appendixB

June 29, 2015

1 Introduction

Here we define the discrete-time replicator dynamic for signaling games with arbitrarily many states, messages, and actions. The formulation provided here treats individual states as independent sender populations and individual messages as independent receiver populations, instead of considering the set of all potential sender strategies as a population and the set of all potential receiver strategies as a population (cf. Hofbauer and Huttegger 2015).

By "devolving" populations in this manner we dramatically reduce the dimensions of the system, while also yielding results that are arguably more transparent. Importantly, it also allows for a fairly straightforward implementation in algebraic terms which is quick to compute.

1.1 Definitions

We start off by defining the components necessary for our analysis. Once we have defined these components we can simulate the game dynamics.

First, we define the payoff matrices for senders and receivers, which depend solely on the utility functions of senders and receivers respectively.

$\mathbf{A}$ is an $n \times n$ matrix such that $\mathbf{A}_{ij} = U_S(t_i, a_j)$:

$$\mathbf{A} = \begin{pmatrix} U_S(t_1,a_1) & \cdots & U_S(t_1,a_j) & \cdots & U_S(t_1,a_n) \\ \vdots & \ddots & \vdots & & \vdots \\ U_S(t_i,a_1) & \cdots & U_S(t_i,a_j) & \cdots & U_S(t_i,a_n) \\ \vdots & & \vdots & \ddots & \vdots \\ U_S(t_n,a_1) & \cdots & U_S(t_n,a_j) & \cdots & U_S(t_n,a_n) \end{pmatrix} \tag{1}$$

$\mathbf{B}$ is an $n \times n$ matrix such that $\mathbf{B}_{ij} = U_R(t_i, a_j)$:

$$\mathbf{B} = \begin{pmatrix} U_R(t_1,a_1) & \cdots & U_R(t_1,a_j) & \cdots & U_R(t_1,a_n) \\ \vdots & \ddots & \vdots & & \vdots \\ U_R(t_i,a_1) & \cdots & U_R(t_i,a_j) & \cdots & U_R(t_i,a_n) \\ \vdots & & \vdots & \ddots & \vdots \\ U_R(t_n,a_1) & \cdots & U_R(t_n,a_j) & \cdots & U_R(t_n,a_n) \end{pmatrix} \tag{2}$$

In this case we'll use the standard utility function for a 2x2 Lewis Signaling Game. Senders and receivers both prefer that the state and action correspond in some prespecified way. For example, the correspondence is usally taken as a bijection between states and actions.

$$U_S(t_i, a_j) = U_R(t_i, a_j) = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{else} \end{cases}$$

We'll define this as a special case of a modified version of the quadratic loss function used by Crawford and Sobel (1982).

```
In [1]: # Import numpy
        import numpy as np
        # Define the utility functions
        def U_S(state, action, b):
            return 1 - (action - state - (1-state)*b)**2
            #return 1 - abs(action - state - (1-state)*b)
        def U_R(state, action):
            return 1 - (action - state)**2
        # Define functions to map integers to interval [0,1]
        def t(i, n):
            return i/float(n)
        def a(i, n):
            return i/float(n)
        # Create payoff matrices
        print "Sender payoff matrix"
        A =  np.matrix([[U_S(t(i, 2-1), a(j,2-1), 0) for j in range(2)] for i in range(2)])
        print A
        print "Receiver payoff matrix"
        B = np.matrix([[U_R(t(i, 2-1), a(j,2-1)) for j in range(2)] for i in range(2)])
        print B
```

```
Sender payoff matrix
[[ 1.  0.]
 [ 0.  1.]]
Receiver payoff matrix
[[ 1.  0.]
 [ 0.  1.]]
```

Second, we define the sender and receiver populations.

$\mathbf{X}$ is a stochastic population matrix such that the proportion of the population in x_i using m_j is x_{ij}, with $\sum_j x_{ij} = 1$.

$$\mathbf{X} = \begin{pmatrix} x_{11} & x_{12} & x_{13} \\ \vdots & \vdots & \vdots \\ x_{i1} & x_{i2} & x_{i3} \\ \vdots & \vdots & \vdots \\ x_{n1} & x_{n2} & x_{n3} \end{pmatrix} \tag{3}$$

Intuitively, each row corresponds to a given state. Each element in the row corresponds to the proportion of use in that population. Each row sums to one because the proportion using the various signals must sum to one.

$\mathbf{Y}$ is a population matrix such that the proportion of the population in y_i responding with action a_j is y_{ij}, with $\sum_j y_{ij} = 1$.

$$\mathbf{Y} = \begin{pmatrix} y_{11} & \cdots & y_{1j} & \cdots & y_{1n} \\ y_{21} & \cdots & y_{2j} & \cdots & y_{2n} \\ y_{31} & \cdots & y_{3j} & \cdots & y_{3n} \end{pmatrix} \tag{4}$$

