments Any keyword arguments to be sent to the Simulation_Investigation object Only relevant if `return_full_data=True`

Returns

times, S, I, R numpy arrays

Or if `return_full_data is True`

full_data Simulation_Investigation object from this we can extract the status history of all nodes We can also plot the network at given times and even create animations using class methods.

SAMPLE USE

```
import EoN
import networkx as nx
import matplotlib.pyplot as plt
import random

N=1000000
G = nx.fast_gnp_random_graph(N, 5/(N-1.))



#set up the code to handle constant transmission rate
#with fixed recovery time.
def trans_time_fxn(source, target, rate):
    return random.expovariate(rate)

def rec_time_fxn(node,D):
    return D

D = 5
tau = 0.3
initial_inf_count = 100
```

(continues on next page)

(continued from previous page)

```
t, S, I, R = EoN.fast_nonMarkov_SIR(G,
                        trans_time_fxn=trans_time_fxn,
                        rec_time_fxn=rec_time_fxn,
                        trans_time_args=(tau,),
                        rec_time_args=(D,),
                        initial_infecteds = range(initial_inf_count))

# note the comma after ``tau`` and ``D``.  This is needed for python
# to recognize these are tuples

# initial condition has first 100 nodes in G infected.
```

EoN.fast_SIS

EoN.**fast_SIS**(*G*, *tau*, *gamma*, *initial_infecteds=None*, *rho=None*, *tmin=0*, *tmax=100*, *transmission_weight=None*, *recovery_weight=None*, *return_full_data=False*, *sim_kwargs=None*)

Fast SIS simulations for epidemics on weighted or unweighted networks, allowing edge and node weights to scale the transmission and recovery rates. Assumes exponentially distributed times to recovery and to transmission.

From figure A.5 of Kiss, Miller, & Simon. Please cite the book if using this algorithm.

Arguments

G networkx Graph The underlying network

tau positive float transmission rate per edge

gamma number recovery rate per node

initial_infecteds node or iterable of nodes if a single node, then this node is initially infected if an iterable, then whole set is initially infected if None, then choose randomly based on rho. If rho is also None, a random single node is chosen. If both initial_infecteds and rho are assigned, then there is an error.

rho number initial fraction infected. number infected is int(round(G.order()*rho))

tmin number (default 0) starting time

tmax number (default 100) stop time

transmission_weight string (default None) the label for a weight given to the edges. transmission rate is G.adj[i][j][transmission_weight]*tau

recovery_weight string (default None) a label for a weight given to the nodes to scale their recovery rates `gamma_i = G.nodes[i][recovery_weight]*gamma`

return_full_data boolean (default False) Tells whether a Simulation_Investigation object should be returned.

sim_kwargs keyword arguments Any keyword arguments to be sent to the Simulation_Investigation object Only relevant if `return_full_data=True`

Returns

times, S, I each a numpy array times and number in each status for corresponding time

or if return_full_data=True

full_data Simulation_Investigation object from this we can extract the status history of all nodes. We can also plot the network at given times and even create animations using class methods.

SAMPLE USE

```
import networkx as nx
import EoN
import matplotlib.pyplot as plt

G = nx.configuration_model([1,5,10]*100000)
initial_size = 10000
gamma = 1.
tau = 0.2
t, S, I = EoN.fast_SIS(G, tau, gamma, tmax = 10,
                               initial_infecteds = range(initial_size))

plt.plot(t, I)
```

EoN.fast_nonMarkov_SIS

EoN.**fast_nonMarkov_SIS**(*G*, *trans_time_fxn=None*, *rec_time_fxn=None*, *trans_and_rec_time_fxn=None*, *trans_time_args=()*, *rec_time_args=()*, *trans_and_rec_time_args=()*, *initial_infecteds=None*, *rho=None*, *tmin=0*, *tmax=100*, *return_full_data=False*, *sim_kwargs=None*)

Similar to fast_nonMarkov_SIR.

Warning

trans_time_fxn (or trans_and_rec_time_fxn) need to return lists of times. Not just the next time. So this is different from the SIR version.

Arguments

G networkx Graph The underlying network

trans_time_fxn User-defined function returning a list

RETURNS A LIST

has slightly different arguments than the SIR version

a user-defined function that returns list of delays until transmission for an edge. All delays are before recovery.

Each entry is the delay from time of infection of node to time of the given transmission (i.e., it's not looking at delays from one transmission to the next)

May depend on various arguments and need not be Markovian.

Called using the form

trans_delays = trans_time_fxn(source_node, target_node, rec_delay, *trans_time_args)

the source_node is the infected node

the target_node is the node that may receive transmission

rec_time_fxn user-designed function returning a float

Returns the duration of infection of a node. May depend on various arguments and need not be Markovian.

Called using the form

duration = rec_time_fxn(node, *rec_time_args)

trans_and_rec_time_fxn user-defined function returning a dict and a float

returns both a dict whose values are lists of delays until transmissions for all edges from source to neighbors and a float giving duration of infection of the source.

can only be **used instead of** `trans_time_fxn` and `rec_time_fxn`. there is an **error** if these are also defined.

Called using the form

trans_delay_dict, duration = trans_and_rec_time_fxn(node, susceptible_neighbors, *trans_and_rec_time_args)

here

trans_delay_dict is a dict whose keys are those neighbors who receive a transmission and whose values are lists of delays

duration is a float.

trans_time_args tuple see trans_time_fxn

rec_time_args tuple see rec_time_fxn

trans_and_rec_time_args tuple see trans_and_rec_time_fxn

initial_infecteds node or iterable of nodes if a single node, then this node is initially infected if an iterable, then whole set is initially infected if None, then choose randomly based on rho. If rho is also None, a random single node is chosen. If both initial_infecteds and rho are assigned, then there is an error.

rho number initial fraction infected. number is int(round(G.order()*rho))

tmin number (default 0) starting time

tmax number (default 100) stop time

return_full_data boolean (default False) Tells whether a Simulation_Investigation object should be returned.

sim_kwargs keyword arguments Any keyword arguments to be sent to the Simulation_Investigation object Only relevant if `return_full_data=True`

Returns

times, S, I each a numpy array giving times and number in each status for corresponding time

or if return_full_data=True:

full_data a Simulation_Investigation object from this we can extract the status history of all nodes We can also plot the network at given times and even create animations using class methods.

SAMPLE USE

EoN.Gillespie_SIR

`EoN.`**`Gillespie_SIR`**(*G*, *tau*, *gamma*, *initial_infecteds=None*, *initial_recovereds=None*, *rho=None*, *tmin=0*, *tmax=inf*, *recovery_weight=None*, *transmission_weight=None*, *return_full_data=False*, *sim_kwargs=None*)

Performs SIR simulations for epidemics.

For unweighted networks, the run time is usually slower than fast_SIR, but they are close. If we add weights, then this Gillespie implementation slows down much more.

I think there are better ways to implement the algorithm to remove this. This will need a new data type that allows us to quickly sample a random event with appropriate weight. I think this is doable through a binary tree and it is in development.

Rather than using figure A.1 of Kiss, Miller, & Simon, this uses a method from Petter Holme

> "Model versions and fast algorithms for network epidemiology"

which focuses on SI edges (versions before 0.99.2 used a method more like fig A.1).

This approach will not work for nonMarkovian transmission. Boguna et al "Simulating non-Markovian stochastic processes"

have looked at how to handle nonMarkovian transmission in a Gillespie Algorithm. At present I don't see a way to efficientl adapt their approach - I think each substep will take O(N) time. So the full algorithm will be O(N^2). For this, it will be much better to use fast_SIR which I believe is O(N log N)

See Also

fast_SIR which has the same inputs but uses a different method to run faster, particularly in the weighted case.

Arguments

G networkx Graph The underlying network

tau positive float transmission rate per edge

gamma number recovery rate per node

initial_infecteds node or iterable of nodes if a single node, then this node is initially infected if an iterable, then whole set is initially infected if None, then choose randomly based on rho. If rho is also None, a random single node is chosen. If both initial_infecteds and rho are assigned, then there is an error.

initial_recovereds iterable of nodes (default None) this whole collection is made recovered. Currently there is no test for consistency with initial_infecteds. Understood that everyone who isn't infected or recovered initially is initially susceptible.

rho number initial fraction infected. number is int(round(G.order()*rho))

tmin number (default 0) starting time

tmax number (default Infinity) stop time

recovery_weight string (default None) the string used to define the node attribute for the weight. Assumes that the recovery rate is gamma*G.nodes[u][recovery_weight]. If None, then just uses gamma without scaling.

transmission_weight string (default None) the string used to define the edge attribute for the weight. Assumes that the transmission rate from u to v is tau*G.adj[u][v][transmission_weight] If None, then just uses tau without scaling.

return_full_data boolean (default False) Tells whether a Simulation_Investigation object should be returned.

sim_kwargs keyword arguments Any keyword arguments to be sent to the Simulation_Investigation object Only relevant if `return_full_data=True`

Returns

times, S, I, R each a numpy array giving times and number in each status for corresponding time

OR if return_full_data=True:

full_data Simulation_Investigation object from this we can extract the status history of all nodes We can also plot the network at given times and even create animations using class methods.

SAMPLE USE

```
import networkx as nx
import EoN
import matplotlib.pyplot as plt

G = nx.configuration_model([1,5,10]*100000)
initial_size = 10000
gamma = 1.
tau = 0.3
t, S, I, R = EoN.Gillespie_SIR(G, tau, gamma,
                               initial_infecteds = range(initial_size))

plt.plot(t, I)
```

EoN.Gillespie_SIS

EoN.**Gillespie_SIS**(*G*, *tau*, *gamma*, *initial_infecteds=None*, *rho=None*, *tmin=0*, *tmax=100*, *recovery_weight=None*, *transmission_weight=None*, *return_full_data=False*, *sim_kwargs=None*)

Performs SIS simulations for epidemics on networks with or without weighted edges.

It assumes that the edges have a weight associated with them and that the transmission rate for an edge is tau*weight[edge]

Based on an algorithm by Petter Holme. It requires a weighted choice of edges and this will be done by tracking the maximum edge weight and then using repeated rejection samples until a successful selection.

See Also

fast_SIS which has the same inputs but uses a faster method (esp for weighted graphs).

Arguments

G (NetworkX Graph) The underlying network

tau (positive float) transmission rate per edge

gamma number recovery rate per node

initial_infecteds node or iterable of nodes if a single node, then this node is initially infected if an iterable, then whole set is initially infected if None, then choose randomly based on rho. If rho is also None, a random single node is chosen. If both initial_infecteds and rho are assigned, then there is an error.

rho number initial fraction infected. number is int(round(G.order()*rho))

tmin number (default 0) starting time

tmax number stop time

recovery_weight string (default None) the string used to define the node attribute for the weight. Assumes that the recovery rate is gamma*G.nodes[u][recovery_weight]. If None, then just uses gamma without scaling.

transmission_weight string (default None) the string used to define the edge attribute for the weight. Assumes that the transmission rate from u to v is tau*G.adj[u][v][transmission_weight]

return_full_data boolean (default False) Tells whether a Simulation_Investigation object should be returned.

sim_kwargs keyword arguments Any keyword arguments to be sent to the Simulation_Investigation object Only relevant if `return_full_data=True`

Returns

times, S, I numpy arrays giving times and number in each status for corresponding time

or if `return_full_data==True`

full_data Simulation_Investigation object from this we can extract the status history of all nodes We can also plot the network at given times and even create animations using class methods.

SAMPLE USE

```
import networkx as nx
import EoN
import matplotlib.pyplot as plt

G = nx.configuration_model([1,5,10]*100000)
initial_size = 10000
gamma = 1.
tau = 0.2
t, S, I = EoN.Gillespie_SIS(G, tau, gamma, tmax = 20,
                            initial_infecteds = range(initial_size))

plt.plot(t, I)
```

EoN.Gillespie_Arbitrary

EoN.**Gillespie_Arbitrary**(*G*, *spontaneous_transition_graph*, *nbr_induced_transition_graph*, *IC*, *return_statuses*, *tmin=0*, *tmax=100*, *spont_kwargs=None*, *nbr_kwargs=None*, *return_full_data=False*, *sim_kwargs=None*)

Calls Gillespie_simple_contagion. This is here for legacy reasons.

Gillespie_Arbitrary has been replaced by Gillespie_simple_contagion. It will be removed in future versions.

EoN.Gillespie_simple_contagion

EoN.**Gillespie_simple_contagion**(*G*, *spontaneous_transition_graph*, *nbr_induced_transition_graph*, *IC*, *return_statuses*, *tmin=0*, *tmax=100*, *spont_kwargs=None*, *nbr_kwargs=None*, *return_full_data=False*, *sim_kwargs=None*)

Performs simulations for epidemics, allowing more flexibility than SIR/SIS.

This does not handle complex contagions. It assumes that when an individual changes status either he/she has received a "transmission" from a *single* neighbor or he/she is changing status independently of any neighbors. So this is like SIS or SIR. Other examples would be like SEIR, SIRS, etc

There is an example below demonstrating an SEIR epidemic.

We allow for nodes to undergo two types of transitions. They can be:

- spontaneous - so a node of status A becomes B without any neighbor's influence

- neighbor-induced - so an edge between status A and status B nodes suddenly becomes an edge between a status A and a status C node because the status B node changes status due to the existence of the edge. (in principle, both could change status, but the code currently only allows the second node to change status).

Both types of transitions can be represented by weighted directed graphs. We describe two weighted graphs whose nodes represent the possible statuses and whose edges represent possible transitions of statuses.

- The spontaneous transitions can be represented by a graph whose nodes are the possible statuses and an edge from `'A'` to `'B'` represent that in the absence of any influence from others an indivdiual of status `'A'` transitions to status `'B'` with default rate given by the weight of the edge in this spontaneous transition graph. The rate may be modified by properties of the nodes in the contact network `G`.
- The neighbor-induced transitions can be represented by a "transitions graph" whose nodes are length-2 tuples. The first entry represents the first individual of a partnership and the second represents the second individual. **only the second individual changes status**. An edge in the transitions graph from the node `('A', 'B')` to the node `('A', 'C')` represents that an `'AB'` partnership in the contact network can cause the second individual to transition to status `'C'`. The weight of the edge in the represents the default transition rate. The rate may be modified by properties of the nodes or the edge in the contact network `G`.

[for reference, if you look at Fig 4.3 on pg 122 of Kiss, Miller & Simon the graphs for **SIS** would be:

`spontaneous_transition_graph`: `'I'`-> `'S'` with the edge weighted by `gamma` and

`nbr_induced_transition_graph`: `('I', 'S')` -> `('I', 'I')` with the edge weighted by `tau`.

For SIR they would be: `spontaneous_transition_graph`: `'I'`->‘‘’R’‘ with weight `gamma` and

`nbr_induced_transition_graph`: `('I', 'S')` -> `('I', 'I')` with rate `tau`.]

These graphs must be defined and then input into the algorithm.

It is possible to weight edges or nodes in the contact network `G` (that is, not the 2 directed networks defined above, but the original contact network) so that some of these transitions have different rates for different individuals/partnerships. These are included as attributes in the contact network. In the most general case, the transition rate depends on some function of the attributes of the nodes or edge in the contact network `G`.

There are two ways we introduce individual or pair-level heterogeneity in the population. The first way is through introducing weights to individuals or their contacts which simply multiply the default rate. The second is through including a function of nodes or pairs of nodes which will explicitly calculate the scaling factor, possibly including more information than we can include with the weights. For a given transition, you can use at most one of these.

- We first describe examples of weighting nodes/edges in the population

So for the SIR case, if some people have higher recovery rate, we might define a node attribute `'recovery_weight'` for each node in `G`, and the recovery would occur with rate G.nodes[node][‘recovery_weight’]*gamma. So a value of 1.1 would correspond to a 10% increased recovery rate. Since I don't know what name you might choose for the weight label as I write this algorithm, in defining the spontaneous transition graph (H), the `'I'`->‘‘’R’‘ edge would be given an attribute `'weight_label'` so that `H.adj['I']['R']['weight_label'] = 'recovery_weight'`. If you define the attribute `'weight_label'` for an edge in `H`, then it will be assumed that every node in `G` has a corresponding weight. If no attribute is given, then it is assumed that all transitions happen with the original rate.

We similarly define the weight_labels as edge attributes in the neighbor-induced transition graph. The edges of the graph `'G'` have a corresponding `G[u,v]['transmission_weight']`

- Alternately we might introduce a function.

So for the SIR case if the recovery rate depends on two attributes of a node (say, age and gender), we define a function `rate_function(G,node)` which will then look at G.nodes[node][‘age’] and

G.nodes[node]['gender'] and then return a factor which will be multiplied by `gamma` to give the recovery rate. Similarly, if we are considering a neighbor induced transition and the rate depends on properties of both nodes we define another function `rate_function(G,u,v)` which may use attributes of u or v or the edge to find the appropriate scaling factor.

Arguments

G NetworkX Graph The underlying contact network. If `G` is directed, we assume that "transmissions" can only go in the same direction as the edge.

spontaneous_transition_graph Directed networkx graph The nodes of this graph are the possible statuses of a node in `G`. An edge in this graph is a possible transition in `G` that occurs without any influence from neighbors.

An edge in this directed graph is labelled with attributes

- `'rate'` [a number, the default rate of the transition]
- **`'weight_label'` (optional) [a string, giving the label of** a **node** attribute in the contact network `G` that scales the transition rate]
- **`'rate_function'` (optional not combinable with `'weight_label'`** for some edge.) [a user-defined function of the contact network and node that will scale the transition rate. This cannot depend on the statuses of any nodes - we must be able to calculate it once at the beginning of the process.] It will be called as `rate_function(G, node,**spont_kwargs)` where `spont_kwargs` is described below.

Only one of `'weight_label'` and `'rate_function'` can be given.

In the description below, let's use

- `rate = spontaneous_transition_graph.adj[Status1][Status2]['rate']`
- `weight_label = spontaneous_transition_graph.adj[Status1][Status2]['weight_label']`
- `rate_function = spontaneous_transition_graph.adj[Status1][Status2]['rate_function']`

For a node `u` whose status is `Status1`, the rate at which `u` transitions to `Status2` is

- `rate` if neither `weight_label` nor `rate_function` is defined.
- `rate*G.nodes[u][weight_label]` if `weight_label` is defined.
- **`rate*rate_function(G, u, **spont_kwargs)` if `rate_function` is** defined.

So for example in the case of an SIR disease, this would be a graph with an isolated node `'S'` and an edge from node `'I'` to `'R'` with `rate` equal to `gamma` (the recovery rate). It would not actually be necessary to have the node `'S'`.

Note that the rate_function version is more general, and in principle we don't need the weight_label option. It's here for backwards compatibility and general simplicity purposes.

nbr_induced_transition_graph Directed networkx graph

The nodes of this graph are tuples with possible statuses of nodes at the end of an edge. The first node in the tuple is the node that could be affecting the second. So for example for the SIR model we would expect a node `('I', 'S')` with an edge to `('I', 'I')`.

An edge in this directed graph is labelled with attributes

- `'rate'` [a number, the default rate of the transition]

- **'weight_label' (optional) [a string, giving the label of** an *edge** attribute in the contact network G that scales the transition rate]
- **'rate_function' (optional not combinable with 'weight_label'** for some edge.) [a user-defined function of the contact network and source and target nodes that will scale the transition rate. This cannot depend on the statuses of any nodes - we must be able to calculate it once at the beginning of the process. It will be called as `rate_function(G, source, target, *nbr_kwargs)`]

Only one of `'weight_label'` and `'rate_function'` can be given.

In the description below, let's use

- `rate = spontaneous_transition_graph.adj[Status1][Status2]['rate']`
- `weight_label = spontaneous_transition_graph.adj[Status1][Status2]['weight_label']`
- `rate_function = spontaneous_transition_graph.adj[Status1][Status2]['rate_function']`

If the transition is (A,B) to (A,C) and we have a source node of status A joined to a target node of status B then the target node transitions from B to C with rate

- `rate` if neither `weight_label` nor `rate_function` is defined.
- `rate*G.adj[source][target][weight_label]` if `weight_label` is defined.
- `rate*rate_function(G, source, target, **nbr_kwargs)` if `rate_function` is defined

So for example in the case of an SIR disease with transmission rate tau, this would be a graph with an edge from the node `('I','S')` to `('I', 'I')`. The attribute `'rate'` for the edge would be `tau`.

IC dict states the initial status of each node in the network.

return_statuses list or other iterable (but not a generator) The statuses that we will return information for, in the order we will return them.

tmin number (default 0) starting time

tmax number (default 100) stop time

spont_kwargs dict or None (default None) Any parameters which might be needed if the user has defined a rate function for a spontaneous transition. If any of the spontaneous transition rate functions accepts these, they all need to (even if not used). It's easiest to define the function as def f(..., **kwargs)

nbr_kwargs dict or None (default None) Any parameters which might be needed if the user has defined a rate function for a neighbor-induced transition. If any of the neighbor-induced transition rate functions accepts these, they all need to (even if not used). It's easiest to define the function as def f(..., **kwargs)

return_full_data boolean Tells whether to return a Simulation_Investigation object or not

sim_kwargs keyword arguments Any keyword arguments to be sent to the Simulation_Investigation object Only relevant if `return_full_data=True`

Returns

(times, status1, status2, ...) tuple of numpy arrays first entry is the times at which events happen. second (etc) entry is an array with the same number of entries as `times` giving the number of nodes of status ordered as they are in `return_statuses`

SAMPLE USE

This does an SEIR epidemic. It treats the nodes and edges as weighted. So some individuals have a higher E->I transition rate than others, and some edges have a higher transmission rate than others. The recovery rate is taken to be the same for all nodes.

