Multi_Message_vs_Single_Message

September 18, 2021

1 Comparison of Single-Agent-Dual-Message model with Dual-Agent-Single-Message model

James Yu, 18 September 2021

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

I moved the code for this notebook to the bottom for readability - the order in which the blocks were run is indicated by the number on the left of each block.

1.1 1-naive 1-strategic benchmark case, one message per period only

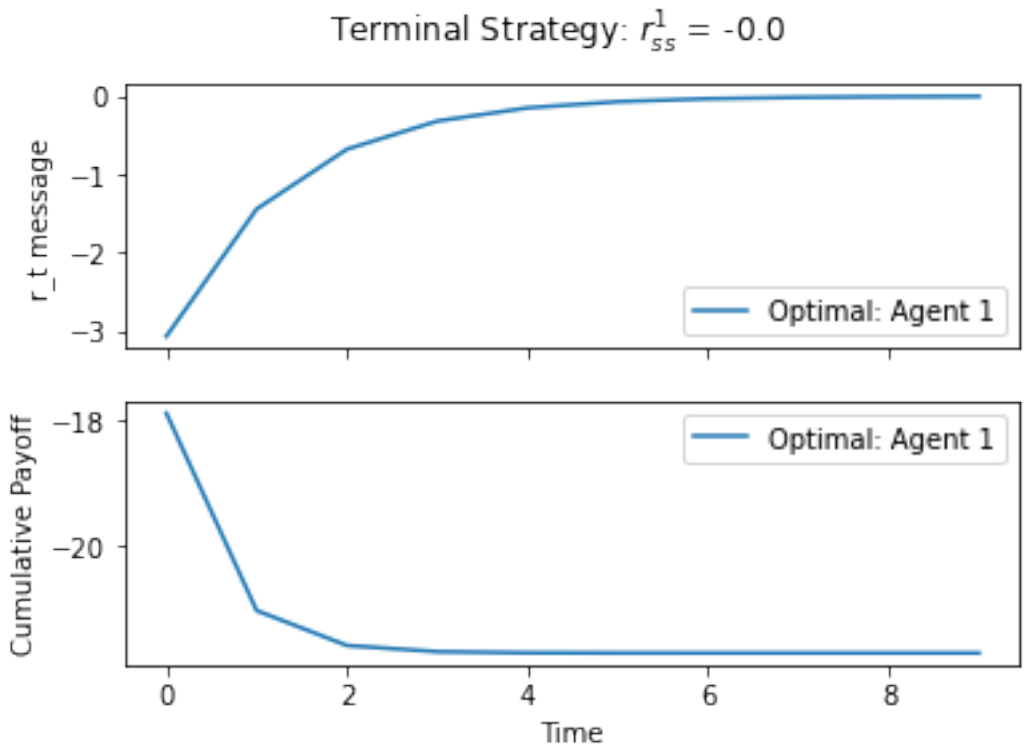
(that is, one naive agent and one strategic agent in this particular model) First, testing using the original system:

[8]:
```
A = np.array([
  [0.7],
], ndmin = 2)

B_1 = np.array([
  0.3
], ndmin = 2).T

B = [B_1]

delta = 0.8
n = 1
m = 1
L = 1
Q = [1 * np.identity(n)]
R = [0.2 * np.identity(m)] # COST

x = [0]
r = [0]
c_base = sum([B[l] @ np.array([[r[l]]], ndmin = 2) for l in range(L)])
c = [c_base + (A - np.identity(n)) @ (x[l] * np.ones((n, 1))) for l in range(L)]
```

```
X_0_1 = np.array([
  4
], ndmin = 2).T # starting with an X_0 > 0
X_0 = [X_0_1]
```

[9]:
```
max_distances, historical_K, historical_k, historical_kappa = solve([Q[0]], [np.
↪zeros((1, n))], [0], A, B, delta, n, m, L, Q, R, x, c)
```

[10]:
```
xs, rs, payoffs = optimal(X_0, historical_K, historical_k, historical_kappa)
do_plot(rs, r, payoffs, num_agents = 1, set_cap = 10)
```



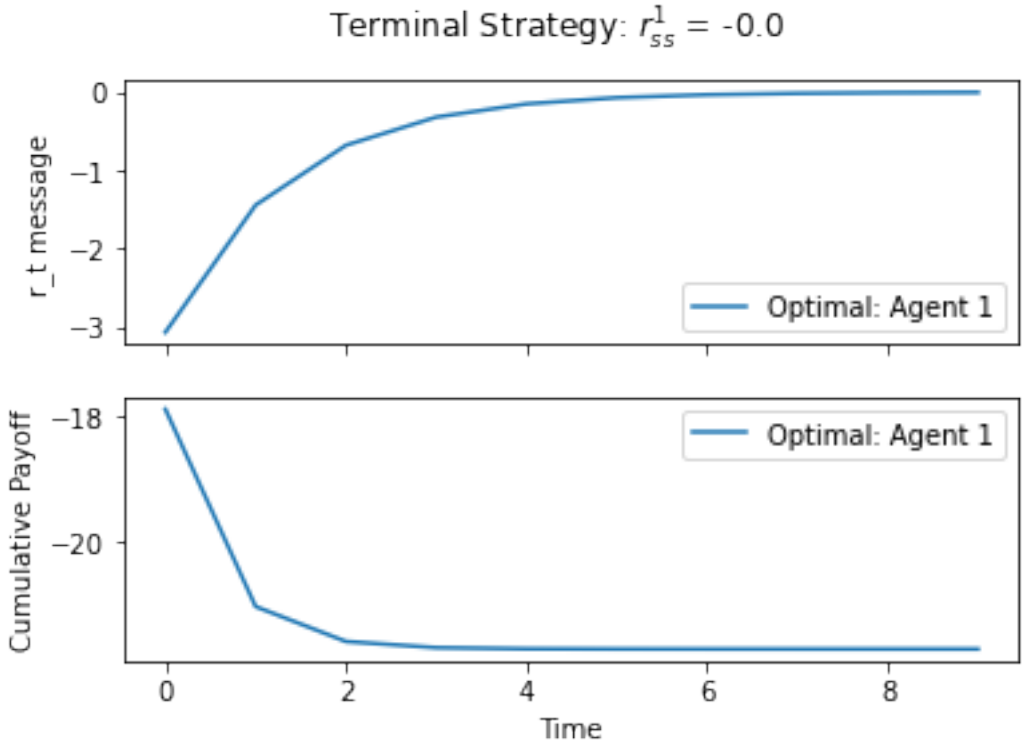
[11]:
```
save_rs = rs
save_payoffs = payoffs
```