Again, intuitively, each row corresponds to a given message. Each element in the row corresponds to the proportion of different responses to the message. Each row sums to one because the proportion using the various responses must sum to one.

```
In [2]: # Import random and set seed
        import random
        random.seed(10)
```

```
        # Create initial sender matrix
        X = np.random.rand(2, 2)
        # Row-normalize the sender matrix
        X /= X.sum(axis=1)[:,np.newaxis]
        print X
        # Create initial receiver matrix
        Y = np.random.rand(2, 2)
        # Row-normalize the receiver matrix
        Y /= Y.sum(axis=1)[:,np.newaxis]
        print Y
```

```
[[ 0.4433709   0.5566291 ]
 [ 0.21282727  0.78717273]]
[[ 0.71254386  0.28745614]
 [ 0.78187104  0.21812896]]
```

$\mathbf{P}$ is a stochastic matrix such that $\forall i \mathbf{P}_i = P(t_1), ..., P(t_n)$. That is, $\mathbf{P}$ is just n rows of the prior probability distribution over states.

$$\mathbf{P} = \begin{pmatrix} P(t_1) & \cdots & P(t_i) & \cdots & P(t_n) \\ \vdots & & \vdots & & \vdots \\ P(t_1) & \cdots & P(t_i) & \cdots & P(t_n) \end{pmatrix} \tag{5}$$

```
In [3]: print "Probability matix"
        P = np.matrix([.5] * 4).reshape(2,2)
        #P = np.matrix([[.75, .25], [.75, .25]])
        print P
```

```
Probability matix
[[ 0.5  0.5]
 [ 0.5  0.5]]
```

For what follows we'll consider the uniform distribution over states, noting that this is a special case. We can generate a beta-binomial distribution over a finite number of states with the following.

```
In [4]: from scipy.special import beta as beta_func
        from scipy.misc import comb
        def beta_binomial(n, alpha, beta):
            return [comb(n-1,k) * beta_func(k+alpha, n-1-k+beta) / beta_func(alpha,beta) for k in range

        print "Probability matix"
        print np.matrix((beta_binomial(2, 1, 1) * 2)).reshape(2,2)
        print "Another probability matix with uneven weights"
        print np.matrix((beta_binomial(2, 2, 1) * 2)).reshape(2,2)
```

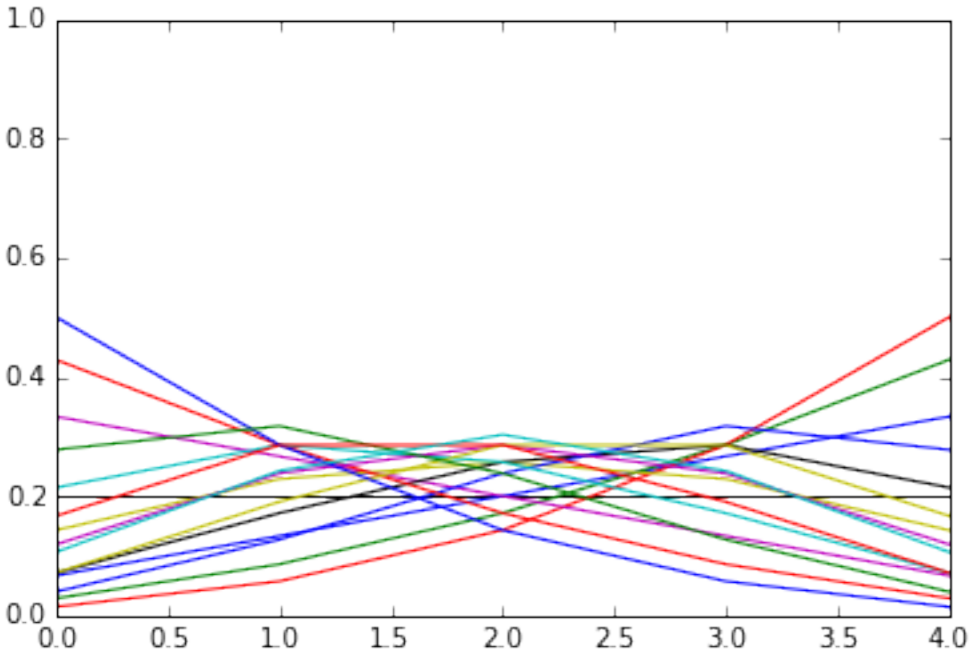
```
Probability matix
[[ 0.5  0.5]
 [ 0.5  0.5]]
Another probability matix with uneven weights
[[ 0.33333333  0.66666667]
 [ 0.33333333  0.66666667]]
```

We can extend this to arbitrarily many states and generate more complex prior distributions. For example, if we have five states, then we can visualize the probabilities for different parametrizations. This includes the uniform distribution as a special case: it's the flat black line at one fifth, which is indeed flat despite the optical illusion.

```
In [5]: % matplotlib inline
        from matplotlib import pyplot as plt
```

```
In [6]: for i in range(5):
            for j in range(5):
                plt.plot(beta_binomial(5, j, i))
        plt.axis((0,4,0,1))
        plt.show()
```

```
/home/cahern-adm/anaconda/lib/python2.7/site-packages/IPython/kernel/__main__.py:4: RuntimeWarning: inval
```