There are more examples in the online documentation at :ref:`simple-contagion-section`.

```
import EoN
import networkx as nx
from collections import defaultdict
import matplotlib.pyplot as plt
import random

N = 100000
G = nx.fast_gnp_random_graph(N, 5./(N-1))

#they will vary in the rate of leaving exposed class.
#and edges will vary in transition rate.
#there is no variation in recovery rate.

node_attribute_dict = {node: 0.5+random.random() for node in G.nodes()}
edge_attribute_dict = {edge: 0.5+random.random() for edge in G.edges()}

nx.set_node_attributes(G, values=node_attribute_dict, name='expose2infect_weight')
nx.set_edge_attributes(G, values=edge_attribute_dict, name='transmission_weight')


H = nx.DiGraph()
H.add_node('S')
H.add_edge('E', 'I', rate = 0.6, weight_label='expose2infect_weight')
H.add_edge('I', 'R', rate = 0.1)

J = nx.DiGraph()
J.add_edge(('I', 'S'), ('I', 'E'), rate = 0.1, weight_label='transmission_weight')
IC = defaultdict(lambda: 'S')
for node in range(200):
    IC[node] = 'I'

return_statuses = ('S', 'E', 'I', 'R')

t, S, E, I, R = EoN.Gillespie_simple_contagion(G, H, J, IC, return_statuses,
                                               tmax = float('Inf'))

plt.semilogy(t, S, label = 'Susceptible')
plt.semilogy(t, E, label = 'Exposed')
plt.semilogy(t, I, label = 'Infected')
plt.semilogy(t, R, label = 'Recovered')
plt.legend()

plt.savefig('SEIR.png')
```

EoN.Gillespie_complex_contagion

EoN.**Gillespie_complex_contagion**(*G*, *rate_function*, *transition_choice*, *get_influence_set*, *IC*, *return_statuses*, *tmin=0*, *tmax=100*, *parameters=None*, *return_full_data=False*, *sim_kwargs=None*)

Initially intended for a complex contagion. However, this can allow influence from any nodes, not just immediate neighbors.

The complex contagion must be something that all nodes do something simultaneously

This is not the same as if node `v` primes node `u` and later node `w` causes `u` to transition. This will require that both `v` and `w` have the relevant states at the moment of transition and it has forgotten any previous history.

Arguments

G (NetworkX Graph) The underlying network

rate_function A function that will take the network, a node, and the statuses of all the nodes and calculate the rate at which the node changes its status.

The function is called like

if parameters is None: rate_function(G, node, status)

else: rate_function(G, node, status, parameters)

where G is the graph, node is the node, status is a dict such that status[u] returns the status of u, and parameters is the parameters passed to the function.

it returns a number, the combined rate at which `node` might change status.

transition_choice A function that takes the network, a node, and the statuses of all the nodes and chooses which event will happen. The function should be called [with or without `parameters`]

if `parameters` is `None`: `transition_choice(G, node, status)`

else: `transition_choice(G, node, status, parameters)`

where `G` is the graph, ``node` is the node, ``status` is a dict such that ``status[u]` returns the status of u, and `parameters` is the parameters passed to the function.

It should return the new status of `node` based on the fact that the node is changing status.

get_influence_set When a node `u` changes status, we want to know which nodes may have their rate altered. We need to update their rates. This function returns all nodes that may be affected by `u` (either in its previous state or its current state). We will go through and recalculate the rates for all of these nodes. For a contagion, we can simply choose all neighbors, but it may be faster to leave out any nodes that it wouldn't have affected before or after its transition (e.g., R or I neighbors in SIR).

if `parameters` is None: `get_influence_set(G, node, status)`

else: `get_influence_set(G, node, status, parameters)`

where G is the graph, node is the node, status is a dict such that status[u] returns the status of u, and parameters is the parameters passed to the function.

it should return the set of nodes whose rates need to be recalculated.

Most likely, it is

def get_influence_set(G, node, status): return G.neighbors(node)

IC A dict. IC[node] returns the status of node.

return_statuses list or other iterable (but not a generator) The statuses that we will return information for, in the order we will return them.

tmin number (default 0) starting time

tmax number (default 100) stop time

return_full_data boolean currently needs to be False. True raises an error.

parameters list/tuple. Any parameters of the functions rate_function, transition_choice, influence_set We assume all three functions can accept parameters. Examples: recovery rate, transmission rate, ...

Returns

(times, status1, status2, ...) tuple of numpy arrays first entry is the times at which events happen. second (etc) entry is an array with the same number of entries as `times` giving the number of nodes of status ordered as they are in `return_statuses`

SAMPLE USE

This simply does an SIR epidemic, by saying that the rate of becoming infected is tau times the number of infected neighbors.

```
import networkx as nx
import EoN
import matplotlib.pyplot as plt
from collections import defaultdict #makes defining the initial condition easier


def rate_function(G, node, status, parameters):
    #This function needs to return the rate at which node changes status.
    #
    tau,gamma = parameters
    if status[node] == 'I':
        return gamma
    elif status[node] == 'S':
        return tau*len([nbr for nbr in G.neighbors(node) if status[nbr] == 'I'])
    else:
        return 0

def transition_choice(G, node, status, parameters):
    #this function needs to return the new status of node.  We already
    #know it is changing status.
    #
    #this function could be more elaborate if there were different
    #possible transitions that could happen.
    if status[node] == 'I':
        return 'R'
    elif status[node] == 'S':
        return 'I'

def get_influence_set(G, node, status, parameters):
    #this function needs to return any node whose rates might change
    #because ``node`` has just changed status.
    #
    #the only neighbors a node might affect are the susceptible ones.

    return {nbr for nbr in G.neighbors(node) if status[nbr] == 'S'}
```

(continues on next page)

(continued from previous page)

```
G = nx.fast_gnp_random_graph(100000,0.00005)

gamma = 1.
tau = 0.5
parameters = (tau, gamma)

IC = defaultdict(lambda: 'S')
for node in range(200):
    IC[node] = 'I'

t, S, I, R = EoN.Gillespie_complex_contagion(G, rate_function,
                            transition_choice, get_influence_set, IC,
                            return_statuses=('S', 'I', 'R'),
                            parameters=parameters)

plt.plot(t, I)
```

EoN.basic_discrete_SIR

EoN.**basic_discrete_SIR**(*G*, *p*, *initial_infecteds=None*, *initial_recovereds=None*, *rho=None*, *tmin=0*, *tmax=inf*, *return_full_data=False*, *sim_kwargs=None*)

Performs simple discrete SIR simulation assuming constant transmission probability p.

From figure 6.8 of Kiss, Miller, & Simon. Please cite the book if using this algorithm.

Does a simulation of the simple case of all nodes transmitting with probability p independently to each neighbor and then recovering.

Arguments

G networkx Graph The network the disease will transmit through.

p number transmission probability

initial_infecteds node or iterable of nodes (default None) if a single node, then this node is initially infected if an iterable, then whole set is initially infected if None, then choose randomly based on rho. If rho is also None, a random single node is chosen. If both initial_infecteds and rho are assigned, then there is an error.

initial_recovereds as for initial_infecteds, but for initially recovered nodes.

rho number (default None) initial fraction infected. number initially infected is int(round(G.order()*rho))

The default results in a single randomly chosen initial infection.

tmin float (default 0) start time

tmax float (default infinity) stop time (if not extinct first).

return_full_data boolean (default False) Tells whether a Simulation_Investigation object should be returned.

sim_kwargs keyword arguments Any keyword arguments to be sent to the Simulation_Investigation object Only relevant if `return_full_data=True`

Returns

if return_full_data is False returns

t, S, I, R numpy arrays

these numpy arrays give all the times observed and the number in each state at each time.

Or `if return_full_data is True` returns full_data Simulation_Investigation object

from this we can extract the status history of all nodes We can also plot the network at given times and even create animations using class methods.

SAMPLE USE

```
import networkx as nx
import EoN
import matplotlib.pyplot as plt
G = nx.fast_gnp_random_graph(1000,0.002)
t, S, I, R = EoN.basic_discrete_SIR(G, 0.6)
plt.plot(t,S)


#This sample may be boring if the randomly chosen initial infection
#doesn't trigger an epidemic.
```

EoN.basic_discrete_SIS

EoN.**basic_discrete_SIS**(*G*, *p*, *initial_infecteds=None*, *rho=None*, *tmin=0*, *tmax=100*, *return_full_data=False*, *sim_kwargs=None*)

Does a simulation of the simple case of all nodes transmitting with probability p independently to each susceptible neighbor and then recovering.

This is not directly described in Kiss, Miller, & Simon.

Arguments

G networkx Graph The network the disease will transmit through.

p number transmission probability

initial_infecteds node or iterable of nodes (default None) if a single node, then this node is initially infected if an iterable, then whole set is initially infected if None, then choose randomly based on rho. If rho is also None, a random single node is chosen. If both initial_infecteds and rho are assigned, then there is an error.

rho number initial fraction infected. number is int(round(G.order()*rho))

return_full_data boolean (default False) Tells whether a Simulation_Investigation object should be returned.

sim_kwargs keyword arguments Any keyword arguments to be sent to the Simulation_Investigation object Only relevant if `return_full_data=True`

Returns

if return_full_data is False t, S, I

All numpy arrays

if return_full_data is True full_data Simulation_Investigation object

from this we can extract the status history of all nodes We can also plot the network at given times and even create animations using class methods.

SAMPLE USE

```
import networkx as nx
import EoN
import matplotlib.pyplot as plt
G = nx.fast_gnp_random_graph(1000,0.002)
t, S, I = EoN.basic_discrete_SIS(G, 0.6, tmax = 20)
plt.plot(t,S)
```

EoN.discrete_SIR

EoN.**discrete_SIR**(*G*, *test_transmission=<function _simple_test_transmission_>*, *args=()*, *initial_infecteds=None*, *initial_recovereds=None*, *rho=None*, *tmin=0*, *tmax=inf*, *return_full_data=False*, *sim_kwargs=None*)

Simulates an SIR epidemic on G in discrete time, allowing user-specified transmission rules

From figure 6.8 of Kiss, Miller, & Simon. Please cite the book if using this algorithm.

Return details of epidemic curve from a discrete time simulation.

It assumes that individuals are infected for exactly one unit of time and then recover with immunity.

This is defined to handle a user-defined function `test_transmission(node1,node2,*args)` which determines whether transmission occurs.

So elaborate rules can be created as desired by the user.

By default it uses `_simple_test_transmission_` in which case args should be entered as (p,)

Arguments

G NetworkX Graph (or some other structure which quacks like a

NetworkX Graph)

The network on which the epidemic will be simulated.

test_transmission function(u,v,*args) (see below for args definition) A function that determines whether u transmits to v. It returns True if transmission happens and False otherwise. The default will return True with probability p, where args=(p,)

This function can be user-defined. It is called like: test_transmission(u,v,*args) Note that if args is not entered, then args=(), and this call is equivalent to test_transmission(u,v)

args a list or tuple The arguments of test_transmission coming after the nodes. If simply having transmission with probability p it should be entered as args=(p,)

[note the comma is needed to tell Python that this is really a tuple]

initial_infecteds node or iterable of nodes (default None) if a single node, then this node is initially infected if an iterable, then whole set is initially infected if None, then choose randomly based on rho. If rho is also None, a random single node is chosen. If both initial_infecteds and rho are assigned, then there is an error.

initial_recovereds as for initial_infecteds, but initially recovered nodes.

rho number (default is None) initial fraction infected. initial number infected is int(round(G.order()*rho)).

The default results in a single randomly chosen initial infection.

tmin start time

tmax stop time (default Infinity).

return_full_data boolean (default False) Tells whether a Simulation_Investigation object should be returned.

sim_kwargs keyword arguments Any keyword arguments to be sent to the Simulation_Investigation object Only relevant if `return_full_data=True`

Returns

t, S, I, R numpy arrays

Or `if return_full_data is True` returns

full_data Simulation_Investigation object from this we can extract the status history of all nodes We can also plot the network at given times and even create animations using class methods.

SAMPLE USE

```
import networkx as nx
import EoN
import matplotlib.pyplot as plt
G = nx.fast_gnp_random_graph(1000,0.002)
t, S, I, R = EoN.discrete_SIR(G, args = (0.6,),
                                    initial_infecteds=range(20))
plt.plot(t,I)
```

Because this sample uses the defaults, it is equivalent to a call to basic_discrete_SIR

EoN.percolate_network

EoN.**percolate_network**(*G*, *p*)

Performs percolation on a network G with each edge persisting with probability p

From figure 6.10 of Kiss, Miller, & Simon. Please cite the book if using this algorithm.

Performs bond percolation on the network G, keeping edges with probability p

Arguments

G networkx Graph The contact network

p number between 0 and 1 the probability of keeping edge

Returns

H NetworkX Graph A network with same nodes as G, but with each edge retained independently with probability p.

SAMPLE USE

```
import networkx as nx
import EoN
import matplotlib.pyplot as plt
G = nx.fast_gnp_random_graph(1000,0.002)
H = EoN.percolate_network(G, 0.6)

#H is now a graph with about 60% of the edges of G
```

EoN.directed_percolate_network

EoN.**directed_percolate_network**(*G*, *tau*, *gamma*, *weights=True*)

performs directed percolation, assuming that transmission and recovery are Markovian

From figure 6.13 of Kiss, Miller, & Simon. Please cite the book if using this algorithm.

This performs directed percolation corresponding to an SIR epidemic assuming that transmission is at rate tau and recovery at rate gamma

See Also

nonMarkov_directed_percolate_network which allows for duration and time to infect to come from other distributions.

nonMarkov_directed_percolate_network which allows for more complex rules

Arguments

G networkx Graph The network the disease will transmit through.

tau positive float transmission rate

gamma positive float recovery rate

weights boolean (default True) if True, then includes information on time to recovery and delay to transmission. If False, just the directed graph.

Returns :

H networkx DiGraph (directed graph) a u->v edge exists in H if u would transmit to v if ever infected.

The edge has a time attribute (time_to_infect) which gives the delay from infection of u until transmission occurs.

Each node u has a time attribute (duration) which gives the duration of its infectious period.

SAMPLE USE

```
import networkx as nx
import EoN

G = nx.fast_gnp_random_graph(1000,0.002)
H = EoN.directed_percolate_network(G, 2, 1)
```

EoN.nonMarkov_directed_percolate_network_with_timing

EoN.**nonMarkov_directed_percolate_network_with_timing**(*G*, *trans_time_fxn*, *rec_time_fxn*, *trans_time_args=()*, *rec_time_args=()*, *weights=True*)

Performs directed percolation on G for user-specified transmission time and recovery time distributions.

A generalization of figure 6.13 of Kiss, Miller & Simon

See Also

directed_percolate_network if it's just a constant transmission and recovery rate.

nonMarkov_directed_percolate_network if your rule for creating the percolated network cannot be expressed as simply calculating durations and delays until transmission.

Arguments

The arguments are very much like in fast_nonMarkov_SIR

G Networkx Graph the input graph

trans_time_fxn user-defined function returns the delay from u's infection to transmission to v.

delay=trans_time_fxn(u, v, *trans_time_args)

rec_time_fxn user-defined function returns the duration of from u's infection (delay to its recovery).

duration = rec_time_fxn(u, *rec_time_args)

Note: if delay == duration, we assume infection happens.

trans_time_args tuple any additional arguments required by trans_time_fxn. For example weights of nodes.

rec_time_args tuple any additional arguments required by rec_time_fxn

weights boolean if true, then return directed network with the delay and duration as weights.

Returns

H directed graph.

The returned graph is built by assigning each node an infection duration and then each edge (in each direction) a delay until transmission. If delay<duration it adds directed edge to `H`.

if weights is True, then `H` contains the duration and delays as weights. Else it's just a directed graph.

EoN.nonMarkov_directed_percolate_network

`EoN.`**`nonMarkov_directed_percolate_network`**(*G*, *xi*, *zeta*, *transmission*)

performs directed percolation on a network following user-specified rules.

From figure 6.18 of Kiss, Miller, & Simon. Please cite the book if using this algorithm.

This algorithm is particularly intended for a case where the duration and delays from infection to transmission are somehow related to one another.

Warning

You probably **shouldn't use this**. Check if nonMarkov_directed_percolate_with_timing fits your needs better.

See Also

`nonMarkov_directed_percolate_network_with_timing`

if your rule for creating the percolated network is based on calculating a recovery time for each node and then calculating a separate transmission time for the edges this will be better.

`directed_percolate_network`

if it's just a constant transmission and recovery rate.

Arguments

G networkx Graph The input graph

xi dict xi[u] gives all necessary information to determine what us infectiousness is.

zeta dict zeta[v] gives everything needed about vs susceptibility

transmission user-defined function transmission(xi[u], zeta[v]) determines whether u transmits to v.

returns True if transmission happens and False if it does not

Returns

H networkx DiGraph (directed graph) Edge (u,v) exists in H if disease will transmit given the opportunity.

SAMPLE USE

for now, I'm being lazy. Look at the sample for estimate_nonMarkov_SIR_prob_size to infer it.

EoN.estimate_SIR_prob_size

EoN.**estimate_SIR_prob_size**(*G*, *p*)

Uses percolation to estimate the probability and size of epidemics assuming constant transmission probability p

From figure 6.12 of Kiss, Miller, & Simon. Please cite the book if using this algorithm.

Provies an estimate of epidemic probability and size assuming a fixed transmission probability p.

The estimate is found by performing bond percolation and then finding the largest connected component in the remaining network.

This assumes that there is a single giant component above threshold.

It will not be an appropriate measure if the network is made up of several densely connected components with very weak connections between these components.

Arguments

G networkx Graph The network the disease will transmit through.

p number transmission probability

Returns

PE, AR both floats between 0 and 1 estimates of the probability and proportion infected (attack rate) in epidemics (the two are equal, but each given for consistency with `estimate_directed_SIR_prob_size`)

SAMPLE USE

```
import networkx as nx
import EoN

G = nx.fast_gnp_random_graph(1000,0.002)
PE, AR = EoN.estimate_SIR_prob_size(G, 0.6)
```

EoN.estimate_SIR_prob_size_from_dir_perc

EoN.**estimate_SIR_prob_size_from_dir_perc**(*H*)

Estimates probability and size of SIR epidemics for an input network after directed percolation

From figure 6.17 of Kiss, Miller, & Simon. Please cite the book if using this algorithm

Arguments

H directed graph The outcome of directed percolation on the contact network G

Returns

PE, AR numbers Estimates of epidemic probability and attack rate found by finding largest strongly connected component and finding in/out components.

SAMPLE USE

```
import networkx as nx
import EoN

G = nx.fast_gnp_random_graph(1000,0.003)
H = some_user_defined_operation_to_do_percolation(G, argument)
PE, AR = EoN.estimate_SIR_prob_size_from_dir_perc(H)
```

EoN.estimate_directed_SIR_prob_size

EoN.**estimate_directed_SIR_prob_size**(*G*, *tau*, *gamma*)

Predicts probability and attack rate assuming continuous-time Markovian SIR disease on network G

From figure 6.17 of Kiss, Miller, & Simon. Please cite the book if using this algorithm

See Also

`estimate_nonMarkov_SIR_prob_size` which handles nonMarkovian versions

Arguments

G networkx Graph The network the disease will transmit through.

tau positive float transmission rate

gamma positive float recovery rate

Returns

PE, AR numbers (between 0 and 1) Estimates of epidemic probability and attack rate found by performing directed percolation, finding largest strongly connected component and finding its in/out components.

SAMPLE USE

```
import networkx as nx
import EoN

G = nx.fast_gnp_random_graph(1000,0.003)
PE, AR = EoN.estimate_directed_SIR_prob_size(G, 2, 1)
```

EoN.estimate_nonMarkov_SIR_prob_size_with_timing

EoN.**estimate_nonMarkov_SIR_prob_size_with_timing**(*G*, *trans_time_fxn*, *rec_time_fxn*, *trans_time_args=()*, *rec_time_args=()*)

estimates probability and size for user-input transmission and recovery time functions.

Arguments

G Networkx Graph the input graph

trans_time_fxn function trans_time_fxn(u, v, *trans_time_args) returns the delay from u's infection to transmission to v.

rec_time_fxn function rec_time_fxn(u, *rec_time_args) returns the delay from u's infection to its recovery.

trans_time_args tuple any additional arguments required by trans_time_fxn. For example weights of nodes.

rec_time_args tuple any additional arguments required by rec_time_fxn

Returns

PE, AR numbers (between 0 and 1) Estimates of epidemic probability and attack rate found by finding largest strongly connected component and finding in/out components.

