PROJECT 2: DESERTIFICATION CONTROL MODELING AND SIMULATION

April 29, 2020

Team 36
Ruixuan Zhang, rzhang438@gatech.edu
Haowen Xu, hxu383@gatech.edu
Repo: https://github.gatech.edu/hxu383/desertification_control_modsim.git

Part1_CA

April 28, 2020

1 Part 1: A Cellular Automata Model

We are using Cellular Automata to simulate our model tp demonstrate the process of desertification control by growing certain desert plant.

1.1 The phenomenon to be modeled and simulated

Suppose we are implementing desertification control by growing two types of desert plant in arid amd semi-arid area. Let's define three types of soil indicating three different degrees of desertification in desert area, that is, N(nondesertification), M(medium desertification), S(serious desertification). We are growing two types of plant. One is Salix Mongolica(Plant 1) and the other one is Haloxylon Ammondendron(Plant 2). They are different in survival rate in different land, as well as the power for improving and maintaining the quality of soil. Specifically, Plant 1 is more suitable on N and M land and have higher probability to help improve the quality of desertified soil. Plant 2 has higher survival rate on M and S land compared to Plant 1, but has lower probability to improve quality of soil. In this case, we will try either Salix Mongolica or Haloxylon Ammondedron in different type of desertified lands, and find out the best solution for desertification control. Under the assumption, we are trying to figure out the best plant for land in different degree of desertification, and calculate the speed of afforestation under certain survival rate and recovery/reversal rate.

1.2 Conceptual model

Let's try using a cellular automaton as the conceptual model. Let's say the world is an $n \times n$ grid $G = G(t) \equiv (g_{ij}(t))$ of cells. The cells vary over discrete time interval of months or years. Every cell of G is a fraction of land with a type of three types of soil that is either N or M or S, and is either empty or occupied by a plant, therefore, a cell will have 6 states as follows: 1. NE: Non-desertified, empty land 2. ME: Medium-desertified, empty land 3. SE: Serious-desertified, empty land 4. NV: Non-desertified land with vegetation 5. MV: Medium-desertified land with vegetation 6. SV: Serious-desertified land with vegetation

```
In [16]: # Possible states:
         EDGE = 0
         SE = 1
         ME = 2
         NE = 3
         SV = 4
```

```
MV = 5
NV = 6
```

Based on our assumption, we'll have these effects from soil or plants to the degree of desertification.

1.2.1 Effects between soil and plant:

When time evolves, the cell will be in one of these states. And they are transitng with the effects below.

Soil to plant: 1. Plant 1 has survival rate r_1. 2. Plant 2 has survival rate r_2.

Plant to soil: 1. When the plant survives, it will improve the quality of soil in conversion rate $c_1 \neq c_2$ 3. If the plant dies, the cell wll become empty again.

Soil to soil: 1. The quality of soil can be improved if over 3 of its neighbors are higher quality with prob p_b and it's not empty. 4. The land might have more severe desetification(lower quality) if it remains empty and its neighbors are all empty, the probability is p_w.

Now let's set a initial version of parameters for our model.

```
In [17]: # Model Parameters:
         N = 10 # number of cells(the min space that can be occupied by a desert plant), repre
         SURV_RATE = 0.8 # survival rate of plant, say Plant 2
         PROB_WORSE = 0.5 # fixed probability of soil's worsening effect
         PROB_BETTER = 0.3 # fixed probability of soil's neighbor-improving effect
         RATE_CONV = 0.4 # conversion/recovery rate of plant, say Plant 2
```

Let's use a 2-D Numpy array to store and implement the grid defined above.

```
In [18]: import matplotlib.pyplot as plt
```

```
In [19]: import numpy as np
         import scipy as sp
         import scipy.sparse

         def count (G):
             """
             Counts the number of locations in a Numpy array, `G`
             """
             return len(np.where(G)[0])

         def find (G):
             """
             Returns the set of locations of a NumPy array, `G`
             """
             assert type(G) is np.ndarray
             return {(i,j) for i,j in zip(*np.where(G))}

         def not_array(G):
             """
             Returns 0 if the element is 1, 1 if the element is 0
```

```
    """
    [rows, cols] = G.shape
    for i in range(1, rows-1):
        for j in range(1, cols-1):
            G[i,j] = 1 - G[i,j]
    return G.astype(int)
```

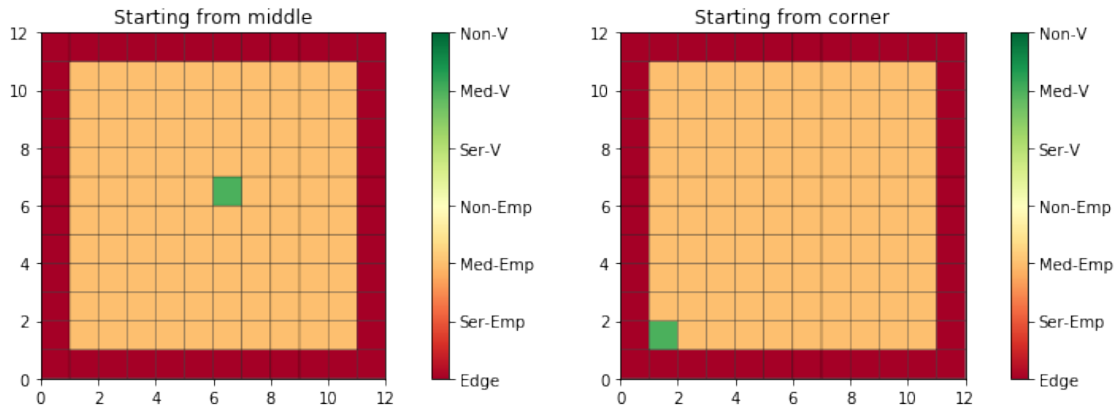
To show the process of simulation, consider a simple case. On t=0, the land is initially empty for all cells with different portion of N, M and S. Initially, let's say the land is a med-desertified area, and we have the first plant grown in the middle of the area. The initial state can be shown in the plot. We also tried another type of initialization, planting from one of the corner of the area.

```
In [73]: from random import seed
         from random import random

         def random_choice(seq, prob, k=1):
             """
             with given prob, choose elements randomly from seq, can be done k times
             prob and seq must be in same size, sum(prob)=1
             default k=1
             return list
                 >>> random_choice(['a','b','c','d'], [0.4, 0.15, 0.1, 0.35])
                 ['d']
                 >>> random_choice('abcd', [0.4, 0.15, 0.1, 0.35], k=5)
                 ['d','d','b','a','d']
             """
             sum_prob = 0
             for i in range(len(prob)):
                 sum_prob += prob[i]
             assert sum_prob == 1

             res = []
             for j in range(k):
                 p = random()
                 for i in range(len(seq)):
                     if sum(prob[:i]) < p <= sum(prob[:i+1]):
                         res.append(seq[i])
             return res


         def create_world (n, emp, init):
             """
             Mid-point intialization
             """
             assert SE<=emp<=NE
             assert SV<=init<=NV
             G = np.zeros((n+2,n+2), dtype=int)
             G[1:-1, 1:-1] = emp
```

```
        i_mid = int ((n+2)/2)
        G[i_mid, i_mid] = init

        return G

    def create_world_side(n, emp, init):

        assert SE<=emp<=NE
        assert SV<=init<=NV
        G = np.zeros((n+2,n+2), dtype=int)
        G[1:-1, 1:-1] = emp
        i_start = 1
        G[i_start, i_start] = init

        return G
```

In [74]:
```
    def show_world(G, vmin=EDGE, vmax=NV, values="states"):
        """
        Show the grid of CA simulation
        """
        assert values in ["states", "bool"]
        if values == "states":
            vticks = range(vmin, vmax+1)
            vlabels = ['Edge', 'Ser-Emp', 'Med-Emp', 'Non-Emp', 'Ser-V', 'Med-V', 'Non-V']
        else:
            vticks = [0,1]
            vlabels = ['False(0)', 'True(1)']

        m,n = G.shape[0]-2, G.shape[1]-2
        plt.pcolor(G, vmin=vmin, vmax=vmax, edgecolor='black')
        plt.set_cmap('RdYlGn')
        cb = plt.colorbar()
        cb.set_ticks (vticks)
        cb.set_ticklabels (vlabels)
        plt.axis('square')
        plt.axis([0, m+2, 0, n+2])


    N=10
    G_0 = create_world(N, ME, MV)
    G_01 = create_world_side(N, ME, MV)

    fig = plt.figure (figsize=(12, 4))
    plt.subplot (1, 2, 1)
    show_world (G_0)
    plt.title ('Starting from middle');

    plt.subplot (1, 2, 2)
```

```
show_world (G_01)
plt.title ('Starting from corner');
```