1.2 Senders

Third, now that we have defined these two components, we can define the expected utility of different strategies and the discrete-time replicator dynamic. But first we'll remind ourselves of the initial state of the sender and receiver populations.

```
In [7]: print "Sender matrix"
        print X
        print "Receiver matrix"
        print Y
```

```
Sender matrix
[[ 0.4433709   0.5566291 ]
 [ 0.21282727  0.78717273]]
Receiver matrix
[[ 0.71254386  0.28745614]
 [ 0.78187104  0.21812896]]
```

The expected utility of sending message m_j in state t_i, where $\mathbf{Y}^T$ is the transpose of $\mathbf{Y}$:

$$E[x_{ij}] = (\mathbf{A}\mathbf{Y}^T)_{ij} \tag{6}$$

```
In [8]: print "Sender expected utility"
        print A * Y.transpose()
```

```
Sender expected utility
[[ 0.71254386  0.78187104]
 [ 0.28745614  0.21812896]]
```

Intuitively, we can read these expected utilities off the receiver matrix. In fact, the resulting expected utilities are just $\mathbf{Y}^T$. This, however, is particular to the utility functions of the Lewis signaling game. For any $b > 0$ this does not hold.

The average expected utility in a sender population x_i:

$$E[x_i] = (\mathbf{X}(\mathbf{A}\mathbf{Y}^T)^T)_{ii} \tag{7}$$

```
In [9]: print "Average sender expected utility"
        print (X * (A * Y.transpose()).transpose()).diagonal()
```

```
Average sender expected utility
[[ 0.75113339  0.23288368]]
```

Note that this average makes sense when we look at the expected utilities of sending different messages in the different populations. That is, it is always somewhere in between the two values, which means things are working as they should.

Let $\hat{\mathbf{X}}$ be the sender expected utility matrix normalized by the average expected utilities such that:

$$\hat{\mathbf{X}}_{ij} = \frac{(\mathbf{A}\mathbf{Y}^T)_{ij}}{(\mathbf{X}(\mathbf{A}\mathbf{Y}^T)^T)_{ii}} \tag{8}$$

```
In [10]: print "Discrete-time replicator dynamic scaling factors"
         X_hat = A * Y.transpose() / ((X * (A * Y.transpose()).transpose()).diagonal()).transpose()
         print X_hat
```

```
Discrete-time replicator dynamic scaling factors
[[ 0.94862494  1.04092169]
 [ 1.23433355  0.93664342]]
```

For both sender and receiver populations, under the discrete-time replicator dynamics strategies grow in proportion to the amount by which they exceed the average payoff in the population. The discrete-time replicator dynamic for message m_j in state t_i:

$$x'_{ij} = x_{ij}\frac{E[x_{ij}]}{E[x_i]} \tag{9}$$

The sender populations at the next point in time are then given by the following, where $\otimes$ indicates the element-wise Hadamard product:

$$\mathbf{X}' = \mathbf{X} \otimes \hat{\mathbf{X}} \tag{10}$$

```
In [11]: print "Current sender populations state"
         print X
         print "Next sender populations state"
         X_next = np.multiply(X, X_hat)
         print X_next
         print "Check that sender populations sum to one"
         print np.sum(X_next, axis=1)
```

```
Current sender populations state
[[ 0.4433709   0.5566291 ]
 [ 0.21282727  0.78717273]]
Next sender populations state
[[ 0.42059269  0.57940731]
 [ 0.26269984  0.73730016]]
Check that sender populations sum to one
[[ 1.]
 [ 1.]]
```

Now that we have defined the discrete-time replicator dynamics for the sender populations, we can do the same for the receiver populations with a few additions. Let $\mathbf{C}$ be the conditional probability of a state given a message. That is, $\mathbf{C}_{ij} = P(t_i|m_j)$, where $\otimes$ indicates element-wise Hadamard multiplication and $\oslash$ indicates the element-wise Hadamard division.

$$\mathbf{C} = (P^T \otimes X) \oslash (PX) \tag{11}$$

```
In [12]: print "Sender matrix"
         print X
         print "Conditional probability of t_i given message m_j"
         C = np.divide(np.multiply(P.transpose(), X), P * X)
         print C
```

```
Sender matrix
[[ 0.4433709   0.5566291 ]
 [ 0.21282727  0.78717273]]
Conditional probability of t_i given message m_j
[[ 0.67566616  0.41421963]
 [ 0.32433384  0.58578037]]
```

The expected utility of receiver responding to message m_i with action a_j:

$$E[y_{ij}] = (\mathbf{B}^T\mathbf{C})_{ji} \tag{12}$$

Since the resulting matrix is $n \times m$, we swap the indices to get the appropriate value. Each column corresponds to a receiver population, and each row corresponds to a response action.

```
In [13]: print "Receiver expected utility"
         print (B.transpose() * C)
         print "Receiver expected utility of responding to m_i with a_j"
         print (B.transpose() * C).transpose()
```

```
Receiver expected utility
[[ 0.67566616  0.41421963]
 [ 0.32433384  0.58578037]]
Receiver expected utility of responding to m_i with a_j
[[ 0.67566616  0.32433384]
 [ 0.41421963  0.58578037]]
```

