StrategicInfluence

May 30, 2021

1 Experimentation with Strategic Influence Network Model

James Yu

29 May 2021

[1]:
```
import matplotlib.pyplot as plt
import numpy as np
```

This notebook implements the finite-horizon example from the lecture notes, then expands to an infinite horizon, then experiments with different starting networks.

[2]:
```
# implementing the example from the lecture notes
N = 5
M = 1
  = 1
Q = 0.2 * np.identity(N)
T = 5 # finite horizon model, for now
```

The A adjacency matrix is a directed graph of inter- and intra-agent influence in the network.
A is NxN where $N = 5$ is the number of agents in the network.
The initial assignment is random but intuitively represents how much each agent's opinion is determined by those of others (and also themselves on the diagonal, i.e. self-confidence).

A bot with a fixed agenda would not be inflenced by anyone other than themselves, so given this bot as agent i, $A_{i,i} = 1$ and $A_{i,j} = 0, \forall j \neq i$.

[3]:
```
A = np.array([
  [0.217,   0.2022,   0.2358,   0.1256,   0.1403],
  [0.2497,   0.0107,   0.2334,   0.1282,   0.378],
  [0.1285,   0.0907,   0.3185,   0.2507,   0.2116],
  [0.1975,   0.0629,   0.2863,   0.2396,   0.2137],
  [0.1256,   0.0711,   0.0253,   0.2244,   0.5536],
])
```

The B listening matrix is a directed bipartite graph representing how much the opinions of the agents in the network are influenced by the messages sent out by the strategic agent.
This can be though of as an "augmentation" of the A matrix by one extra agent, the agent being the strategic agent with an agenda.

B is NxM where $N = 5$ is the number of agents in the network and $M = 1$ is the number of channels through which the strategic agent can broadcast messages to the network.

[4]:
```
B = np.array([
  0.0791,
  0,
  0,
  0,
  0,
])
```

The x_t opinion vector is an N-vector representing the opinions each agent has on the strategic agent's agenda.
These are normalized to zero.
This acts as the "state" of the dynamic system.

[5]:
```
x = np.array([
  -0.98,
  -4.62,
  2.74,
  4.67,
  2.15,
])
```

The r_t agent message vector is an M-vector representing the actual messages being broadcast by the agent.
This acts as the "control" of the dynamic system, and will be solved by the following algorithm.

Agents have a transition function $x_{t+1} = Ax_t + Br_t$.
Intuitively this says opinions of the agents across time are a linear combination of, from the previous periods:
- the influence of others, as given by A - self-confidence, as given by the diagonal in A - the influence of the strategic agent via message broadcast channels, as given by B

This means the sum of each row in A plus the correponding row in B must be equal to 1.

The strategic agent wishes to maximize their influence to push their agenda as accurately as possible. This is done using a standard time-discounted linear-quadratic regulator. In this example, since $\delta = 1$, time is not discounted.
However, since $Q = \frac{1}{5}I$, the absolute value of given opinions is discounted as the quadratic regulator equates to maximizing $-\sum_{t=0}^{T} \frac{1}{5}(x_t \cdot x_t)$. Intuitively, payoff is less suceptible to noise.

The optimal strategy is bijective to Sargent's Recursive Macroeconomic Theory, equation 5.2.9, as the formulations are the same (control variable - the messages being broadcast - does not factor into payoff, and thus its term is zero).

The optimal payoff is therefore the quadratic form $x'K_0x$ achieved through backwards induction from $K_T = Q$.
This is a truncated form of the infinite-horizon problem where $K_\infty = 0$ and there exists a steady-state K^*, as in the equation for the finite-horizon problem, $K_{T+1} = 0$.

[6]:
```
K = Q # the initial K_T

K_t = [0 for i in range(T+1)] # generate a list of T+1 elements from 0 to T
K_t[T] = K                    # set K_T

for i in range(T - 1, -1, -1):
    # iteratively construct each K_t using the discrete Riccati difference␣
 ↪equation
    K = ( * A.T * (K - (K @ B * (1/(B.T @ K @ B)) * B.T @ K)) @ A) + Q
    K_t[i] = K
```

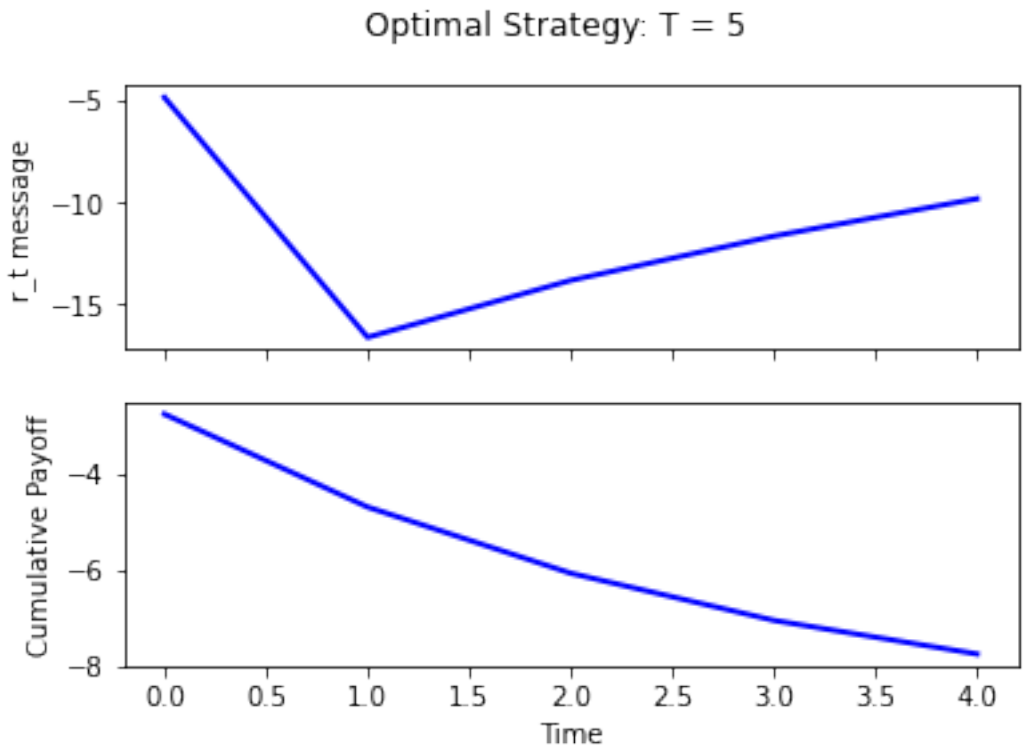
The above code evaluates the iteratively-defined K_t and stores it in a list of values for later. This can then be used to determine the optimal strategy.
This strategy is defined as $r_t(x_t) = L_t x_t$ such that $L_t = -(B'K_{t+1}B)^{-1}B'K_{t+1}$, the policy function which also appears in Recursive Macroeconomic Theory.