SAMPLE USE

```
#mimicking the standard version with transmission rate tau
#and recovery rate gamma
#equivalent to
#PE, AR = EoN.estimate_SIR_prob_size(G, tau, gamma)

import networkx as nx
import EoN
import random
from collections import defaultdict

G=nx.fast_gnp_random_graph(1000,0.002)

tau = 2

gamma = 1

def trans_time_fxn(u,v, tau):

    return random.expovariate(tau)

def rec_time_fxn(u, gamma):

    return random.expovariate(gamma)

PE, AR = EoN.estimate_nonMarkov_SIR_prob_size(G,
                                              trans_time_fxn=trans_time_fxn,
                                              rec_time_fxn = rec_time_fxn,
                                              trans_time_args = (tau,),
                                              rec_time_args = (gamma,)
                                              )
```

EoN.estimate_nonMarkov_SIR_prob_size

EoN.**estimate_nonMarkov_SIR_prob_size**(*G*, *xi*, *zeta*, *transmission*)

Predicts epidemic probability and size using nonMarkov_directed_percolate_network.

This is not directly described in Kiss, Miller, & Simon, but is based on (fig 6.18).

Warning

You probably DON'T REALLY WANT TO USE THIS. Check if estimate_nonMarkov_prob_size_with_timing fits your needs better.

Arguments

G (networkx Graph) The input graph

xi dict xi[u] gives all necessary information to determine what u's infectiousness is.

zeta dict zeta[v] gives everything needed about v's susceptibility

transmission user-defined function transmission(xi[u], zeta[v]) determines whether u transmits to v. Returns True or False depending on whether the transmission would happen

Returns

PE, AR numbers (between 0 and 1) Estimates of epidemic probability and attack rate found by finding largest strongly connected component and finding in/out components.

SAMPLE USE

```
#mimicking the standard version with transmission rate tau
#and recovery rate gamma

import networkx as nx
import EoN
import random
from collections import defaultdict

G=nx.fast_gnp_random_graph(1000,0.002)
tau = 2
gamma = 1

xi = {node:random.expovariate(gamma) for node in G.nodes()}
#xi[node] is duration of infection of node.

zeta = defaultdict(lambda : tau) #every node has zeta=tau, so same
                                 #transmission rate

def my_transmission(infection_duration, trans_rate):
    #infect if duration is longer than time to infection.
    if infection_duration > random.expovariate(trans_rate):
        return True
    else:
        return False

PE, AR = EoN.estimate_nonMarkov_SIR_prob_size(G, xi, zeta,
                                              my_transmission)
```

EoN.get_infected_nodes

EoN.**get_infected_nodes**(*G*, *tau*, *gamma*, *initial_infecteds=None*, *initial_recovereds=None*)
Finds all eventually infected nodes in an SIR simulation, through a percolation approach

From figure 6.15 of Kiss, Miller, & Simon. Please cite the book if using this algorithm

Assumes that the intial infecteds are as given and transmission occurs with rate tau and recovery with rate gamma.

Note that the output of this algorithm is stochastic.

This code has similar run-time whether an epidemic occurs or not. There are much faster ways to implement an algorithm giving the same output, for example by actually running one of the epidemic simulation tools.

Warning

why are you using this command? If it's to better understand the relationship between percolation and SIR epidemics, that's fine. But this command IS NOT an efficient way to calculate anything. Don't do it like this. Use one of the other algorithms. Try `fast_SIR`, for example.

Arguments

G networkx Graph The network the disease will transmit through.

tau positive float transmission rate

gamma positive float recovery rate

initial_infecteds node or iterable of nodes if a single node, then this node is initially infected if an iterable, then whole set is initially infected if None, then a randomly chosen node is initially infected.

initial_recovereds node or iterable of nodes if a single node, then this node is initially recovered if an iterable, then whole set is initially recovered

Returns

infected_nodes set the set of nodes infected eventually in a simulation.

SAMPLE USE

```
import networkx as nx
import EoN

G = nx.fast_gnp_random_graph(1000,0.002)

finalR = EoN.get_infected_nodes(G, 2, 1, initial_infecteds=[0, 5])

#finds the nodes infected if 0 and 5 are the initial nodes infected
#and tau=2, gamma=1
```

EoN.percolation_based_discrete_SIR

EoN.**percolation_based_discrete_SIR**(*G*, *p*, *initial_infecteds=None*, *initial_recovereds=None*, *rho=None*, *tmin=0*, *tmax=inf*, *return_full_data=False*, *sim_kwargs=None*)
perfoms a simple SIR epidemic but using percolation as the underlying method.

From figure 6.10 of Kiss, Miller, & Simon. Please cite the book if using this algorithm.

The simple case of all nodes transmitting with probability p independently to each neighbor and then recovering, but using a percolation-based approach.

Warning

You probably **shouldn't use this algorithm**.

See `basic_discrete_SIR` which produces equivalent outputs.

That algorithm will be faster than this one.

The value of this function is that by performing many simulations we can see that the outputs of the two are equivalent.

This algorithm leads to a better understanding of the theory, but it's not computationally efficient.

Arguments

G networkx Graph The network the disease will transmit through.

p number transmission probability

initial_infecteds node or iterable of nodes (default None) if a single node, then this node is initially infected if an iterable, then whole set is initially infected if None, then choose randomly based on rho. If rho is also None, a random single node is chosen. If both initial_infecteds and rho are assigned, then there is an error.

initial_recovereds as for initial_infecteds, but initially recovered nodes.

rho number initial fraction infected. number is int(round(G.order()*rho))

tmin start time

tmax stop time (if not extinct first). Default step is 1.

return_full_data boolean (default False) Tells whether a Simulation_Investigation object should be returned.

sim_kwargs keyword arguments Any keyword arguments to be sent to the Simulation_Investigation object Only relevant if `return_full_data=True`

Returns

t, S, I, R numpy arrays

OR if `return_full_data is True`:

full_data Simulation_Investigation object from this we can extract the status history of all nodes We can also plot the network at given times and even create animations using class methods.

SAMPLE USE

```
import networkx as nx
import EoN
import matplotlib.pyplot as plt
G = nx.fast_gnp_random_graph(1000,0.002)
t, S, I, R = EoN.percolation_based_discrete_SIR(G, p)
plt.plot(t,S)
```

This is equivalent to basic_discrete_epidemic (but many simulations may be needed before it's clear, since these are stochastic)

Short descriptions

- Event-based algorithms:

  These algorithms use an efficient approach to simulate epidemics. `fast_SIR` and `fast_SIS` assume constant transmission and recovery rates, while `fast_nonMarkov_SIR` and `fast_nonMarkov_SIS` allow the user to specify more detailed rules for transmission.

  - **fast_SIR**
  - **fast_nonMarkov_SIR**
  - **fast_SIS**
  - **fast_nonMarkov_SIS**

- Gillespie Algorithms

  These algorithms simulate epidemics assuming constant transmission and recovery rates. They are commonly used, but in many cases are slower than the event driven methods. I do not see evidence that they are ever significantly faster. It is not very practical to get away from the constant rate assumptions so I prefer to avoid them. However, `Gillespie_simple_contagion` allows the user to do SEIR, SIRS, or any of a number of other more exotic "simple contagion" scenarios that are not in the event-driven code. `Gillespie_complex_contagion` handles complex contagions, in which an individual requires multiple partners to have a given state before it changes status. For legacy reasons, `Gillespie_Arbitrary` is included, it simply calls `Gillespie_simple_contagion`, and will be removed in future versions.

  - **Gillespie_SIR**
  - **Gillespie_SIS**
  - **Gillespie_Arbitrary**
  - **Gillespie_simple_contagion**
  - **Gillespie_complex_contagion**

- Discrete-time algorithms

  These algirthms are appropriate for where we separate infection into generations. We assume infection lasts a single time step. The `basic_*` algorithms assume that transmission occurs with probability p for all edges. In contrast `discrete_SIR` allows for very general user-specified transmission rules.

  - **basic_discrete_SIR**
  - **basic_discrete_SIS**
  - **discrete_SIR**

- Percolation-based approaches

  There is a close relation between percolation and SIR disease which is described in Chapter 6 of the book. Many of these algorithms are related to demonstrating the equivalence as outlined in the book, and are not really the most efficient way to simulate an epidemic. However, these algorithms will be useful for estimating probability and size of epidemics.

  - **percolate_network** (undirected percolation corresponding to fixed transmission probability)
  - **directed_percolate_network** (directed percolation corresponding to constant transmission and recovery rates)
  - **nonMarkov_directed_percolate_network_with_timing** (uses user-generated duration and transmission time distributions)
  - **nonMarkov_directed_percolate_network** (uses user-generated transmission rules)

- **estimate_SIR_prob_size** (estimates prob/size from an undirected percolated network - only appropriate if constant p)
- **estimate_SIR_prob_size_from_dir_perc** (estimates epi prob and size from a given percolated network)
- **estimate_directed_SIR_prob_size** (estimates based on constant transmission and recovery rates)
- **estimate_nonMarkov_SIR_prob_size_with_timing** (estimates based on user-generated transmission and recovery time distributions)
- **estimate_nonMarkov_SIR_prob_size** (estimates based on user-generated transmission rules)
- **get_infected_nodes** (simulates epidemic and returns final infected nodes)
- **percolation_based_discrete_SIR**

2.3.3 Simulation Investigation toolkit

We can study simulations in detail through the Simulation_Investigation class. This includes automated generation of animations.

This is particularly useful if we want to look at time series or at animations of the network as the disease spreads.

When EoN performs a simulation with `return_full_data` set to `True`, it returns a Simulation_Investigation object. At it's core, this has the data about when each node changed status and what its new status became. This allows us to generate plots of the network at any given instance in time and to produce animations.

The basic display produced by a Simulation_Investigation object shows the network at a given snapshot in time on the left, and on the right it shows the time series of S, I, and (if SIR) R. It has the option to add additional curves that might have been calculated by an analytic model, or perhaps another simulation.

In general, any of the dynamic simulations will produce a Simulation_Investigation object if we pass it `return_full_data = True`.

Some examples appear in section *Visualizing or animating disease spread*.

Quick list

display(time[, ts_plots, ts_list, nodelist, ...])	Provides a plot of the network at a specific time and (optionally) some of the time series
animate([frame_times, ts_plots, ts_list, ...])	As in display, but this produces an animation.
node_history(node)	returns the history of a node.
node_status(node, time)	returns the status of a given node at a given time.
get_statuses([nodelist, time])	returns the status of nodes at a given time.
summary([nodelist])	Provides the population-scale summary of the dynamics.
t()	Returns the times of events Generally better to get these all through summary()
S()	If `'S'` is a state, then this will return the number susceptible at each time.
I()	See notes for S
R()	See notes for S
transmissions()	Returns a list of tuples (t,u,v) stating that node u infected node v at time t.
transmission_tree()	Produces a MultiDigraph whose edges correspond to transmission events.

Continued on next page

Table 2 – continued from previous page

add_timeseries(ts_data[, color_dict, label, tex])	This allows us to include some additional timeseries for comparision with the simulation.
update_ts_kwargs(ts, **kwargs)	Allows us to change some of the matplotlib key word arguments for a timeseries object
update_ts_label(ts, label)	updates the label for time series plots
update_ts_color_dict(ts, color_dict)	updates the color_dict for time series plots
update_ts_tex(ts, tex)	updates the tex flag for time series plots
sim_update_kwargs(**kwargs)	Allows us to change some of the matplotlib key word arguments for the simulation.
sim_update_label(label)	updates the label for the simulation in the time series plots
sim_update_color_dict(color_dict)	updates the color_dict for the simulation
sim_update_tex(tex)	updates the tex flag for the simulation in the time series plots and in the network plots
set_pos(pos)	Set the position of the nodes.

EoN.Simulation_Investigation.display

Simulation_Investigation.**display**(*time*, *ts_plots=None*, *ts_list=None*, *nodelist=None*, *status_order=False*, *timelabel='t'*, *pos=None*, *statuses_to_plot=None*, ***nx_kwargs*)

Provides a plot of the network at a specific time and (optionally) some of the time series

By default it plots the network and all time series. The time series are plotted in 3 (for SIR) or 2 (for SIS) different plots to the right of the network. There are options to control how many plots appear and which time series objects are plotted in it.

We can make the number of time series plots to the right be zero by setting ts_plots to be an empty list.

Arguments

time float the time for the snapshot of the network.

ts_plots (list of strings, defaults to `statuses_to_plot`, which defaults

to `self._possible_statuses_`)

if `[]` or `False` then the display only shows the network.

lists such as `[['S'], ['I'], ['R']]` or `[['S', 'I'], ['R']]`

equivalently `['S', 'I', 'R']` and `['SI', 'R']` will do the same but is problematic if a status has a string longer than 1.

denotes what should appear in the timeseries plots. The length of the list determines how many plots there are. If entry i is `['A', 'B']` then plot i has both `'A'` and `'B'` plotted. . So `[['S'], ['I'], ['R']]` or `['SIR']` will result in 3 plots, one with just `'S'`, one with just `'I'` and one with just `'R'`

while `[['S', 'I'], ['R']]` or `['SI', 'R']` will result in 2 plots, one with both `'S'` and `'I'` and one with just `'R'`.

Defaults to the possible_statuses

ts_list (list of timeseries objects - default None) If multiple time series have been added, we might want to plot only some of them. This says which ones to plot. The simulation is always included.

nodelist (list, default None) which nodes should be included in the network plot. By default this is the entire network. This also determines which nodes are on top of each other (particularly if `status_order` is `False`).

status_order list of statuses default `False` Each status will appear on top of all later statuses. If list empty or `False`, will ignore. Any statuses not appearing in list will simply be below those on the list and will not have priority by status.

timelabel (string, default `'$t$'`) the horizontal label to be used on the time series plots

pos overrides self.pos for this display (but does not overwrite self.pos. Use set_pos if you want to do this)

statuses_to_plot list of statuses to plot. If given, then the other nodes will be left invisible when plotting but I think this requires networkx v2.3 or later.

****nx_kwargs** any networkx keyword arguments to go into the network plot.

Returns

network_ax, ts_ax_list (axis, list of axises) The axes for the network plot and a list of all the axes for the timeseries plots

Notes :

If you only want to plot the graph, set ts_plots equal to [].

If you want S, I, and R on a single plot, set ts_plots equal to ['SIR']

If you only want some of the timeseries objects, set ts_list to be those (the simulation time series will always be plotted).

Examples :

To show a plot where sim is the Simulation_Investigation object simply do

```
sim.display()
plt.show()
```

To save it,

```
sim.display()
plt.savefig(filename).
```

If you want to do more detailed modifications of the plots, this returns the axes:

```
network_ax, timeseries_axes = sim.display()
```

EoN.Simulation_Investigation.animate

Simulation_Investigation.**animate**(*frame_times=None*, *ts_plots=None*, *ts_list=None*, *nodelist=None*, *status_order=False*, *timelabel='t'*, *pos=None*, *statuses_to_plot=None*, ***nx_kwargs*)

As in display, but this produces an animation.

To display an animation where sim is the Simulation_Investigation object simply do

```
sim.animate()
plt.show()
```

To save an animation [on a mac with appropriate additional libraries installed], you can do

```
ani = sim.animate()
ani.save(filename, fps=5, extra_args=['-vcodec', 'libx264'])
```

here `ani` is a matplotlib animation. See

https://matplotlib.org/api/_as_gen/matplotlib.animation.Animation.save.html

for more about the save command for matplotlib animations.

Arguments

The same as in display, except that time is replaced by frame_times

frame_times (list/numpy array) The times for animation frames. If nothing is given, then it uses 101 times between 0 and t[-1]

ts_plots (list of strings, defaults to `statuses_to_plot`, which defaults

to `self._possible_statuses_`)

if `[]` or `False` then the display only shows the network.

lists such as `[['S'], ['I'], ['R']]` or `[['S', 'I'], ['R']]`

equivalently `['S', 'I', 'R']` and `['SI', 'R']` will do the same but is problematic if a status has a string longer than 1.

denotes what should appear in the timeseries plots. The length of the list determines how many plots there are. If entry i is `['A', 'B']` then plot i has both `'A'` and `'B'` plotted. . So `[['S'], ['I'], ['R']]` or `['SIR']` will result in 3 plots, one with just `'S'`, one with just `'I'` and one with just `'R'`

while `[['S', 'I'], ['R']]` or `['SI', 'R']` will result in 2 plots, one with both `'S'` and `'I'` and one with just `'R'`.

Defaults to the possible_statuses

ts_list list of timeseries objects (default None) If multiple time series have been added, we might want to plot only some of them. This says which ones to plot. The simulation is always included.

nodelist list (default None) which nodes should be included in the network plot. By default this is the entire network. This also determines which nodes are on top of each other (particularly if status_order is `False`).

status_order list of statuses default `False` Each status will appear on top of all later statuses. If list empty or `False`, will ignore. Any statuses not appearing in list will simply be below those on the list and will not have priority by status.

timelabel string (default 't') the horizontal label to be used on the time series plots

pos dict (default None) overrides self.pos for this display (but does not overwrite self.pos. Use set_pos if you want to do this)

statuses_to_plot list of statuses to plot. If given, then the other nodes will be left invisible when plotting but I think this requires networkx v2.3 or later.

****nx_kwargs** any networkx keyword arguments to go into the network plot.

EoN.Simulation_Investigation.node_history

`Simulation_Investigation.`**`node_history`**`(`*`node`*`)`

returns the history of a node.

Arguments

node the node

Returns

timelist, statuslist lists

the times at which the node changes status and what the new status is at each time.

EoN.Simulation_Investigation.node_status

Simulation_Investigation.**node_status**(*node*, *time*)

returns the status of a given node at a given time.

Arguments

node the node

time float the time of interest.

Returns

status string (such as *'S'*, *'I'*, or *'R'*) status of node at time.

EoN.Simulation_Investigation.get_statuses

Simulation_Investigation.**get_statuses**(*nodelist=None*, *time=None*)

returns the status of nodes at a given time.

Arguments

nodelist iterable (default None): Some sort of iterable of nodes. If default value, then returns statuses of all nodes.

time float (default None) the time of interest. if default value, then returns initial time

Returns

status dict A dict whose keys are the nodes in nodelist giving their status at time.

EoN.Simulation_Investigation.summary

Simulation_Investigation.**summary**(*nodelist=None*)

Provides the population-scale summary of the dynamics. It returns a numpy array t as well as numpy arrays for each of the `possible_statuses` giving how many nodes had that status at the corresponding time.

Assumes that all entries in node_history start with same tmin

Arguments

nodelist (default None) The nodes that we want to focus on. By default this is all nodes. If you want all nodes, the most efficient thing to do is to not include `'nodelist'`. Otherwise it will recalculate everything.

Returns

summary tuple a pair (t, D) where - t is a numpy array of times and - D is a dict whose keys are the possible statuses and whose values

are numpy arrays giving the count of each status at the specific times.

If nodelist is empty, this is for the entire graph. Otherwise it is just for the node in nodelist.

EoN.Simulation_Investigation.t

Simulation_Investigation.**t**()
Returns the times of events Generally better to get these all through summary()

EoN.Simulation_Investigation.S

Simulation_Investigation.**S**()
If `'S'` is a state, then this will return the number susceptible at each time.

Else it raises an error

Generally better to get these all through `summary()`

EoN.Simulation_Investigation.I

Simulation_Investigation.**I**()
See notes for S

Returns the number infected at each time Generally better to get these all through summary()

EoN.Simulation_Investigation.R

Simulation_Investigation.**R**()
See notes for S

Returns the number recovered at each time Generally better to get these all through summary()

EoN.Simulation_Investigation.transmissions

Simulation_Investigation.**transmissions**()
Returns a list of tuples (t,u,v) stating that node u infected node v at time t. In the standard code, if v was already infected at tmin, then the source is None

Note - this only includes successful transmissions. So if u tries to infect v, but fails because v is already infected this is not recorded.

EoN.Simulation_Investigation.transmission_tree

Simulation_Investigation.**transmission_tree**()
Produces a MultiDigraph whose edges correspond to transmission events. If SIR, then this is a tree (or a forest).

Returns

T a directed Multi graph T has all the information in `transmissions`. An edge from u to v with time t means u transmitted to v at time t.

Warning

Although we refer to this as a "tree", if the disease is SIS, there are likely to be cycles and/or repeated edges. If the disease is SIR but there are multiple initial infections, then this will be a "forest".

If it's an SIR, then this is a tree (or forest).

The graph contains only those nodes that are infected at some point.

EoN.Simulation_Investigation.add_timeseries

Simulation_Investigation.**add_timeseries**(*ts_data*, *color_dict=None*, *label=None*, *tex=None*, ***kwargs*)

This allows us to include some additional timeseries for comparision with the simulation. So for example, if we perform a simulation and want to plot the simulation but also a prediction, this is what we would use.

Arguments

ts_data a pair (t, D) where t is a numpy array of times and D is a dict where D[status] is the number of individuals of given status at corresponding time.

color_dict dict (default None) a dictionary mapping statuses to the color desired for their plots. Defaults to the same as the simulation

label (string) The label to be used for these plots in the legend.

tex (boolean) Tells whether status should be rendered in tex's math mode in labels. Defaults to whatever was done for creation of this simulation_investigation object.

****kwargs** any matplotlib key word args to affect how the curve is shown.

Returns

ts timeseries object

Modifies

This adds the timeseries object `ts` to the internal `_time_series_list_`

EoN.Simulation_Investigation.update_ts_kwargs

Simulation_Investigation.**update_ts_kwargs**(*ts*, ***kwargs*)

Allows us to change some of the matplotlib key word arguments for a timeseries object

Arguments

ts (timeseries object) the timeseries object whose key word args we are updating.