[12]:
```
payoffs[0][-1]
```

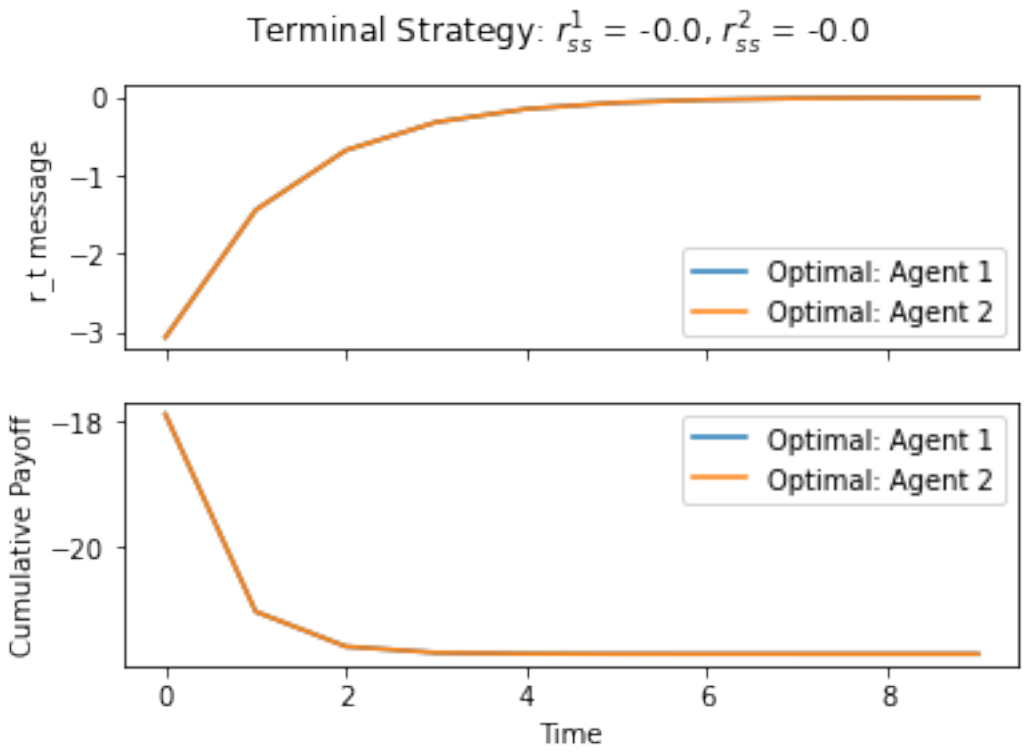
[12]:
```
-21.719082733041443
```

Next, testing my new code to ensure the new code works properly:

[13]:
```
A = np.array([
  [0.7],
], ndmin = 2)

B = np.array([
  [0.3]
], ndmin = 2)

delta = 0.8
Q = 1 * np.identity(1)
R = 0.2 * np.identity(1)
x = np.array([
  [4],
], ndmin = 2)

r_ts, x_ts, payoffs, Ks = optimal_single(1, delta, A, B, R, Q, x)
save2_r_ts = r_ts
save2_payoffs = payoffs
do_plot({0:r_ts}, [0], {0:payoffs}, num_agents = 1, set_cap = 10)
```



[14]:
```
do_plot({0:r_ts, 1:save_rs[0]}, [0, 0], {0:payoffs, 1:save_payoffs[0]},␣
↪num_agents = 2, set_cap = 10)
```



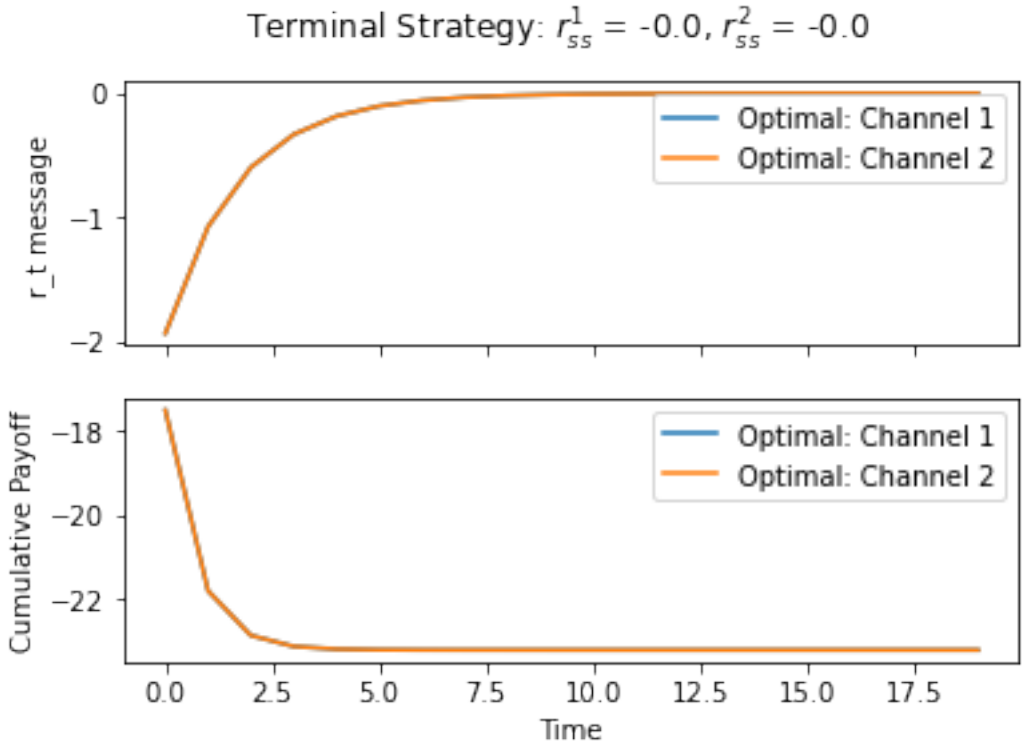
They are the exact same, as expected, meaning there are no immediate issues (there shouldn't be, since the code is just the original single-agent code with a few scale tweaks).