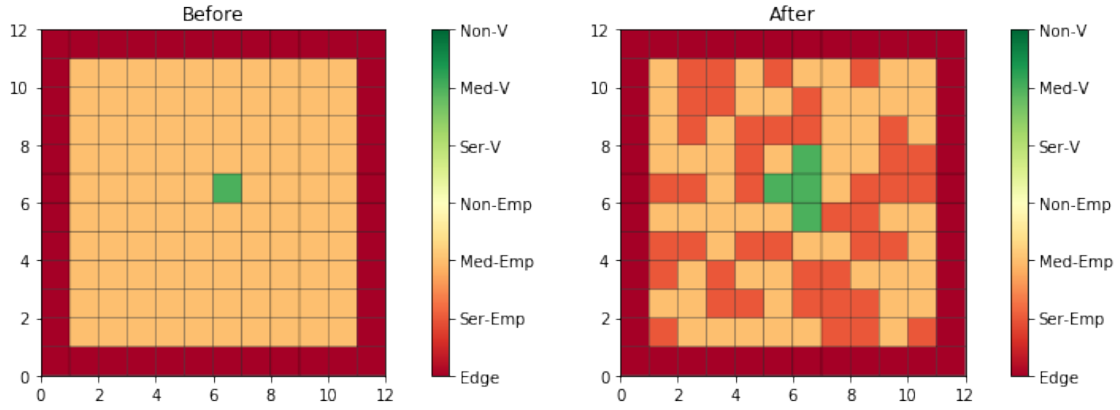


In [22]:
```
def non_emp (G):
    """
    Count the number of Nondesertified empty cells
    """
    return (G == NE).astype (int)

def med_emp (G):
    """
    Count the number of Med-desertified empty cells
    """
    return (G == ME).astype (int)

def ser_emp (G):
    """
    Count the number of Ser-desertified empty cells
    """
    return (G == SE).astype (int)

def non_v (G):
    """
    Count the number of Nondesertified cells covered with vegetation
    """
    return (G == NV).astype (int)

def med_v (G):
    """
    Count the number of Med-desertified cells covered with vegetation
    """
    return (G == MV).astype (int)
```

```
        def ser_v (G):
            """
            Count the number of Ser-desertified cells covered with vegetation
            """
            return (G == SV).astype (int)

In [23]: def exposed_to_greening (G):
            """
            Determines the cells for implantation of next step
            """
            E = np.zeros (G.shape, dtype=int)
            I = ser_v(G)|med_v(G)|non_v(G)
            E[1:-1, 1:-1] = I[0:-2, 1:-1] | I[1:-1, 2:] | I[2:, 1:-1] | I[1:-1, 0:-2]
            return E


        def exposed_to_better_neighbor(G):
            """
            Count the number of cells with neighbors who have less desertification level
            """
            E = np.zeros (G.shape, dtype=int)
            [rows, cols] = G.shape
            for i in range(1,rows-1):
                for j in range(1,cols-1):
                    if G[i,j] > NE:
                        num_better = int(G[i-1,j]>G[i,j]) + int(G[i+1,j]>G[i,j]) + int(G[i,j-
                        if (num_better >= 3):
                            E[i,j] = 1
            return E

        def exposed_to_desert(G):
            """
            Count the number of cells that is empty and have empty neighbors
            """
            E = np.zeros (G.shape, dtype=int)
            [rows, cols] = G.shape
            for i in range(1,rows-1):
                for j in range(1,cols-1):
                    if G[i,j] <= NE:
                        num_empty = int(G[i-1,j]<=NE) + int(G[i+1,j]<=NE) + int(G[i,j-1]<=NE)
                        if (num_empty >= 4):
                            E[i,j] = 1
            return E

In [24]: def survival (G, r = SURV_RATE):
            """
            Determines if the plant can survive with survival rate r
            """
```

```
    I = np.zeros (G.shape, dtype=int)
    I = ser_emp(G) | med_emp(G) | non_emp(G)
    random_draw = np.random.uniform (size=G.shape)
    G_s = (I) * exposed_to_greening (G) * (random_draw < r)
    return G_s.astype (int)


def desertification(G, pw = PROB_WORSE):
    """
    Determines the number of more serious desertified cells
    """
    E = np.zeros (G.shape, dtype=int)
    E = med_emp(G)|ser_emp(G)
    random_draw = np.random.uniform(size=G.shape)
    G_s = (E) * exposed_to_desert(G) * (random_draw < pw)
    return G_s.astype (int)

def improved_neffect(G, pb = PROB_BETTER):
    """
    Determines the number of cells that will get better by neighbor-improving effect
    """
    S = np.zeros (G.shape, dtype=int)
    S = med_v(G)|ser_v(G)
    random_draw = np.random.uniform(size=G.shape)
    G_s = (S) * exposed_to_better_neighbor(G) * (random_draw < pb)
    return G_s.astype (int)



def improved_peffect(G, pv = RATE_CONV):
    """
    Determines the number of cells that will get better because of vegetation's recov
    """
    S = np.zeros (G.shape, dtype=int)
    S = med_v(G)|ser_v(G)
    random_draw = np.random.uniform(size=G.shape)
    G_s = (S) * (random_draw < pv)
    return G_s.astype (int)


def sim_onestep(G, r = SURV_RATE, pw = PROB_WORSE, pb = PROB_BETTER, pv = RATE_CONV):
    # if the planted tree survives: plant trees around the cell
    # sim one step with desert worsening effect, neighbor-improving effect and vegeta

    [rows,cols] = G.shape
    G = G - desertification(G,pw) + improved_neffect(G,pb) + improved_peffect(G,pv)
    P = survival(G, r)
    for i in range(1,rows-1):
```

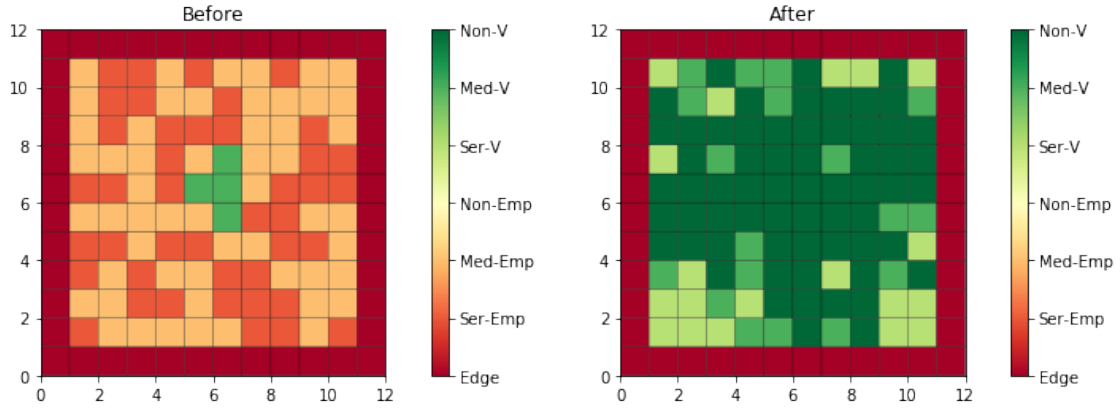
```
            for j in range(1,cols-1):
                if P[i,j] == 1:
                    if G[i,j] == NE:
                        G[i,j] = NV
                    elif G[i,j] == ME:
                        G[i,j] = MV
                    elif G[i,j] == SE:
                        G[i,j] = SV

        for i in range(1,rows-1):
            for j in range(1,cols-1):
                if G[i,j] == EDGE:
                    G[i,j] = SE
                elif G[i,j] > NV:
                    G[i,j] = NV
        return G

    np.random.seed (1602034230) # Fixed seed, for debugging
    G_1 = sim_onestep (G_0)

    fig = plt.figure (figsize=(12, 4))
    plt.subplot (1, 2, 1)
    show_world (G_0)
    plt.title ('Before');

    plt.subplot (1, 2, 2)
    show_world (G_1)
    plt.title ('After');
```



"Before" is the initial state of our sim world, while "After" shows the state after one-step simulation. To test the simulation, we run 10 steps below and the "After" plot shows the state of 10 steps forward in the simulation. Basicly, our simulation will stop when the whole area is non-desertified and covered with vegetations. Then we can compute an average time of stop to view

the efficiency of the desertification control. At each step, we plant trees around the existing trees, for the surviving plant, there's probablity that the quality of soil will be improved by the vegetation itself or the vegetation of neighborhood, for those bald area, there's also probability that the desertification will be expanded. The greener the cell is, the smaller degree of desertification.

```
In [25]: G_new = sim_onestep (G_0)
         for i in range (1, 10):
             G_new = sim_onestep (G_new)

         fig = plt.figure (figsize=(12, 4))
         plt.subplot (1, 2, 1)
         show_world (G_1)
         plt.title ('Before');

         plt.subplot (1, 2, 2)
         show_world (G_new)
         plt.title ('After');
```