It should be noted that all of the code here is specifically designed to operate with a $M = 1$ single-message network. This is because Python can't invert 1 by 1 matrices. Different code is necessary to handle more messages.

[7]:
```
def L(t):
    return -1 * (1/(B.T @ K_t[t+1] @ B)) * B.T @ K_t[t+1] @ A

x_t = x
payoff = 0
r_ts = []
payoffs = []
for t in range(T):
    r_t = L(t) @ x_t
    r_ts.append(r_t)
    x_t = A @ x_t + B * r_t
    payoff += -1 * (x_t.T @ Q @ x_t)
    payoffs.append(payoff)
```

The above code computes the optimal strategy and stores the choices of r in a list alongside the cumulative payoff. This plots as follows:

[8]:
```
fig, sub = plt.subplots(2, sharex=True)
fig.suptitle("Optimal Strategy: T = 5")

sub[0].plot(range(T), r_ts, 'b', linewidth=2)
sub[0].set(ylabel = "r_t message")

sub[1].plot(range(T), payoffs, 'b', linewidth=2)
sub[1].set(xlabel = "Time", ylabel =  "Cumulative Payoff")

plt.show()
```



As in the diagrams from the lecture notes, as time increases, the message level decreases before increasing again. Cumulative payoff is continuously decreasing, also as in the notes.

To do an infinite-horizon problem, a steady-state must be found in the computation of K_t. This can be done as follows:

[9]:
```
K = np.zeros((N, N)) # the initial K is zero

K_t = [K, Q] # now the K_t matrices will be added on-demand
K = Q        # start here to avoid division by zero caused by inverse of zero␣
↪matrix

while True:
    # iteratively construct each K_t using the discrete Riccati difference␣
↪equation
    K_new = ( * A.T * (K - (K @ B * (1/(B.T @ K @ B)) * B.T @ K)) @ A) + Q
    K_t.append(K_new)
    current_difference = np.max(np.abs(K - K_new))
    K = K_new
    print(current_difference)
    if current_difference == 0:
        break
```

```
0.05735777400000003
```

```
0.018849471778646265
0.0067461258790245116
0.0024623325659086093
0.0009042138120677334
0.00033258472349328994
0.00012238685361176 67
4.5042563995456586e-05
1.6577808048701126e-05
6.101484085307973e-06
2.2456655787150837e-06
8.265231525306227e-07
3.0420409807829785e-07
1.1196315397032919e-07
4.120834706800025e-08
1.5166845668268536e-08
5.582199369413843e-09
2.0545438683683415e-09
7.56180562611064e-10
2.7831431603786427e-10
1.0243433878898145e-10
3.770123102597722e-11
1.3876066962126288e-11
5.107081424426951e-12
1.8797186029928525e-12
6.918354777951663e-13
2.5462965069777965e-13
9.370282327836321e-14
3.447242491461111e-14
1.2712053631958042e-14
4.6629367034256575e-15
1.7208456881689926e-15
6.661338147750939e-16
2.220446049250313e-16
1.1102230246251565e-16
2.0816681711721685e-17
2.168404344971009e-18
0.0
```

Conveniently this converges to exactly zero, although that may not always happen (due to numerical errors) so a less strict tolerance could be used instead.
The policy is therefore:

[10]:
```
K_t.reverse()

x_t = x
payoff = 0
r_ts = []
payoffs = []
```

```
for t in range(len(K_t) - 1):
    r_t = L(t) @ x_t
    r_ts.append(r_t)
    x_t = A @ x_t + B * r_t
    payoff += -1 * (x_t.T @ Q @ x_t)
    payoffs.append(payoff)
```