1.2 Comparing the single message case to the dual message case in the 1-strategic model

First, just the dual message case:

[15]:
```
A = np.array([
  [0.7],
], ndmin = 2)

B = np.array([
  [0.15, 0.15] # here the agent now has two channels through which to send
], ndmin = 2)

delta = 0.8
Q = 1 * np.identity(1)
R = 0.2 * np.identity(2)
x = np.array([
  [4],
], ndmin = 2)
```

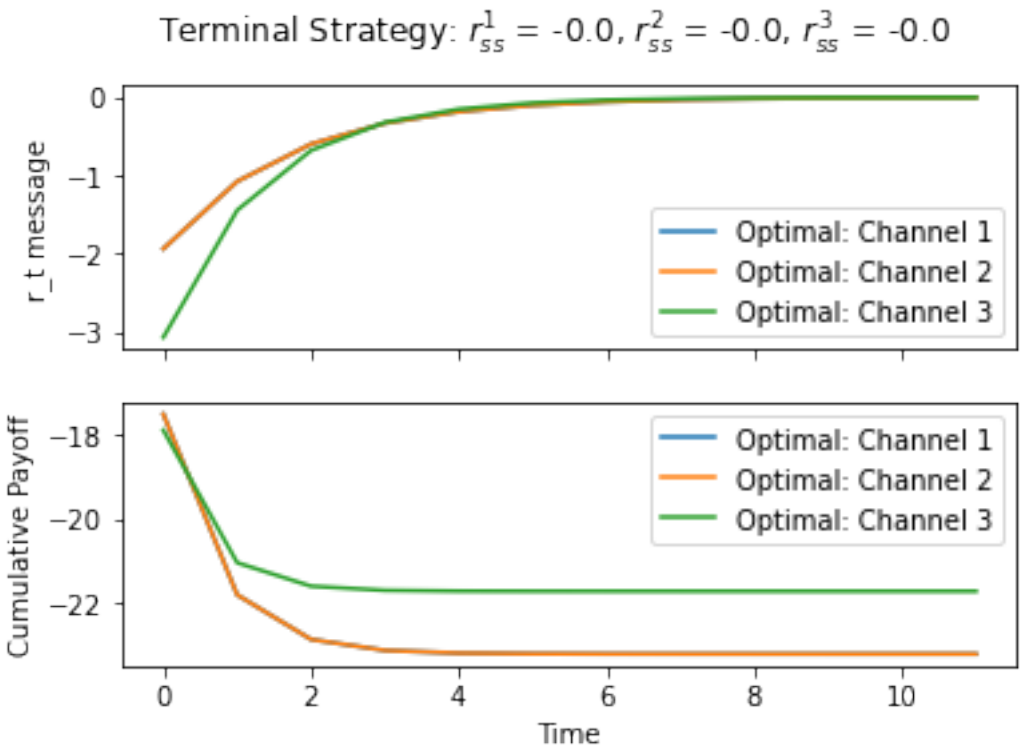
```
r_ts, x_ts, payoffs, Ks = optimal_single(1, delta, A, B, R, Q, x)
do_plot({0:[i[0] for i in r_ts], 1: [i[1] for i in r_ts]}, [0, 0], {0:payoffs,␣
↪1:payoffs}, num_agents = 2, set_cap = 20, flag = True)
```



What we can see off the bat with the dual message case is that the messages are closer to zero, and both messages on a per-period basis are always identical. Payoff does appear to be worse in this specific case.

Now, we can compare this more directly to the one-message model:

[16]:
```
do_plot({0:[i[0] for i in r_ts], 1: [i[1] for i in r_ts], 2:save2_r_ts}, [0, 0,␣
↪0], {0:payoffs, 1:payoffs, 2:save2_payoffs}, num_agents = 3, set_cap = 12,␣
↪flag = True)
```



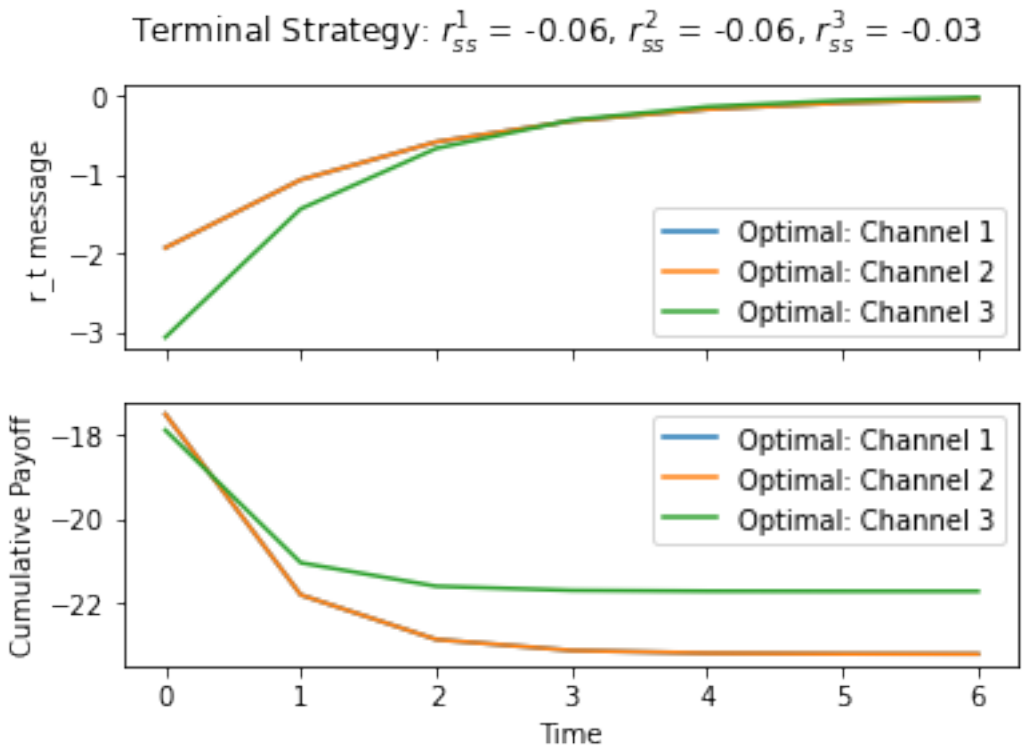
Here Channel 3 refers to the agent of the one-message case, not a third message.