The average expected utility in a receiver population y_i:

$$E[y_i] = (Y(\mathbf{B}^T\mathbf{C})_{ji})_{ii} \tag{13}$$

```
In [14]: print "Average receiver expected utility"
         print (Y * (B.transpose() * C)).diagonal()
```

```
Average receiver expected utility
[[ 0.57467353  0.451642  ]]
```

Let $\hat{\mathbf{Y}}$ be the population normalized matrix for receiver expected utilities such that:

$$\hat{\mathbf{Y}}_{ji} = \frac{(\mathbf{B}^T\mathbf{C})_{ji}}{(\mathbf{Y}(\mathbf{B}^T\mathbf{C})_{ji})_{ii}} \tag{14}$$

```
In [15]: print "Discrete-time replicator dynamic scaling factors"
         Y_hat = (B.transpose() * C) / (Y * (B.transpose() * C)).diagonal()
         print Y_hat.transpose()
```

```
Discrete-time replicator dynamic scaling factors
[[ 1.17573914  0.5643793 ]
 [ 0.91714153  1.29700153]]
```

The discrete-time replicator dynamic for action a_j in response to message m_i:

$$y'_{ij} = y_{ij}\frac{E[y_{ij}]}{E[y_i]} \tag{15}$$

The receiver populations at the next point in time are then given by the following:

$$\mathbf{Y}' = \mathbf{Y} \otimes \hat{\mathbf{Y}}^T \tag{16}$$

```
In [16]: print "Current receiver populations state"
         print Y
         print "Next receiver populations state"
         Y_next = np.multiply(Y, Y_hat.transpose())
         print Y_next
         print "Check that sender populations sum to one"
         print np.sum(Y_next, axis=1)
```

```
Current receiver populations state
[[ 0.71254386  0.28745614]
 [ 0.78187104  0.21812896]]
Next receiver populations state
[[ 0.83776571  0.16223429]
 [ 0.7170864   0.2829136 ]]
Check that sender populations sum to one
[[ 1.]
 [ 1.]]
```

1.3 Simulations

Now that we have defined the components of the discrete-time replicator dynamics and how to calculate the state of the populations at the next point in time, we can calculate the game dynamics from an initial state.

```
In [17]: def discrete_time_replicator_dynamics(n_steps, X, Y, A, B, P):
             """Calculate the discrete-time replicator dynamics for"""
             # Get the number of states, signals, and actions
             X_nrow = X.shape[0]
             X_ncol = X.shape[1]
             Y_nrow = Y.shape[0] # Same as X_ncol
             Y_ncol = Y.shape[1] # Often, but not necessarily, the same as X_nrow
             # Create empty arrays to hold the population states over time
             X_t = np.empty(shape=(n_steps, X_nrow*X_ncol), dtype=float)
```

```
    Y_t = np.empty(shape=(n_steps, X_nrow*X_ncol), dtype=float)
    # Set the initial state
    X_t[0,:] = X.ravel()
    Y_t[0,:] = Y.ravel()
    # Iterate forward over (n-1) steps
    for i in range(1,n_steps):
        # Get the previous state
        X_prev = X_t[i-1,:].reshape(X_nrow, X_ncol)
        Y_prev = Y_t[i-1,:].reshape(Y_nrow, Y_ncol)
        # Calculate the scaling factors
        X_hat = A * Y_prev.transpose() / ((X_prev * (A * Y_prev.transpose()).transpose()).diag
        C = np.divide(np.multiply(P.transpose(), X_prev), P * X_prev)
        Y_hat = (B.transpose() * C) / (Y_prev * (B.transpose() * C)).diagonal()
        #if i == 1:
        #    print X_hat
        #    print Y_hat
        # Calculate next states
        X_t[i,:] = np.multiply(X_prev, X_hat).ravel()
        Y_t[i,:] = np.multiply(Y_prev, Y_hat.transpose()).ravel()
    return X_t, Y_t
```

We can verify that this simulation matches our calculations above for a single step of the game dynamics. They do, which suggests we're doing everything correctly. We can then move on to iterating over a larger number of time steps and plotting the results.

```
In [18]: X_hist, Y_hist = discrete_time_replicator_dynamics(2, X, Y, A, B, P)
         print "Initial sender populations state"
         print X_hist[0].reshape(2,2)
         print "Next sender populations state"
         print X_hist[1].reshape(2,2)
         print "Initial receiver populations state"
         print Y_hist[0].reshape(2,2)
         print "Next receiver populations state"
         print Y_hist[1].reshape(2,2)
```

```
Initial sender populations state
[[ 0.4433709   0.5566291 ]
 [ 0.21282727  0.78717273]]
Next sender populations state
[[ 0.42059269  0.57940731]
 [ 0.26269984  0.73730016]]
Initial receiver populations state
[[ 0.71254386  0.28745614]
 [ 0.78187104  0.21812896]]
Next receiver populations state
[[ 0.83776571  0.16223429]
 [ 0.7170864   0.2829136 ]]
```

We should note that this is pretty quick. It takes about two seconds to step through 10,000 steps of the simplest non-trivial signaling game. This will obviously change as we increase the size of the matrices, but is a good sign that everything is set up well.