1.3 Simulation

Next, after the setup of model, we'll simulates to compare the two plants which are different on the survival rate and conversion rate. For example, we'll use the same setting as above, simulating on the Medium-desertified area with different survival rate and conversion rate. The parameters and the state at each step can be modified and observed as below.

```
In [63]: def summarize (G_t, verbose=True):
             """
             Summarizes the number of each state
             """
             n_NV = count (non_v (G_t))
             n_MV = count (med_v (G_t))
             n_SV = count (ser_v (G_t))
             n_NE = count (non_emp (G_t))
```

```
    n_ME = count (med_emp (G_t))
    n_SE = count (ser_emp (G_t))

    if verbose:
        print ("# Non-desertified with vegetation covered:", n_NV)
        print ("# Med-desertified with vegetation covered:", n_MV)
        print ("# Serious-desertified with vegetation covered:", n_SV)
        print ("# Non-desertified empty land:", n_NE)
        print ("# Med-desertified empty land:", n_ME)
        print ("# Serious-desertified empty land:", n_SE)

    return n_NV, n_MV, n_SV, n_NE, n_ME, n_SE

def sim (G_0, max_steps, r = SURV_RATE, pw = PROB_WORSE, pb = PROB_BETTER, pv = RATE_
    """
    Starting from a given initial state, `G_0`, this
    function simulates up to `max_steps` time steps of
    the cellular automaton.
    """
    t, G_t = 0, G_0.copy ()
    [m,n] = G_t.shape[0]-2, G_t.shape[1]-2
    (num_NV, _, _, _, _, _) = summarize (G_t, verbose=False)
    while (num_NV < m*n) and (t < max_steps):
        t = t + 1
        G_t = sim_onestep (G_t, r)
        (num_NV, _, _, _, _, _) = summarize (G_t, verbose=False)
    return (t, G_t)


from ipywidgets import interact

def isim (n, max_steps=0, r = SURV_RATE, pw = PROB_WORSE, pb = PROB_BETTER, pv = RATE_
    np.random.seed (seed)
    G_0 = np.zeros((n+2,n+2), dtype=int)
    G_0[1:-1, 1:-1] = ME
    i_mid = int ((n+2)/2)
    j_mid = int ((n+2)/2)
    G_0[i_mid, j_mid] = MV
    (_, G_t) = sim (G_0, max_steps, r, pw, pb, pv)
    show_world (G_t)


interact (isim
          , n=(1, 50, 1)
          , max_steps=(0, 100, 1)
          , r=(0.0, 1.0, 0.1)
          , pw=(0.0, 1.0, 0.1)
          , pb=(0.0, 1.0, 0.1)
```

```
                  , pv=(0.0, 1.0, 0.1)
                  , seed=(0, 50, 1)
                 );
```

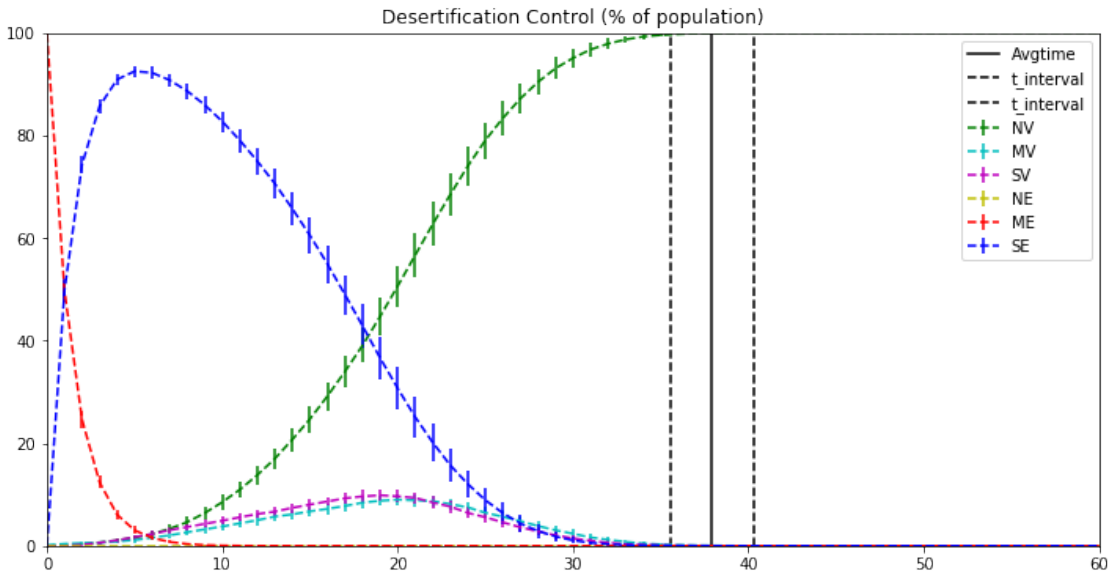
```
interactive(children=(IntSlider(value=25, description='n', max=50, min=1), IntSlider(value=0, d
```

In [64]:
```
# === Simulation parameters ===

N = 25 # World is N x N
SURV_RATE = 0.5
PROB_WORSE = 0.5
PROB_BETTER = 0.3
RATE_CONV = 0.6
MAX_STEPS = 60
NUM_SIMS = 100

# === Holds simulation results ===
nNV = np.zeros ((MAX_STEPS+1, NUM_SIMS))
nMV = np.zeros ((MAX_STEPS+1, NUM_SIMS))
nSV = np.zeros ((MAX_STEPS+1, NUM_SIMS))
nNE = np.zeros ((MAX_STEPS+1, NUM_SIMS))
nME = np.zeros ((MAX_STEPS+1, NUM_SIMS))
nSE = np.zeros ((MAX_STEPS+1, NUM_SIMS))
T_stop = np.zeros (NUM_SIMS)

# === Define initial condition ===
G_0 = create_world(N, ME, MV)


# === Simulation ===

def sim_once (G_0, r = SURV_RATE, pw = PROB_WORSE, pb = PROB_BETTER, pv = RATE_CONV, t
    nNV = np.zeros (t_max+1)
    nMV = np.zeros (t_max+1)
    nSV = np.zeros (t_max+1)
    nNE = np.zeros (t_max+1)
    nME = np.zeros (t_max+1)
    nSE = np.zeros (t_max+1)

    nNV[0], nMV[0], nSV[0], nNE[0], nME[0], nSE[0] = summarize (G_0, verbose=False)
    t, G_t = 0, G_0.copy ()
    [m, n] = G_0.shape[0]-2, G_0.shape[1]-2
    while (nNV[t] < m*n) and (t < t_max):
        t, G_t = t+1, sim_onestep (G_t, r, pw, pb, pv)
        nNV[t], nMV[t], nSV[t], nNE[t], nME[t], nSE[t] = summarize (G_t, verbose=Fals

    if t < MAX_STEPS: # Fill in steady-state values, if any
```

```
                nNV[t+1:] = nNV[t]
                nMV[t+1:] = nSV[t]
                nSV[t+1:] = nSV[t]
                nNE[t+1:] = nNE[t]
                nME[t+1:] = nSE[t]
                nSE[t+1:] = nSE[t]

            return t, nNV, nMV, nSV, nNE, nME, nSE

        for k in range (NUM_SIMS):
            T_stop[k], nNV[:, k], nMV[:, k], nSV[:, k], nNE[:, k], nME[:, k], nSE[:, k] = sim
```