[17]:
```
print("Payoff in one-message case:", save2_payoffs[-1])
print("Payoff in two-message case:", payoffs[-1])
```

```
Payoff in one-message case: -21.719082733041443
Payoff in two-message case: -23.21599696956931
```

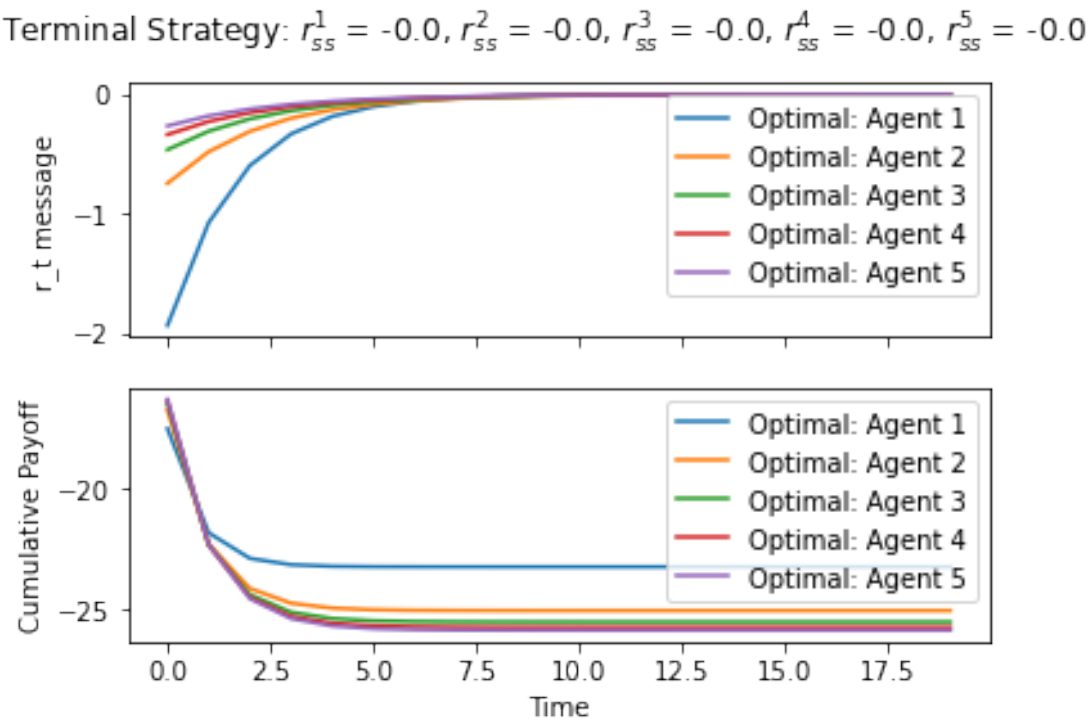
We can also plot just the first 7 iterations:

[18]:
```
do_plot({0:[i[0] for i in r_ts], 1: [i[1] for i in r_ts], 2:save2_r_ts}, [0, 0,␣
↪0], {0:payoffs, 1:payoffs, 2:save2_payoffs}, num_agents = 3, set_cap = 7,␣
↪flag = True)
```



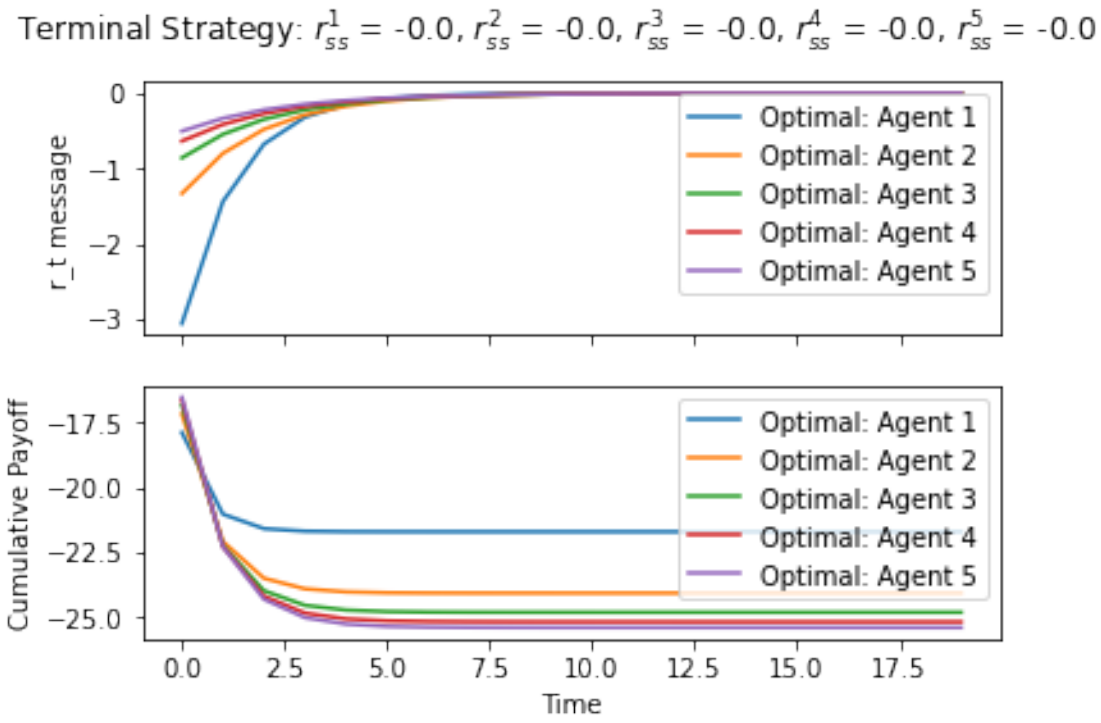
One observation is that there is a point where both lines cross in both plots. The effect of this can be observed by varying some of the parameters. Let's try varying R.