****kwargs** the new matplotlib key word arguments

EoN.Simulation_Investigation.update_ts_label

Simulation_Investigation.**update_ts_label**(*ts*, *label*)

updates the label for time series plots

Arguments

ts timeseries object the timeseries object whose key word args we are updating.

label string the new label

EoN.Simulation_Investigation.update_ts_color_dict

Simulation_Investigation.**update_ts_color_dict**(*ts*, *color_dict*)
updates the color_dict for time series plots

Arguments

ts timeseries object the timeseries object whose key word args we are updating.

color_dict dict the new color_dict

EoN.Simulation_Investigation.update_ts_tex

Simulation_Investigation.**update_ts_tex**(*ts*, *tex*)
updates the tex flag for time series plots

Arguments

ts (timeseries object) the timeseries object whose key word args we are updating.

tex the new value for `tex`

EoN.Simulation_Investigation.sim_update_kwargs

Simulation_Investigation.**sim_update_kwargs**(***kwargs*)
Allows us to change some of the matplotlib key word arguments for the simulation. This is identical to update_ts_kwargs except we don't need to tell it which time series to use.

Arguments

****kwargs** the new matplotlib key word arguments

EoN.Simulation_Investigation.sim_update_label

Simulation_Investigation.**sim_update_label**(*label*)
updates the label for the simulation in the time series plots

Arguments

label string the new `label`

EoN.Simulation_Investigation.sim_update_color_dict

Simulation_Investigation.**sim_update_color_dict**(*color_dict*)
updates the color_dict for the simulation

Arguments

color_dict dict the new color_dict

EoN.Simulation_Investigation.sim_update_tex

Simulation_Investigation.**sim_update_tex**(*tex*)
updates the tex flag for the simulation in the time series plots and in the network plots

Arguments

tex string the new value of `tex`

EoN.Simulation_Investigation.set_pos

Simulation_Investigation.**set_pos**(*pos*)
Set the position of the nodes.

Arguments

pos (dict) as in `nx.draw_networkx`

Short description

- Visualizations

  There are two main commands for visualization. We can either produce a snapshot at a given time, or produce an animation. In either case we can optionally include plots of S, I, (and R) as functions of time.

  - **display (allows us to plot a graph at a specific time** point, and to optionally include the calculated time series)
  - **animate (allows us to plot a graph at many time points** We can create a visualization such as mp4, or save each individual frame of an animation. The time series are optional.

- Data about simulation

  Often we'll want to be able to check what happened to specific nodes in the network, or we'll want to know what the time history of the outbreak looked like

  - **node_history**
  - **node_status** returns the status of a node at a given time
  - **get_statuses returns the status of a collection of nodes at** a given time (in a dict).
  - **summary** returns t, S, I, (and if SIR R) for the population (or a subset of the population)
  - **t**
  - **S**
  - **I**
  - **R**
  - **transmissions** returns a list of 3-tuples of the form (t, u, v) stating that u transmitted to v at time t.
  - **transmission_tree** returns a MultiDiGraph where an edge from u to v with attribute time = t means that u transmitted to v at time t. (For SIR this is a tree or a forest).

- Details for plotting

  The remaining commands are to do with the specifics of how the plots appear

  - **update_ts_kwargs**

  - **update_ts_label**
  - **update_ts_colordict**
  - **update_ts_tex**
  - **sim_update_kwargs**
  - **sim_update_label**
  - **sim_update_colordict**
  - **sim_update_tex**
  - **set_pos**
- Plotting of transmission tree

2.3.4 Analytic Toolkit

This submodule deals with solution to systems of equations appearing in the book. The majority of these also have a version that take a graph G. There are additional functions that calculate properties which these need.

Quick list

SIS_individual_based(G, tau, gamma[, rho, ...])	Encodes System (3.7) of Kiss, Miller, & Simon.
SIS_individual_based_pure_IC(G, tau, gamma, ...)	Encodes System (3.7) of Kiss, Miller, & Simon.
SIS_pair_based(G, tau, gamma[, rho, ...])	Encodes System (3.26) of Kiss, Miller, & Simon.
SIS_pair_based_pure_IC(G, tau, gamma, ...[, ...])	Encodes System (3.26) of Kiss, Miller, & Simon, using a "pure initial condition".
SIR_individual_based(G, tau, gamma[, rho, ...])	Encodes System (3.30) of Kiss, Miller, & Simon.
SIR_individual_based_pure_IC(G, tau, gamma, ...)	Encodes System (3.30) of Kiss, Miller, & Simon.
SIR_pair_based(G, tau, gamma[, rho, ...])	Encodes System (3.39) of Kiss, Miller, & Simon.
SIR_pair_based_pure_IC(G, tau, gamma, ...[, ...])	Encodes System (3.39) of Kiss, Miller, & Simon, using a "pure initial condition".
SIS_homogeneous_meanfield(S0, I0, n, tau, gamma)	Encodes System (4.8) of Kiss, Miller, & Simon.
SIR_homogeneous_meanfield(S0, I0, R0, n, ...)	Encodes System (4.9) of Kiss, Miller, & Simon.
SIS_homogeneous_pairwise(S0, I0, SI0, SS0, ...)	Encodes System (4.10) of Kiss, Miller, & Simon.
SIS_homogeneous_pairwise_from_graph(G, tau, ...)	Calls SIS_homogeneous_pairwise after calculating S0, I0, SI0, SS0, n based on the graph G and initial fraction infected rho.
SIR_homogeneous_pairwise(S0, I0, R0, SI0, ...)	Encodes System (4.11) of Kiss, Miller, & Simon.
SIR_homogeneous_pairwise_from_graph(G, tau, ...)	Calls SIR_homogeneous_pairwise after calculating S0, I0, R0, SI0, SS0, n, based on the graph G and initial fraction infected rho.

Continued on next page

Table 3 – continued from previous page

SIS_heterogeneous_meanfield(Sk0, Ik0, tau, gamma)	Encodes System (5.10) of Kiss, Miller, & Simon.
SIS_heterogeneous_meanfield_from_graph(G, ...)	Calls SIS_heterogeneous_meanfield after calculating Sk0, Ik0 based on the graph G and random fraction infected rho.
SIR_heterogeneous_meanfield(Sk0, Ik0, Rk0, ...)	Encodes System (5.11) of Kiss, Miller, & Simon.
SIR_heterogeneous_meanfield_from_graph(G, ...)	Calls SIR_heterogeneous_meanfield after calculating Sk0, Ik0, Rk0 based on a graph G and initial fraction infected rho.
SIS_heterogeneous_pairwise(Sk0, Ik0, SkSl0, ...)	Encodes System (5.13) of Kiss, Miller, & Simon.
SIS_heterogeneous_pairwise_from_graph(G, ...)	Calls SIS_heterogeneous_pairwise after calculating Sk0, Ik0, SkSl0, SkIl0, IkIl0 from a graph G and initial fraction infected rho.
SIR_heterogeneous_pairwise(Sk0, Ik0, Rk0, ...)	Encodes System (5.15) of Kiss, Miller, & Simon.
SIR_heterogeneous_pairwise_from_graph(G, ...)	Calls SIR_heterogeneous_pairwise after calculating Sk0, Ik0, Rk0, SkSl0, SkIl0 from a graph G and initial fraction infected rho.
SIS_compact_pairwise(Sk0, Ik0, SI0, SS0, ...)	Encodes system (5.18) of Kiss, Miller, & Simon.
SIS_compact_pairwise_from_graph(G, tau, gamma)	Calls SIS_compact_pairwise after calculating Sk0, Ik0, SI0, SS0, II0 from the graph G and initial fraction infected rho.
SIR_compact_pairwise(Sk0, I0, R0, SS0, SI0, ...)	Encodes system (5.19) of Kiss, Miller, & Simon.
SIR_compact_pairwise_from_graph(G, tau, gamma)	Calls SIR_compact_pairwise after calculating Sk0, I0, R0, SS0, SI0 from the graph G and initial fraction infected rho.
SIS_super_compact_pairwise(S0, I0, SS0, SI0, ...)	Encodes system (5.20) of Kiss, Miller, & Simon.
SIS_super_compact_pairwise_from_graph(G, ...)	Calls SIS_super_compact_pairwise after calculating S0, I0, SS0, SI0, II0 from the graph G and initial fraction infected rho
SIR_super_compact_pairwise(R0, SS0, SI0, N, ...)	Encodes system (5.22) of Kiss, Miller, & Simon.
SIR_super_compact_pairwise_from_graph(G, ...)	Calls SIR_super_compact_pairwise after calculating R0, SS0, SI0 from the graph G and initial fraction infected rho
SIS_effective_degree(Ssi0, Isi0, tau, gamma)	Encodes system (5.36) of Kiss, Miller, & Simon. Please cite the
SIS_effective_degree_from_graph(G, tau, gamma)	Calls SIS_effective_degree after calculating Ssi0, Isi0 from the graph G and initialf fraction infected rho.
SIR_effective_degree(S_si0, I0, R0, tau, gamma)	Encodes system (5.38) of Kiss, Miller, & Simon.
SIR_effective_degree_from_graph(G, tau, gamma)	Calls SIR_effective_degree after calculating S_si0, I0, R0 from the graph G and initial fraction infected rho
SIR_compact_effective_degree(Skappa0, I0, ...)	Encodes system (5.43) of Kiss, Miller, & Simon.

Continued on next page

Table 3 – continued from previous page

SIR_compact_effective_degree_from_graph(G, ...)	Calls SIR_compact_effective_degree after calculating Skappa0, I0, R0, SI0 from the graph G and initial fraction infected rho.
SIS_compact_effective_degree(Sk0, Ik0, SI0, ...)	Encodes system (5.44) of Kiss, Miller, & Simon.
SIS_compact_effective_degree_from_graph(G, ...)	Because the SIS compact effective degree model is identical to the compact pairwise model, simply calls SIS_compact_pairwise_from_graph
Epi_Prob_discrete(Pk, p[, number_its])	Encodes System (6.2) of Kiss, Miller, & Simon.
Epi_Prob_cts_time(Pk, tau, gamma[, umin, ...])	Encodes System (6.3) of Kiss, Miller, & Simon.
Epi_Prob_non_Markovian(Pk, Pxidxi, po[, ...])	Encodes system (6.5) of Kiss, Miller, & Simon.
Attack_rate_discrete(Pk, p[, rho, Sk0, ...])	Encodes systems (6.6) and (6.10) of Kiss, Miller, & Simon.
Attack_rate_discrete_from_graph(G, p[, ...])	if initial_infecteds and initial_recovereds is defined, then it will find Sk0, phiS0, and phiR0 and then call Attack_rate_discrete.
Attack_rate_cts_time(Pk, tau, gamma[, ...])	Encodes system (6.7) of Kiss, Miller, & Simon.
Attack_rate_cts_time_from_graph(G, tau, gamma)	Given a graph, predicts the attack rate for Configuration Model networks with the given degree distribution.
Attack_rate_non_Markovian(Pk, Pzetadzeta, pi)	Encodes system (6.8) of Kiss, Miller, & Simon.
Attack_rate_discrete(Pk, p[, rho, Sk0, ...])	Encodes systems (6.6) and (6.10) of Kiss, Miller, & Simon.
EBCM_discrete(N, psihat, psihatPrime, p, phiS0)	Encodes system (6.11) of Kiss, Miller, & Simon.
EBCM_discrete_from_graph(G, p[, ...])	Takes a given graph, finds the degree distribution (from which it gets psi), assumes a constant proportion of the population is infected at time 0, and then uses the discrete EBCM model.
EBCM(N, psihat, psihatPrime, tau, gamma, phiS0)	Encodes system (6.12) of Kiss, Miller, & Simon.
EBCM_uniform_introduction(N, psi, psiPrime, ...)	Handles the case that the disease is introduced uniformly as opposed to depending on degree.
EBCM_from_graph(G, tau, gamma[, ...])	Given network G and rho, calculates N, psihat, psihatPrime, and calls EBCM.
EBCM_pref_mix(N, Pk, Pnk, tau, gamma[, rho, ...])	Encodes the system derived in exercise 6.21 of Kiss, Miller, & Simon.
EBCM_pref_mix_from_graph(G, tau, gamma[, ...])	Takes a given graph, finds degree correlations, and calls EBCM_pref_mix
EBCM_pref_mix_discrete(N, Pk, Pnk, p[, rho, ...])	Encodes the discrete version of exercise 6.21 of Kiss, Miller, & Simon.
EBCM_pref_mix_discrete_from_graph(G, p[, ...])	Takes a given graph, finds degree correlations, and calls EBCM_pref_mix_discrete
get_Pk(G)	Used in several places so that we can input a graph and then we can call the methods that depend on the degree distribution
get_PGF(Pk)	Given a degree distribution (as a dict), returns the probability generating function
get_PGFPrime(Pk)	Given a degree distribution (as a dict) returns the function $\psi'(x)$
get_PGFDPrime(Pk)	Given a degree distribution (as a dict) returns the function $\psi''(x)$

Continued on next page

Table 3 – continued from previous page

estimate_R0(G[, tau, gamma, transmissibility])	provides the estimate of the reproductive number R_0 = T <K^2-K>/<K>

EoN.SIS_individual_based

EoN.**SIS_individual_based**(*G*, *tau*, *gamma*, *rho=None*, *Y0=None*, *nodelist=None*, *tmin=0*, *tmax=100*, *tcount=1001*, *transmission_weight=None*, *recovery_weight=None*, *return_full_data=False*)

Encodes System (3.7) of Kiss, Miller, & Simon. Please cite the book if using this algorithm.

See also: Hadjichrysanthou and Sharkey Epidemic control analysis: Desigining targeted intervention

strategies against epidemics propagated on contact networks,

Journal of Theoretical Biology

<dot{Y}_i> = tau sum_j g_{ij} (1-<Y_i>)<Y_j> - gamma_i <Y_i>

Arguments

G networkx Graph

tau positive float transmission rate of disease

gamma number global recovery rate

rho float between 0 and 1 (default `None`) initial uniformly random probability of being infected. Cannot define both rho and Y0.

Y0 numpy array (default `None`) the array of initial infection probabilities. If Y0 is defined, nodelist must also be defined. If Y0 is defined, rho cannot be defined.

nodelist list (default `None`) list of nodes in `G` in the same order as in Y0. If rho is defined and nodelist is not, then nodelist= G.nodes() Only affects returned values if return_full_data=True.

tmin number (default 0) minimum report time

tmax number (default 100) maximum report time

tcount integer (default 1001) number of reports

transmission_weighta string (default `None`) the label for a weight given to the edges. G.edge[i][j][transmission_weight] = g_{ij}

recovery_weight string (default `None`) a label for a weight given to the nodes to scale their recovery rates so gamma_i = G.nodes[i][recovery_weight]*gamma

return_full_data (default False) If True, returns times, Ss, Is if False, returns times, S, I

Returns

if return_full_data is True: returns **times, Ss, Is** where `times` is a numpy array of times, `Ss` is a 2D numpy array `Ss[i,j]` gives probability `nodelist[i]` is susceptible at time `times[j]`. Similarly for Is.

if return_full data is False: returns **times, S, I** all are numpy arrays. gives times, and expected number susceptible and expected number infected.

SAMPLE USE

```
import networkx as nx
import EoN as EoN

G = nx.configuration_model([3,10]*1000)
N = G.order()
rho = 1./N
t, S, I = EoN.SIS_individual_based(G, 0.3, 1, rho=rho, tmax = 20)
```

EoN.SIS_individual_based_pure_IC

EoN.**SIS_individual_based_pure_IC**(*G*, *tau*, *gamma*, *initial_infecteds*, *nodelist=None*, *tmin=0*, *tmax=100*, *tcount=1001*, *transmission_weight=None*, *recovery_weight=None*, *return_full_data=False*)

Encodes System (3.7) of Kiss, Miller, & Simon. Please cite the book if using this algorithm.

The difference between this and SIS_individual_based is that this one assumes a "pure initial condition", that is, we know exactly what the statuses of the nodes are at the initial time.

<dot{Y}_i> = tau sum_j g_{ij} (1-<Y_i>)<Y_j> - gamma_i <Y_i>

Arguments

G networkx Graph The contact network

tau positive float transmission rate of disease

gamma number (default None) global recovery rate

initial_infecteds list or set the set of nodes initially infected

nodelist list (default None) list of nodes in G in desired order. (only matters if return_full_data==True)

tmin number (default 0) minimum report time

tmax number (default 100) maximum report time

tcount integer (default 1001) number of reports

transmission_weight string (default None) the label for a weight given to the edges. G.edge[i][j][transmission_weight] = g_{ij}

recovery_weight string (default None) a label for a weight given to the nodes to scale their recovery rates

gamma_i = G.nodes[i][recovery_weight]*gamma

return_full_data boolean (default False)

Returns

if return_full_data is True, returns **times, Ss, Is**

if return_full_data is False, returns **times, S, I**

SAMPLE USE

```
import networkx as nx
import EoN
import matplotlib.pyplot as plt
```

(continues on next page)

(continued from previous page)

```
G = nx.configuration_model([3,10]*1000)
nodelist = G.nodes()
initial_infecteds = range(100)
t, S, I = EoN.SIS_individual_based(G, 0.3, 1, initial_infecteds, nodelist,
                                   tmax = 20)
plt.plot(t,I)
```

EoN.SIS_pair_based

EoN.**SIS_pair_based**(*G*, *tau*, *gamma*, *rho=None*, *nodelist=None*, *Y0=None*, *XY0=None*, *XX0=None*, *tmin=0*, *tmax=100*, *tcount=1001*, *transmission_weight=None*, *recovery_weight=None*, *return_full_data=False*)

Encodes System (3.26) of Kiss, Miller, & Simon. Please cite the book if using this algorithm.

WARNING: this does NOT solve the pairwise equations. Look at SIS_homogeneous_pairwise and SIS_heterogeneous_pairwise for that.

This system solves equations for an SIS disease model spreading on a given graph.

It captures the dependence with pairs, but not triples.

It does not include corrections for triangles (or any other cycles).

The corrections for triangles are provided in the text, but not implemented here.

There are some inefficiencies in the implementation: we track all pairs, rather than just those pairs in edges, but this is unlikely to significantly affect the calculation time.

This makes it much easier to vectorize things.

We track pairs in both directions: e.g., XX[1,2] and XX[2,1].

Arguments

G networkx Graph

tau positive float transmission rate of disease

gamma number global recovery rate

rho (float between 0 and 1, default `None`) proportion assumed initially infected. If `None`, then `Y0` is used if `Y0` is also `None`, then `rho = 1./N`

nodelist list list of nodes in `G` in some prescribed order (just since there is no guarantee that `G` returns nodes in the same order if things change a bit.)

Y0 numpy array the array of initial infection probabilities for each node in the same order as in nodelist

XY0 2D numpy array (default `None`) (each dimension has length number of nodes of G) XY0[i,j] is probability node i is susceptible and j is infected. if None, then assumes that infections are introduced randomly according to Y0.

XX0 2D numpy array (default `None`) (each dimension has length number of nodes of G) XX0[i,j] is probability nodes i and j are susceptible. if None, then assumes that infections are introduced randomly according to Y0.

tmin number (default 0) minimum report time

tmax number (default 100) maximum report time

tcount integer (default 1001) number of reports

transmission_weight string the label for a weight given to the edges. G.edge[i][j][transmission_weight] = g_{ij}

recovery_weight string (default `None`) a label for a weight given to the nodes to scale their recovery rates

gamma_i = G.nodes[i][recovery_weight]*gamma

return_full_data boolean (default False)

if True: returns times, S, I, R, Xs, Ys, Zs, XY, XX

if False: returns times, S, I, R

Returns

if return_full_data is True: returns **times, S, I, Xs, Ys, XY, XX**

if False: returns times, **S, I**

SAMPLE USE

```
import networkx as nx
import EoN

G = nx.fast_gnp_random_graph(1000,0.004)
nodelist = G.nodes()
Y0 = np.array([1 if node<10 else 0 for node in nodelist]) #infect first 10
t, S, I = EoN.SIS_pair_based(G, 2, 0.5, nodelist, Y0, tmax = 4, tcount = 101)
plt.plot(t,I)
```

EoN.SIS_pair_based_pure_IC

EoN.**SIS_pair_based_pure_IC**(*G*, *tau*, *gamma*, *initial_infecteds*, *nodelist=None*, *tmin=0*, *tmax=100*, *tcount=1001*, *transmission_weight=None*, *recovery_weight=None*, *return_full_data=False*)

Encodes System (3.26) of Kiss, Miller, & Simon, using a "pure initial condition". That is, we can specify the exact status of all nodes at tmin

Please cite the book if using this algorithm.

Arguments

G networkx Graph

tau positive float transmission rate of disease

gamma number global recovery rate

initial_infecteds list or set the set of nodes initially infected

nodelist list list of nodes in `G` in some prescribed order (just since there is no guarantee that `G` returns nodes in the same order if things change a bit.)

tmin number (default 0) minimum report time

tmax number (default 100) maximum report time

tcount integer (default 1001) number of reports

transmission_weight string the label for a weight given to the edges. G.edge[i][j][transmission_weight] = g_{ij}

recovery_weight string (default `None`) a label for a weight given to the nodes to scale their recovery rates

gamma_i = G.nodes[i][recovery_weight]*gamma

return_full_data boolean (default False)

if True: returns times, S, I, R, Xs, Ys, Zs, XY, XX

if False: returns times, S, I, R

Returns

if return_full_data is True: returns **times, S, I, Xs, Ys, XY, XX**

if False: returns times, **S, I**

EoN.SIR_individual_based

`EoN.`**`SIR_individual_based`**(*G*, *tau*, *gamma*, *rho=None*, *Y0=None*, *X0=None*, *nodelist=None*, *tmin=0*, *tmax=100*, *tcount=1001*, *transmission_weight=None*, *recovery_weight=None*, *return_full_data=False*)

Encodes System (3.30) of Kiss, Miller, & Simon. Please cite the book if using this algorithm.

