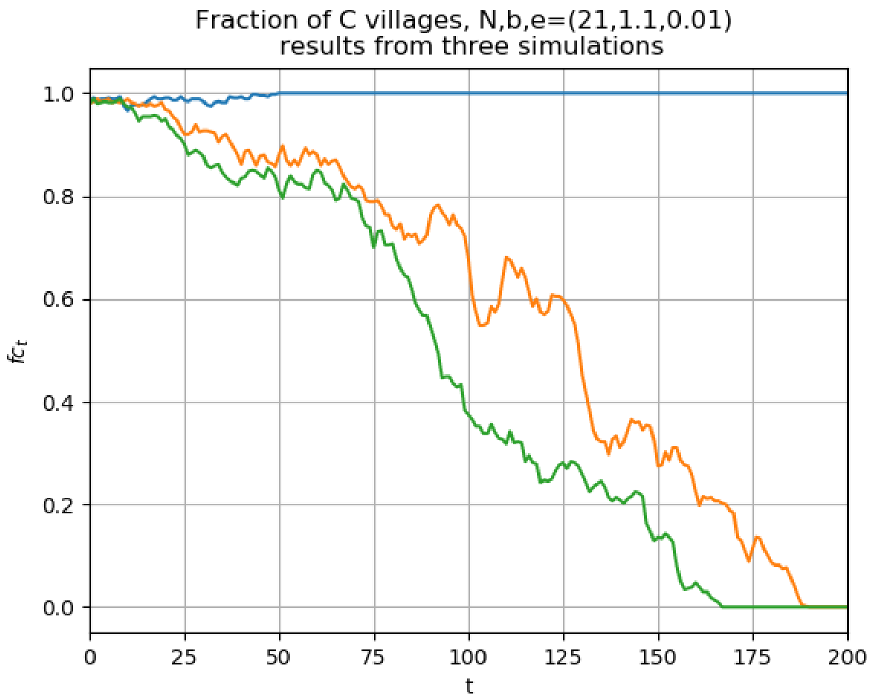
M3C 2018 Homework 1 solution

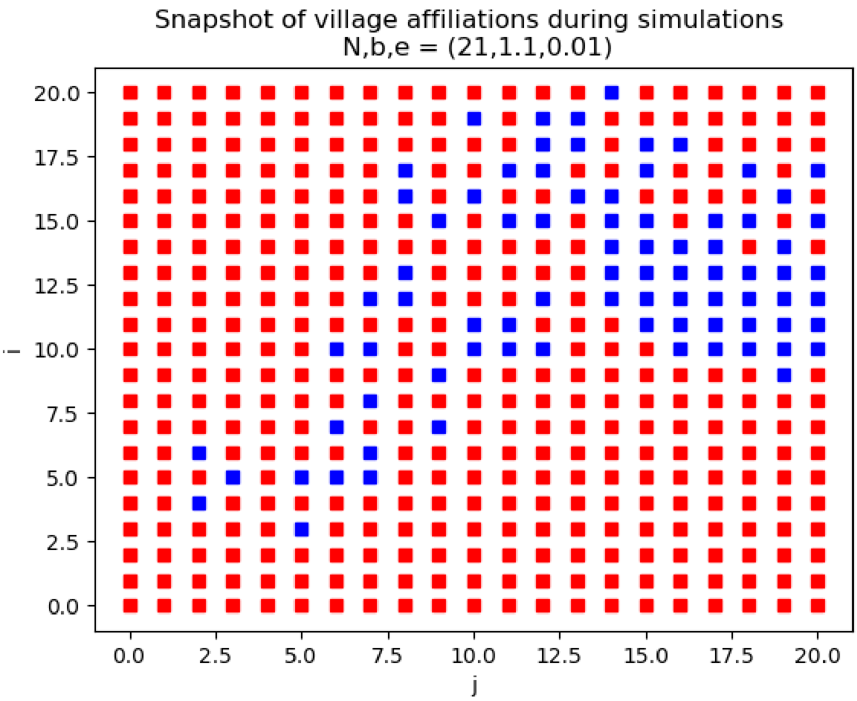
Part 0. (5 pts) You should develop your code in a new git repository. If using Bitbucket, the online repo **must** be private. You should make at least a few commits as you work through the assignment. Once you have completed the assignment, generate a text file, *hw1.txt* at the command line in a Unix terminal which contains the log for your repo. Edit the file so that the first line contains the command used to first generate the file.