[19]:
```
rsx = {}
psx = {}
cap = 5
for j, i in enumerate(np.linspace(0.2, 2, cap)):
    R = i * np.identity(2)
    r_ts, x_ts, payoffs, Ks = optimal_single(1, delta, A, B, R, Q, x)
    rsx[j] = [i[0] for i in r_ts]
    psx[j] = payoffs
do_plot(rsx, [0 for i in range(cap)], psx, num_agents = cap, set_cap = 20)
```



The blue line is what we had before, and we can clearly see that increasing cost R lowers payoff and brings messages closer to zero. If we do this in the one-agent case:

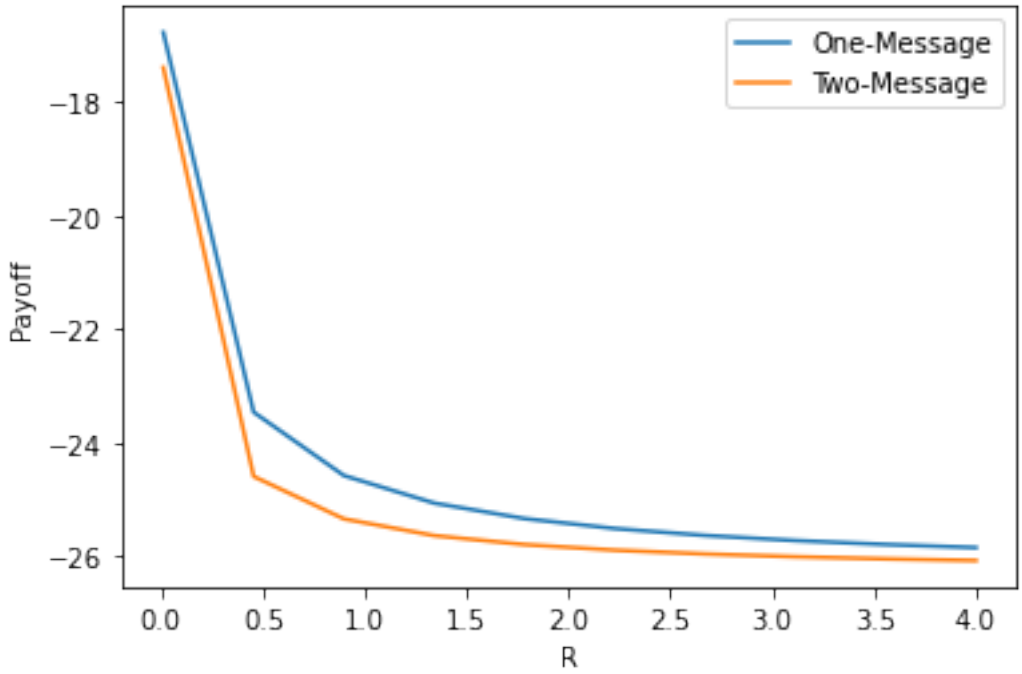
[20]:
```
B = np.array([
  [0.3]
], ndmin = 2)

rsx = {}
psx = {}
cap = 5
for j, i in enumerate(np.linspace(0.2, 2, cap)):
    R = i * np.identity(1)
    r_ts, x_ts, payoffs, Ks = optimal_single(1, delta, A, B, R, Q, x)
    rsx[j] = [i[0] for i in r_ts]
    psx[j] = payoffs
do_plot(rsx, [0 for i in range(cap)], psx, num_agents = cap, set_cap = 20)
```



we observe the same pattern. The question now is whether there is a point where the two-message model can do better than the one-message model. Instead of plotting each iteration, we can try plotting final payoff as a function of R in both cases.

[21]:
```
# start with one message model
B = np.array([
  [0.3]
], ndmin = 2)
payoffx = []
for j, i in enumerate(np.linspace(0.01, 4, 10)):
    R = i * np.identity(1)
    r_ts, x_ts, payoffs, Ks = optimal_single(1, delta, A, B, R, Q, x)
    payoffx.append(payoffs[-1])

# and then two message model
B = np.array([
  [0.15, 0.15]
], ndmin = 2)
payoffx2 = []
for j, i in enumerate(np.linspace(0.01, 4, 10)):
    R = i * np.identity(2)
    r_ts, x_ts, payoffs, Ks = optimal_single(1, delta, A, B, R, Q, x)
    payoffx2.append(payoffs[-1])
```

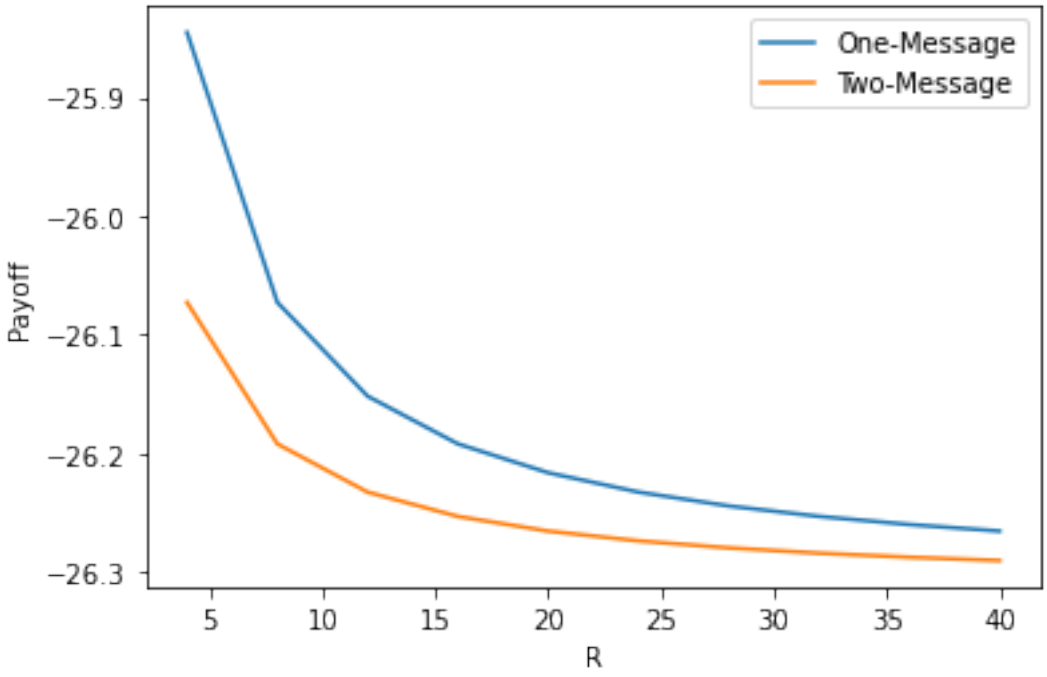
```
fig, ax = plt.subplots()
plt.plot(np.linspace(0.01, 4, 10), payoffx, label = "One-Message")
plt.plot(np.linspace(0.01, 4, 10), payoffx2, label = "Two-Message")
plt.xlabel("R")
plt.ylabel("Payoff")
ax.legend()
plt.show()
```



At the extrema, the payoffs appear to be similar between the two models, but the one-message model seems to be consistently on top.