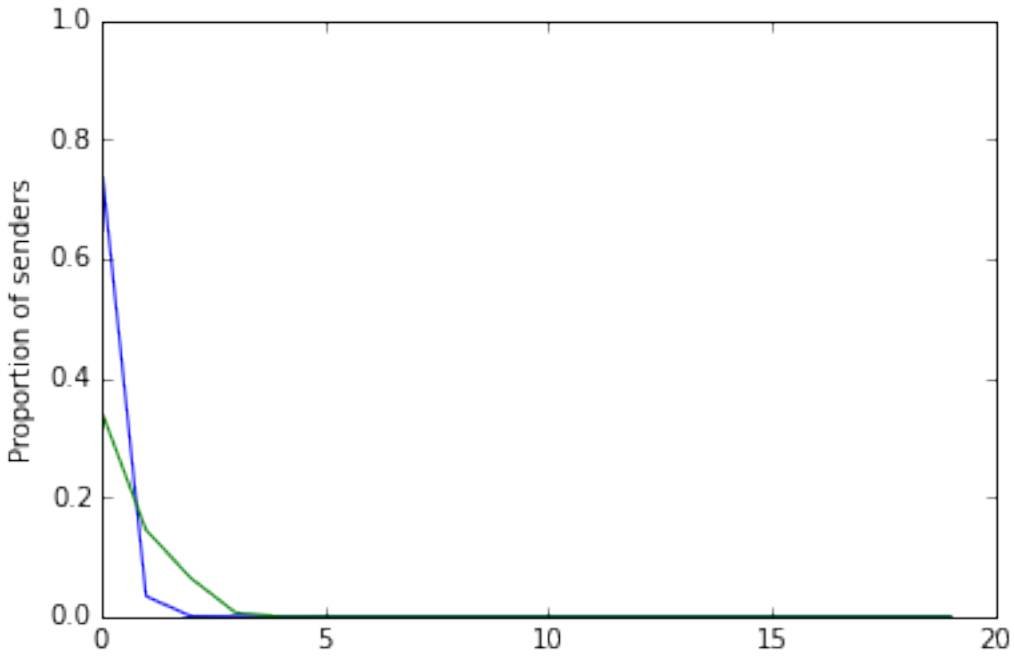
```
In [19]: import timeit
         start = timeit.default_timer()
         X_hist, Y_hist = discrete_time_replicator_dynamics(10000, X, Y, A, B, P)
```

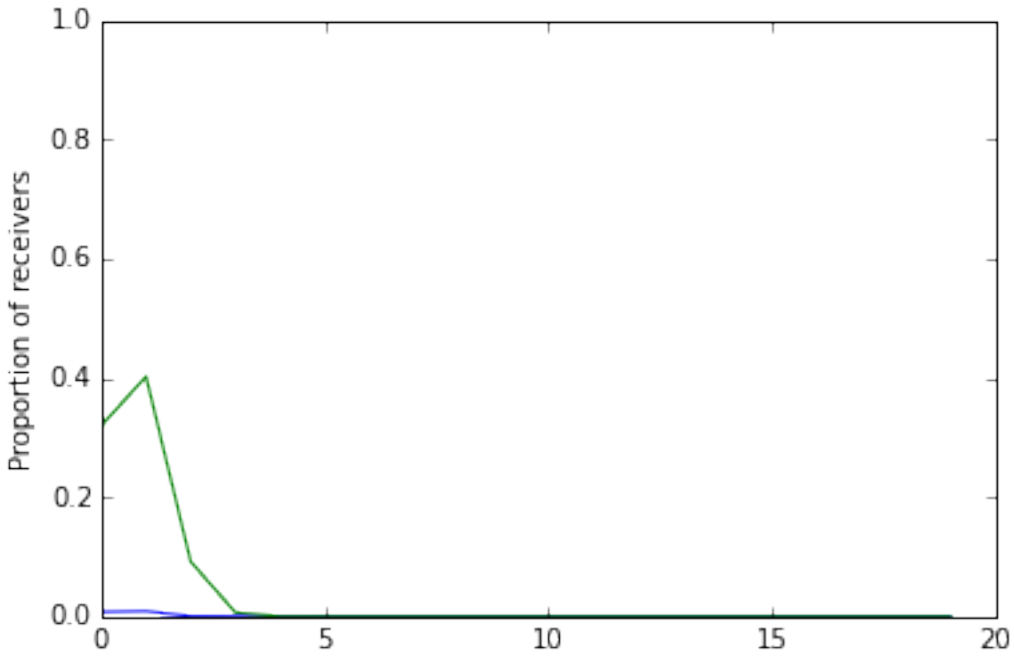
```
stop = timeit.default_timer()
print stop - start
```

```
1.82154607773
```

Now that we've established that the populations are updating as they should, we can move our attention to the long-term behavior of the system. We'll start by running a bit longer and examining the end state. From analytic results, we know that when states are equiprobable that the signaling equilibria are the only asymptotically stable states. This means that the populations should always evolve to one of two signaling systems. We can look at this visually by plotting the proportion in the sender and receiver populations. A signaling system results when all sender and receiver populations converge to one or zero.