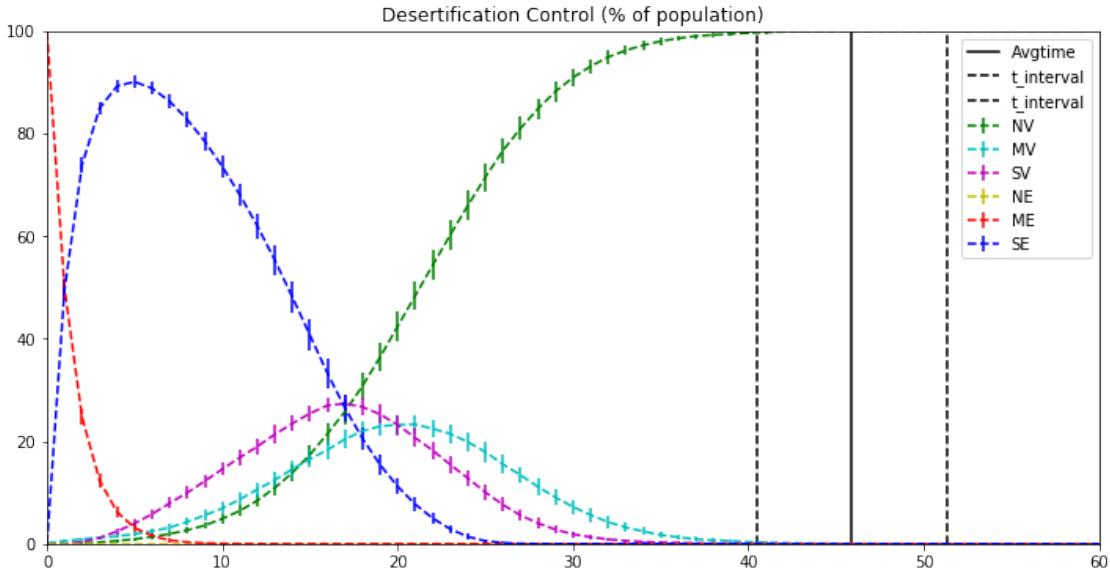
In [65]:
```
        # === Visualize the sim results ===

        # Computes the averages and stds
        NV_avg = np.mean (nNV, axis=1)
        MV_avg = np.mean (nMV, axis=1)
        SV_avg = np.mean (nSV, axis=1)
        NE_avg = np.mean (nNE, axis=1)
        ME_avg = np.mean (nME, axis=1)
        SE_avg = np.mean (nSE, axis=1)
        t_stop_avg = np.mean (T_stop)

        NV_std = np.std (nNV, axis=1)
        MV_std = np.std (nMV, axis=1)
        SV_std = np.std (nSV, axis=1)
        NE_std = np.std (nNE, axis=1)
        ME_std = np.std (nME, axis=1)
        SE_std = np.std (nSE, axis=1)
        t_stop_std = np.std (T_stop)

        T = np.arange (MAX_STEPS+1)
        fig = plt.figure (figsize=(12, 6))

        SCALE = 1e2 / (N**2)
        plt.errorbar (T, NV_avg*SCALE, yerr=NV_std*SCALE, fmt='g--')
        plt.errorbar (T, MV_avg*SCALE, yerr=MV_std*SCALE, fmt='c--')
        plt.errorbar (T, SV_avg*SCALE, yerr=SV_std*SCALE, fmt='m--')
        plt.errorbar (T, NE_avg*SCALE, yerr=NE_std*SCALE, fmt='y--')
        plt.errorbar (T, ME_avg*SCALE, yerr=ME_std*SCALE, fmt='r--')
        plt.errorbar (T, SE_avg*SCALE, yerr=SE_std*SCALE, fmt='b--')
        plt.plot ([t_stop_avg, t_stop_avg], [0., 100.], 'k-')
        plt.plot ([t_stop_avg-t_stop_std, t_stop_avg-t_stop_std], [0., 100.], 'k--')
        plt.plot ([t_stop_avg+t_stop_std, t_stop_avg+t_stop_std], [0., 100.], 'k--')
        plt.axis ([0, MAX_STEPS, 0.0, 100.0])
        plt.legend (['Avgtime' ,'t_interval', 't_interval', 'NV', 'MV', 'SV', 'NE', 'ME', 'SE
        plt.title ("Desertification Control (% of population)")
```

```
    # Sanity check
    assert (np.abs ((NV_avg + MV_avg + SV_avg + NE_avg + ME_avg + SE_avg)/(N**2) - 1.0) <=
```



In [66]:
```
# === Simulation parameters ===

N = 25 # World is N x N
SURV_RATE = 0.7
PROB_WORSE = 0.5
PROB_BETTER = 0.3
RATE_CONV = 0.2
MAX_STEPS = 60
NUM_SIMS = 100

# === Holds simulation results ===
nNV = np.zeros ((MAX_STEPS+1, NUM_SIMS))
nMV = np.zeros ((MAX_STEPS+1, NUM_SIMS))
nSV = np.zeros ((MAX_STEPS+1, NUM_SIMS))
nNE = np.zeros ((MAX_STEPS+1, NUM_SIMS))
nME = np.zeros ((MAX_STEPS+1, NUM_SIMS))
nSE = np.zeros ((MAX_STEPS+1, NUM_SIMS))
T_stop = np.zeros (NUM_SIMS)

# === Define initial condition ===
G_0 = create_world(N, ME, MV)


# === Simulation ===
```

```
for k in range (NUM_SIMS):
    T_stop[k], nNV[:, k], nMV[:, k], nSV[:, k], nNE[:, k], nME[:, k], nSE[:, k] = sim

# === Visualize the sim results ===

# Computes the averages and stds
NV_avg = np.mean (nNV, axis=1)
MV_avg = np.mean (nMV, axis=1)
SV_avg = np.mean (nSV, axis=1)
NE_avg = np.mean (nNE, axis=1)
ME_avg = np.mean (nME, axis=1)
SE_avg = np.mean (nSE, axis=1)
t_stop_avg = np.mean (T_stop)

NV_std = np.std (nNV, axis=1)
MV_std = np.std (nMV, axis=1)
SV_std = np.std (nSV, axis=1)
NE_std = np.std (nNE, axis=1)
ME_std = np.std (nME, axis=1)
SE_std = np.std (nSE, axis=1)
t_stop_std = np.std (T_stop)

T = np.arange (MAX_STEPS+1)
fig = plt.figure (figsize=(12, 6))

SCALE = 1e2 / (N**2)
plt.errorbar (T, NV_avg*SCALE, yerr=NV_std*SCALE, fmt='g--')
plt.errorbar (T, MV_avg*SCALE, yerr=MV_std*SCALE, fmt='c--')
plt.errorbar (T, SV_avg*SCALE, yerr=SV_std*SCALE, fmt='m--')
plt.errorbar (T, NE_avg*SCALE, yerr=NE_std*SCALE, fmt='y--')
plt.errorbar (T, ME_avg*SCALE, yerr=ME_std*SCALE, fmt='r--')
plt.errorbar (T, SE_avg*SCALE, yerr=SE_std*SCALE, fmt='b--')
plt.plot ([t_stop_avg, t_stop_avg], [0., 100.], 'k-')
plt.plot ([t_stop_avg-t_stop_std, t_stop_avg-t_stop_std], [0., 100.], 'k--')
plt.plot ([t_stop_avg+t_stop_std, t_stop_avg+t_stop_std], [0., 100.], 'k--')
plt.axis ([0, MAX_STEPS, 0.0, 100.0])
plt.legend (['Avgtime' ,'t_interval', 't_interval','NV', 'MV', 'SV', 'NE', 'ME', 'SE'
plt.title ("Desertification Control (% of population)")

# Sanity check
assert (np.abs ((NV_avg + MV_avg + SV_avg + NE_avg + ME_avg + SE_avg)/(N**2) - 1.0) <=
```

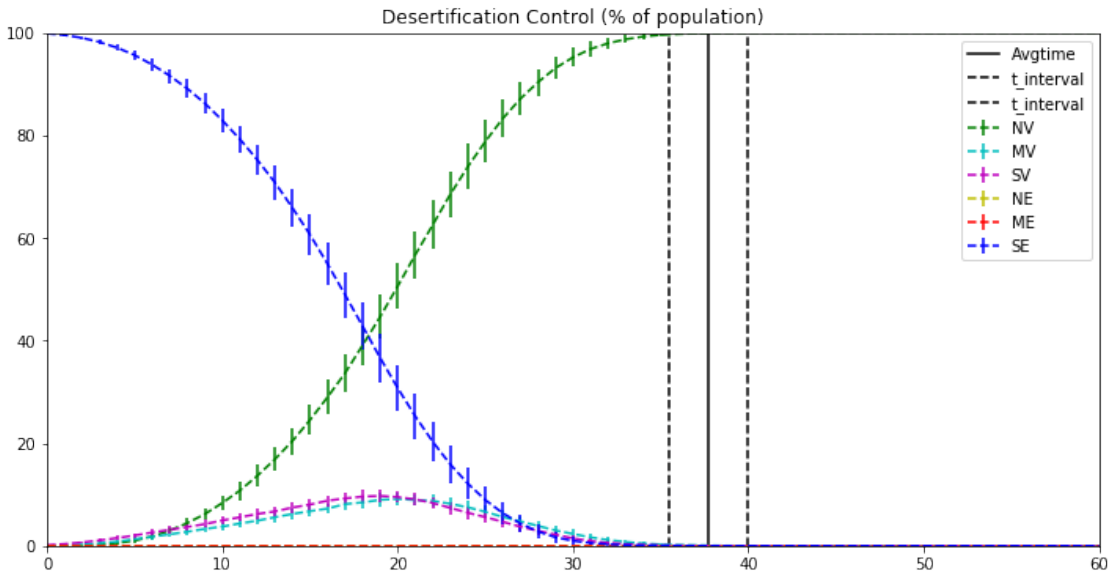


From the plot we can tell that based on our setting, there is a huge growing for serious-desertified empty area because of at the very beginning the area is exposed to empty and easily eroded circumstances, the desertification effect is strong. But once we begin the greening, the degree of desertification is soon suppressed to a lower level. Finally, the overall land will converge to a non-desertified area with vegetation. Comparing the simulation over Plant 1 and Plant 2 with preset parameters(survival rate of Plant 1 = 0.5 while the rate of Plant 2 = 0.7, the conversion rate of Plant 1 and Plant 2 is 0.6 and 0.2 respectively, and we fixed the p_w and p_b to be 0.5 and 0.3), we found that Plant 1 with higher conversion rate convergence faster (lower Avgtime), that is, the one with higher conversion rate and lower survival rate afforests the land quicker. Then in the following steps, we'll compare their effect on serious-desertified land with med-desertified land above.