[22]:
```
# start with one message model
B = np.array([
  [0.3]
], ndmin = 2)
payoffx = []
for j, i in enumerate(np.linspace(4, 40, 10)):
    R = i * np.identity(1)
    r_ts, x_ts, payoffs, Ks = optimal_single(1, delta, A, B, R, Q, x)
    payoffx.append(payoffs[-1])

# and then two message model
B = np.array([
  [0.15, 0.15]
], ndmin = 2)
```

```
payoffx2 = []
for j, i in enumerate(np.linspace(4, 40, 10)):
    R = i * np.identity(2)
    r_ts, x_ts, payoffs, Ks = optimal_single(1, delta, A, B, R, Q, x)
    payoffx2.append(payoffs[-1])

fig, ax = plt.subplots()
plt.plot(np.linspace(4, 40, 10), payoffx, label = "One-Message")
plt.plot(np.linspace(4, 40, 10), payoffx2, label = "Two-Message")
plt.xlabel("R")
plt.ylabel("Payoff")
ax.legend()
plt.show()
```



This indicates that the limit behavior is likely equality.

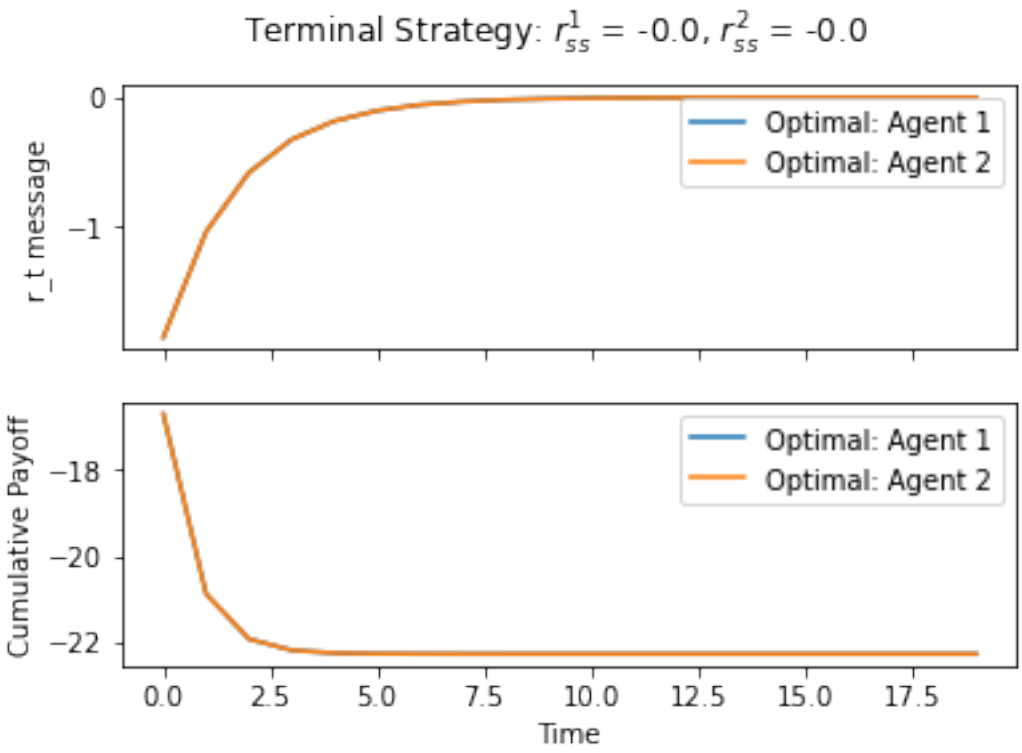
1.3 Comparison between the 1-strategic 2-channel case and the 2-strategic 1-channel case (the baseline multiple strategic agent model)

[23]:
```
A = np.array([
  [0.7],
], ndmin = 2)

B_1 = np.array([
```

```
  0.15, # split the channel of the strategic agent from the previous model␣
 ↪amongst the two strategic agents here
], ndmin = 2).T

B = [B_1, B_1]

X_0_1 = np.array([
  4,
], ndmin = 2).T
X_0 = [X_0_1, X_0_1]

delta = 0.8
n = 1
m = 1
L = 2 # two agents now
Q = [1 * np.identity(n), 1 * np.identity(n)]
R = [0.2 * np.identity(m), 0.2 * np.identity(m)]

x = [0, 0] # identical agents
r = [0, 0]
c_base = sum([B[l] @ np.array([[r[l]]], ndmin = 2) for l in range(L)])
c = [c_base + (A - np.identity(n)) @ (x[l] * np.ones((n, 1))) for l in range(L)]
```

[24]:
```
max_distances, historical_K, historical_k, historical_kappa = solve(Q, [np.
 ↪zeros((1, n)), np.zeros((1, n))], [0, 0], A, B, delta, n, m, L, Q, R, x, c,␣
 ↪tol = 1000)
```

[25]:
```
xs2, rs2, payoffs2 = optimal(X_0, historical_K, historical_k, historical_kappa)
do_plot(rs2, r, payoffs2, num_agents = 2, set_cap = 20)
```