ANS: The following will generate the required file:

```
$ git log > hw1.txt
```

Part 1. (50 pts) Complete the function *simulate1* so that it simulates this model with *N*, *b*, *e*, and *Nt* provided as input. Here *Nt* is the number of iterations (years) to run the simulation. The initial village configuration has been provided in the matrix, *S*. The C villages correspond to coordinates with $S_{i,j} = 1$ and *M* villages correspond to $S_{i,j} = 0$. The function should return the final value of *S* after your simulation. Additionally, the function should return the array *fc* which contains the fraction of villages which are Cs at the *Nt+1* timesteps. A simple function for visualizing *S* (*plot_S*) has been provided for you, and you can use (or not use) this as you wish.

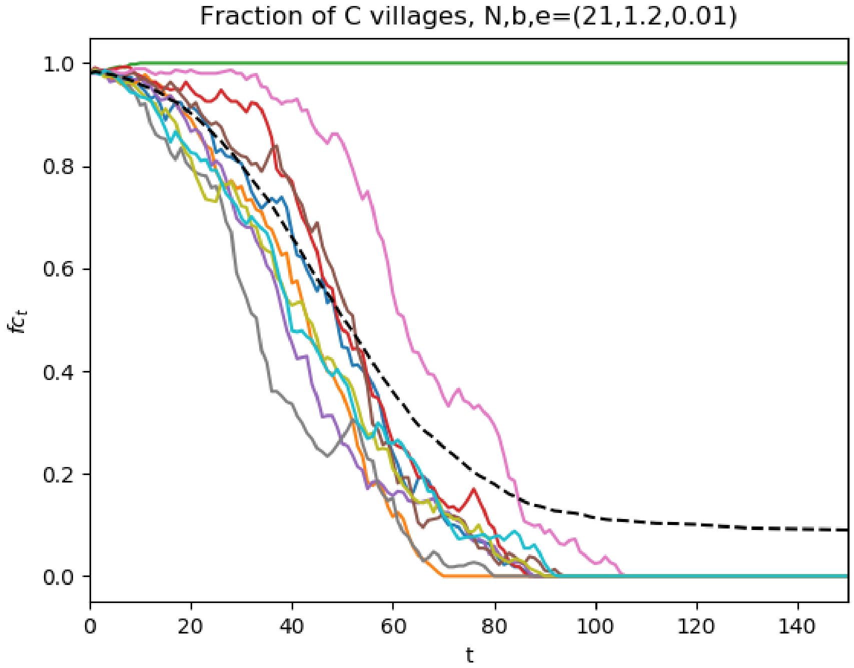


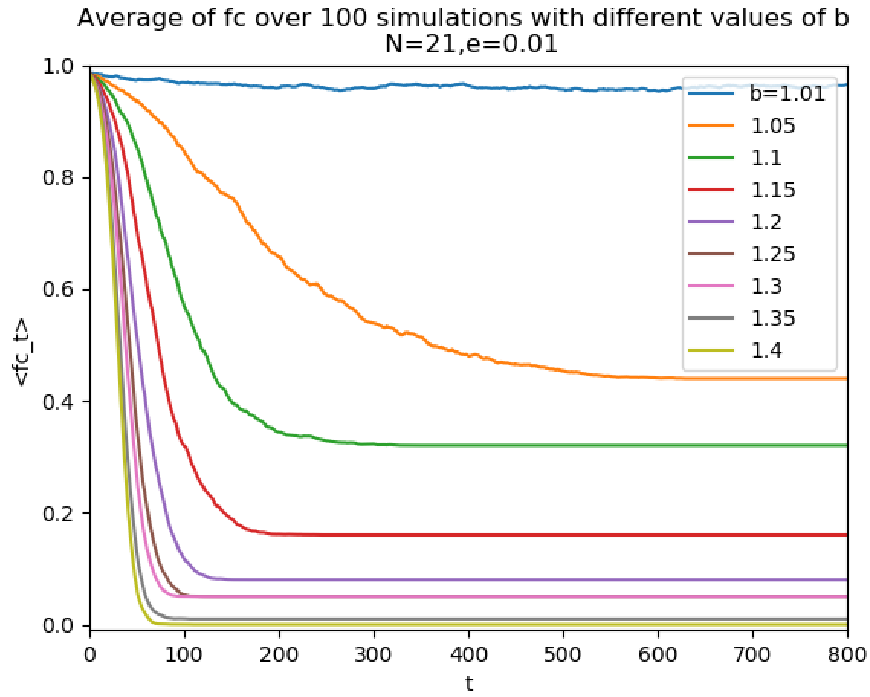


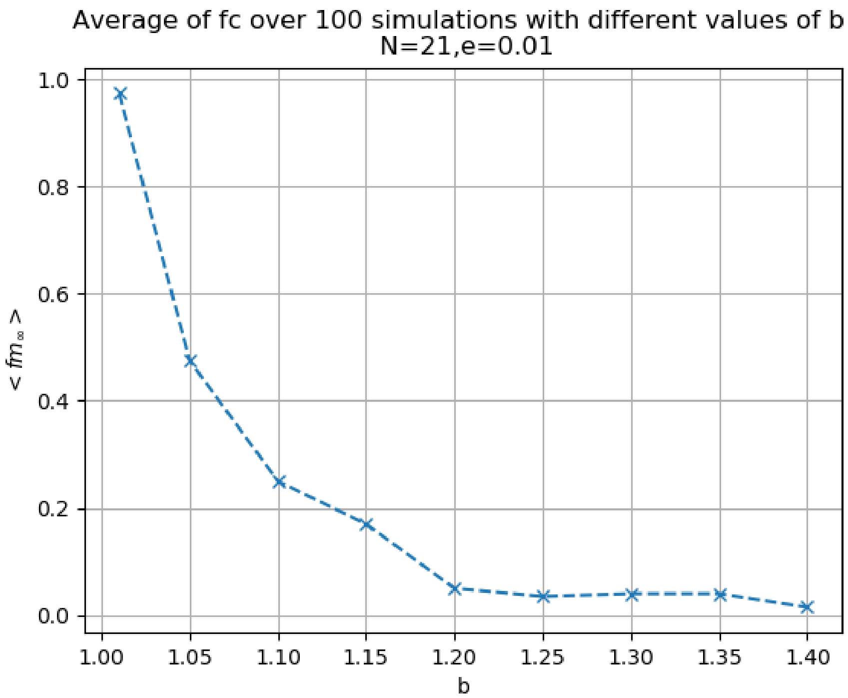
ANS: A few illustrative results are shown in the above two figures. Numerical experiments indicate that at long enough times, the fraction, *fc*, will go to either 0 or 1. With b very close to 1, *fc*, nearly always goes to 1 very quickly, and with b much larger than 1, *fc* always goes to 0 very quickly. However, in the first figure, we see that with b=1.1, fc can go to zero or one, and in part 2, we consider the frequency with which fc approaches these limits with different values of b. In the 2nd figure, we see a representative snapshot of a b=1.1 simulation with irregular clusters of C villages struggling for survival in a sea of red.