See also:

Arguments

G networkx Graph

tau positive float transmission rate of disease

gamma number (default `None`) global recovery rate

rho number between 0 and 1 (default `None`) probability random node is infected. Cannot be defined along with `X0` and `Y0`. At least one of `rho` and `Y0` must be defined.

Y0 numpy array (default `None`) the array of initial infection probabilities. If not defined, set to be rho uniformly.

X0 numpy array (default `None`) the array of initial susceptibility probabilities. If not defined set to be 1-Y0. Cannot define X0 without Y0.

nodelist list (default `None`) list of nodes in `G` in the same order as in X0 and Y0. Only relevant to returned data if return_full_data=True

tmin number (default 0) minimum report time

tmax number (default 100) maximum report time

tcount integer (default 1001) number of reports

transmission_weight string (default `None`) the label for a weight given to the edges. G.edge[i][j][transmission_weight] = g_{ij}

recovery_weight string (default `None`) a label for a weight given to the nodes to scale their recovery rates

gamma_i = G.nodes[i][recovery_weight]*gamma

return_full_data (default False) If True, returns times, S, I, R, Ss, Is, Rs if False, returns times, S, I, R

Returns

if return_full_data is True: returns **times, Ss, Is, Rs** where times is a numpy array of times, Ss is a 2D numpy array Ss[i,j] gives probability nodelist[i] is susceptible at time times[j]. Similarly for Is ans Rs

if return_full data is False: returns **times, S, I, R** all are numpy arrays. gives times, and expected number susceptible, expected number infected, and expected number recovered

SAMPLE USE

```
import networkx as nx
import EoN
import matplotlib.pyplot as plt

G = nx.configuration_model([3,10]*10000)
#G has 20000 nodes, half with degree 3 and half with degree 10

tau = 0.3
gamma = 1
N = G.order()
rho = 1./N

t, S, I, R = EoN.SIR_individual_based(G, tau, gamma, rho=rho, tmax = 20)
plt.plot(t,I)
```

EoN.SIR_individual_based_pure_IC

EoN.**SIR_individual_based_pure_IC**(*G*, *tau*, *gamma*, *initial_infecteds*, *initial_recovereds=None*, *nodelist=None*, *tmin=0*, *tmax=100*, *tcount=1001*, *transmission_weight=None*, *recovery_weight=None*, *return_full_data=False*)

Encodes System (3.30) of Kiss, Miller, & Simon. Please cite the book if using this algorithm.

The difference between this and SIR_individual_based is that this one assumes a "pure initial condition", that is, we know exactly what the statuses of the nodes are at the initial time.

<dot{Y}_i> = tau sum_j g_{ij} (1-<Y_i>)<Y_j> - gamma_i <Y_i>

Arguments

G networkx Graph The contact network

tau positive float transmission rate of disease

gamma number (default None) global recovery rate

initial_infecteds list or set the set of nodes initially infected

nodelist list (default None) list of nodes in G in desired order. (only matters if return_full_data==True)

initial_recovereds list or set (default None) initially recovered nodes if equal to None, then all non-index nodes are initially susceptible.

tmin number (default 0) minimum report time

tmax number (default 100) maximum report time

tcount integer (default 1001) number of reports

transmission_weight string (default None) the label for a weight given to the edges. G.edge[i][j][transmission_weight] = g_{ij}

recovery_weight string (default None) a label for a weight given to the nodes to scale their recovery rates

gamma_i = G.nodes[i][recovery_weight]*gamma

return_full_data boolean (default False)

Returns

if return_full_data is True, returns **times, S, I, R, Ss, Is, Rs**

if return_full_data is False, returns **times, S, I, R**

EoN.SIR_pair_based

EoN.**SIR_pair_based**(*G*, *tau*, *gamma*, *rho=None*, *nodelist=None*, *Y0=None*, *X0=None*, *XY0=None*, *XX0=None*, *tmin=0*, *tmax=100*, *tcount=1001*, *transmission_weight=None*, *recovery_weight=None*, *return_full_data=False*)

Encodes System (3.39) of Kiss, Miller, & Simon. Please cite the book if using this algorithm.

WARNING This does not solve the pairwise equations. Look at SIR_homogeneous_pairwise and SIR_heterogeneous_pairwise for that.

This system solves equations for an SIR disease model spreading on a given graph. It captures the dependence with pairs, but not triples.

It will be exact for a tree.

There are NO CORRECTIONS for the existence of TRIANGLES or any other CYCLES.

Some corrections for triangles are provided in the text, but not implemented here.

See also: Hadjichrysanthou and Sharkey Epidemic control analysis: Desigining targeted intervention

strategies against epidemics propagated on contact networks,

Journal of Theoretical Biology

Arguments

G networkx Graph

tau positive float transmission rate of disease

gamma number global recovery rate

rho float between 0 and 1 (default `None`) proportion assumed initially infected. If None, then Y0 is used if Y0 is also None, then rho = 1./N

nodelist list list of nodes in `G` in the some prescribed order (just since there is no guarantee that `G` returns nodes in the same order if things change a bit.)

Y0 numpy array the array of initial infection probabilities for each node in order as in nodelist/

X0 numpy array (default `None`) probability a random node is initially susceptible. the probability of initially recovered will be 1-X0-Y0. By default we assume no initial recoveries, so X0=1-Y0 will be assumed unless both Y0 and X0 are given.

XY0 2D numpy array (default `None`) (each dimension has length number of nodes of G) XY0[i,j] is probability node i is susceptible and j is

infected.

if None, then assumes that infections are introduced randomly according to Y0.

XX0 2D numpy array (default `None`) (each dimension has length number of nodes of G) XX0[i,j] is probability nodes i and j are susceptible. if None, then assumes that infections are introduced

randomly according to Y0.

tmin number (default 0) minimum report time

tmax number (default 100) maximum report time

tcount integer (default 1001) number of reports

transmission_weight string the label for a weight given to the edges. G.edge[i][j][transmission_weight] = g_{ij}

recovery_weight string (default `None`) a label for a weight given to the nodes to scale their recovery rates

gamma_i = G.nodes[i][recovery_weight]*gamma

return_full_data boolean (default False)

if True: returns times, S, I, R, Xs, Ys, Zs, XY, XX

if False: returns times, S, I, R

Returns

if return_full_data is True: returns times, S, I, R, Xs, Ys, Zs, XY, XX

if . . . is False: returns times, S, I, R

SAMPLE USE

```
import networkx as nx
import EoN

G = nx.fast_gnp_random_graph(1000,0.004)
nodelist = G.nodes()
Y0 = np.array([1 if node<10 else 0 for node in nodelist]) #infect first 10
t, S, I, R = EoN.SIR_pair_based(G, nodelist, Y0, 2, 0.5, tmax = 4, tcount = 101)
plt.plot(t,I)
```

EoN.SIR_pair_based_pure_IC

EoN.**SIR_pair_based_pure_IC**(*G*, *tau*, *gamma*, *initial_infecteds*, *initial_recovereds=None*, *nodelist=None*, *tmin=0*, *tmax=100*, *tcount=1001*, *transmission_weight=None*, *recovery_weight=None*, *return_full_data=False*)

Encodes System (3.39) of Kiss, Miller, & Simon, using a "pure initial condition". Please cite the book if using this algorithm.

uses SIR_pair_based after finding the appropriate initial conditions.

Arguments

G networkx Graph

tau positive float transmission rate of disease

gamma number global recovery rate

initial_infecteds list or set the set of nodes initially infected

initial_recovereds list or set (default `None`) the set of nodes that are already recovered.

nodelist list list of nodes in `G` in a prescribed order (just since there is no guarantee that `G` returns nodes in the same order if things change a bit.)

tmin number (default 0) minimum report time

tmax number (default 100) maximum report time

tcount integer (default 1001) number of reports

transmission_weight string the label for a weight given to the edges. G.edge[i][j][transmission_weight] = g_{ij}

recovery_weight string (default `None`) a label for a weight given to the nodes to scale their recovery rates

gamma_i = G.nodes[i][recovery_weight]*gamma

return_full_data boolean (default False)

if True: returns times, S, I, R, Xs, Ys, Zs, XY, XX

if False: returns times, S, I, R

Returns

if return_full_data is True: returns **times, S, I, R, Xs, Ys, Zs, XY, XX**

if ... is False: returns times, S, I, R

EoN.SIS_homogeneous_meanfield

EoN.**SIS_homogeneous_meanfield**(*S0*, *I0*, *n*, *tau*, *gamma*, *tmin=0*, *tmax=100*, *tcount=1001*)

Encodes System (4.8) of Kiss, Miller, & Simon. Please cite the book if using this algorithm.

In the text this is often referred to as the "mean-field model closed at the level of pairs"

[dot{S}] = gamma [I] - tau n[S][I]/N

[dot{I}] = au n[S][I]/N - gamma [I]

This is the SIS version of the "Kermack-McKendrick equations".

Arguments

S0 number initial number susceptible

I0 number initial number infected

n number (average) degree of all nodes.

tau positive float transmission rate

gamma number recovery rate

tmin number (default 0) minimum report time

tmax number (default 100) maximum report time

tcount integer (default 1001) number of reports

Returns

times, S, I all numpy arrays

SAMPLE USE

```
import networkx as nx
import EoN
S0 = 999
I0 = 1
n = 4 #degree
tau = 1
gamma = 2
t, S, I = EoN.SIS_homogeneous_meanfield(S0, I0, n, tau, gamma)
```

EoN.SIR_homogeneous_meanfield

EoN.**SIR_homogeneous_meanfield**(*S0*, *I0*, *R0*, *n*, *tau*, *gamma*, *tmin=0*, *tmax=100*, *tcount=1001*)

Encodes System (4.9) of Kiss, Miller, & Simon. Please cite the book if using this algorithm.

In the text this is often referred to as the "mean-field model closed at the level of pairs"

These are often referred to as the "Kermack-McKendrick equations"

[dot{S}] = - tau n[S][I]/N [dot{I}] = au n[S][I]/N - gamma [I] [dot{R}] = gamma [I]

Arguments

S0 number initial number susceptible

I0 number initial number infected

R0 number initial number recovered

n number (average) degree of each node

tau positive float transmission rate

gamma number recovery rate

tmin number (default 0) minimum report time

tmax number (default 100) maximum report time

tcount integer (default 1001) number of reports

Returns

times, S, I, R all numpy arrays

SAMPLE USE

```
import networkx as nx
import EoN
S0 = 999
I0 = 1
n = 4 #degree
tau = 1
gamma = 2
t, S, I, R = EoN.SIR_homogeneous_meanfield(S0, I0, 0, n, tau, gamma)
```

EoN.SIS_homogeneous_pairwise

EoN.**SIS_homogeneous_pairwise**(*S0*, *I0*, *SI0*, *SS0*, *n*, *tau*, *gamma*, *tmin=0*, *tmax=100*, *tcount=1001*, *return_full_data=False*)

Encodes System (4.10) of Kiss, Miller, & Simon. Please cite the book if using this algorithm.

In the text this is often referred to as the "mean-field model closed at the level of triples"

Arguments

S0 number initial number susceptible

I0 number initial number infected

SI0 number initial number of SI edges

SS0 number initial number of SS edges

n integer (common) degree of nodes.

tau positive float transmission rate

gamma number recovery rate

tmin number (default 0) minimum report time

tmax number (default 100) maximum report time

tcount integer (default 1001) number of reports

return_full_data boolean tells whether to just return times, S, I or all calculated data.

Returns

if return_full_data is True: t, S, I, SI, SS, II

if return_full_data is False: t, S, I

SAMPLE USE

```
import networkx as nx
import EoN
S0 = 990
I0 = 10
SI0 = 50
SS0 = 4900
n = 5
tau = 1
gamma = 2
t, S, I = EoN.SIS_homogeneous_pairwise(S0, I0, SI0, SS0, n, tau, gamma,
                                       tmax = 20)
```

EoN.SIS_homogeneous_pairwise_from_graph

EoN.**SIS_homogeneous_pairwise_from_graph**(*G*, *tau*, *gamma*, *initial_infecteds=None*, *rho=None*, *tmin=0*, *tmax=100*, *tcount=1001*, *return_full_data=False*)

Calls SIS_homogeneous_pairwise after calculating S0, I0, SI0, SS0, n based on the graph G and initial fraction infected rho.

Arguments

G networkx Graph the contact network

tau positive float transmission rate

gamma number recovery rate

initial infecteds node or iterable of nodes (default `None`) if a single node, then this node is initially infected if an iterable, then whole set is initially infected if None, then choose randomly based on rho. If rho is also None, a random single node is chosen. If both initial_infecteds and rho are assigned, then there is an error.

rho float between 0 and 1 (default `None`) the fraction to be randomly infected at time 0 If None, then rho=1/N is used where N = G.order()

tmin number (default 0) minimum report time

tmax number (default 100) maximum report time

tcount integer (default 1001) number of reports

return_full_data boolean (default False) tells whether to just return times, S, I, or all calculated data. if True, then returns times, S, I, SI, SS

Returns

if return_full_data is True: t, S, I, SI, SS, II

if return_full_data is False: t, S, I

SAMPLE USE

```
import networkx as nx
import EoN
G = nx.fast_gnp_random_graph(10000,0.0005)
tau = 1
gamma = 3
rho = 0.02
t, S, I = EoN.SIS_homogeneous_pairwise_from_graph(G, tau, gamma, rho, tmax = 20)
```

EoN.SIR_homogeneous_pairwise

EoN.**SIR_homogeneous_pairwise**(*S0*, *I0*, *R0*, *SI0*, *SS0*, *n*, *tau*, *gamma*, *tmin=0*, *tmax=100*, *tcount=1001*, *return_full_data=False*)

Encodes System (4.11) of Kiss, Miller, & Simon. Please cite the book if using this algorithm.

In the text this is often referred to as the "mean-field model closed at the level of triples"

[dot{S}] = - tau [SI] [dot{I}] = au [SI] - gamma [I] [dot{R}] = gamma [I] ; [R] = N-[S]-[I] [dot{SI}] = -gamma [SI]+ au ((n-1)/n) [SI]([SS]-[SI])/[S]

- au [SI]

[dot{SS}] = - 2 au ((n-1)/n) [SI][SS]/[S]

conserved quantities: [S]+[I]+[R] also [SS]+[II]+[RR] + 2([SI] + [SR] + [IR])

Arguments

S0 float Initial number susceptible

I0 float Initial number infected

R0 float Initial number recovered

SI0 float Initial number of SI edges

SS0 float Initial number of SS edges

n float Degree of nodes

tau positive float transmission rate

gamma number recovery rate

tmin number (default 0) minimum report time

tmax number (default 100) maximum report time

tcount integer (default 1001) number of reports

return_full_data boolean (default False) tells whether to just return times, S, I, R or all calculated data. if True, then returns times, S, I, R, SI, SS

Returns

if return_full_data is True: times, S, I, R, SI, SS

if return_full_data is False: **times, S, I, R **

SAMPLE USE

```
import networkx as nx
import EoN
S0 = 990
I0 = 10
R0 = 1
SI0 = 45
SS0 = 4900
n = 5
tau = 1
gamma = 2
t, S, I, R = EoN.SIR_homogeneous_pairwise(S0, I0, R0, SI0, SS0, n, tau, gamma,
                                          tmax = 20)
```

EoN.SIR_homogeneous_pairwise_from_graph

EoN.**SIR_homogeneous_pairwise_from_graph**(*G*, *tau*, *gamma*, *initial_infecteds=None*, *initial_recovereds=None*, *rho=None*, *tmin=0*, *tmax=100*, *tcount=1001*, *return_full_data=False*)

Calls SIR_homogeneous_pairwise after calculating S0, I0, R0, SI0, SS0, n, based on the graph G and initial fraction infected rho.

Arguments

G networkx Graph the contact network

tau positive float transmission rate

gamma number recovery rate

initial infecteds node or iterable of nodes (default `None`) if a single node, then this node is initially infected if an iterable, then whole set is initially infected if None, then choose randomly based on rho. If rho is also None, a random single node is chosen. If both initial_infecteds and rho are assigned, then there is an error.

initial_recovereds iterable of nodes (default `None`) this whole collection is made recovered. Currently there is no test for consistency with initial_infecteds. Understood that everyone who isn't infected or recovered initially is initially susceptible.

rho float between 0 and 1 (default `None`) the fraction to be randomly infected at time 0 If None, then rho=1/N is used where N = G.order()

tmin number (default 0) minimum report time

tmax number (default 100) maximum report time

tcount integer (default 1001) number of reports

return_full_data boolean (default False) tells whether to just return times, S, I, R or all calculated data. if True, then returns times, S, I, R, SI, SS

Returns

if return_full_data is True: t, S, I, SI, SS, II

if return_full_data is False: t, S, I

SAMPLE USE

```
import networkx as nx
import EoN
G = nx.fast_gnp_random_graph(10000,0.0005)
tau = 1
gamma = 3
rho = 0.02
t, S, I, R = EoN.SIR_homogeneous_pairwise_from_graph(G, tau, gamma, rho,
                                                     tmax = 20)
```

EoN.SIS_heterogeneous_meanfield

EoN.**SIS_heterogeneous_meanfield**(*Sk0*, *Ik0*, *tau*, *gamma*, *tmin=0*, *tmax=100*, *tcount=1001*, *return_full_data=False*)

Encodes System (5.10) of Kiss, Miller, & Simon. Please cite the book if using this algorithm.

In the text this is often referred to as the "heterogeneous mean-field model closed at the level of pairs"

This is also called Degree-baded Mean Field or Mean Field Social Heterogeneity

a few notes on the inputs: Sk0 is an array (or a list).

It is not a dict.

Sk0[k] is the *number* of nodes that are susceptible and have degree k (even if some degrees missing).

A dict like this can be converted into an array by Sk0 = np.array([Sk0dict.get(k,0)

for k in range(max(Sk0dict.keys())+1)])

Ik0 is similar to Sk0.

[dot{S}_k] = gamma [I_k] - au k [S_k] pi_I [dot{I}_k] = -(above) pi_I = sum_k k [I_k] / sum_k k [N_k]

Arguments

Sk0 numpy array number susceptible for each k

Ik0 numpy array number infected for each k

tau positive float transmission rate

gamma number recovery rate

tmin number (default 0) minimum report time

tmax number (default 100) maximum report time

tcount integer (default 1001) number of reports

return_full_data boolean (default False) tells whether to just return times, S, I or all calculated data. if True, returns t, S, I, Sk, Ik

Returns

if return_full_data is True: times, S, I, Sk (Sk is numpy 2D arrays)

if return_full_data is False: times, S, I (all numpy arrays)

SAMPLE USE

```
import networkx as nx
import EoN
Sk0 = [995, 995, 995, 995, 995]
Ik0 = [5, 5, 5, 5, 5]
tau = 1
gamma = 2
t, S, I = EoN.SIS_heterogeneous_meanfield(Sk0, Ik0, tau, gamma, tmax = 10)
```

EoN.SIS_heterogeneous_meanfield_from_graph

EoN.**SIS_heterogeneous_meanfield_from_graph**(*G*, *tau*, *gamma*, *initial_infecteds=None*, *rho=None*, *tmin=0*, *tmax=100*, *tcount=1001*, *return_full_data=False*)

Calls SIS_heterogeneous_meanfield after calculating Sk0, Ik0 based on the graph G and random fraction infected rho.

Arguments

G networkx Graph the contact network

tau positive float transmission rate

gamma number recovery rate

initial infecteds node or iterable of nodes (default `None`) if a single node, then this node is initially infected if an iterable, then whole set is initially infected if None, then choose randomly based on rho. If rho is also None, a random single node is chosen. If both initial_infecteds and rho are assigned, then there is an error.

rho float between 0 and 1 (default `None`) the fraction to be randomly infected at time 0 If None, then rho=1/N is used where N = G.order()

tmin number (default 0) minimum report time

tmax number (default 100) maximum report time

tcount integer (default 1001) number of reports

return_full_data boolean tells whether to just return times, S, I, R or all calculated data.

Returns

if return_full_data is True: times, S, I, Sk, Ik (the Xk are numpy 2D arrays)

if return_full_data is False: times, S, I (all numpy arrays)

SAMPLE USE

```
import networkx as nx
import EoN
G = nx.configuration_model([1,2,3,4]*1000)
tau = 1
gamma = 2
t, S, I = EoN.SIS_heterogeneous_meanfield_from_graph(G, tau, gamma,
                                                     tmax = 15)
```

EoN.SIR_heterogeneous_meanfield

EoN.**SIR_heterogeneous_meanfield**(*Sk0*, *Ik0*, *Rk0*, *tau*, *gamma*, *tmin=0*, *tmax=100*, *tcount=1001*, *return_full_data=False*)

Encodes System (5.11) of Kiss, Miller, & Simon. Please cite the book if using this algorithm.