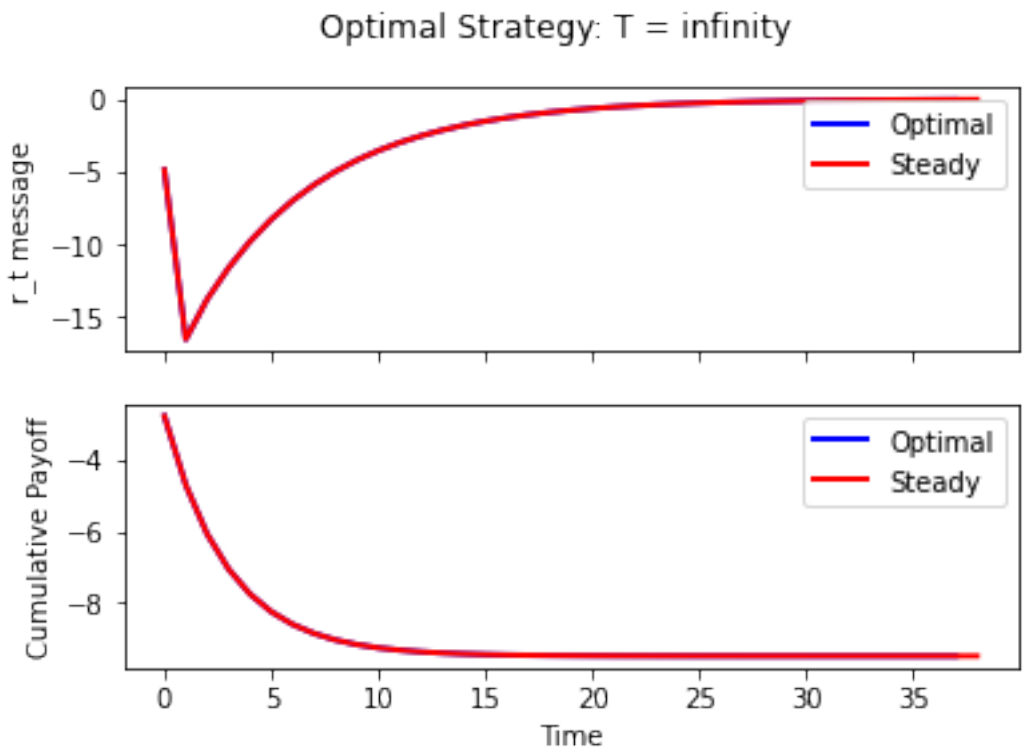
```
c:\users\jbrig\appdata\local\programs\python\python37\lib\site-
packages\ipykernel_launcher.py:2: RuntimeWarning: divide by zero encountered in
double_scalars

c:\users\jbrig\appdata\local\programs\python\python37\lib\site-
packages\ipykernel_launcher.py:2: RuntimeWarning: invalid value encountered in
multiply

c:\users\jbrig\appdata\local\programs\python\python37\lib\site-
packages\ipykernel_launcher.py:2: RuntimeWarning: invalid value encountered in
matmul
```

A strategy purely using the steady-state K_t can also be used:

[11]:
```
def steady_L(t):
    return -1 * (1/(B.T @ K_t[0] @ B)) * B.T @ K_t[0] @ A

steady_x_t = x
steady_payoff = 0
steady_r_ts = []
steady_payoffs = []
for t in range(len(K_t) - 1):
    steady_r_t = steady_L(t) @ steady_x_t
    steady_r_ts.append(steady_r_t)
    steady_x_t = A @ steady_x_t + B * steady_r_t
    steady_payoff += -1 * (steady_x_t.T @ Q @ steady_x_t)
    steady_payoffs.append(steady_payoff)
```

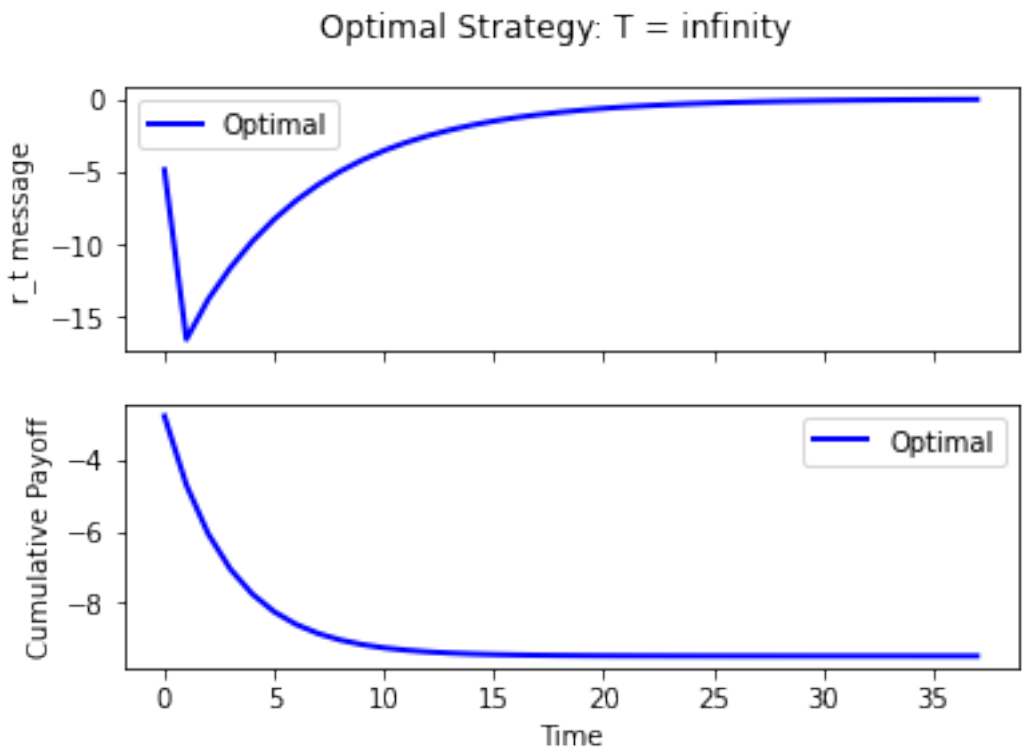
[12]:
```
fig, sub = plt.subplots(2, sharex=True)
fig.suptitle("Optimal Strategy: T = infinity")

sub[0].plot(range(len(K_t) - 1), r_ts, 'b', label = "Optimal", linewidth=2)
sub[0].plot(range(len(K_t) - 1), steady_r_ts, 'r', label = "Steady",␣
 ↪linewidth=2)
sub[0].set(ylabel = "r_t message")

sub[1].plot(range(len(K_t) - 1), payoffs, 'b', label = "Optimal", linewidth=2)
sub[1].plot(range(len(K_t) - 1), steady_payoffs, 'r', label = "Steady",␣
 ↪linewidth=2)
sub[1].set(xlabel = "Time", ylabel =  "Cumulative Payoff")
```

```
sub[0].legend()
sub[1].legend()
plt.show()
```