In [20]:
```
# Generate random starting states
X = np.random.rand(2, 2)
X /= X.sum(axis=1)[:,np.newaxis]
Y = np.random.rand(2, 2)
Y /= Y.sum(axis=1)[:,np.newaxis]
# Iterate under discrete-time replicator dynamics
X_hist, Y_hist = discrete_time_replicator_dynamics(20, X, Y, A, B, P)
# Sender plots
plt.plot(X_hist[:,0], 'b') # Proportion of t_0 sending m_0
plt.plot(X_hist[:,3], 'g') # Proportion of t_1 sending m_1
plt.ylim(0,1)
plt.ylabel("Proportion of senders")
plt.show()
# Receiver plots
plt.plot(Y_hist[:,0], 'b') # Proportion of t_0 sending m_0
plt.plot(Y_hist[:,3], 'g') # Proportion of t_1 sending m_1
plt.ylim(0,1)
plt.ylabel("Proportion of receivers")
plt.show()
```





With simulations of signaling under the discrete-time replicator dynamics in place, we might consider how they compare to the continuous-time replicator dynamics.

1.4 Comparison with continous-time replicator dynamics

The continuous replicator dynamic for message m_j in state t_i:

$$\dot{x}_{ij} = x_{ij}(E[x_{ij}] - E[x_i]) \tag{17}$$

From above, we know that this can be expressed as:

$$\dot{x}_{ij} = x_{ij}((\mathbf{A}\mathbf{Y}^T)_{ij} - (\mathbf{X}(\mathbf{A}\mathbf{Y}^T)^T)_{ii}) \tag{18}$$

Likewise the replicator dynamic for action a_j in response to message m_i:

$$\dot{y}_{ij} = y_{ij}(E[y_{ij}] - E[y_i]) \tag{19}$$

Or, alternatively:

$$\dot{y}_{ij} = y_{ij}((\mathbf{B}^T\mathbf{C})_{ji} - (\mathbf{Y}(\mathbf{B}^T\mathbf{C}))_{ii}) \tag{20}$$

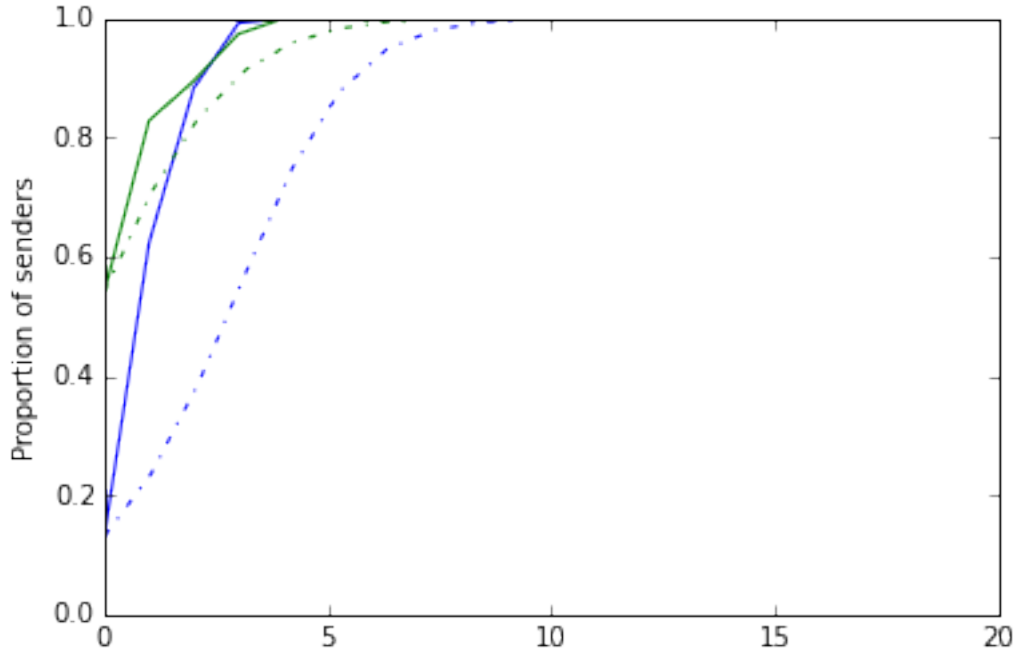
We won't get into the details of translating the matrix notation into a function to integrate.