In [67]:
```
# === Simulation parameters ===

N = 25 # World is N x N
SURV_RATE = 0.5
PROB_WORSE = 0.5
PROB_BETTER = 0.3
RATE_CONV = 0.6
MAX_STEPS = 60
NUM_SIMS = 100

# === Holds simulation results ===
nNV = np.zeros ((MAX_STEPS+1, NUM_SIMS))
nMV = np.zeros ((MAX_STEPS+1, NUM_SIMS))
nSV = np.zeros ((MAX_STEPS+1, NUM_SIMS))
nNE = np.zeros ((MAX_STEPS+1, NUM_SIMS))
nME = np.zeros ((MAX_STEPS+1, NUM_SIMS))
nSE = np.zeros ((MAX_STEPS+1, NUM_SIMS))
```

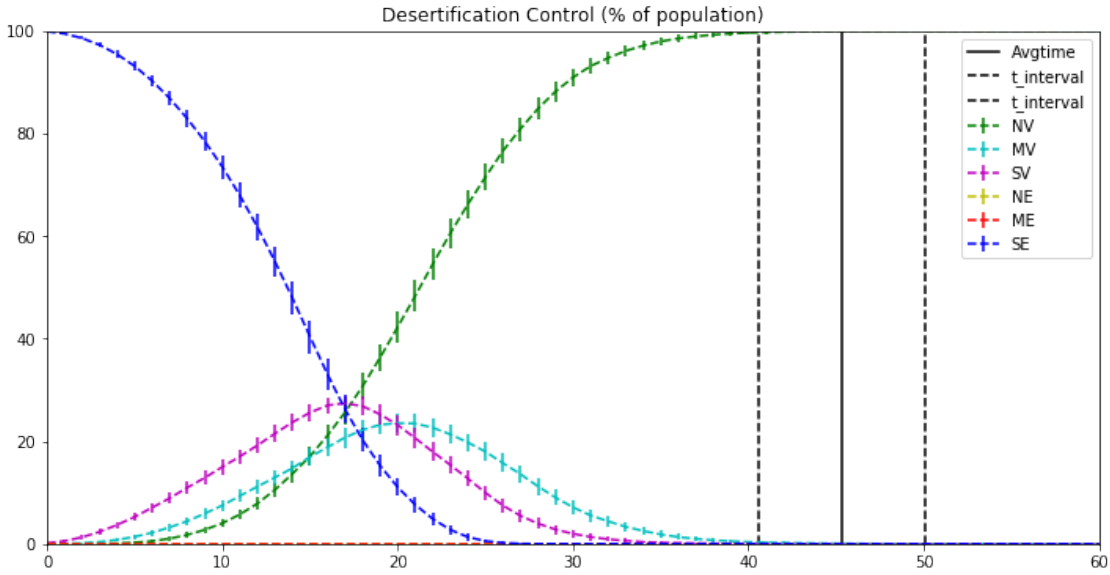
```
T_stop = np.zeros (NUM_SIMS)

# === Define initial condition ===
G_0 = create_world(N, SE, SV)


# === Simulation ===

for k in range (NUM_SIMS):
    T_stop[k], nNV[:, k], nMV[:, k], nSV[:, k], nNE[:, k], nME[:, k], nSE[:, k] = sim


# === Visualize the sim results ===

# Computes the averages and stds
NV_avg = np.mean (nNV, axis=1)
MV_avg = np.mean (nMV, axis=1)
SV_avg = np.mean (nSV, axis=1)
NE_avg = np.mean (nNE, axis=1)
ME_avg = np.mean (nME, axis=1)
SE_avg = np.mean (nSE, axis=1)
t_stop_avg = np.mean (T_stop)

NV_std = np.std (nNV, axis=1)
MV_std = np.std (nMV, axis=1)
SV_std = np.std (nSV, axis=1)
NE_std = np.std (nNE, axis=1)
ME_std = np.std (nME, axis=1)
SE_std = np.std (nSE, axis=1)
t_stop_std = np.std (T_stop)

T = np.arange (MAX_STEPS+1)
fig = plt.figure (figsize=(12, 6))

SCALE = 1e2 / (N**2)
plt.errorbar (T, NV_avg*SCALE, yerr=NV_std*SCALE, fmt='g--')
plt.errorbar (T, MV_avg*SCALE, yerr=MV_std*SCALE, fmt='c--')
plt.errorbar (T, SV_avg*SCALE, yerr=SV_std*SCALE, fmt='m--')
plt.errorbar (T, NE_avg*SCALE, yerr=NE_std*SCALE, fmt='y--')
plt.errorbar (T, ME_avg*SCALE, yerr=ME_std*SCALE, fmt='r--')
plt.errorbar (T, SE_avg*SCALE, yerr=SE_std*SCALE, fmt='b--')
plt.plot ([t_stop_avg, t_stop_avg], [0., 100.], 'k-')
plt.plot ([t_stop_avg-t_stop_std, t_stop_avg-t_stop_std], [0., 100.], 'k--')
plt.plot ([t_stop_avg+t_stop_std, t_stop_avg+t_stop_std], [0., 100.], 'k--')
plt.axis ([0, MAX_STEPS, 0.0, 100.0])
plt.legend (['Avgtime' ,'t_interval', 't_interval','NV', 'MV', 'SV', 'NE', 'ME', 'SE']
plt.title ("Desertification Control (% of population)")
```

```
    # Sanity check
    assert (np.abs ((NV_avg + MV_avg + SV_avg + NE_avg + ME_avg + SE_avg)/(N**2) - 1.0) <=
```



In [68]:
```
# === Simulation parameters ===

N = 25 # World is N x N
SURV_RATE = 0.7
PROB_WORSE = 0.5
PROB_BETTER = 0.3
RATE_CONV = 0.2
MAX_STEPS = 60
NUM_SIMS = 100

# === Holds simulation results ===
nNV = np.zeros ((MAX_STEPS+1, NUM_SIMS))
nMV = np.zeros ((MAX_STEPS+1, NUM_SIMS))
nSV = np.zeros ((MAX_STEPS+1, NUM_SIMS))
nNE = np.zeros ((MAX_STEPS+1, NUM_SIMS))
nME = np.zeros ((MAX_STEPS+1, NUM_SIMS))
nSE = np.zeros ((MAX_STEPS+1, NUM_SIMS))
T_stop = np.zeros (NUM_SIMS)

# === Define initial condition ===
G_0 = create_world(N, SE, SV)


# === Simulation ===
```

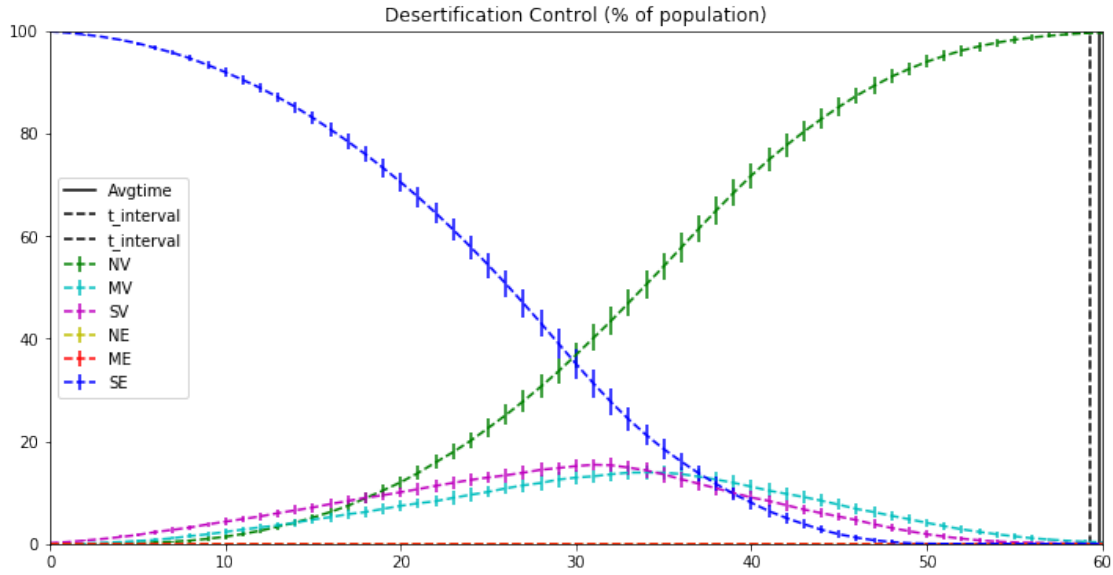
```
for k in range (NUM_SIMS):
    T_stop[k], nNV[:, k], nMV[:, k], nSV[:, k], nNE[:, k], nME[:, k], nSE[:, k] = sim

# === Visualize the sim results ===

# Computes the averages and stds
NV_avg = np.mean (nNV, axis=1)
MV_avg = np.mean (nMV, axis=1)
SV_avg = np.mean (nSV, axis=1)
NE_avg = np.mean (nNE, axis=1)
ME_avg = np.mean (nME, axis=1)
SE_avg = np.mean (nSE, axis=1)
t_stop_avg = np.mean (T_stop)

NV_std = np.std (nNV, axis=1)
MV_std = np.std (nMV, axis=1)
SV_std = np.std (nSV, axis=1)
NE_std = np.std (nNE, axis=1)
ME_std = np.std (nME, axis=1)
SE_std = np.std (nSE, axis=1)
t_stop_std = np.std (T_stop)

T = np.arange (MAX_STEPS+1)
fig = plt.figure (figsize=(12, 6))

SCALE = 1e2 / (N**2)
plt.errorbar (T, NV_avg*SCALE, yerr=NV_std*SCALE, fmt='g--')
plt.errorbar (T, MV_avg*SCALE, yerr=MV_std*SCALE, fmt='c--')
plt.errorbar (T, SV_avg*SCALE, yerr=SV_std*SCALE, fmt='m--')
plt.errorbar (T, NE_avg*SCALE, yerr=NE_std*SCALE, fmt='y--')
plt.errorbar (T, ME_avg*SCALE, yerr=ME_std*SCALE, fmt='r--')
plt.errorbar (T, SE_avg*SCALE, yerr=SE_std*SCALE, fmt='b--')
plt.plot ([t_stop_avg, t_stop_avg], [0., 100.], 'k-')
plt.plot ([t_stop_avg-t_stop_std, t_stop_avg-t_stop_std], [0., 100.], 'k--')
plt.plot ([t_stop_avg+t_stop_std, t_stop_avg+t_stop_std], [0., 100.], 'k--')
plt.axis ([0, MAX_STEPS, 0.0, 100.0])
plt.legend (['Avgtime' ,'t_interval', 't_interval','NV', 'MV', 'SV', 'NE', 'ME', 'SE'
plt.title ("Desertification Control (% of population)")

# Sanity check
assert (np.abs ((NV_avg + MV_avg + SV_avg + NE_avg + ME_avg + SE_avg)/(N**2) - 1.0) <=
```