The steady-state and optimal strategies are the same, just like in the lecture notes. If this is not convincing, the underlying optimal strategy can be plotted independently:

[13]:
```
fig, sub = plt.subplots(2, sharex=True)
fig.suptitle("Optimal Strategy: T = infinity")

sub[0].plot(range(len(K_t) - 1), r_ts, 'b', label = "Optimal", linewidth=2)
sub[0].set(ylabel = "r_t message")

sub[1].plot(range(len(K_t) - 1), payoffs, 'b', label = "Optimal", linewidth=2)
sub[1].set(xlabel = "Time", ylabel =  "Cumulative Payoff")

sub[0].legend()
sub[1].legend()
plt.show()
```



Now that the algorithm matches the lecture notes, various modifications can be tested with the network structure.

The first modification of interest is that with a "robot" agent, or a bot with a fixed agenda. Given one of these in our five-agent network, A would be modified to be:

[14]:
```
A = np.array([
  [0.217,    0.2022,   0.2358,   0.1256,   0.1403],
  [0,        1,        0,        0,        0     ],
  [0.1285,   0.0907,   0.3185,   0.2507,   0.2116],
  [0.1975,   0.0629,   0.2863,   0.2396,   0.2137],
  [0.1256,   0.0711,   0.0253,   0.2244,   0.5536],
])
```

This "bot" has some influence over the other agents, but they themselves are a computer program not influenced by anyone (including the strategic agent). We must also give this robot an opinion which counteracts the strategic agent:

[15]:
```
x = np.array([
  -0.98,
  2, # the robot, which is against the strategic agent
  2.74,
  4.67,
  2.15,
```

```
])
```

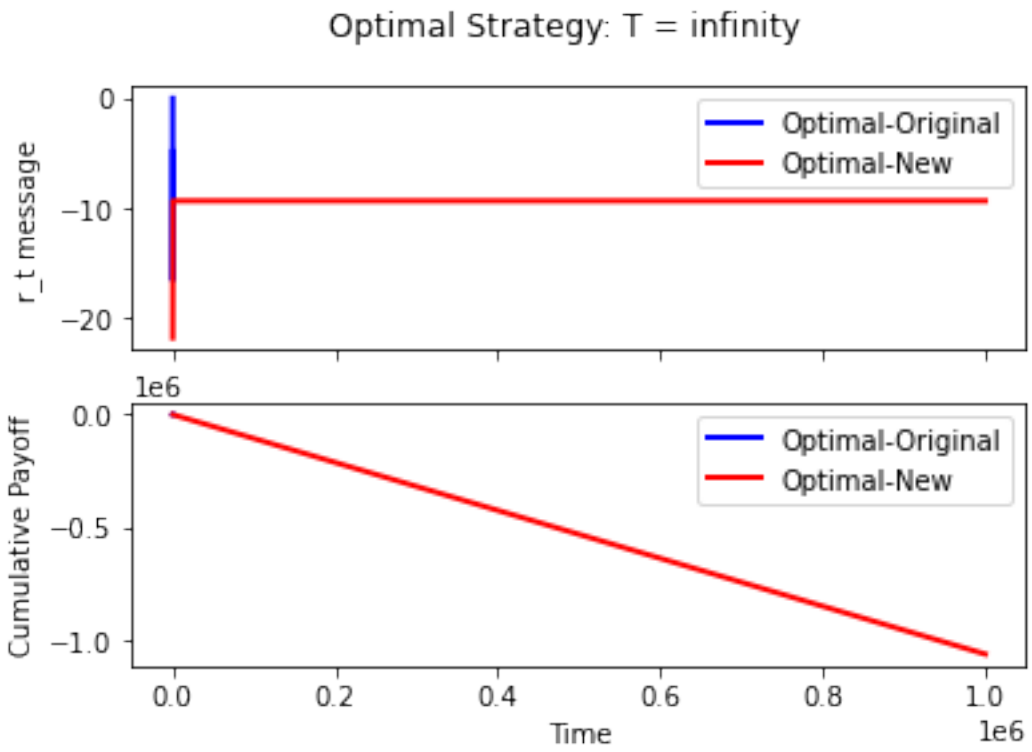
[16]:
```
old_length = len(K_t)
K = np.zeros((N, N)) # initial K

K_t = [K, Q] # saved K
K = Q
i = 0
while True:
    K_new = (  * A.T * (K - (K @ B * (1/(B.T @ K @ B)) * B.T @ K)) @ A) + Q
    K_t.append(K_new)
    current_difference = np.max(np.abs(K - K_new))
    K = K_new
    i += 1
    if i % 1000000 == 0:
        print(i, current_difference)
        break
    if current_difference == 0:
        break
```

```
1000000 0.1912960536137689
```

[17]:
```
K_t.reverse()

x_t = x
payoff = 0
r_ts2 = []
payoffs2 = []
for t in range(len(K_t) - 1):
    r_t = L(t) @ x_t
    r_ts2.append(r_t)
    x_t = A @ x_t + B * r_t
    payoff += -1 * (x_t.T @ Q @ x_t)
    payoffs2.append(payoff)
```