In the text this is often referred to as the "heterogeneous mean-field model closed at the level of pairs"

This is also called Degree-baded Mean Field or Mean Field Social Heterogeneity

Ik0 and Rk0 are similar to Sk0.

[S_k] = [S_k](0) theta^k [I_k] = [N_k] - [S_k] - [R_k] [dot{R}_k] = gamma [I_k] pi_I = sum_k k[I_k]

Arguments

Sk0 array Sk0[k] is the number of nodes that are susceptible and have degree k (even if some degrees missing).

Ik0 array as in Sk0

Rk0 array as in Sk0

tau positive float transmission rate

gamma number recovery rate

tmin number (default 0) minimum report time

tmax number (default 100) maximum report time

tcount integer (default 1001) number of reports

return_full_data boolean tells whether to just return times, S, I, R or all calculated data.

Returns

if return_full_data is True: times, S, I, R, Sk, Ik, Rk (the Xk are numpy 2D arrays)

if return_full_data is False: times, S, I, R (all numpy arrays)

SAMPLE USE

```
import networkx as nx
import EoN
Sk0 = [995, 995, 995, 995, 995]
Ik0 = [5, 5, 5, 5, 5]
Rk0 = [0,0,0,0,0]
tau = 1
gamma = 2
t, S, I, R = EoN.SIR_heterogeneous_meanfield(Sk0, Ik0, Rk0, tau, gamma,
                                             tmax = 10)
```

EoN.SIR_heterogeneous_meanfield_from_graph

EoN.**SIR_heterogeneous_meanfield_from_graph**(*G*, *tau*, *gamma*, *initial_infecteds=None*, *initial_recovereds=None*, *rho=None*, *tmin=0*, *tmax=100*, *tcount=1001*, *return_full_data=False*)

Calls SIR_heterogeneous_meanfield after calculating Sk0, Ik0, Rk0 based on a graph G and initial fraction infected rho.

Arguments

G networkx Graph the contact network

tau positive float transmission rate

gamma number recovery rate

initial infecteds node or iterable of nodes (default `None`) if a single node, then this node is initially infected if an iterable, then whole set is initially infected if None, then choose randomly based on rho. If rho is also None, a random single node is chosen. If both initial_infecteds and rho are assigned, then there is an error.

initial_recovereds iterable of nodes (default `None`) this whole collection is made recovered. Currently there is no test for consistency with initial_infecteds. Understood that everyone who isn't infected or recovered initially is initially susceptible.

rho float between 0 and 1 (default `None`) the fraction to be randomly infected at time 0 If None, then rho=1/N is used where N = G.order()

tmin number (default 0) minimum report time

tmax number (default 100) maximum report time

tcount integer (default 1001) number of reports

return_full_data boolean tells whether to just return times, S, I, R or all calculated data.

Returns

if return_full_data is True times, Sk, Ik, Rk (the Xk are numpy 2D arrays)

if False, times, S, I, R (all numpy arrays)

SAMPLE USE

```
import networkx as nx
import EoN
G = nx.configuration_model([1,2,3,4]*1000)
tau = 1
gamma = 2
t, S, I, R = EoN.SIR_heterogeneous_meanfield_from_graph(G, tau, gamma,
                                                        tmax = 10)
```

EoN.SIS_heterogeneous_pairwise

EoN.**SIS_heterogeneous_pairwise**(*Sk0*, *Ik0*, *SkSl0*, *SkIl0*, *IkIl0*, *tau*, *gamma*, *tmin=0*, *tmax=100*, *tcount=1001*, *return_full_data=False*, *Ks=None*)

Encodes System (5.13) of Kiss, Miller, & Simon. Please cite the book if using this algorithm.

In the text this is often referred to as the heterogeneous mean-field model closed at the level of triples

Arguments

Sk0 array. (if Ks is defined, the definition changes slightly, see below)

Sk0[k] is the number of nodes that are susceptible and have degree k. If one is empty, it becomes 0.

Ik0 array (if Ks is defined, the definition changes slightly, see below)

similar to Sk0, but for infected.

SkSl0 2D numpy array (if Ks is defined, the definition changes slightly, see below)

SkSl0[k][l] is [S_kS_l] at 0 see below for constraints these should satisfy related to Sk0 and Ik0. The code does not enforce these constraints.

SkIl0 2D numpy array as in SkSl0

IkIl0 2D numpy array as in SkSl0

tau positive float transmission rate

gamma number recovery rate

tmin number (default 0) minimum report time

tmax number (default 100) maximum report time

tcount integer (default 1001) number of reports

return_full_data boolean (default False) If True, return times, Sk, Ik, SkIl, SkSl, IkIl If False, return times, S, I

Ks numpy array. (default `None`) (helps prevent memory errors) if some degrees are not observed, then the corresponding entries of these arrays are zero. This can lead to memory errors in the case of a network with many missing degrees. So Ks is an (assumed) ordered vector stating which Ks are actually observed. Then the Sk0[i] is the number of nodes that are susceptible and have degree Ks[i]. Similarly for Ik0 and SkIl0 etc.

In principle, there are constraints relating Sk with SkSl and SkIl and similarly relating Ik with IkIl and SkIl.T.

No attempt is made to enforce these.

It is assumed the user will ensure acceptible inputs.

We could also remove Sk0 and Ik0 as inputs and infer them from the others, but for consistency with elsewhere, this is not done here.

Returns

if return_full_data is True: returns **times, S, I, Sk, Ik, SkIl, SkSl, IkIl**

if return_full_data is False: returns **times, S, I**

SAMPLE USE

```
import networkx as nx
import EoN
import numpy as np

Sk0 = 100 * np.ones(4)
Ik0 = np.zeros(4)
Ik0[3]=1
SkSl0 = np.array([[0, 0,0,0],[0,100,0,0],[0,0,200,0],[0,0,0,294]])
#only interact within a degree class, so the deg 1 and 2 are safe.
SkIl0 = np.zeros((4,4))
SkIl0[3,3] = 3
IkIl0 = np.zeros((4,4))
tau = 1
gamma = 1

t, S, I = EoN.SIS_heterogeneous_pairwise(Sk0, Ik0, SkSl0, SkIl0, IkIl0, tau,
                                         gamma)
```

EoN.SIS_heterogeneous_pairwise_from_graph

EoN.**SIS_heterogeneous_pairwise_from_graph**(*G*, *tau*, *gamma*, *initial_infecteds=None*, *rho=None*, *tmin=0*, *tmax=100*, *tcount=1001*, *return_full_data=False*)

Calls SIS_heterogeneous_pairwise after calculating Sk0, Ik0, SkSl0, SkIl0, IkIl0 from a graph G and initial fraction infected rho.

Arguments

G networkx Graph The contact network

tau positive float transmission rate

gamma number recovery rate

initial infecteds node or iterable of nodes (default `None`) if a single node, then this node is initially infected if an iterable, then whole set is initially infected if None, then choose randomly based on rho. If rho is also None, a random single node is chosen. If both initial_infecteds and rho are assigned, then there is an error.

rho float between 0 and 1 (default `None`) the fraction to be randomly infected at time 0 If None, then rho=1/N is used where N = G.order()

tmin number (default 0) minimum report time

tmax number (default 100) maximum report time

tcount integer (default 1001) number of reports

return_full_data boolean (default False) tells whether to just return times, S, I, or all calculated data. if True, then returns times, S, I, SI, SS

Returns

if return_full_data is True: returns **times, S, I, Sk, Ik, SkIl, SkSl, IkIl**

if return_full_data is False: returns **times, S, I**

SAMPLE USE

```
import networkx as nx
import EoN
G = nx.fast_gnp_random_graph(10000,0.0005)
tau = 1
gamma = 3
rho = 0.02
t, S, I = EoN.SIS_heterogeneous_pairwise_from_graph(G, tau, gamma, rho, tmax = 20)
```

WARNING

```
This can have segmentation faults if there are too many degrees in the
graph.  This appears to happen because of trouble in numpy, and I have
not been able to find a way around it.
```

EoN.SIR_heterogeneous_pairwise

EoN.**SIR_heterogeneous_pairwise**(*Sk0*, *Ik0*, *Rk0*, *SkSl0*, *SkIl0*, *tau*, *gamma*, *tmin=0*, *tmax=100*, *tcount=1001*, *return_full_data=False*, *Ks=None*)

Encodes System (5.15) of Kiss, Miller, & Simon. Please cite the book if using this algorithm.

In the text this is often referred to as the "heterogeneous mean-field model closed at the level of triples"

[dot{S}_k] = -tau [S_k I] [dot{I}_k] = tau [S_k I] - gamma [I_k] [dot{R}_k] = gamma [I_k] (but using Rk=Nk-Sk-Ik for this equation) [dot{S_kI_l}] = -gamma[S_k I_l] + tau([S_k S_l I] - [I S_k I_l]

- [S_k I_l])

[dot{S_kS_l}] = -tau([S_k S_l I] + [I S_k S_l])

[A_l S_k I] = ((k-1)/k) [A_l S_k] [S_k I]/ [S_k] [I S_k A_l] = ((k-1)/k) [I S_k] [S_k A_l]/ [S_k]

Arguments

Sk0 numpy array, (if Ks is defined, the definition changes slightly, see below)

Sk0[k] is number of degree k susceptible at time 0.

Ik0 numpy array (if Ks is defined, the definition changes slightly, see below)

as in Sk0

Rk0 numpy array (if Ks is defined, the definition changes slightly, see below)

as in Sk0

SkSl0 numpy 2D array (if Ks is defined, the definition changes slightly, see below)

SkSl0[k][l] is [S_kS_l] at 0

SkIl0 numpy 2D array (if Ks is defined, the definition changes slightly, see below)

as in SkSl0

tau positive float transmission rate

gamma number recovery rate

tmin number (default 0) minimum report time

tmax number (default 100) maximum report time

tcount integer (default 1001) number of reports

return_full_data boolean (default False) If True, return times, Sk, Ik, Rk, SkIl, SkSl If False, return times, S, I, R

Ks numpy array. (default `None`) (helps prevent memory errors) if some degrees are not observed, then the corresponding entries of these arrays are zero. This can lead to memory errors in the case of a network with many missing degrees. So Ks is an (assumed) ordered vector stating which Ks are actually observed. Then the Sk0[i] is the number of nodes that are susceptible and have degree Ks[i]. Similarly for Ik0 and SkIl0 etc.

Returns

if return_full_data is True returns **times, S, I, R, Sk, Ik, Rk, SkIl, SkSl**

if return_full_data is False return **times, S, I, R**

SAMPLE USE

```
import networkx as nx
import EoN
```

EoN.SIR_heterogeneous_pairwise_from_graph

EoN.**SIR_heterogeneous_pairwise_from_graph**(*G*, *tau*, *gamma*, *initial_infecteds=None*, *initial_recovereds=None*, *rho=None*, *tmin=0*, *tmax=100*, *tcount=1001*, *return_full_data=False*)

Calls SIR_heterogeneous_pairwise after calculating Sk0, Ik0, Rk0, SkSl0, SkIl0 from a graph G and initial fraction infected rho.

Arguments

G networkx Graph The contact network

tau positive float transmission rate

gamma number recovery rate

initial infecteds node or iterable of nodes (default `None`) if a single node, then this node is initially infected if an iterable, then whole set is initially infected if None, then choose randomly based on rho. If rho is also None, a random single node is chosen. If both initial_infecteds and rho are assigned, then there is an error.

initial_recovereds iterable of nodes (default `None`) this whole collection is made recovered. Currently there is no test for consistency with initial_infecteds. Understood that everyone who isn't infected or recovered initially is initially susceptible.

rho float between 0 and 1 (default `None`) the fraction to be randomly infected at time 0 If None, then rho=1/N is used where N = G.order()

tmin number (default 0) minimum report time

tmax number (default 100) maximum report time

tcount integer (default 1001) number of reports

return_full_data boolean (default False) If True, return times, Sk, Ik, Rk, SkIl, SkSl If False, return times, S, I, R

Returns

if return_full_data is True returns **times, S, I, R, Sk, Ik, Rk, SkIl, SkSl**

if return_full_data is False return **times, S, I, R**

WARNING This can have segmentation faults if there are too many degrees in the graph. This appears to happen because of trouble in numpy, and I have not been able to find a way around it.

EoN.SIS_compact_pairwise

EoN.**SIS_compact_pairwise**(*Sk0*, *Ik0*, *SI0*, *SS0*, *II0*, *tau*, *gamma*, *tmin=0*, *tmax=100*, *tcount=1001*, *return_full_data=False*)

Encodes system (5.18) of Kiss, Miller, & Simon. Please cite the book if using this algorithm.

[dot{S}_k] = gamma[I_k] - tau k [S_k] [SI]/[SX]

[dot{I}_k] = tau * k *[S_k] [SI]/SX - gamma [I_k] = -[dot{S}_k]

[dot{SI}] = gamma([II]-[SI]) + tau([SS]-[SI])[SI]Q - tau[SI]

[dot{SS}] = 2 gamma[SI] - 2 tau [SS] [SI] Q

[dot{II}] = 2 tau[SI] = 2 gamma[II] + 2 tau[SI]^2Q

[SX] = sum_k k [S_k]

Q = (1/[SX]^2) sum_k (k-1)k[S_k]

Conserved quantities

[Sk]+[Ik]

SS + II + 2SI

Arguments

Sk0 numpy array number susceptible for each k

Ik0 numpy array number infected for each k

SI0 number number of SI edges

SS0 number number of SS edges

II0 number number of II edges

tau positive float transmission rate

gamma number recovery rate

tmin number (default 0) minimum report time

tmax number (default 100) maximum report time

tcount integer (default 1001) number of reports

return_full_data boolean (default False) if True, return times, S, I, Sk, Ik, SI, SS, II if False, return times, S, I

Returns

All numpy arrays

if return_full_data return **times, S, I, Sk, Ik, SI, SS, II**

else return **times, S, I**

SAMPLE USE

```
import networkx as nx
import EoN
```

EoN.SIS_compact_pairwise_from_graph

EoN.**SIS_compact_pairwise_from_graph**(*G*, *tau*, *gamma*, *initial_infecteds=None*, *rho=None*, *tmin=0*, *tmax=100*, *tcount=1001*, *return_full_data=False*)

Calls SIS_compact_pairwise after calculating Sk0, Ik0, SI0, SS0, II0 from the graph G and initial fraction infected rho.

Arguments

G networkx Graph The contact network

tau positive float transmission rate

gamma number recovery rate

initial infecteds node or iterable of nodes (default `None`) if a single node, then this node is initially infected if an iterable, then whole set is initially infected if None, then choose randomly based on rho. If rho is also None, a random single node is chosen. If both initial_infecteds and rho are assigned, then there is an error.

rho float between 0 and 1 (default `None`) the fraction to be randomly infected at time 0 If None, then rho=1/N is used where N = G.order()

tmin number (default 0) minimum report time

tmax number (default 100) maximum report time

tcount integer (default 1001) number of reports

return_full_data boolean (default False) if True, return times, S, I, Sk, Ik, SI, SS, II if False, return times, S, I

Returns

All numpy arrays if return_full_data:

return **times, S, I, Sk, Ik, SI, SS, II**

else: return **times, S, I**

EoN.SIR_compact_pairwise

EoN.**SIR_compact_pairwise**(*Sk0*, *I0*, *R0*, *SS0*, *SI0*, *tau*, *gamma*, *tmin=0*, *tmax=100*, *tcount=1001*, *return_full_data=False*)

Encodes system (5.19) of Kiss, Miller, & Simon. Please cite the book if using this algorithm.

[dot{S}_k] = -tau k [S_k] [SI]/[SX]

[dot{SS}] = -2 tau [SS] [SI] Q

[dot{SI}] = -gamma [SI] + tau([SS]-[SI])[SI]Q - tau [SI]

[dot{R} = gamma [I]

[SX] = sum_k k[S_k]

Q = (1/[SX]^2) sum_k (k-1) k [S_k]

[S] = sum [S_k]

I = N-[S]-R

Arguments

Sk0 numpy array initial number of suscetibles of each degree k

I0 number initial number infected

R0 number initial number recovered

SS0 number initial number of SS edges

SI0 number initial number of SI edges

tau positive float transmission rate

gamma number recovery rate

tmin number (default 0) minimum report time

tmax number (default 100) maximum report time

tcount integer (default 1001) number of reports

return_full_data boolean tells whether to just return times, S, I, R or all calculated data.

Returns

if return_full_data: times, Sk, I, R, SS, SI

else: times, S, I, R

SAMPLE USE

```
import networkx as nx
import EoN
```

EoN.SIR_compact_pairwise_from_graph

EoN.**SIR_compact_pairwise_from_graph**(*G*, *tau*, *gamma*, *initial_infecteds=None*, *initial_recovereds=None*, *rho=None*, *tmin=0*, *tmax=100*, *tcount=1001*, *return_full_data=False*)

Calls SIR_compact_pairwise after calculating Sk0, I0, R0, SS0, SI0 from the graph G and initial fraction infected rho.

Arguments

G networkx Graph The contact network

tau positive float transmission rate

gamma number recovery rate

initial infecteds node or iterable of nodes (default `None`) if a single node, then this node is initially infected if an iterable, then whole set is initially infected if None, then choose randomly based on rho. If rho is also None, a random single node is chosen. If both initial_infecteds and rho are assigned, then there is an error.

initial_recovereds iterable of nodes (default `None`) this whole collection is made recovered. Currently there is no test for consistency with initial_infecteds. Understood that everyone who isn't infected or recovered initially is initially susceptible.

rho float between 0 and 1 (default `None`) the fraction to be randomly infected at time 0 If None, then rho=1/N is used where N = G.order()

tmin number (default 0) minimum report time

tmax number (default 100) maximum report time

tcount integer (default 1001) number of reports

return_full_data boolean tells whether to just return times, S, I, R or all calculated data.

Returns

if return_full_data: times, Sk, I, R, SS, SI

else: times, S, I, R

EoN.SIS_super_compact_pairwise

EoN.**SIS_super_compact_pairwise**(*S0*, *I0*, *SS0*, *SI0*, *II0*, *tau*, *gamma*, *k_ave*, *ksquare_ave*, *kcube_ave*, *tmin=0*, *tmax=100*, *tcount=1001*, *return_full_data=False*)

Encodes system (5.20) of Kiss, Miller, & Simon. Please cite the book if using this algorithm.

Arguments

S0 number initial number susceptible

I0 number initial number infected

SS0 number initial number of susceptible-susceptible edges

SI0 number initial number of susceptible-infected edges

II0 number initial number of infected-infected edges.

tau positive float transmission rate

gamma number recovery rate

k_ave number average value of degree k

ksquare_ave number average value of k**2

kcube_ave number average value of k**3

tmin number (default 0) minimum report time

tmax number (default 100) maximum report time

tcount integer (default 1001) number of reports

return_full_data boolean tells whether to just return times, S, I, R or all calculated data.

Returns

if return_full_data is True returns **times, S, I, SS, SI, II**

if return_full_data is False returns **times, S, I**

SAMPLE USE

```
import networkx as nx
import EoN
```

EoN.SIS_super_compact_pairwise_from_graph

EoN.**SIS_super_compact_pairwise_from_graph**(*G*, *tau*, *gamma*, *initial_infecteds=None*, *rho=None*, *tmin=0*, *tmax=100*, *tcount=1001*, *return_full_data=False*)

Calls SIS_super_compact_pairwise after calculating S0, I0, SS0, SI0, II0 from the graph G and initial fraction infected rho

EoN.SIR_super_compact_pairwise

EoN.**SIR_super_compact_pairwise**(*R0*, *SS0*, *SI0*, *N*, *tau*, *gamma*, *psihat*, *psihatPrime*, *psihatDPrime*, *tmin=0*, *tmax=100*, *tcount=1001*, *return_full_data=False*)

Encodes system (5.22) of Kiss, Miller, & Simon. Please cite the book if using this algorithm.

dot{theta} = -tau [SI]/N*psihat(theta)

[dot{SS}] = -2 tau [SS] [SI] Q

[dot{SI}] = -gamma[SI] + tau([SS]-[SI])[SI]Q - tau*[SI]

[dot{R}] = gamma*[I]

[S] = N psihat(theta)

[I] = N-[S]-[R]

Q = psihat_xx(theta)/N(psihat_x(theta))^2

Arguments

R0 number initial number of R nodes.

SS0 number initial number of SS edges

SI0 number initial number of SI edges

tau positive float transmission rate

gamma number recovery rate

tmin number (default 0) minimum report time

tmax number (default 100) maximum report time

tcount integer (default 1001) number of reports

return_full_data boolean tells whether to just return times, S, I, R or all calculated data.