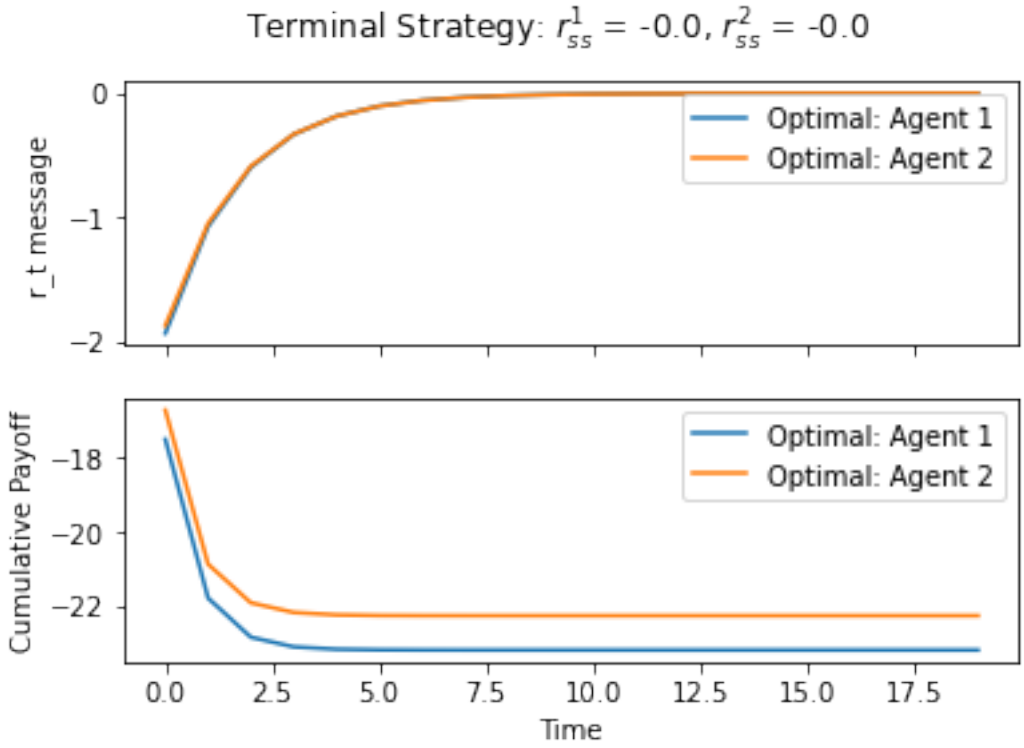
This is the baseline two-strategic-agent model. Here cumulative payoff is:

[26]:
```
payoffs2[0][-1]
```

[26]:
```
-22.28448611624381
```

[27]:
```
A = np.array([
  [0.7],
], ndmin = 2)

B = np.array([
  [0.15, 0.15] # here the agent now has two channels through which to send
], ndmin = 2)

delta = 0.8
Q = 1 * np.identity(1)
R = 0.2 * np.identity(2)
x = np.array([
  [4],
], ndmin = 2)

r_ts, x_ts, payoffs, Ks = optimal_single(1, delta, A, B, R, Q, x)
```

```
do_plot({0:[i[0] for i in r_ts], 1:rs2[1]}, [0, 0], {0:payoffs, 1:payoffs2[1]},␣
↪num_agents = 2, set_cap = 20)
```



Here "Agent 1" refers to the single agent case with two messages, and "Agent 2" is one of the two agents in the dual agent case with one message. The strategies appear to almost be identical, but the payoff is different. In fact, the single strategic agent is worse off (-23). This makes sense because, if the same strategy is being used for both models, the single agent has to take on the cost of both messages instead of just one of them in the dual case.

1.4 Primary Code:

[2]:
```
def M(K, B, R, L, delta):
    """Computes M_{t-1} given B_l \forall l, K_t^l \forall l,
        R_l \forall l, number of strategic agents L, and delta."""
    # handle the generic structure first:
    template = [(B[l].T @ K[l] @ B[l]).item() for l in range(L)]
    base = [template.copy() for l_prime in range(L)]
    # then change the diagonals to construct M_{t-1}:
    for l in range(L): base[l][l] = (B[l].T @ K[l] @ B[l] + R[l]/delta).item()
    return np.array(base, ndmin = 2)

def H(B, K, A, L):
    """Computes H_{t-1} given B_l \forall l, K_t^l \forall l,
```

```
        A, and number of strategic agents L."""
    return np.concatenate(tuple(B[l].T @ K[l] @ A for l in range(L)), axis = 0)

def C_l(B, K, k, h, L, c, x, n):
    """Computes C_{t-1}^h (displayed as C_{t-1}^l) given B_l \forall l, K_t^l␣
 ↪\forall l,
        k_t^l \forall l, a specific naive agent h, number of strategic agents␣
 ↪L,
        c_l \forall l, x_l \forall l, and number of naive agents n"""
    return np.concatenate(tuple(B[l].T @ K[l] @ A @ ((x[h] - x[l]) * np.
 ↪ones((n, 1)))
                            + B[l].T @ K[l] @ c[l]
                            + 0.5 * B[l].T @ k[l].T for l in range(L)), axis = 0)

def E(M_, H_):
    """Computes the generic E_{t-1} given M_{t-1} and H_{t-1}."""
    return np.linalg.inv(M_) @ H_

def F(M_, C_l_, l):
    """Computes F_{t-1}^l given M_{t-1}, C_{t-1}^l, and specific naive agent l.
 ↪"""
    return (np.linalg.inv(M_) @ C_l_)[l:l+1, :]

def G(A, B, E_, L):
    """Computes the generic G_{t-1} given A, B_l \forall l,
        E_{t-1}, and number of strategic agents L."""
    return A - sum([B[l] @ E_[l:l+1, :] for l in range(L)])

def g_l(B, E_, h, x, F_, L):
    """Computes g_{t-1}^l given B_l \forall l, E_{t-1}^l,
        a particular naive agent h, x_l \forall l, F_{t-1}^l \forall l,
        number of strategic agents L, number of naive agents n, and c_h."""
    return - sum([B[l] @ (E_[l:l+1, :] @ ((x[h] - x[l]) * np.ones((n, 1))) +␣
 ↪F_[l]) for l in range(L)]) + c[h]
```

[3]:
```
def K_t_minus_1(Q, K, E_, R, G_, L, delta):
    return [Q[l] + E_[l:l+1, :].T @ R[l] @ E_[l:l+1, :]
            + delta * G_.T @ K[l] @ G_ for l in range(L)]

def k_t_minus_1(K, k, G_, g, E_, F_, R, L, delta):
    return [2*delta* g[l].T @ K[l] @ G_ + delta * k[l] @ G_
            + 2 * F_[l].T @ R[l] @ E_[l:l+1, :] for l in range(L)]

def kappa_t_minus_1(K, k, kappa, g_, F_, R, L, delta):
    return [-delta * (g_[l].T @ K[l] @ g_[l] + k[l] @ g_[l] - kappa[l])
            - (F_[l].T @ R[l] @ F_[l]) for l in range(L)]
```