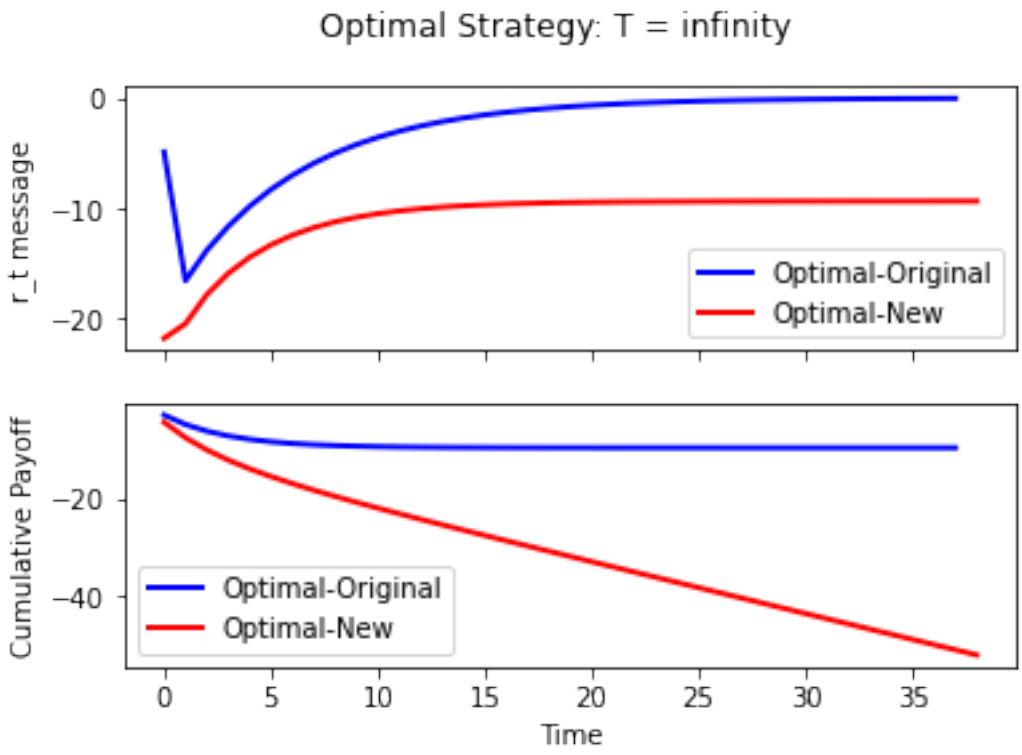
```
c:\users\jbrig\appdata\local\programs\python\python37\lib\site-
packages\ipykernel_launcher.py:2: RuntimeWarning: divide by zero encountered in
double_scalars

c:\users\jbrig\appdata\local\programs\python\python37\lib\site-
packages\ipykernel_launcher.py:2: RuntimeWarning: invalid value encountered in
multiply

c:\users\jbrig\appdata\local\programs\python\python37\lib\site-
packages\ipykernel_launcher.py:2: RuntimeWarning: invalid value encountered in
matmul
```

[18]:
```
fig, sub = plt.subplots(2, sharex=True)
fig.suptitle("Optimal Strategy: T = infinity")

sub[0].plot(range(old_length - 1), r_ts, 'b', label = "Optimal-Original",␣
 ↪linewidth=2)
sub[0].plot(range(len(K_t) - 1), r_ts2, 'r', label = "Optimal-New", linewidth=2)
sub[0].set(ylabel = "r_t message")

sub[1].plot(range(old_length - 1), payoffs, 'b', label = "Optimal-Original",␣
 ↪linewidth=2)
sub[1].plot(range(len(K_t) - 1), payoffs2, 'r', label = "Optimal-New",␣
 ↪linewidth=2)
sub[1].set(xlabel = "Time", ylabel =  "Cumulative Payoff")

sub[0].legend()
sub[1].legend()
plt.show()
```



Under these conditions the system has reached a local, eternally decreasing state of consistency. Payoff never diminishes, and the intensity of the message never changes as the strategic agent apparently has to fight the influence of the robot indefinitely. Intuitively this may make sense because fighting a continuous program acting as a powerhouse of misinformation would take considerable resources to counteract, but this deserves further study.

[19]:
```
fig, sub = plt.subplots(2, sharex=True)
fig.suptitle("Optimal Strategy: T = infinity")

sub[0].plot(range(old_length - 1), r_ts, 'b', label = "Optimal-Original",␣
 ↪linewidth=2)
sub[0].plot(range(old_length - 1), r_ts2[:old_length - 1], 'r', label =␣
 ↪"Optimal-New", linewidth=2)
sub[0].set(ylabel = "r_t message")

sub[1].plot(range(old_length - 1), payoffs, 'b', label = "Optimal-Original",␣
 ↪linewidth=2)
sub[1].plot(range(old_length - 1), payoffs2[:old_length - 1], 'r', label =␣
 ↪"Optimal-New", linewidth=2)
sub[1].set(xlabel = "Time", ylabel =  "Cumulative Payoff")

sub[0].legend()
sub[1].legend()
plt.show()
```



Locally there are similar patterns: exponentially decreasing cumulative payoff leading to a linear steady state, and logarithmically increasing message intensity leading to a linear steady state. The difference is the line has a slope of zero in the payoff function without the bot, and the line slopes downward in the payoff function with the bot.