Part 2. (45 pts) As *b* is increased (with *e* and *N* held fixed), we expect M to be more and more succesful, and in the second part of the project, you should develop Python code in the function *analyze* which… analyzes if and to what degree this trend is observed. You are encouraged to develop your own approach to this problem, but it may be helpful to consider the evolution of *fc* for different values of *b*. Another point to consider is whether or not *fc* reaches a relatively stable value at long times for some or all values of *b*. Your function should produce 2-4 figures which illustrate the most important qualitative trends which you have identified, and the docstring of the function should contain a clear, concise discussion of these trends and your main conclusions. Save your figures as *.png* files with the names *hw11.png*, *hw12.png*, …, and submit them with your codes. These final results should be generated with *N=21*, *e=0.01*. b… The code in the *name==main* section of the module should call *analyze* and generate the figures you are submitting.

If you decide to modify your *simulate1* code for part 2, please place the modified code in the *simulate2* function provided and call this function from *analyze* instead of *simulate1*.







ANS: The first of the three figures above shows results from both individual simulations and the average, $< fc >$, taken over 100 simulations. The 2nd figure shows the dependence of $< fc >$ on b. For b=1.01, the (average) fraction stays very close to 1, however as b is increased, there tends to be an initial adjustment away from the initial <*fc*> down to a constant value. Both the duration of this adjustment and the 'long-time' value of <*fc*> decrease as b increases. The third figure shows <*fc*> at t=800, and while this clearly shows the key qualitative trend, further refinement (with both more simulations and values of b) is needed before a functional form can be conjectured (though exponential decay would be a good first guess).