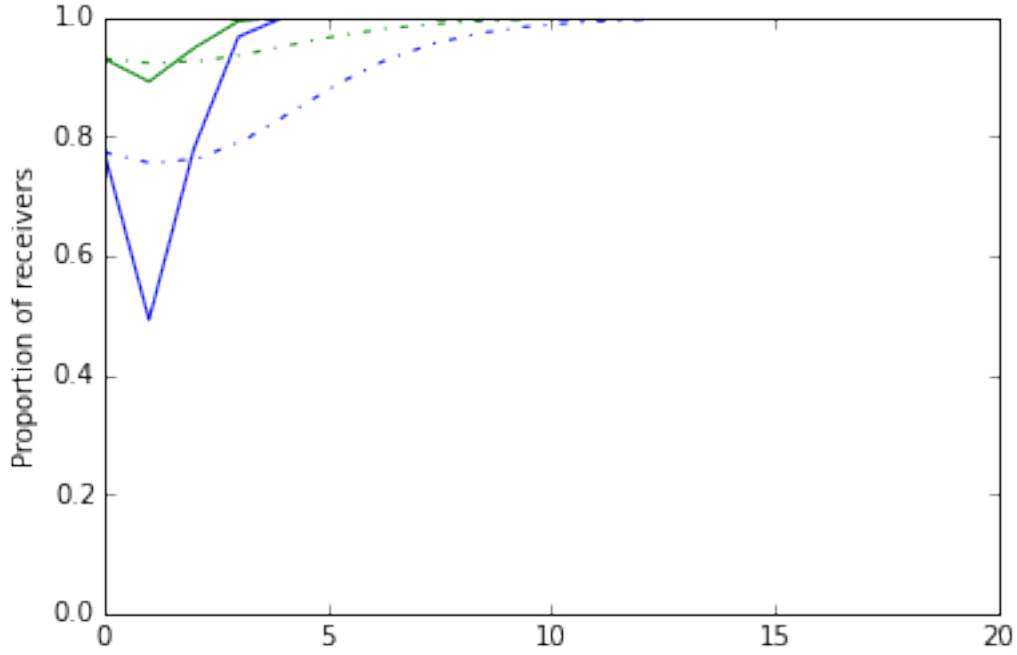
```
In [21]: from scipy.integrate import odeint
         # Signaling system to integrate over
         def signaling(p_vec, t, param):
             # Unpack the position vector
             p0, p1, q0, q1 = p_vec
             # Unpack the parametera
             a = param[0] # probability distribution P(t_0) = P(t_1) = a
             # Construct system of ODEs
             p0_diff = p0*(1-p0)*(q0 - (1 - q1))
```

```
    p1_diff = p1*(1-p1)*(q1 - (1 - q0))
    q0_diff = q0*(1-q0)*(a*p0 - (1-a)*(1-p1))
    q1_diff = q1*(1-q1)*((1-a)*p1 - a*(1-p0))
    # Return system of ODEs
    return [p0_diff, p1_diff, q0_diff, q1_diff]
```

We can compare the solution trajectories of the discrete- and continuous-time replicator dynamics. There are several things to note: * The shape of both are generally the same, the discrete-time isn't always "smooth" * The discrete-time dynamic looks like a condensed version of the continuous-time dynamics * The two kinds of dynamics sometimes yield different solutions

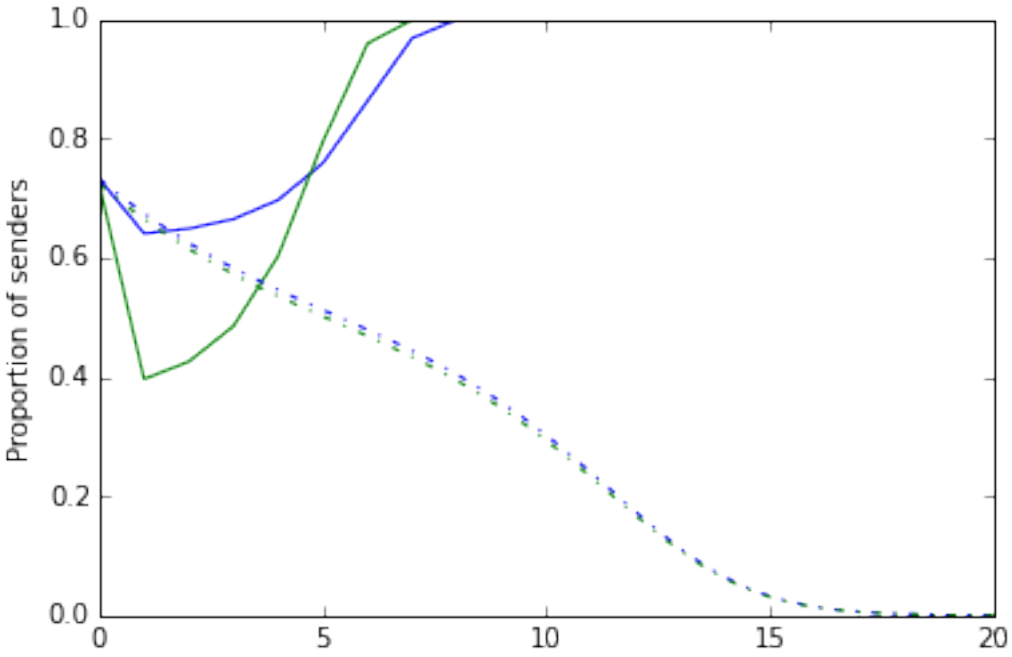
In [22]:
```
# Create random initial states
X = np.random.rand(2, 2)
X /= X.sum(axis=1)[:,np.newaxis]
Y = np.random.rand(2, 2)
Y /= Y.sum(axis=1)[:,np.newaxis]
# Iterate discrete-time replicator dynamics
X_hist, Y_hist = discrete_time_replicator_dynamics(20, X, Y, A, B, P)
# Format initial state
p0_vec = X.diagonal().tolist() + Y.diagonal().tolist()
params = [P[0,0]]
t_output = np.linspace(0, 20, num=20)
# Integrate continuous-time replicator dynamics
p_vec_result = odeint(signaling, p0_vec, t_output, args=(params,))
# Sender comparison
plt.plot(X_hist[:,0], 'b') # Proportion of t_0 sending m_0
plt.plot(X_hist[:,3], 'g') # Proportion of t_1 sending m_1
plt.plot(t_output, p_vec_result[:,0], 'b-.')
plt.plot(t_output, p_vec_result[:,1], 'g-.')
plt.ylim(0,1)
plt.ylabel("Proportion of senders")
plt.show()
# Receiver comparison
plt.plot(Y_hist[:,0], 'b') # Proportion of t_0 sending m_0
plt.plot(Y_hist[:,3], 'g') # Proportion of t_1 sending m_1
plt.plot(t_output, p_vec_result[:,2], 'b-.')
plt.plot(t_output, p_vec_result[:,3], 'g-.')
plt.ylim(0,1)
plt.ylabel("Proportion of receivers")
plt.show()
```