Another approach that can be observed, without doing too many changes yet, is a complete uniform graph where everyone listens to everyone equally. This looks like:

[20]:
```
A = np.array([
  [0.1666,   0.1666,   0.1666,   0.1666,   0.1666],
  [0.1666,   0.1666,   0.1666,   0.1666,   0.1666],
  [0.1666,   0.1666,   0.1666,   0.1666,   0.1666],
  [0.1666,   0.1666,   0.1666,   0.1666,   0.1666],
  [0.1666,   0.1666,   0.1666,   0.1666,   0.1666],
])
```

[21]:
```
B = np.array([
  0.1666,
  0.1666,
  0.1666,
  0.1666,
  0.1666,
])
```

Let's use the original initial opinions, which are:

[22]:
```
x = np.array([
  -0.98,
  -4.62,
  2.74,
  4.67,
  2.15,
])
```

Then the optimal solution is:

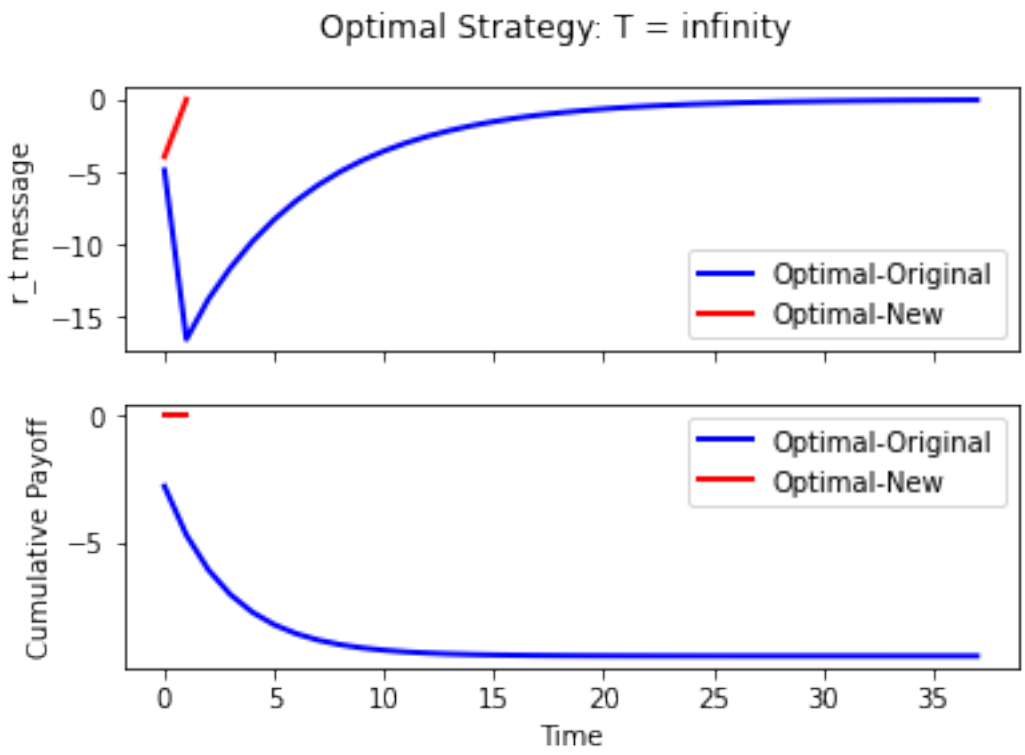
[23]:
```
K = np.zeros((N, N)) # initial K

K_t = [K, Q] # saved K
K = Q
while True:
    K_new = (  * A.T * (K - (K @ B * (1/(B.T @ K @ B)) * B.T @ K)) @ A) + Q
    K_t.append(K_new)
    current_difference = np.max(np.abs(K - K_new))
    K = K_new
    print(current_difference)
    if current_difference == 0:
        break
```

```
5.203055541969093e-19
0.0
```

[24]:
```
K_t.reverse()

x_t = x
payoff = 0
r_ts2 = []
payoffs2 = []
for t in range(len(K_t) - 1):
    r_t = L(t) @ x_t
    r_ts2.append(r_t)
    x_t = A @ x_t + B * r_t
    payoff += -1 * (x_t.T @ Q @ x_t)
    payoffs2.append(payoff)
```

```
c:\users\jbrig\appdata\local\programs\python\python37\lib\site-
packages\ipykernel_launcher.py:2: RuntimeWarning: divide by zero encountered in
double_scalars

c:\users\jbrig\appdata\local\programs\python\python37\lib\site-
packages\ipykernel_launcher.py:2: RuntimeWarning: invalid value encountered in
matmul
```

[25]:
```
fig, sub = plt.subplots(2, sharex=True)
fig.suptitle("Optimal Strategy: T = infinity")

sub[0].plot(range(old_length - 1), r_ts, 'b', label = "Optimal-Original",␣
 ↪linewidth=2)
sub[0].plot(range(len(K_t) - 1), r_ts2, 'r', label = "Optimal-New", linewidth=2)
sub[0].set(ylabel = "r_t message")

sub[1].plot(range(old_length - 1), payoffs, 'b', label = "Optimal-Original",␣
 ↪linewidth=2)
sub[1].plot(range(len(K_t) - 1), payoffs2, 'r', label = "Optimal-New",␣
 ↪linewidth=2)
sub[1].set(xlabel = "Time", ylabel =  "Cumulative Payoff")

sub[0].legend()
sub[1].legend()
plt.show()
```



With an entirely uniform network, the model converges almost immediately. Equal weighting means not much work is required to get the agenda through, especially since some initial opinions were biased in favour. However, suppose they were not:

[26]:
```
x = np.array([
  10,
  10,
  10,
  10,
  10,
])
```

[27]:
```
K = np.zeros((N, N)) # initial K

K_t = [K, Q] # saved K
K = Q
while True:
    K_new = (  * A.T * (K - (K @ B * (1/(B.T @ K @ B)) * B.T @ K)) @ A) + Q
    K_t.append(K_new)
    current_difference = np.max(np.abs(K - K_new))
    K = K_new
    print(current_difference)
    if current_difference == 0:
```