After comparing on med-desertified and serious-desertified land, we did not found much difference. Let's look at the different strategy on planting.

In [70]:
```
# === Simulation parameters ===

N = 25 # World is N x N
SURV_RATE = 0.7
PROB_WORSE = 0.5
PROB_BETTER = 0.3
RATE_CONV = 0.2
MAX_STEPS = 60
NUM_SIMS = 100

# === Holds simulation results ===
nNV = np.zeros ((MAX_STEPS+1, NUM_SIMS))
nMV = np.zeros ((MAX_STEPS+1, NUM_SIMS))
nSV = np.zeros ((MAX_STEPS+1, NUM_SIMS))
nNE = np.zeros ((MAX_STEPS+1, NUM_SIMS))
nME = np.zeros ((MAX_STEPS+1, NUM_SIMS))
nSE = np.zeros ((MAX_STEPS+1, NUM_SIMS))
T_stop = np.zeros (NUM_SIMS)

# === Define initial condition ===
G_0 = create_world_side(N, SE, SV)


# === Simulation ===

for k in range (NUM_SIMS):
```

```
    T_stop[k], nNV[:, k], nMV[:, k], nSV[:, k], nNE[:, k], nME[:, k], nSE[:, k] = sim_


# === Visualize the sim results ===

# Computes the averages and stds
NV_avg = np.mean (nNV, axis=1)
MV_avg = np.mean (nMV, axis=1)
SV_avg = np.mean (nSV, axis=1)
NE_avg = np.mean (nNE, axis=1)
ME_avg = np.mean (nME, axis=1)
SE_avg = np.mean (nSE, axis=1)
t_stop_avg = np.mean (T_stop)

NV_std = np.std (nNV, axis=1)
MV_std = np.std (nMV, axis=1)
SV_std = np.std (nSV, axis=1)
NE_std = np.std (nNE, axis=1)
ME_std = np.std (nME, axis=1)
SE_std = np.std (nSE, axis=1)
t_stop_std = np.std (T_stop)

T = np.arange (MAX_STEPS+1)
fig = plt.figure (figsize=(12, 6))

SCALE = 1e2 / (N**2)
plt.errorbar (T, NV_avg*SCALE, yerr=NV_std*SCALE, fmt='g--')
plt.errorbar (T, MV_avg*SCALE, yerr=MV_std*SCALE, fmt='c--')
plt.errorbar (T, SV_avg*SCALE, yerr=SV_std*SCALE, fmt='m--')
plt.errorbar (T, NE_avg*SCALE, yerr=NE_std*SCALE, fmt='y--')
plt.errorbar (T, ME_avg*SCALE, yerr=ME_std*SCALE, fmt='r--')
plt.errorbar (T, SE_avg*SCALE, yerr=SE_std*SCALE, fmt='b--')
plt.plot ([t_stop_avg, t_stop_avg], [0., 100.], 'k-')
plt.plot ([t_stop_avg-t_stop_std, t_stop_avg-t_stop_std], [0., 100.], 'k--')
plt.plot ([t_stop_avg+t_stop_std, t_stop_avg+t_stop_std], [0., 100.], 'k--')
plt.axis ([0, MAX_STEPS, 0.0, 100.0])
plt.legend (['Avgtime' ,'t_interval', 't_interval','NV', 'MV', 'SV', 'NE', 'ME', 'SE']
plt.title ("Desertification Control (% of population)")

# Sanity check
assert (np.abs ((NV_avg + MV_avg + SV_avg + NE_avg + ME_avg + SE_avg)/(N**2) - 1.0) <=
```



Using the parameters for Plant 2 and serious-desertification assumption, it's obvious that planting trees starting from the middle of the land is faster from starting from the corner. We can play around with this model with more variation such as the p_b and p_w can be changed due to the characteristics and location of the land, or imports more effects and settings to simulates the real world desertification control.

Part2_Mean_field

April 28, 2020

1 Part 2: Mean-field models

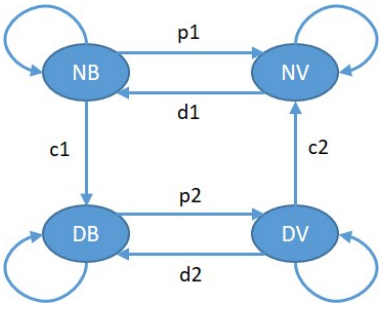
The CA model will become infeasible when the number of state variables becomes large (too many cells). Here, we derive a mean-field approximation to model desertification control process. In this model, we will ignore neighborhood influence and assume that cells are fully-connected. Then we only need to analysis the change of fraction of the population.

We further simplify the conditions of land into two: N represents non-desertification. D represents desertification. We also simplify the type of trees into one. So a piece of land could be either B (bald and no trees), or V (vegatated). Overall, our mean-field model has four states and we define four time-dependent variable for each state.

- NB_t be the fraction of land that is bald but not desertified at time t;
- NV_t be the fraction of land that is covered with vegetation and not desertified at time t;
- DB_t be the fraction of land that is bald and desertified at time t;
- DV_t be the fraction of land that is covered with vegetation and desertified at time t,

where $NB_t + NV_t + DB_t + DV_t = 1$.

The transition between states can be represented in the figure below



where

- c_1 is the conversion rate of non-desertification to desertification when the land is bald.
- c_2 is the conversion rate of desertification to non-desertification when the land is cover with vegetation.
- d_1 is the dying rate of trees on non_desertified land.
- d_2 is the dying rate of trees on desertified land.
- p_1 is the maximum fraction of non-desertified land that we plant trees at each time step.
- p_2 is the maximum fraction of desertified land that we plant trees at each time step.
- $\hat{p_1}$ is the real fraction of non-desertified land that we plant trees at each time step. $\hat{p_1} = \min\{p_1, NB_t(1 - c_1)\}$.

- $\hat{p_2}$ is the real fraction of desertified land that we plant trees at each time step. $\hat{p_2} = \min\{p_2, DB_t\}$.

Then, our discrete-time dynamical system is:

$$NB_{t+1} = NB_t * (1 - c_1) + NV_t * d_1 - \min\{p_1, NB_t(1 - c_1)\} \quad (1)$$
$$NV_{t+1} = NV_t * (1 - d_1) + DV_t * c_2 + \min\{p_1, NB_t(1 - c_1)\} \quad (2)$$
$$DB_{t+1} = DB_t + NB_t * c_1 + DV_t * d_2 - \min\{p_2, DB_t\} \quad (3)$$
$$DV_{t+1} = DV_t * (1 - c_2 - d_2) + \min\{p_2, DB_t\} \quad (4)$$

1.1 Implementing the mean-field model

Let's first build a forward step function to calculate the state variables of the next time step.

```
In [22]: import numpy as np
         import matplotlib.pyplot as plt

In [23]: def forward_step(x, c1, c2, d1, d2, p1, p2):
             x_next = x.copy()
             NB = 0
             NV = 1
             DB = 2
             DV = 3
             x_next[NB] = x[NB] * (1 - c1) + x[NV] * d1 - min(p1, x[NB] * (1 - c1))
             x_next[NV] = x[NV] * (1 - d1) + x[DV] * c2 + min(p1, x[NB] * (1 - c1))
             x_next[DB] = x[DB] + x[NB] * c1 + x[DV] * d2- min(p2, x[DB])
             x_next[DV] = x[DV] * (1 - c2 - d2) + min(p2, x[DB])
             return x_next
```

Next we build a function that simulates the system for t_{max} time steps. Assume $DB = 1$ as the initial state.