[4]:
```
def solve(K_t, k_t, kappa_t, A, B, delta, n, m, L, Q, R, x, c, tol = 300):
    historical_K = [K_t]
    historical_k = [k_t]
    historical_kappa = [kappa_t]
    max_distances = defaultdict(list)
    counter = 0
    while True:
        M_ = M(K_t, B, R, L, delta)
        H_ = H(B, K_t, A, L)
        E_ = E(M_, H_)
        G_ = G(A, B, E_, L)
        K_new = K_t_minus_1(Q, K_t, E_, R, G_, L, delta)
        F_ = [F(M_, C_l(B, K_t, k_t, l, L, c, x, n), l) for l in range(L)]
        g = [g_l(B, E_, h, x, F_, L) for h in range(L)]
        k_new = k_t_minus_1(K_t, k_t, G_, g, E_, F_, R, L, delta)
        kappa_new = kappa_t_minus_1(K_t, k_t, kappa_t, g, F_, R, L, delta)
        cd_K = [np.max(np.abs(K_t[l] - K_new[l])) for l in range(L)]
        cd_k = [np.max(np.abs(k_t[l] - k_new[l])) for l in range(L)]
        cd_kappa = [np.max(np.abs(kappa_t[l] - kappa_new[l])) for l in range(L)]
        K_t = K_new
        k_t = k_new
        kappa_t = kappa_new
        historical_K.insert(0, K_t)
        historical_k.insert(0, k_t)
        historical_kappa.insert(0, kappa_t)
        for l in range(L):
            max_distances[(l+1, "K")].append(cd_K[l])
            max_distances[(l+1, "k")].append(cd_k[l])
            max_distances[(l+1, "kappa")].append(cd_kappa[l])
        counter += 1
        if sum(cd_K + cd_k + cd_kappa) == 0 or counter > tol:
            return max_distances, historical_K, historical_k, historical_kappa
```

[5]:
```
def optimal(X_init, historical_K, historical_k, historical_kappa, infinite =␣
↪True):
    X_t = [a.copy() for a in X_init]
    xs = defaultdict(list)
    for l in range(L):
        xs[l].append(X_t[l])

    rs = defaultdict(list)
    payoffs = defaultdict(list)
    payoff = defaultdict(lambda: 0)
    i = 0
    while [i < len(historical_K), True][infinite]:
        K_t = historical_K[[i, 0][infinite]]
        k_t = historical_k[[i, 0][infinite]]
```

```
        M_ = M(K_t, B, R, L, delta)
        H_ = H(B, K_t, A, L)
        E_ = E(M_, H_)
        G_ = G(A, B, E_, L)
        F_ = [F(M_, C_l(B, K_t, k_t, l, L, c, x, n), l) for l in range(L)]
        g = [g_l(B, E_, h, x, F_, L) for h in range(L)]
        for l in range(L):
            Y_new = -1 * E_[l:l+1, :] @ X_t[l] - F(M_, C_l(B, K_t, k_t, l, L,␣
↪c, x, n), l)
            rs[l].append(Y_new)
            payoff[l] += (-1 * delta**i * (X_t[l].T @ Q[l] @ X_t[l])).item() +␣
↪(-1 * delta**i * (Y_new.T @ R[l] @ Y_new)).item()
            payoffs[l].append(payoff[l])
            X_new = G_ @ X_t[l] + g[l]
            xs[l].append(X_new)
            if infinite == True and np.max(X_t[l] - X_new) == 0:
                return xs, rs, payoffs
            X_t[l] = X_new
        i += 1

    return xs, rs, payoffs
```

[6]:
```
def do_plot(rs, r, payoffs, num_agents = 1, set_cap = np.inf, flag = False,␣
↪legend = True):
    fig, sub = plt.subplots(2, sharex=True)
    if legend:
        fig.suptitle(f"Terminal Strategy: {', '.join(['$r_{ss}^' + str(l+1) +␣
↪'$ = ' + str(round(rs[l][:min(len(rs[l]), set_cap)][-1].item() + r[l], 2))␣
↪for l in range(num_agents)])}")

    for l in range(num_agents):
        sub[0].plot(range(min(len(rs[l]), set_cap)), [a.item() + r[l] for a in␣
↪rs[l][:min(len(rs[l]), set_cap)]], label = f"Optimal: {['Agent',␣
↪'Channel'][flag]} {l+1}")
    sub[0].set(ylabel = "r_t message")

    for l in range(num_agents):
        sub[1].plot(range(min(len(payoffs[l]), set_cap)), payoffs[l][:
↪min(len(payoffs[l]), set_cap)], label = f"Optimal: {['Agent',␣
↪'Channel'][flag]} {l+1}")
    sub[1].set(xlabel = "Time", ylabel =  "Cumulative Payoff")
    if legend:
        sub[0].legend()
        sub[1].legend()
    plt.show()
```

[7]:
```
def optimal_single(num_agents, delta, A, B, R, Q, x):
    K = np.zeros((num_agents, num_agents))
    K_t = [Q]
    K = Q
    while True:
        K_new = delta * (A.T @ (K - (K @ B @ np.linalg.inv((B.T @ K @ B) + R/
↪delta) @ B.T @ K)) @ A) + Q
        K_t.insert(0, K_new)
        current_difference = np.max(np.abs(K - K_new))
        K = K_new
        if current_difference == 0:
            break

    def L_single(K_ent):
        return -1 * np.linalg.inv((B.T @ K_ent @ B) + R/delta) @ B.T @ K_ent @ A

    x_t = x
    r_ts = []

    payoff = 0
    payoffs = []
    x_ts = [x]
    i = 0
    while True:
        r_t = L_single(K_t[0]) @ x_t
        r_ts.append(r_t)
        payoff += (-1 * delta**i * (x_t.T @ Q @ x_t)).item() + (-1 * delta**i *␣
↪(r_t.T @ R @ r_t)).item()
        payoffs.append(payoff)
        x_t_new = A @ x_t + B @ r_t
        x_ts.append(x_t_new)
        if np.max(x_t_new - x_t) == 0:
            break
        x_t = x_t_new
        i += 1

    return r_ts, x_ts, payoffs, K_t
```