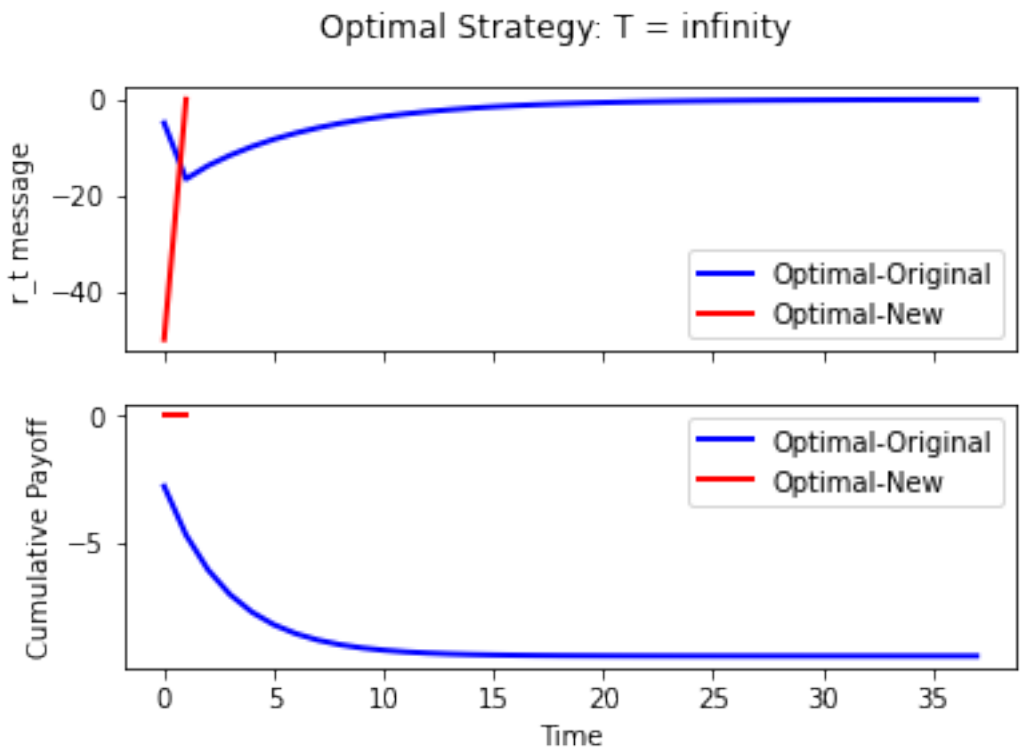
```
        break
```

```
5.203055541969093e-19
0.0
```

[28]:
```
K_t.reverse()

x_t = x
payoff = 0
r_ts2 = []
payoffs2 = []
for t in range(len(K_t) - 1):
    r_t = L(t) @ x_t
    r_ts2.append(r_t)
    x_t = A @ x_t + B * r_t
    payoff += -1 * (x_t.T @ Q @ x_t)
    payoffs2.append(payoff)
```

```
c:\users\jbrig\appdata\local\programs\python\python37\lib\site-
packages\ipykernel_launcher.py:2: RuntimeWarning: divide by zero encountered in
double_scalars

c:\users\jbrig\appdata\local\programs\python\python37\lib\site-
packages\ipykernel_launcher.py:2: RuntimeWarning: invalid value encountered in
matmul
```

[29]:
```
fig, sub = plt.subplots(2, sharex=True)
fig.suptitle("Optimal Strategy: T = infinity")

sub[0].plot(range(old_length - 1), r_ts, 'b', label = "Optimal-Original",␣
 ↪linewidth=2)
sub[0].plot(range(len(K_t) - 1), r_ts2, 'r', label = "Optimal-New", linewidth=2)
sub[0].set(ylabel = "r_t message")

sub[1].plot(range(old_length - 1), payoffs, 'b', label = "Optimal-Original",␣
 ↪linewidth=2)
sub[1].plot(range(len(K_t) - 1), payoffs2, 'r', label = "Optimal-New",␣
 ↪linewidth=2)
sub[1].set(xlabel = "Time", ylabel =  "Cumulative Payoff")

sub[0].legend()
sub[1].legend()
plt.show()
```



The physical amount of time it took to converge did not change at all. However, the intensity of the initial message required to convince everyone did. This seems to hint on two primary factors at play. Amount of time it takes to reach a steady-state is related to the influencibility of the agents in the network, while intensity of the messages required is related to the initial opinions of the agents in the network.

Finally, consider a network where there existed one "super bot" who not only is unmovable in terms of agenda, but also exerts just as much influence on everyone else as the strategic agent does. Suppose we had a setup like:

[30]:
```
A = np.array([
  [1,       0,       0,       0,       0],
  [0.8,     0,       0,       0,       0],
  [0.8,     0,       0,       0,       0],
  [0.2,     0,       0,       0,       0],
  [0.2,     0,       0,       0,       0],
])
```

[31]:
```
B = np.array([
  0,
  0.2,
  0.2,
  0.8,
```

```
  0.8,
])
```

This is a network with a very strong existing follower base. Two of the agents are big followers of the first agent, where the first agent is a devout supporter of their own agenda. The two remaining agents are highly influenced by the strategic agent.

Suppose we start with the superbot advertising a negative campaign against the strategic agent's agenda, and the followers were split between sides:

[32]:
```
x = np.array([
  10,
  1,
  1,
  -1,
  -1,
])
```

With the considerable power the superbot has over half of the entire network, it would be difficult for the strategic agent to optimize against. The optimal solution is:

[33]:
```
K = np.zeros((N, N)) # initial K

K_t = [K, Q] # saved K
K = Q

i = 0
while True:
    K_new = (  * A.T * (K - (K @ B * (1/(B.T @ K @ B)) * B.T @ K)) @ A) + Q
    K_t.append(K_new)
    current_difference = np.max(np.abs(K - K_new))
    K = K_new
    i += 1
    if i % 1000000 == 0:
        print(i, current_difference)
        break
    if current_difference == 0:
        break
```