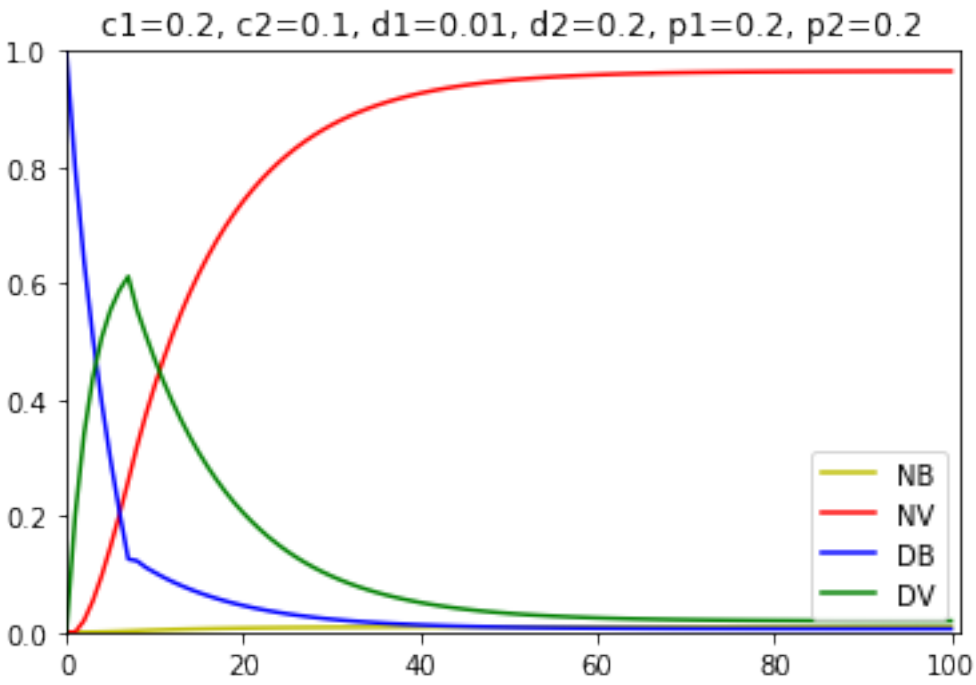
```
In [24]: def sim(t_max, c1, c2, d1, d2, p1, p2):
             X = np.zeros((4, t_max+1))
             X[:, 0] = np.array([0, 0, 1, 0])
             for t in range(t_max):
                 X[:, t+1] = forward_step(X[:, t], c1, c2, d1, d2, p1, p2)
             return X

         def summarize_sim(X, c1, c2, d1, d2, p1, p2):
             t_max = X.shape[1] - 1
             print('Simulation results: t_max = {}, c1={}, c2={}, d1={}, d2={}, p1={}, p2={}'.
             NB = X[0, -1]
             NV = X[1, -1]
             DB = X[2, -1]
             DV = X[3, -1]
             print("- NB_{{{}}} = {:.3f}".format(t_max, NB))
```

```
        print("- NV_{{{}}} = {:.3f}".format(t_max, NV))
        print("- DB_{{{}}} = {:.3f}".format(t_max, DB))
        print("- DV_{{{}}} = {:.3f}".format(t_max, DV))

    def plot_sim(X, c1, c2, d1, d2, p1, p2):
        t_max = X.shape[1] - 1
        T = np.arange(t_max + 1)
        plt.plot(T, X[0, :], 'y-')
        plt.plot(T, X[1, :], 'r-')
        plt.plot(T, X[2, :], 'b-')
        plt.plot(T, X[3, :], 'g-')
        plt.legend(['NB', 'NV', 'DB', 'DV'])
        plt.axis([0, t_max+1, 0, 1])
        plt.title('c1={}, c2={}, d1={}, d2={}, p1={}, p2={}'.format(c1, c2, d1, d2, p1, p2
```

Now we set the parameters as $c_1 = 0.1$, $c_2 = 0.1$, $d_1 = 0.01$, $d_2 = 0.2$, $p_1 = 0.2$, $p_2 = 0.2$ and run the simulation to see the result.

```
In [25]: t_max = 100
         c1 = 0.2
         c2 = 0.1
         d1 = 0.01
         d2 = 0.2
         p1 = 0.2
         p2 = 0.2
         X = sim(t_max, c1, c2, d1, d2, p1, p2)
         summarize_sim(X, c1, c2, d1, d2, p1, p2)
         plot_sim(X, c1, c2, d1, d2, p1, p2)
```

```
Simulation results: t_max = 100, c1=0.2, c2=0.1, d1=0.01, d2=0.2, p1=0.2, p2=0.2
- NB_{100} = 0.010
- NV_{100} = 0.965
- DB_{100} = 0.006
- DV_{100} = 0.019
```



In the simulation experiment above, the desert is converted into green land in about 50 time steps.

If we

1.2 Finding a good strategy that minimizes the cost

After implementing our simulation model, we want to find a good strategy of planting trees that minimizes the cost to convert a bald desert land into green land. Here, strategies are represented by two values: p_1 and p_2. Cost is the total number of trees died during a given length of time steps. A successful desertification control means the fraction of non-desertified land covered with vegetation (NV) becomes larger than 0.9 after a given time.

Let's assume the number of time steps we are given to control the desertification of a piece of land is 50, i.e. $t_{max} = 50$. In order to find the best strategy, we will apply grid search over p_1 and p_2 to find the best pair of values that minimizes the number of trees died while satisfies $NV \geq 0.9$ at time step 50.

First, we need to modify our 'forward_step' function to return the number of trees die during a time step.

```
In [26]: def forward_step_new(x, c1, c2, d1, d2, p1, p2):
             x_next = x.copy()
             NB = 0
             NV = 1
             DB = 2
             DV = 3
             x_next[NB] = x[NB] * (1 - c1) + x[NV] * d1 - min(p1, x[NB] * (1 - c1))
             x_next[NV] = x[NV] * (1 - d1) + x[DV] * c2 + min(p1, x[NB] * (1 - c1))
```

```
    x_next[DB] = x[DB] + x[NB] * c1 + x[DV] * d2- min(p2, x[DB])
    x_next[DV] = x[DV] * (1 - c2 - d2) + min(p2, x[DB])
    tree_died = x[NV] * d1 + x[DV] * d2
    return x_next, tree_died
```

Second, we need to modify our 'sim' function to return the total number of trees die during t_{max} time steps.

```
In [27]: def sim_new(t_max, c1, c2, d1, d2, p1, p2):
             X = np.zeros((4, t_max+1))
             X[:, 0] = np.array([0, 0, 1, 0])
             total_tree_died = 0
             for t in range(t_max):
                 X[:, t+1], tree_died = forward_step_new(X[:, t], c1, c2, d1, d2, p1, p2)
                 total_tree_died += tree_died
             return X, total_tree_died

         def summarize_sim_new(X, c1, c2, d1, d2, p1, p2):
             t_max = X.shape[1] - 1
             print('Simulation results: t_max = {}, c1={}, c2={}, d1={}, d2={}, p1={}, p2={}'.
             NB = X[0, -1]
             NV = X[1, -1]
             DB = X[2, -1]
             DV = X[3, -1]
             print("- NB_{{{}}} = {:.3f}".format(t_max, NB))
             print("- NV_{{{}}} = {:.3f}".format(t_max, NV))
             print("- DB_{{{}}} = {:.3f}".format(t_max, DB))
             print("- DV_{{{}}} = {:.3f}".format(t_max, DV))

         def plot_sim_new(X, c1, c2, d1, d2, p1, p2):
             t_max = X.shape[1] - 1
             T = np.arange(t_max + 1)
             plt.plot(T, X[0, :], 'y-')
             plt.plot(T, X[1, :], 'r-')
             plt.plot(T, X[2, :], 'b-')
             plt.plot(T, X[3, :], 'g-')
             plt.legend(['NB', 'NV', 'DB', 'DV'])
             plt.axis([0, t_max+1, 0, 1])
             plt.title('c1={}, c2={}, d1={}, d2={}, p1={}, p2={}'.format(c1, c2, d1, d2, p1, p
```

Next, we need to write a script to perform grid search over p_1 and p_2. We will keep the values of c_1, c_2, d_1, d_2 the same as our first experiment.

```
In [28]: t_max = 50
         c1 = 0.2
         c2 = 0.1
         d1 = 0.01
         d2 = 0.2
         p1_lst = [0.001, 0.002, 0.004, 0.008, 0.016, 0.032, 0.064]
```