Returns

if return_full_data: return **times, S, I, R, SS, SI**

else: return **times, S, I, R**

EoN.SIR_super_compact_pairwise_from_graph

EoN.**SIR_super_compact_pairwise_from_graph**(*G*, *tau*, *gamma*, *initial_infecteds=None*, *initial_recovereds=None*, *rho=None*, *tmin=0*, *tmax=100*, *tcount=1001*, *return_full_data=False*)

Calls SIR_super_compact_pairwise after calculating R0, SS0, SI0 from the graph G and initial fraction infected rho

Arguments

G networkx Graph The contact network

tau positive float transmission rate

gamma number recovery rate

initial infecteds node or iterable of nodes (default `None`) if a single node, then this node is initially infected if an iterable, then whole set is initially infected if None, then choose randomly based on rho. If rho is also None, a random single node is chosen. If both initial_infecteds and rho are assigned, then there is an error.

initial_recovereds iterable of nodes (default `None`) this whole collection is made recovered. Currently there is no test for consistency with initial_infecteds. Understood that everyone who isn't infected or recovered initially is initially susceptible.

rho float between 0 and 1 (default `None`) the fraction to be randomly infected at time 0 If None, then rho=1/N is used where N = G.order()

tmin number (default 0) minimum report time

tmax number (default 100) maximum report time

tcount integer (default 1001) number of reports

return_full_data boolean tells whether to just return times, S, I, R or all calculated data.

Returns

if return_full_data: return **times, S, I, R, SS, SI**

else: return **times, S, I, R**

EoN.SIS_effective_degree

EoN.**SIS_effective_degree**(*Ssi0*, *Isi0*, *tau*, *gamma*, *tmin=0*, *tmax=100*, *tcount=1001*, *return_full_data=False*)

Encodes system (5.36) of Kiss, Miller, & Simon. Please cite the book if using this algorithm.

Arguments

Ssi0 and Isi0 [(square) numpy 2D arrays of same shape.] Entries are initial number susceptible or infected with given initial number of susceptible/infected neighbors.

tau positive float transmission rate

gamma number recovery rate

tmin number (default 0) minimum report time

tmax number (default 100) maximum report time

tcount integer (default 1001) number of reports

return_full_data boolean (default False) tells whether to just return times, S, I, R or all calculated data. if True,

return times, S, I, Ssi, Isi

if False, return times, S, I

Returns

if return_full_data: return times, S, I, Ssi, Isi

else: return times, S, I

EoN.SIS_effective_degree_from_graph

EoN.**SIS_effective_degree_from_graph**(*G*, *tau*, *gamma*, *initial_infecteds=None*, *rho=None*, *tmin=0*, *tmax=100*, *tcount=1001*, *return_full_data=False*)

Calls SIS_effective_degree after calculating Ssi0, Isi0 from the graph G and initialf fraction infected rho.

WARNING

```
This can have segmentation faults if there are too many degrees in the
graph.  This appears to happen because of trouble in numpy, and I have
not been able to find a way around it.
```

EoN.SIR_effective_degree

EoN.**SIR_effective_degree**(*S_si0*, *I0*, *R0*, *tau*, *gamma*, *tmin=0*, *tmax=100*, *tcount=1001*, *return_full_data=False*)

Encodes system (5.38) of Kiss, Miller, & Simon. Please cite the book if using this algorithm.

dot{S}_{s,i} = - tau i S_{s,i} + gamma((i+1)S_{s,i+1} - i S_{s,i})

- tau [ISS]((s+1)S_{s+1,i-1} - sS_{s,i})/[SS]

dot{R} = gamma I S = sum_{s,i} S_{s,i} I = N-S-R

Arguments

S_si0 (square) numpy 2-D array S_{s,i} at time 0

I0 number number of infected individuals at time 0

R0 number number of recovered individuals at time 0

tau positive float transmission rate

gamma number recovery rate

tmin number (default 0) minimum report time

tmax number (default 100) maximum report time

tcount integer (default 1001) number of reports

return_full_data boolean tells whether to just return times, S, I, R or all calculated data.

Returns

if return_full_data==False times np.array of times

S np.array of number susceptible

I np.array of number infected

R np.array of number recovered

else times as before

S number susceptible

I number infected

R number recovered

S_si S_{s,i} at each time in times

EoN.SIR_effective_degree_from_graph

EoN.**SIR_effective_degree_from_graph**(*G*, *tau*, *gamma*, *initial_infecteds=None*, *initial_recovereds=None*, *rho=None*, *tmin=0*, *tmax=100*, *tcount=1001*, *return_full_data=False*)

Calls SIR_effective_degree after calculating S_si0, I0, R0 from the graph G and initial fraction infected rho

Arguments

G networkx Graph The contact network

tau positive float transmission rate

gamma number recovery rate

initial infecteds node or iterable of nodes (default `None`) if a single node, then this node is initially infected if an iterable, then whole set is initially infected if None, then choose randomly based on rho. If rho is also None, a random single node is chosen. If both initial_infecteds and rho are assigned, then there is an error.

initial_recovereds iterable of nodes (default `None`) this whole collection is made recovered. Currently there is no test for consistency with initial_infecteds. Understood that everyone who isn't infected or recovered initially is initially susceptible.

rho float between 0 and 1 (default `None`) the fraction to be randomly infected at time 0 If None, then rho=1/N is used where N = G.order()

tmin number (default 0) minimum report time

tmax number (default 100) maximum report time

tcount integer (default 1001) number of reports

return_full_data boolean tells whether to just return times, S, I, R or all calculated data.

Returns

if return_full_data==False times np.array of times

S np.array of number susceptible

I np.array of number infected

R np.array of number recovered

else times as before

S number susceptible

I number infected

R number recovered

S_si S_{s,i} at each time in times

WARNING

This can have segmentation faults if there are too many degrees in the graph. This appears to happen because of trouble in numpy, and I have not been able to find a way around it.

EoN.SIR_compact_effective_degree

EoN.**SIR_compact_effective_degree**(*Skappa0*, *I0*, *R0*, *SI0*, *tau*, *gamma*, *tmin=0*, *tmax=100*, *tcount=1001*, *return_full_data=False*)

Encodes system (5.43) of Kiss, Miller, & Simon. Please cite the book if using this algorithm.

dot{S}_kappa = <I> [-(tau+gamma) kappa S_kappa

- gamma(kappa+1)S_{kappa+1}

[dot{SI}] = -(tau+gamma)[SI]

- tau(<I> - 2 <I>^2) sum_{kappa} kappa(kappa-1) S_kappa

dot{R} = gamma I <I> = [SI]/sum_kappa kappa S_kappa S = sum_kappa S_kappa I = N - S - R

Arguments

Skappa0 [numpy array] from S_0(0) up to S_kappamax(0) of number susceptible with each effective degree Skappa = number of nodes that are susceptible and have kappa non-recovered neighbors

I0 number number of infected individuals at time 0

R0 number initial number recovered

SI0 number initial number of SI edges

tau positive float transmission rate

gamma number recovery rate

tmin number (default 0) minimum report time

tmax number (default 100) maximum report time

tcount integer (default 1001) number of reports

return_full_data boolean tells whether to just return times, S, I, R or all calculated data.

Returns

if return_full_data==False times np.array of times

S np.array of number susceptible

I np.array of number infected

R np.array of number recovered

else times as before

S number susceptible

I number infected

R number recovered

SI S_{s,i} number of SI edges

EoN.SIR_compact_effective_degree_from_graph

EoN.**SIR_compact_effective_degree_from_graph**(*G*, *tau*, *gamma*, *initial_infecteds=None*, *initial_recovereds=None*, *rho=None*, *tmin=0*, *tmax=100*, *tcount=1001*, *return_full_data=False*)

Calls SIR_compact_effective_degree after calculating Skappa0, I0, R0, SI0 from the graph G and initial fraction infected rho. :Arguments:

G networkx Graph The contact network

tau positive float transmission rate

gamma number recovery rate

initial infecteds node or iterable of nodes (default `None`) if a single node, then this node is initially infected if an iterable, then whole set is initially infected if None, then choose randomly based on rho. If rho is also None, a random single node is chosen. If both initial_infecteds and rho are assigned, then there is an error.

initial_recovereds iterable of nodes (default `None`) this whole collection is made recovered. Currently there is no test for consistency with initial_infecteds. Understood that everyone who isn't infected or recovered initially is initially susceptible.

rho float between 0 and 1 (default `None`) the fraction to be randomly infected at time 0 If None, then rho=1/N is used where N = G.order()

tmin number (default 0) minimum report time

tmax number (default 100) maximum report time

tcount integer (default 1001) number of reports

return_full_data boolean tells whether to just return times, S, I, R or all calculated data.

Returns

if return_full_data==False times np.array of times

S np.array of number susceptible

I np.array of number infected

R np.array of number recovered

else times as before

S number susceptible

I number infected

R number recovered

SI S_{s,i} number of SI edges

EoN.SIS_compact_effective_degree

EoN.**SIS_compact_effective_degree**(*Sk0*, *Ik0*, *SI0*, *SS0*, *II0*, *tau*, *gamma*, *tmin=0*, *tmax=100*, *tcount=1001*, *return_full_data=False*)

Encodes system (5.44) of Kiss, Miller, & Simon. Please cite the book if using this algorithm.

This model is identical to the SIS compact pairwise model, so it simply calls SIS_compact_pairwise()

EoN.SIS_compact_effective_degree_from_graph

EoN.**SIS_compact_effective_degree_from_graph**(*G*, *tau*, *gamma*, *initial_infecteds=None*, *rho=None*, *tmin=0*, *tmax=100*, *tcount=1001*, *return_full_data=False*)

because the SIS compact effective degree model is identical to the compact pairwise model, simply calls SIS_compact_pairwise_from_graph

EoN.Epi_Prob_discrete

EoN.**Epi_Prob_discrete**(*Pk*, *p*, *number_its=100*)

Encodes System (6.2) of Kiss, Miller, & Simon. Please cite the book if using this algorithm.

Arguments

Pk dict Pk[k] is probability a node has degree k.

p number transmission probability

number_its int number of iterations before assumed converged. default value is 100

Returns

PE float Calculated Epidemic probability in the limit of a negligible fraction initially infected (assuming configuration model)

EoN.Epi_Prob_cts_time

EoN.**Epi_Prob_cts_time**(*Pk*, *tau*, *gamma*, *umin=0*, *umax=10*, *ucount=1001*, *number_its=100*)
Encodes System (6.3) of Kiss, Miller, & Simon. Please cite the book if using this algorithm.

The equations are rescaled by setting $u=gamma T$. Then it becomes

P = 1- int_0^infty psi(alpha(u/gamma)) e^{-u} du alpha_d(u/gamma) = 1- p(u/gamma)

- p(u/gamma) int_0^infty

  (psiPrime(alpha(hat{u}/gamma))/<K>) e^{-u}du

where p(u/gamma) = 1 - e^{-tau u/gamma}

Define hat{p}(u) = p(u/gamma), and hat{alpha}(u) = alpha(u/gamma)

and then drop hats to get

P = 1-int_0^infty psi(alpha(u)) e^{-u} du alpha(u) = 1-p(u) + p(u)

int_0^infty (psiPrime(alpha(u))/<K>)e^{-u} du

with initial guess alpha_1(u) = e^{-tau u/gamma}

and p(u) = 1-e^{-tau u/gamma}

Arguments

Pk dict Pk[k] is probability a node has degree k.

tau float transmission rate

gamma float recovery rate

umin minimal value of gamma T used in calculation **umax** maximum value of gamma T used in calculation **ucount** number of points taken for integral.

So this integrates from umin to umax using simple Riemann sum.

number_its int number of iterations before assumed converged. default value is 100

Returns

PE float Calculated Epidemic probability (assuming configuration model)

EoN.Epi_Prob_non_Markovian

EoN.**Epi_Prob_non_Markovian**(*Pk*, *Pxidxi*, *po*, *number_its=100*)
Encodes system (6.5) of Kiss, Miller, & Simon. Please cite the book if using this algorithm.

Arguments

Pk dict Pk[k] is probability a node has degree k.

Pxidxi dict Pxidxi[xi] is P(xi)dxi for user-selected xi. The algorithm will replace the integral with sum_{xi in Pxidxi.keys()} psi(alpha(xi)) Pxidxi(xi)

po a function. returns p_o(xi), the probability a node will transmit to a random neighbor given xi.

number_its int number of iterations before assumed converged. default value is 100

Returns

PE float Calculated Epidemic probability (assuming configuration model)

EoN.Attack_rate_discrete

EoN.**Attack_rate_discrete**(*Pk*, *p*, *rho=None*, *Sk0=None*, *phiS0=None*, *phiR0=0*, *number_its=100*)

Encodes systems (6.6) and (6.10) of Kiss, Miller, & Simon. Please cite the book if using this algorithm.

To use system (6.6), leave rho and Sk0 as None.

Arguments

Pk dict Pk[k] is the probability a randomly selected node has degree k.

tau positive float per-edge transmission rate.

gamma number per-node recovery rate

number_its int The solution is found iteratively, so this determines the number of iterations.

rho Number (default `None`) proportion of the population to be randomly infected at time 0 only one of rho and Sk0 can be defined. The other (or both) should remain None. if rho=0, then calculates the limiting attack rate as rho->0 (assuming an epidemic happens)

Sk0 dict (default `None`) only one of rho and Sk0 can be defined. The other (or both) should remain None. Sk0 is a dict such that Sk0[k] is the probability that a degree k node is susceptible at start.

phiS0 number (default `None`) Should only be used if Sk0 is not None. If it is None, then assumes that initial introduction is randomly introduced

phiR0 number (default 0) As with phiS0, only used if Sk0 is not None.

Returns

AR float the predicted fraction infected.

EoN.Attack_rate_discrete_from_graph

EoN.**Attack_rate_discrete_from_graph**(*G*, *p*, *initial_infecteds=None*, *initial_recovereds=None*, *rho=None*, *number_its=100*)

if initial_infecteds and initial_recovereds is defined, then it will find Sk0, phiS0, and phiR0 and then call Attack_rate_discrete.

Otherwise it calls attack_rate_discrete with rho.

EoN.Attack_rate_cts_time

EoN.**Attack_rate_cts_time**(*Pk*, *tau*, *gamma*, *number_its=100*, *rho=None*, *Sk0=None*, *phiS0=None*, *phiR0=0*)

Encodes system (6.7) of Kiss, Miller, & Simon. Please cite the book if using this algorithm.

This system predicts the fraction of nodes infected if an epidemic occurs in a Configuration Model network assuming a continuous-time Markovian SIR disease.

This gives the limit of the attack rate of epidemics as the initial fraction infected approaches 0.

If we look for the limit of a nonzero initial fraction infected, we introduce rho or Sk0

Arguments

Pk dict the probability a randomly selected node has degree k.

tau positive float per-edge transmission rate.

gamma number per-node recovery rate

number_its int The solution is found iteratively, so this determines the number of iterations.

rho number, optional The initial proportion infected (defaults to None). If None, then result is limit of rho->0.

Sk0 dict (default `None`) only one of rho and Sk0 can be defined. The other (or both) should remain None. rho gives the fraction of nodes randomly infected. Sk0 is a dict such that Sk0[k] is the probability that a degree k node is susceptible at start.

Returns

AR float the predicted fraction infected.

EoN.Attack_rate_cts_time_from_graph

EoN.**Attack_rate_cts_time_from_graph**(*G*, *tau*, *gamma*, *initial_infecteds=None*, *initial_recovereds=None*, *rho=None*, *number_its=100*)

Given a graph, predicts the attack rate for Configuration Model networks with the given degree distribution. This does not account for any structure in G beyond degree distribution.

First calculates the degree distribution and then calls *Attack_rate_cts_time*.

See also: *estimate_SIR_prob_size(G, p)* which accounts for entire structure of G, not just degree distribution.

EoN.Attack_rate_non_Markovian

EoN.**Attack_rate_non_Markovian**(*Pk*, *Pzetadzeta*, *pi*, *number_its=100*)

Encodes system (6.8) of Kiss, Miller, & Simon. Please cite the book if using this algorithm.

Arguments

Pk dict Pk[k] is probability of degree k.

Pzetadzeta a dict. gives P(zeta)dzeta for user-selected zeta. The algorithm will replace the integral with sum_{zeta in Pzetadzeta.keys()} psi(alpha(zeta)) Pzetadzeta(zeta)

pi a function. returns p_i(zeta), the probability a node will receive a transmission from random infected neighbor given zeta.

number_its int (default 100)

number of iterations before assumed converged. default value is 100

Returns

AR float attack rate

Comments Because of the symmetry for epidemic probability, this works by simply calling Epi_Prob_non_Markovian.

EoN.EBCM_discrete

EoN.**EBCM_discrete**(*N*, *psihat*, *psihatPrime*, *p*, *phiS0*, *phiR0=0*, *R0=0*, *tmin=0*, *tmax=100*, *return_full_data=False*)

Encodes system (6.11) of Kiss, Miller, & Simon. Please cite the book if using this algorithm.

theta(t) = (1-p) + p(phi_R(0)

- phi_S(0) psihatPrime(theta(t-1))/psihatPrime(1))

R(t) = R(t-1) + I(t-1) S(t) = N psihat(theta(t)) I(t) = N-S-R

Arguments

N positive integer size of population

psihat function psihat(x) = sum_k S(k,0) x^k

psihatPrime function psihatPrime(x) = d psihat(x)/dx = sum_k k S(k,0) x^{k-1}

p number per edge transmission probability

phiS0 number initial proportion of edges (of susceptible nodes) connecting to susceptible nodes

phiR0 number initial proportion of edges (of susceptible nodes) connecting to recovered nodes

R0 number number of recovered nodes at time 0

tmax number maximum time

return_full_data boolean

if False, return t, S, I, R

if True return t, S, I, R, and theta

Returns

if return_full_data == False: returns **t, S, I, R**, all numpy arrays

if . . . == True returns **t, S, I, R** and **theta**, all numpy arrays

EoN.EBCM_discrete_from_graph

EoN.**EBCM_discrete_from_graph**(*G*, *p*, *initial_infecteds=None*, *initial_recovereds=None*, *rho=None*, *tmin=0*, *tmax=100*, *return_full_data=False*)

Takes a given graph, finds the degree distribution (from which it gets psi), assumes a constant proportion of the population is infected at time 0, and then uses the discrete EBCM model.

Arguments

G Networkx Graph the contact network

p number per edge transmission probability

initial infecteds node or iterable of nodes (default `None`) if a single node, then this node is initially infected if an iterable, then whole set is initially infected if None, then choose randomly based on rho. If rho is also None, a random single node is chosen. If both initial_infecteds and rho are assigned, then there is an error.

initial_recovereds iterable of nodes (default `None`) this whole collection is made recovered. Currently there is no test for consistency with initial_infecteds. Understood that everyone who isn't infected or recovered initially is initially susceptible.

rho float between 0 and 1 (default `None`) the fraction to be randomly infected at time 0 If None, then rho=1/N is used where N = G.order()

tmax number maximum time

return_full_data boolean

if False, return t, S, I, R and if True return t, S, I, R, and theta

Returns

if return_full_data == False: returns **t, S, I, R**, all numpy arrays

if . . . == True returns **t, S, I, R** and **theta**, all numpy arrays

EoN.EBCM

EoN.**EBCM**(*N*, *psihat*, *psihatPrime*, *tau*, *gamma*, *phiS0*, *phiR0=0*, *R0=0*, *tmin=0*, *tmax=100*, *tcount=1001*, *return_full_data=False*)

Encodes system (6.12) of Kiss, Miller, & Simon. Please cite the book if using this algorithm.

note : R0 is R(0), not the reproductive number

Arguments

N positive integer size of population

psihat function psihat(x) = sum_k S(k,0) x^k

psihatPrime function psihatPrime(x) = d psihat(x)/dx = sum_k k S(k,0) x^{k-1} where S(k,0) is the probability a random node has degree k and is susceptible at start time.

tau positive float per edge transmission rate

gamma positive float per node recovery rate

phiS0 positive float initial proportion of edges (of susceptible nodes) connecting to susceptible nodes

phiR0 positive float initial proportion of edges (of susceptible nodes) connecting to recovered nodes

R0 positive integer number of recovered nodes at time 0

tmin number start time

tmax number stop time

tcount positive integer number of distinct times to calculate

return_full_data boolean

if False, return t, S, I, R

if True return t, S, I, R, and theta

Returns

if return_full_data == False: returns **t, S, I, R**, all numpy arrays

if . . . == True returns **t, S, I, R** and **theta**, all numpy arrays

EoN.EBCM_uniform_introduction

EoN.**EBCM_uniform_introduction**(*N*, *psi*, *psiPrime*, *tau*, *gamma*, *rho*, *tmin=0*, *tmax=100*, *tcount=1001*, *return_full_data=False*)

Handles the case that the disease is introduced uniformly as opposed to depending on degree.

Arguments

N positive integer size of population

psi function psihat(x) = sum_k S(k,0) x^k

psiPrime function psihatPrime(x) = d psihat(x)/dx = sum_k k S(k,0) x^{k-1}

tau positive float per edge transmission rate

gamma number per node recovery rate

rho number initial proportion infected

tmin number start time

tmax number stop time

tcount integer number of distinct times to calculate

return_full_data boolean

if False, return t, S, I, R

if True return t, S, I, R, and theta

Returns

if return_full_data == False: returns **t, S, I, R**, all numpy arrays

if . . . == True returns **t, S, I, R** and **theta**, all numpy arrays

EoN.EBCM_from_graph

EoN.**EBCM_from_graph**(*G*, *tau*, *gamma*, *initial_infecteds=None*, *initial_recovereds=None*, *rho=None*, *tmin=0*, *tmax=100*, *tcount=1001*, *return_full_data=False*)

Given network G and rho, calculates N, psihat, psihatPrime, and calls EBCM.