The full solution code is below. Note that the only loops are used for time marching in simulate1 and simulate2.

```
"""M3C 2018 Homework 1 solution code

"""
import numpy as np
import matplotlib.pyplot as plt


def simulate1(N,Nt,b,e,S=0):
    """Simulate C vs. M competition on N x N grid over
    Nt generations. b and e are model parameters
    to be used in fitness calculations
    Output: S: Status of each gridpoint at tend of somulation, 0=M, 1=C
    fc: fraction of villages which are C at all Nt+1 times
    Do not modify input or return statement without instructor's permission.
    """
    #Set initial condition
    if S==0:
        S  = np.ones((N,N),dtype=int) #Status of each gridpoint: 0=M, 1=C
        j = int((N-1)/2)
        S[j-1:j+2,j-1:j+2] = 0
    N2inv = 1./(N*N)

    fc = np.zeros(Nt+1) #Fraction of points which are C
    fc[0] = S.sum()*N2inv

    #Initialize matrices
    NB = np.zeros((N,N),dtype=int) #Number of neighbors for each point
    NC = np.zeros((N,N),dtype=int) #Number of neighbors who are Cs
    S2 = np.zeros((N+2,N+2),dtype=int) #S + border of zeros
    F = np.zeros((N,N)) #Fitness matrix
    F2 = np.zeros((N+2,N+2)) #Fitness matrix + border of zeros
    A = np.ones((N,N)) #Fitness parameters, each of N^2 elements is 1 or b
    P = np.zeros((N,N)) #Probability matrix
    Pden = np.zeros((N,N))
    R = np.random.rand(N,N,Nt) #Random numbers used to update S every time step
```

```
    #---------------------

    #Calculate number of neighbors for each point
    NB[:,:] = 8
    NB[0,1:-1],NB[-1,1:-1],NB[1:-1,0],NB[1:-1,-1] = 5,5,5,5
    NB[0,0],NB[-1,-1],NB[0,-1],NB[-1,0] = 3,3,3,3
    NBinv = 1.0/NB
    #-------------

    #----Time marching-----
    for t in range(Nt):

        #Set up coefficients for fitness calculation
        A = np.ones((N,N))
        ind0 = np.where(S==0)
        A[ind0] = b

        #Add boundary of zeros to S
        S2[1:-1,1:-1] = S

        #Count number of C neighbors for each point
        NC = S2[:-2,:-2]+S2[:-2,1:-1]+S2[:-2,2:]+S2[1:-1,:-2] + S2[1:-1,2:] + S2[2:,:-2] + S2[2:,1:-1

        #Calculate fitness matrix
        F = NC*A
        F[ind0] = F[ind0] + (NB[ind0]-NC[ind0])*e
        F = F*NBinv

        #Calculate probability matrix
        F2[1:-1,1:-1]=F
        F2S2 = F2*S2
        P = F2S2[:-2,:-2]+F2S2[:-2,1:-1]+F2S2[:-2,2:]+F2S2[1:-1,:-2] + F2S2[1:-1,1:-1] + F2S2[1:-1,2:
        Pden = F2[:-2,:-2]+F2[:-2,1:-1]+F2[:-2,2:]+F2[1:-1,:-2] + F2[1:-1,1:-1] + F2[1:-1,2:] + F2[2:
        P = P/Pden

        #Set new affiliations based on probability matrix and random numbers stored in R
        #Some/many students will have used np.random.random_choice instead
        S[:,:] = 0
        S[R[:,:,t]<=P] = 1

        fc[t+1] = S.sum()*N2inv
        #plot_S(S)
        #----Finish time marching-----

    return S,fc,P


def plot_S(S):
    """Simple function to create plot from input S matrix
    """
    ind_s0 = np.where(S==0) #C locations
    ind_s1 = np.where(S==1) #M locations
    plt.plot(ind_s0[1],ind_s0[0],'rs')
    plt.hold(True)
    plt.plot(ind_s1[1],ind_s1[0],'bs')
    plt.hold(False)
    plt.show()
    plt.pause(0.05)
    return None


def simulate2(N,Nt,bvalues,e,M,S=0):
    """N,Nt,e are the same as in simulate1
    M simulations are run for each of the values of b stored in bvalues,
    and fc from all simulations are returned to analyze for further... analysis.
    """
    Nb = len(bvalues)
    bv = np.array(bvalues)

    #Set initial condition
    if S==0:
        S  = np.ones((N,N,M,Nb),dtype=int) #Status of each gridpoint: 0=M, 1=C
```

```