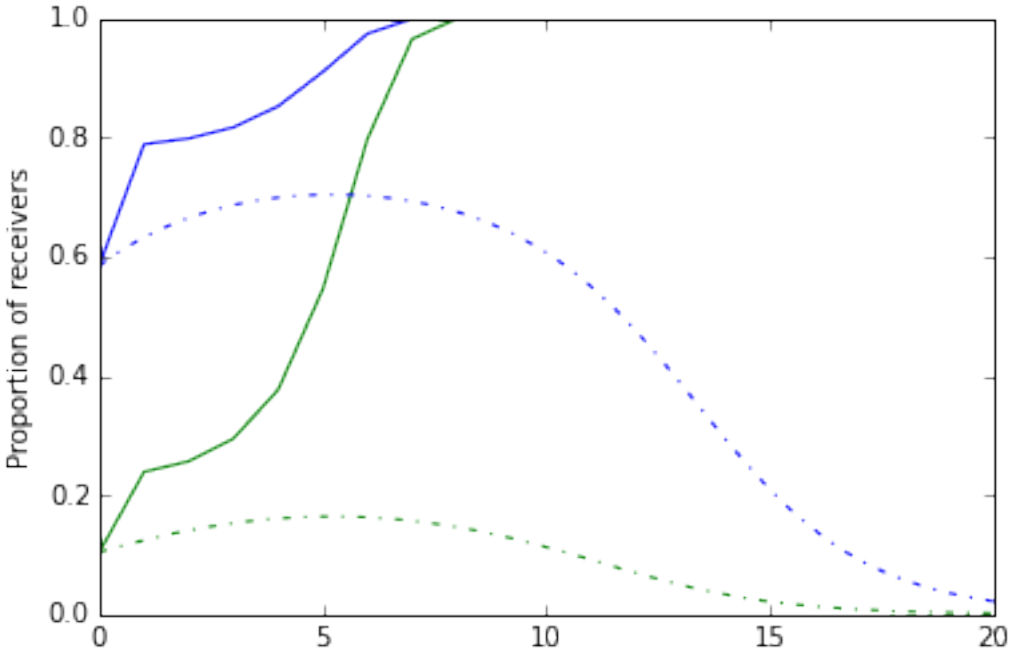




Let's look at a particular starting condition that yields different solution trajectories. It looks like the scaling factors for the discrete-time dynamics is rather large. That is, there's a really big jump at the first iteration. This may be enough to affect the rest of the iterations; perhaps the discretization is sufficient that it derails the rest. One way to look at this would be to simulate random starting points and to see when they trajectories diverge if the total movement in the first step exceeds some threshold.

In [23]:
```
X = np.array([[ 0.73162521,  0.26837479], [ 0.27700021,  0.72299979]])
Y = np.array([[ 0.58517496,  0.41482504], [ 0.89572339,  0.10427661]])
#
X_hist, Y_hist = discrete_time_replicator_dynamics(20, X, Y, A, B, P)
# Format initial state
p0_vec = X.diagonal().tolist() + Y.diagonal().tolist()
params = [P[0,0]]
t_output = np.linspace(0, 20)
# Integrate continuous-time replicator dynamics
p_vec_result = odeint(signaling, p0_vec, t_output, args=(params,))
# Sender comparison
plt.plot(X_hist[:,0], 'b') # Proportion of t_0 sending m_0
plt.plot(X_hist[:,3], 'g') # Proportion of t_1 sending m_1
plt.plot(t_output, p_vec_result[:,0], 'b-.')
plt.plot(t_output, p_vec_result[:,1], 'g-.')
plt.ylim(0,1)
plt.ylabel("Proportion of senders")
plt.show()
# Receiver comparison
plt.plot(Y_hist[:,0], 'b') # Proportion of t_0 sending m_0
plt.plot(Y_hist[:,3], 'g') # Proportion of t_1 sending m_1
plt.plot(t_output, p_vec_result[:,2], 'b-.')
plt.plot(t_output, p_vec_result[:,3], 'g-.')
plt.ylim(0,1)
plt.ylabel("Proportion of receivers")
plt.show()
```




Proportion of receivers
1.0
0.8
0.6
0.4
0.2
0.0
0
5
10
15
20