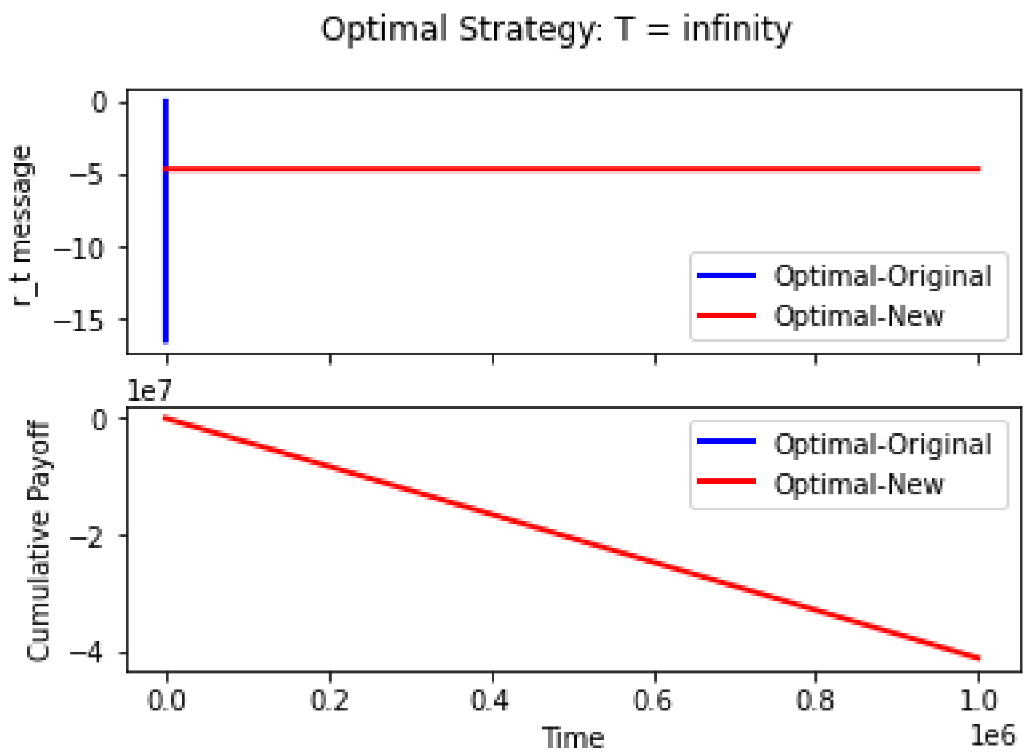
```
1000000 0.18494117649970576
```

[34]:
```
K_t.reverse()

x_t = x
payoff = 0
r_ts2 = []
payoffs2 = []
x_ts = []
```

```
for t in range(len(K_t) - 1):
    r_t = L(t) @ x_t
    r_ts2.append(r_t)
    x_t = A @ x_t + B * r_t
    x_ts.append(x_t)
    payoff += -1 * (x_t.T @ Q @ x_t)
    payoffs2.append(payoff)
```

```
c:\users\jbrig\appdata\local\programs\python\python37\lib\site-
packages\ipykernel_launcher.py:2: RuntimeWarning: divide by zero encountered in
double_scalars

c:\users\jbrig\appdata\local\programs\python\python37\lib\site-
packages\ipykernel_launcher.py:2: RuntimeWarning: invalid value encountered in
multiply

c:\users\jbrig\appdata\local\programs\python\python37\lib\site-
packages\ipykernel_launcher.py:2: RuntimeWarning: invalid value encountered in
matmul
```

[35]:
```
fig, sub = plt.subplots(2, sharex=True)
fig.suptitle("Optimal Strategy: T = infinity")

sub[0].plot(range(old_length - 1), r_ts, 'b', label = "Optimal-Original",␣
 ↪linewidth=2)
sub[0].plot(range(len(K_t) - 1), r_ts2, 'r', label = "Optimal-New", linewidth=2)
sub[0].set(ylabel = "r_t message")

sub[1].plot(range(old_length - 1), payoffs, 'b', label = "Optimal-Original",␣
 ↪linewidth=2)
sub[1].plot(range(len(K_t) - 1), payoffs2, 'r', label = "Optimal-New",␣
 ↪linewidth=2)
sub[1].set(xlabel = "Time", ylabel =  "Cumulative Payoff")

sub[0].legend()
sub[1].legend()
plt.show()
```



It appears having 50% of the population following is insufficient to counter the bot without having to broadcast the message indefinitely.
We can actually observe what the steady-state opinions on the strategic agent's agenda are given that the superbot is continuously requiring the strategic agent to broadcast above their preferred normalized level:

[45]:
```
for i in range(0, 1000000, 100000):
    print(x_ts[i])
```

```
[10.          7.05882353  7.05882353 -1.76470588 -1.76470588]
[10.          7.05882353  7.05882353 -1.76470588 -1.76470588]
[10.          7.05882353  7.05882353 -1.76470588 -1.76470588]
[10.          7.05882353  7.05882353 -1.76470588 -1.76470588]
[10.          7.05882353  7.05882353 -1.76470588 -1.76470588]
[10.          7.05882353  7.05882353 -1.76470588 -1.76470588]
[10.          7.05882353  7.05882353 -1.76470588 -1.76470588]
[10.          7.05882353  7.05882353 -1.76470588 -1.76470588]
[10.          7.05882353  7.05882353 -1.76470588 -1.76470588]
[10.          7.05882353  7.05882353 -1.76470588 -1.76470588]
```

The opinions actually converge. The agents that were most influenced by the superbot had their opinions skyrocket towards that of the superbot. Because of the strategic agent having to continuously broadcast at -5, this actually drove the agents that supported the strategic agent further away from zero (-5 because the agent attempted to counteract the relatively unmovable agents

supporting the superbot).

This seems to indicate a bit of an issue - when faced with a split network of competing interests, attempting to maximize inflence doesn't actually bring anyone closer to the strategic agent's platform, but perhaps does prevent the agents from moving even further away from it.