            j = int((N-1)/2)
            S[j-1:j+2,j-1:j+2,:,:] = 0
        N2inv = 1./(N*N)

        fc = np.zeros((Nt+1,M,Nb)) #Fraction of points which are C
        fc[0,:,:] = S.sum(axis=(0,1))*N2inv

        #Initialize matrices
        NB = np.zeros((N,N,M,Nb),dtype=int) #Number of neighbors for each point
        NC = np.zeros((N,N,M,Nb),dtype=int) #Number of neighbors who are Cs
        S2 = np.zeros((N+2,N+2,M,Nb),dtype=int) #S + border of zeros
        F = np.zeros((N,N,M,Nb)) #Fitness matrix
        F2 = np.zeros((N+2,N+2,M,Nb)) #Fitness matrix + border of zeros
        A = np.ones((N,N,M,Nb)) #Fitness parameters, each of N^2 elements is 1 or b
        P = np.zeros((N,N,M,Nb)) #Probability matrix
        Pden = np.zeros((N,N,M,Nb))
        #---------------------

        #Calculate number of neighbors for each point
        NB[:,:,:,:] = 8
        NB[0,1:-1,:,:],NB[-1,1:-1,:,:],NB[1:-1,0,:,:],NB[1:-1,-1,:,:] = 5,5,5,5
        NB[0,0,:,:],NB[-1,-1,:,:],NB[0,-1,:,:],NB[-1,0,:,:] = 3,3,3,3
        NBinv = 1.0/NB
        #-------------

        #----Time marching-----
        for t in range(Nt):
            R = np.random.rand(N,N,M,Nb) #Random numbers used to update S every time step

            #Set up coefficients for fitness calculation
            A = np.ones((N,N,M,Nb))
            ind0 = np.where(S==0)
            A[ind0] = bv[ind0[3]]

            #Add boundary of zeros to S
            S2[1:-1,1:-1,:,:] = S

            #Count number of C neighbors for each point
            NC = S2[:-2,:-2,:,:]+S2[:-2,1:-1,:,:]+S2[:-2,2:,:,:]+S2[1:-1,:-2,:,:] + S2[1:-1,2:,:,:] + S2[

            #Calculate fitness matrix
            F = NC*A
            F[ind0] = F[ind0] + (NB[ind0]-NC[ind0])*e
            F = F*NBinv

            #Calculate probability matrix
            F2[1:-1,1:-1,:,:]=F
            F2S2 = F2*S2
            P = F2S2[:-2,:-2,:,:]+F2S2[:-2,1:-1,:,:]+F2S2[:-2,2:,:,:]+F2S2[1:-1,:-2,:,:] + F2S2[1:-1,1:-1
            Pden = F2[:-2,:-2,:,:]+F2[:-2,1:-1,:,:]+F2[:-2,2:,:,:]+F2[1:-1,:-2,:,:] + F2[1:-1,1:-1,:,:] +
            P = P/Pden

            #Set new affiliations based on probability matrix and random numbers stored in R
            #Some/many students will have used np.random.random_choice instead
            S[:,:,:,:] = 0
            S[R<=P] = 1

            fc[t+1,:,:] = S.sum(axis=(0,1))*N2inv
            #plot_S(S)
            #----Finish time marching-----

        return S,fc



    def analyze(N,Nt,bvalues,e,M):
        """ For each of the bvalues, M simulations are run and figures are
        constructed using the computed fc arrays
        """

        S,fc = simulate2(N,Nt,bvalues,e,M)
        fm = fc.mean(axis=1)
```

```
    plt.figure()
    plt.plot(fc[:,::15,4],'--')
    plt.plot(fm[:,4],'k-',linewidth=2)
    plt.xlim([0,150])
    plt.xlabel('t')
    plt.ylabel('$f_c$')
    plt.title('Sample simulations (dashed) and mean (solid black) with b=%f' %(bvalues[4]))

    plt.figure()
    plt.plot(fm)
    plt.xlabel('t')
    plt.ylabel('$<f_c>$')
    plt.title('Average of fc over %d simulations, %3.2f<=b<=%3.2f \n N=%d, e=%d' %(M,np.min(bvalues),

    plt.figure()
    plt.plot(bvalues,fm[-1,:])
    plt.xlabel('b')
    plt.ylabel('$<f_c>$')
    plt.title('Average of fc(t=%d) over %d simulations \n N=%d, e=%d' %(Nt,M,N,e))

    return None

if __name__ == '__main__':
    #The code here should call analyze and generate the
    #figures that you are submitting with your code
    bvalues = np.linspace(1,1.5,11); bvalues[0]=1.01
    N,Nt,e,M = 21,800,0.01,100
    output = analyze(N,Nt,bvalues,e,M)
```