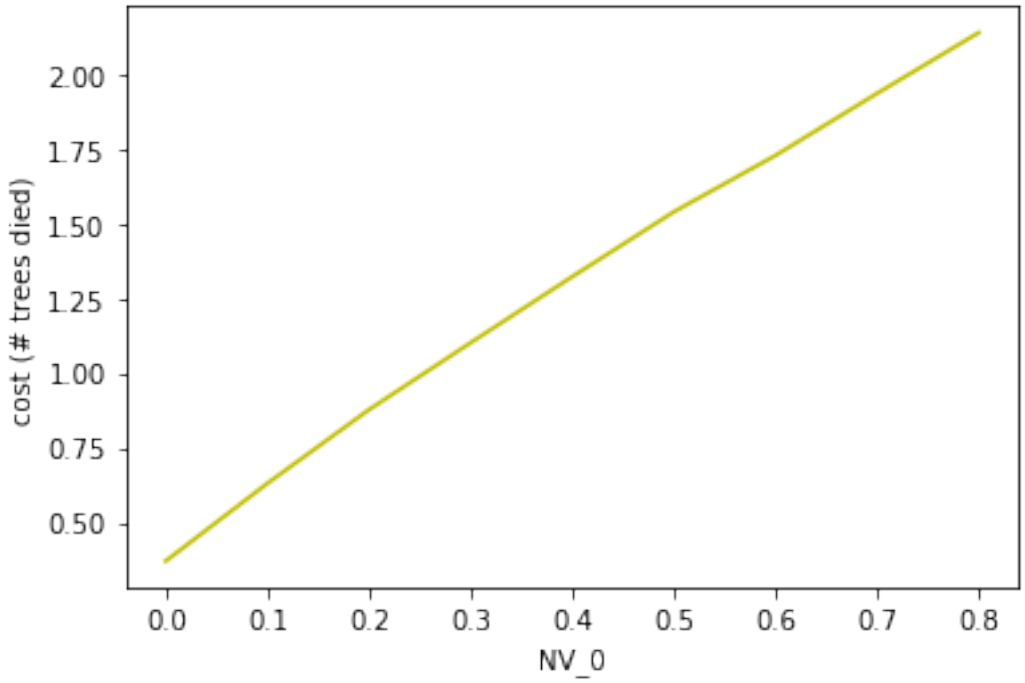
```
        p2_lst = [0.01, 0.02, 0.04, 0.08, 0.16, 0.32, 0.64]
        NV_matrix = np.zeros((len(p1_lst), len(p2_lst)))
        total_tree_died_matrix = np.zeros((len(p1_lst), len(p2_lst)))
        best_cost = 1000
        for i in range(len(p1_lst)):
            for j in range(len(p2_lst)):
                p1 = p1_lst[i]
                p2 = p2_lst[j]
                X, total_tree_died = sim_new(t_max, c1, c2, d1, d2, p1, p2)
                NV_matrix[i, j] = X[1, -1]
                total_tree_died_matrix[i, j] = total_tree_died
                if NV_matrix[i, j] > 0.9 and total_tree_died_matrix[i, j] < best_cost:
                    best_cost = total_tree_died
                    best_p1 = p1
                    best_p2 = p2
        np.set_printoptions(precision=4)
        print('NV_matrix:')
        print(NV_matrix)
        print('\n')
        print('total_tree_died_matrix:')
        print(total_tree_died_matrix)
        print('\n')
        print('best strategy: p1={}, p2={}'.format(best_p1, best_p2))
```

```
NV_matrix:
[[0.1471 0.2795 0.5325 0.837  0.8555 0.8574 0.8579]
 [0.1475 0.2942 0.559  0.8525 0.8706 0.8725 0.8729]
 [0.1475 0.2951 0.5884 0.8816 0.9003 0.9025 0.903 ]
 [0.1475 0.2951 0.5901 0.9177 0.9473 0.9513 0.9522]
 [0.1475 0.2951 0.5901 0.9177 0.9473 0.9513 0.9522]
 [0.1475 0.2951 0.5901 0.9177 0.9473 0.9513 0.9522]
 [0.1475 0.2951 0.5901 0.9177 0.9473 0.9513 0.9522]]


total_tree_died_matrix:
[[0.3455 0.6896 1.375  2.3102 2.5412 2.6054 2.6252]
 [0.3455 0.6909 1.3792 2.2791 2.5012 2.5623 2.5809]
 [0.3455 0.6909 1.3818 2.2331 2.4327 2.4857 2.5018]
 [0.3455 0.6909 1.3818 2.2084 2.3688 2.4092 2.4216]
 [0.3455 0.6909 1.3818 2.2084 2.3688 2.4092 2.4216]
 [0.3455 0.6909 1.3818 2.2084 2.3688 2.4092 2.4216]
 [0.3455 0.6909 1.3818 2.2084 2.3688 2.4092 2.4216]]


best strategy: p1=0.008, p2=0.08
```

1.3 How will the initial state affect the simulation result?

Now we are going to study how will the initial state affect our simulation model. Assume at time t_0, every piece of land could only be in two state: NV or DB (i.e. $NV_0 + DB_0 = 1, NB_0 = DV_0 = 0$). Let's see how many trees will die before $NV > 0.9$, if we start from different initial states.

```
In [29]: t_max = 50
         c1 = 0.2
         c2 = 0.1
         d1 = 0.01
         d2 = 0.2
         p1 = 0.008
         p2 = 0.08
         DB0 = [0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0]
         cost_lst = []
         for i in range(len(DB0)):
             x = np.zeros((4,1))
             x[1] = 1 - DB0[i]
             x[2] = DB0[i]
             total_cost = 0
             for t in range(t_max):
                 x, cost = forward_step_new(x, c1, c2, d1, d2, p1, p2)
                 total_cost += cost
                 if x[1] > 0.9:
                     cost_lst.append(total_cost)
                     break

         plt.plot(NV0, cost_lst, 'y-')
         plt.xlabel('NV_0')
         plt.ylabel('cost (# trees died)')

Out[29]: Text(0, 0.5, 'cost (# trees died)')
```



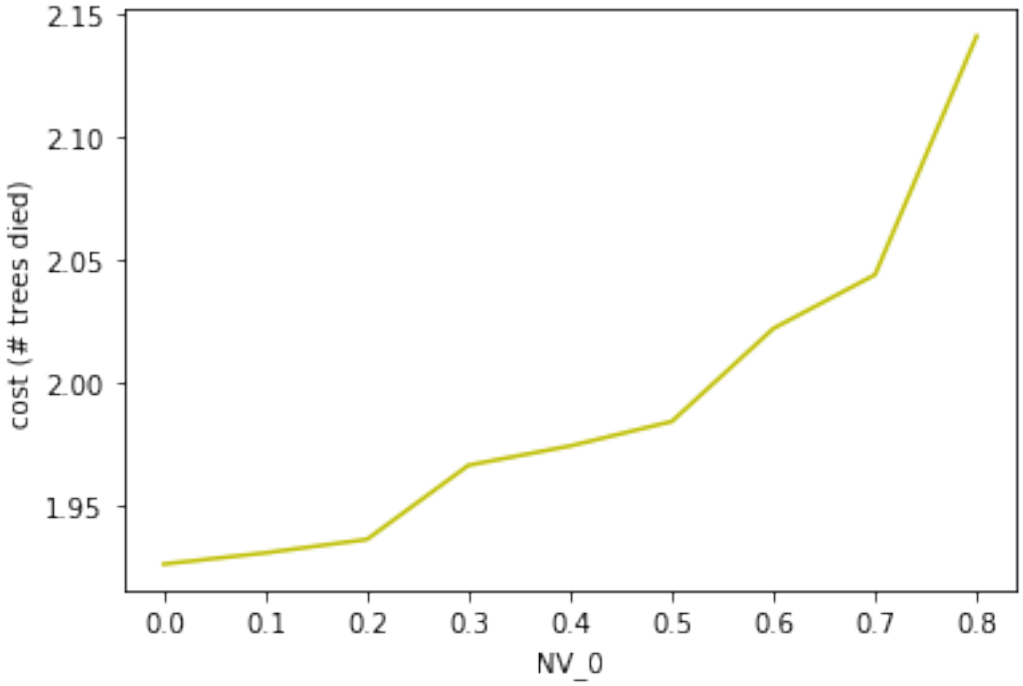
It shows that the cost decreases almost linearily when DB_0 increases. The result superised me a little bit. I have expected a larger curvature than this. It is intuitive to think that the severer the desertification is, the harder it is to control the desertification. However, our models gives an almost linear relationship.

It is interesting to see that if we change the assumption about the initial state to "$NB_0 + DB_0 = 1, NV_0 = DV_0 = 0$, the result is in accord with our intuition. The non-linearity in the show means it is much harder to control the desertification if the desertification is already very severe.

```
In [30]: t_max = 50
         c1 = 0.2
         c2 = 0.1
         d1 = 0.01
         d2 = 0.2
         p1 = 0.008
         p2 = 0.08
         NB0 = [0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0]
         cost_lst = []
         for i in range(len(NB0)):
             x = np.zeros((4,1))
             x[0] = 1 - NB0[i]
             x[2] = NB0[i]
             total_cost = 0
             for t in range(t_max):
                 x, cost = forward_step_new(x, c1, c2, d1, d2, p1, p2)
                 total_cost += cost
                 if x[1] > 0.9:
```

```
                cost_lst.append(total_cost)
                break

        plt.plot(NV0, cost_lst, 'y-')
        plt.xlabel('NV_0')
        plt.ylabel('cost (# trees died)')
```

```
Out[30]: Text(0, 0.5, 'cost (# trees died)')
```



WORK DISTRIBUTION

The project is equally contributed. Part 1 CA is done by Ruixuan Zhang, and Part 2 Mean-field is done by Haowen Xu.