EoN.EBCM_pref_mix

EoN.**EBCM_pref_mix**(*N*, *Pk*, *Pnk*, *tau*, *gamma*, *rho=None*, *tmin=0*, *tmax=100*, *tcount=1001*, *return_full_data=False*)

Encodes the system derived in exercise 6.21 of Kiss, Miller, & Simon. Please cite the book if using this algorithm.

I anticipate eventually adding an option so that the initial condition is not uniformly distributed. So could give rho_k

Arguments

N positive integer number of nodes.

Pk dict (could also be an array or a list) Pk[k] is the probability a random node has degree k.

Pnk dict of dicts (possibly array/list) Pnk[k1][k2] is the probability a neighbor of a degree k1 node has degree k2.

tau positive float transmission rate

gamma number recovery rate

rho number (optional) initial proportion infected. Defaults to 1/N.

tmin number (default 0) minimum time

tmax number (default 100) maximum time

tcount integer (default 1001) number of time points for data (including end points)

return_full_data boolean (default False) whether to return theta or not

Returns

if return_full_data == False: returns **t, S, I, R**, all numpy arrays

if . . . == True returns **t, S, I, R** and **theta** where theta[k] is a numpy array giving theta for degree k

EoN.EBCM_pref_mix_from_graph

EoN.**EBCM_pref_mix_from_graph**(*G*, *tau*, *gamma*, *rho=None*, *tmin=0*, *tmax=100*, *tcount=1001*, *return_full_data=False*)

Takes a given graph, finds degree correlations, and calls EBCM_pref_mix

I anticipate eventually adding an option so that the initial condition is not uniformly distributed. So could give rho_k

Arguments

G networkx Graph The contact network

tau positive float transmission rate

gamma positive float recovery rate

rho positive float (default `None`) initial proportion infected. Defaults to 1/N.

tmin number (default 0) minimum time

tmax number (default 100) maximum time

tcount integer (default 1001) number of time points for data (including end points)

return_full_data boolean (default False) whether to return theta or not

Returns

if return_full_data == False: returns **t, S, I, R**, all numpy arrays

if . . . == True returns **t, S, I, R** and **theta**, where theta[k] is a numpy array giving theta for degree k

EoN.EBCM_pref_mix_discrete

EoN.**EBCM_pref_mix_discrete**(*N*, *Pk*, *Pnk*, *p*, *rho=None*, *tmin=0*, *tmax=100*, *return_full_data=False*)

Encodes the discrete version of exercise 6.21 of Kiss, Miller, & Simon. Please cite the book if using this algorithm.

I anticipate eventually adding an option so that the initial condition is not uniformly distributed. So could give rho_k

Arguments

N positive integer number of nodes.

Pk dict (could also be an array or a list) Pk[k] is the probability a random node has degree k.

Pnk dict of dicts (possibly array/list) Pnk[k1][k2] is the probability a neighbor of a degree k1 node has degree k2.

p positive float (0 <= p <= 1) transmission probability

rho number (optional) initial proportion infected. Defaults to 1/N.

tmin number (default 0) minimum time

tmax number (default 100) maximum time

tcount integer (default 1001) number of time points for data (including end points)

return_full_data boolean (default False) whether to return theta or not

Returns

if return_full_data == False: returns **t, S, I, R**, all numpy arrays

if . . . == True returns **t, S, I, R** and **theta**, where theta is a dict and theta[k] is the thetas for given k.

EoN.EBCM_pref_mix_discrete_from_graph

EoN.**EBCM_pref_mix_discrete_from_graph**(*G*, *p*, *rho=None*, *tmin=0*, *tmax=100*, *return_full_data=False*)

Takes a given graph, finds degree correlations, and calls EBCM_pref_mix_discrete

SAMPLE USE

```
import networkx as nx
import EoN
import matplotlib.pyplot as plt
G = nx.bipartite.configuration_model([5]*300000, [2]*750000)
t, S, I, R = EoN.basic_discrete_SIR(G, 0.6, rho = 0.002)
tx, Sx, Ix, Rx = EoN.EBCM_pref_mix_discrete_from_graph(G, 0.6, rho=0.002, tmax=t[-
↪1])
plt.plot(t, I, label = 'simulation')
plt.plot(tx, Ix, '--', label = 'analytic prediction')
plt.legend(loc='upper right')
plt.show()
```

EoN.get_Pk

EoN.**get_Pk**(*G*)

Used in several places so that we can input a graph and then we can call the methods that depend on the degree distribution

Arguments

G networkx Graph

Returns

Pk dict `Pk[k]` is the proportion of nodes with degree `k`.

EoN.get_PGF

EoN.**get_PGF**(*Pk*)

Given a degree distribution (as a dict), returns the probability generating function

Arguments

Pk dict Pk[k] is the proportion of nodes with degree k.

Returns

psi function $\psi(x) = \sum_k P_k[k]x^k$

EoN.get_PGFPrime

EoN.**get_PGFPrime**(*Pk*)

Given a degree distribution (as a dict) returns the function $\psi'(x)$

Arguments

Pk dict Pk[k] is the proportion of nodes with degree k.

Returns

psiPrime (function) $\psi'(x) = \sum_k kP_k[k]x^{k-1}$

EoN.get_PGFDPrime

EoN.**get_PGFDPrime**(*Pk*)

Given a degree distribution (as a dict) returns the function $\psi''(x)$

Arguments

Pk dict Pk[k] is the proportion of nodes with degree k.

Returns

psiDPrime function $\psi''(x) = \sum_k k(k-1)P_k[k]x^{k-2}$

EoN.estimate_R0

EoN.**estimate_R0**(*G*, *tau=None*, *gamma=None*, *transmissibility=None*)
provides the estimate of the reproductive number R_0 = T <K^2-K>/<K>

This handles the Markovian continuous time case with T = tau/(tau+gamma) and it handles other cases by the user inputting an average transmissibility

For <K^2-K>/<K> it measures the network.

Arguments

G networkx Graph

tau positive float (default None) transmission rate

Either both tau and gamma must be given or transmissibility is given.

gamma positive float (default None) recovery rate

Either both tau and gamma must be given or transmissibility is given.

transmissibility positive float (default None) average transmission probability

Either both tau and gamma must be given or transmissibility is given.

Returns

R_0 float Reproductive number $\mathcal{R}_0 = T\langle K^2 - K\rangle/\langle K\rangle$

Short description

These come from the book. The numbers given below are the equation numbers in the book.

- Chapter 3

  This chapter deals with models assuming we know the full network structure.

  - System (3.7): SIS model: Closes equations by assuming that knowing the probabilities for nodes to have each status is enough to predict impact of their interactions (ignores temporal correlation between statuses of neighbors).
    - **SIS_individual_based**
    - **SIS_individual_based_pure_IC**

    The pure_IC version assumes that some nodes begin infected with probability 1 and the others are susceptible with probability 1.
  - System (3.26): assumes that tracking pair correlations is enough. Many more equations than individual-based.
    - **SIS_pair_based**
    - **SIS_pair_based_pure_IC**
  - System (3.30) SIR equivalent of corresponding SIS model.
    - **SIR_individual_based**
    - **SIR_individual_based_pure_IC**
  - System (3.39) SIR equivalent of corresponding SIS model.
    - **SIR_pair_based**

        * **SIR_pair_based_pure_IC**

- Chapter 4

  This chapter attempts to approximate the exact dynamics by ignoring heterogeneity in degree.

  - System (4.8) Assumes dynamics determined by average number of contacts and number of nodes of each status.
    * **SIS_homogeneous_meanfield**
  - System (4.9) As for SIS.
    * **SIR_homogeneous_meanfield**
  - System (4.10) Assumes dynamics are determined by the average number of contacts, nodes of each status, and pairs of each status.
    * **SIS_homogeneous_pairwise**
    * **SIS_homogeneous_pairwise_from_graph** (reads properties from input graph)
  - System (4.11)
    * **SIR_homogeneous_pairwise**
    * **SIR_homogeneous_pairwise_from_graph**

- Chapter 5

  This chapter attempts to approximate the exact dynamics and incorporates heterogeneity in degree (at the cost of more complex models)

  - System (5.10)
    * **SIS_heterogeneous_meanfield**
    * **SIS_heterogeneous_meanfield_from_graph**
  - System (5.11)
    * **SIR_heterogeneous_meanfield**
    * **SIR_heterogeneous_meanfield_from_graph**
  - System (5.13)
    * **SIS_heterogeneous_pairwise**
    * **SIS_heterogeneous_pairwise_from_graph**
  - System (5.15)
    * **SIR_heterogeneous_pairwise**
    * **SIR_heterogeneous_pairwise_from_graph**
  - System (5.18)
    * **SIS_compact_pairwise**
    * **SIS_compact_pairwise_from_graph**
  - System (5.19)
    * **SIR_compact_pairwise**
    * **SIR_compact_pairwise_from_graph**
  - System (5.20)

        - **SIS_super_compact_pairwise**
        - **SIS_super_compact_pairwise_from_graph**
    - System (5.22)
        - **SIR_super_compact_pairwise**
        - **SIR_super_compact_pairwise_from_graph**
    - System (5.36)
        - **SIS_effective_degree**
        - **SIS_effective_degree_from_graph**
    - System (5.38)
        - **SIR_effective_degree**
        - **SIR_effective_degree_from_graph**
    - System (5.43)
        - **SIR_compact_effective_degree**
        - **SIR_compact_effective_degree_from_graph**
    - System (5.44)
        - **SIS_compact_effective_degree**
        - **SIS_compact_effective_degree_from_graph**
- Chapter 6

  This chapter uses percolation-based techniques to explore epidemic properties.
    - System (6.2) Given a degree distribution and uniform transmission probability, find epidemic probability.
        - **Epi_Prob_discrete**
    - System (6.3) As in 6.2, but assuming constant transmission and recovery rates.
        - **Epi_Prob_cts_time**
    - System (6.5) As in 6.2, but with user-specified transmission rules
        - **Epi_Prob_non_Markovian**
    - System (6.6) Given a degree distribution, initial proportion infected, and transmission probability, find attack rate. See also System (6.10).
        - **Attack_rate_discrete**
        - **Attack_rate_discrete_from_graph**
    - System (6.7) as in 6.6, but assuming constant transmission and recovery rates.
        - **Attack_rate_cts_time**
        - **Attack_rate_cts_time_from_graph**
    - System (6.8) As in 6.6, but with user-specified transmission rules
        - **Attack_rate_non_Markovian**
    - System (6.10) See code for System (6.6).
    - System (6.11) Perform EBCM calculations for discrete-time.

    - **EBCM_discrete**
    - **EBCM_discrete_from_graph**
  - System (6.12) Perform EBCM calculations for continuous-time.
    - **EBCM** allows initial status to be degree dependant.
    - **EBCM_uniform_introduction** assumes disease introduced at t=0 uniformly at random
    - **EBCM_from_graph** assumes disease introduced at t=0 uniformly at random in network of given degree distribution.
  - **exercise 6.21 Deals with the EBCM model assuming preferential mixing.**
    - **EBCM_pref_mix**
    - **EBCM_pref_mix_from_graph**
    - **EBCM_pref_mix_discrete**
    - **EBCM_pref_mix_discrete_from_graph**

2.3.5 Auxiliary Functions

We have a few additional functions which are of value.

Quick List

get_time_shift(times, L, threshold)	Identifies the first time at which list/array L crosses a threshold.
subsample(report_times, times, status1[, ...])	Given a list/array of times to report at, returns the number of nodes of each status at those times.
hierarchy_pos(G[, root, width, vert_gap, ...])	If the graph is a tree this will return the positions to plot this in a hierarchical layout.

EoN.get_time_shift

EoN.**get_time_shift**(*times*, *L*, *threshold*)

Identifies the first time at which list/array L crosses a threshold. Useful for shifting times.

Arguments

times list or numpy array (ordered) the times we have observations

L a list or numpy array order of L corresponds to times

threshold number the threshold value

Returns

t number the first time at which L reaches or exceeds threshold.

SAMPLE USE

```
import networkx as nx
import EoN
import numpy as np
import matplotlib.pyplot as plt

""" in this example we will run 20 stochastic simulations.
    We plot the unshifted curves (grey) and the curves shifted
    so that t=0 when 1% have been infected (I+R = 0.01N) (red)
"""
plt.clf() # just clearing any previous plotting.

N=100000
kave = 10.
G = nx.fast_gnp_random_graph(N,kave/(N-1.))
tau = 0.2
gamma = 1.
report_times = np.linspace(0,5,101)
Ssum = np.zeros(len(report_times))
Isum = np.zeros(len(report_times))
Rsum = np.zeros(len(report_times))
iterations = 20
for counter in range(iterations):
    R=[0]
    while R[-1]<1000: #if an epidemic doesn't happen, repeat
        t, S, I, R = EoN.fast_SIR(G, tau, gamma)
        print R[-1]
    plt.plot(t, I, linewidth = 1, color = 'gray', alpha=0.4)
    tshift = EoN.get_time_shift(t, I+R, 0.01*N)
    plt.plot(t-tshift, I, color = 'red', linewidth = 1, alpha = 0.4)
plt.savefig("timeshift_demonstration.pdf")
```

EoN.subsample

EoN.**subsample**(*report_times*, *times*, *status1*, *status2=None*, *status3=None*)

Given a list/array of times to report at, returns the number of nodes of each status at those times.

returns them subsampled at specific report_times.

Arguments

report_times iterable (ordered) times at which we want to know state of system

times iterable (ordered) times at which we have the system state (assumed no change between these times)

status1 iterable generally S, I, or R

number of nodes in given status at corresponding time in times.

status2 iterable (optional, default None) generally S, I, or R

number of nodes in given status at corresponding time in times.

status3 iterable (optional, default None) generally S, I, or R

number of nodes in given status at corresponding time in times.

Returns

If only status1 is defined report_status1 numpy array gives `status1` subsampled just at `report_times`.

If more are defined then it returns a list, either [report_status1, report_status2]

or [report_status1, report_status2, report_status3]

In each case, these are subsampled just at report_times.

SAMPLE USE

```
import networkx as nx
import EoN
import numpy as np
import matplotlib.pyplot as plt

""" in this example we will run 100 stochastic simulations.
    Each simulation will produce output at a different set
    of times.  In order to calculate an average we will use
    subsample to find the epidemic sizes at a specific set
    of times given by report_times.
"""

G = nx.fast_gnp_random_graph(10000,0.001)
tau = 1.
gamma = 1.
report_times = np.linspace(0,5,101)
Ssum = np.zeros(len(report_times))
Isum = np.zeros(len(report_times))
Rsum = np.zeros(len(report_times))
iterations = 100
for counter in range(iterations):
    t, S, I, R = EoN.fast_SIR(G, tau, gamma, initial_infecteds = range(10))
    #t, S, I, and R have an entry for every single event.
    newS, newI, newR = EoN.subsample(report_times, t, S, I, R)
    #could also do: newI = EoN.subsample(report_times, t, I)
    plt.plot(report_times, newS, linewidth=1, alpha = 0.4)
    plt.plot(report_times, newI, linewidth=1, alpha = 0.4)
    plt.plot(report_times, newR, linewidth=1, alpha = 0.4)
    Ssum += newS
    Isum += newI
    Rsum += newR
Save = Ssum / float(iterations)
Iave = Isum / float(iterations)
Rave = Rsum / float(iterations)
plt.plot(report_times, Save, "--", linewidth = 5, label = "average")
plt.plot(report_times, Iave, "--", linewidth = 5)
plt.plot(report_times, Rave, "--", linewidth = 5)
plt.legend(loc = "upper right")
plt.savefig("tmp.pdf")
```

If only one of the sample times is given then returns just that.

If report_times goes longer than times, then this simply assumes the system freezes in the final state.

This uses a recursive approach if multiple arguments are defined.

EoN.hierarchy_pos

EoN.**hierarchy_pos**(*G*, *root=None*, *width=1.0*, *vert_gap=0.2*, *vert_loc=0*, *leaf_vs_root_factor=0.5*)

If the graph is a tree this will return the positions to plot this in a hierarchical layout.

Based on Joel's answer at https://stackoverflow.com/a/29597209/2966723, but with some modifications.

We include this because it may be useful for plotting transmission trees, and there is currently no networkx equivalent (though it may be coming soon).

There are two basic approaches we think of to allocate the horizontal location of a node.

- Top down: we allocate horizontal space to a node. Then its `k` descendants split up that horizontal space equally. This tends to result in overlapping nodes when some have many descendants.
- Bottom up: we allocate horizontal space to each leaf node. A node at a higher level gets the entire space allocated to its descendant leaves. Based on this, leaf nodes at higher levels get the same space as leaf nodes very deep in the tree.

We use use both of these approaches simultaneously with `leaf_vs_root_factor` determining how much of the horizontal space is based on the bottom up or top down approaches. `0` gives pure bottom up, while 1 gives pure top down.

Arguments

G the graph (must be a tree)

root the root node of the tree - if the tree is directed and this is not given, the root will be found and used - if the tree is directed and this is given, then the positions will be

just for the descendants of this node.

- if the tree is undirected and not given, then a random choice will be used.

width horizontal space allocated for this branch - avoids overlap with other branches

vert_gap gap between levels of hierarchy

vert_loc vertical location of root

leaf_vs_root_factor

xcenter: horizontal location of root

Short Description

- **get_time_shift** (allows us to shift plots to eliminate the effect of early-time stochasticity)
- **subsample** (allows us to take output given at a stochastic set of times and get output at given times - particularly useful to allow for averaging multiple simulations)
- **hierarchy_pos** (Provides positions that help visualize the transmission chain from a Simulation_Investigation object)

2.4 Changes from v 1.0

2.4.1 New in v 1.2rc1

- Corrected bug affecting code with rates weighted by node for new networkx. Due to this change, those parts of the code require networkx 2.0 or greater.
- updated `Gillespie_simple_contagion` so that if both random and numpy.random keys are set, the code will produce reproducible results.

2.4.2 New in v 1.1

- `Hierarchy_Pos` has an extraneous print statement removed.
- `Gillespie_simple_contagion` should now accept a directed graph `G`.
- Small bug fix in `Gillespie_simple_contagion` which would cause any attempt to assign a rate function to crash

2.4.3 New in v 1.0.8

- Bug fixes in `basic_discrete_SIS`.
- The `Simulation_Investigation` objects can now handle arbitrary statuses, rather than just SIS and SIR.
- The `display` and `animate` functions now allow an optional `statuses_to_plot` argument, allowing us to leave some statuses out. This may require networkx v2.3 or later to work right.
- The `Simulation_Investigation` code now handles plotting things like `'S+V'` if we add a time series appropriately. The last example of *Visualizing or animating disease spread* shows this.
- The `Gillespie_simple_contagion` and `Gillespie_complex_contagion` code can now handle `return_full_data=True`.
- `Gillespie_simple_contagion` is now more flexible in how it handles heterogeneity. The user can now define a function which will give the 'transmission' rates between a pair of nodes and the 'recovery' rates of individual nodes. So it can be more general than the original version. (a heterogeneous SIRS example is now provided)
- There is now a `hierarchy_pos` function which allows us to plot transmission trees in a nice way.
- Changed the discrete SIS and SIR code so that the initial infections occur at t=-1 for the `simulation_investigation` objects.
- Small change to the default color for infected nodes (FF2020->FF2000) in `simulation_investigation`

2.4.4 New in v 1.0.7

No changes (fixing an error in a tag)

2.4.5 New in v 1.0.6

Documentation for `Gillespie_complex_contagion` now includes an example.

Removed print command (left over from debugging) from `Gillespie_complex_contagion.`

2.4.6 New in v 1.0.5

Reintroduced `Gillespie_Arbitrary` which just calls `Gillespie_simple_contagion` and provides a warning that it will be discontinued later.

2.4.7 New in v 1.0.4

Have added `Gillespie_complex_contagion` which can handle complex contagions.

The old `Gillespie_Arbitrary` has been renamed `Gillespie_simple_contagion`. I have fixed a bug in previous versions that prevented it from handling weighted graphs.

`Gillespie_Arbitrary` is now back-compatible to networkx 1.11 (but it has been renamed – see above).

Readthedocs is now providing documentation for each function.

2.4.8 New in v 1.0.3

No changes to package, but a small change attempting to get readthedocs to correctly build.

2.4.9 New in v 1.0.2

No changes (I accidentally made a typo just before uploading v1.0.1 to pypi and I can't reupload with the same name).

2.4.10 New in v 1.0.1

Returning transmission chains

When simulations have `return_full_data=True`, the returned object now includes information on who infected whom at each time. This can be accessed through:

transmissions which returns a list of tuples (t,u,v) stating that node u infected node v at time t.

transmission_tree which returns a directed multi graph where an edge from u to v with attribute 'time' equal to t means u infected v at time t.

(note that in an SIS epidemic, this "tree" may have cycles and repeated edges)

(addresses issue 21)

Non-SIS/SIR processes

It is now possible to run a wide range of non-SIS/SIR processes spreading in a network. These processes include competing diseases, SIRS disease, SEIR disease, and quite a few other options. This is done using:

Gillespie_Arbitrary.

Examples are here.

Currently this does not accept `return_full_data=True`, and it requires that the events all occur as Poisson processes (that is, it makes sense to say that there is a rate at which things happen, and that rate depends on the status of the nodes and perhaps some property of the node or the partnership, but nothing else).

(addresses issues 13 & 17)

Index

H

I

N

P

R

S

T

U