Opposite_Bias

October 30, 2021

1 Opposite Bias Model

James Yu, 30 October 2021

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

Just a quick test of one agent with an agenda of 10 and one agent with an agenda of -5 both targeting a single naive agent.

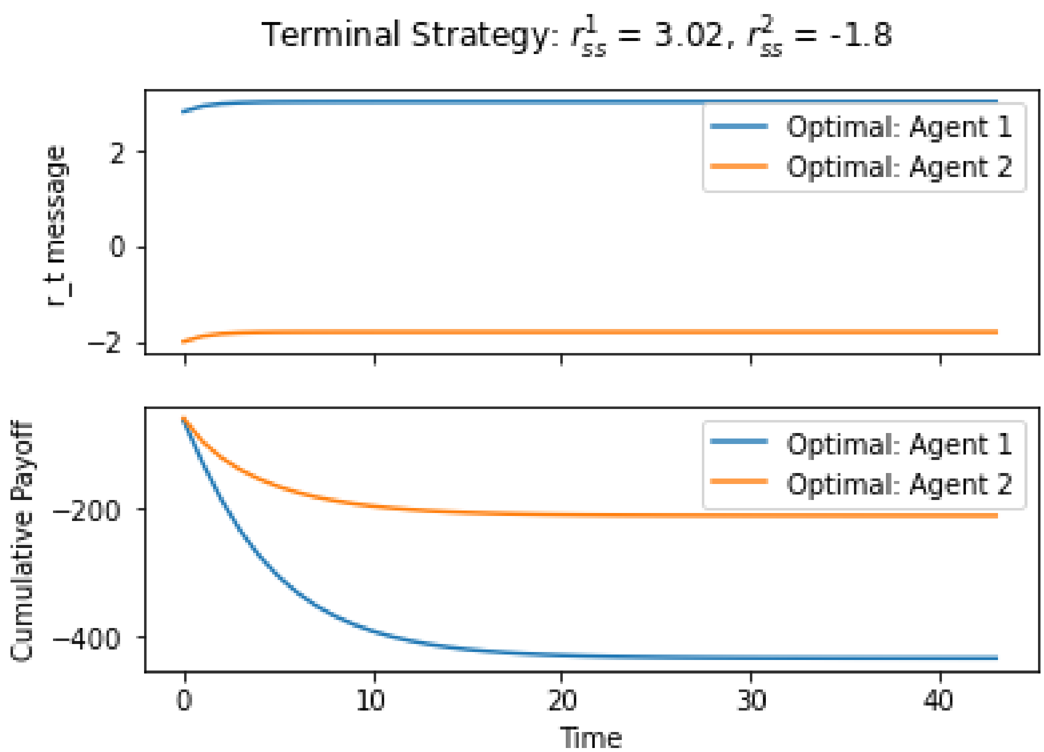
1.1 Baseline, infinite time

[7]:
```
A = np.array([
  [0.5],
], ndmin = 2)

B_1 = np.array([
  0.25,
], ndmin = 2).T

B_2 = np.array([
  0.25,
], ndmin = 2).T

B = [B_1, B_2]

X_0_1 = np.array([
  2.5 - 10, # somewhat in between, will vary it later
], ndmin = 2).T

X_0_2 = np.array([
  2.5 + 5,
], ndmin = 2).T
X_0 = [X_0_1, X_0_2]

delta = 0.8
n = 1
```

```
m = 1
L = 2
Q = [1 * np.identity(n), 1 * np.identity(n)]
R = [1 * np.identity(m), 1 * np.identity(m)]

x = [10, -5] # here we have difference in agendas
r = [0, 0] # a message of nonzero weight has cost
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)]
```

[8]:
```
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)
```

[9]:
```
xs2, rs2, payoffs2 = optimal(X_0, historical_K, historical_k, historical_kappa)
do_plot(rs2, r, payoffs2, num_agents = 2)
```



[10]:
```
payoffs2[0][-1] # agent 1 with the 10
```

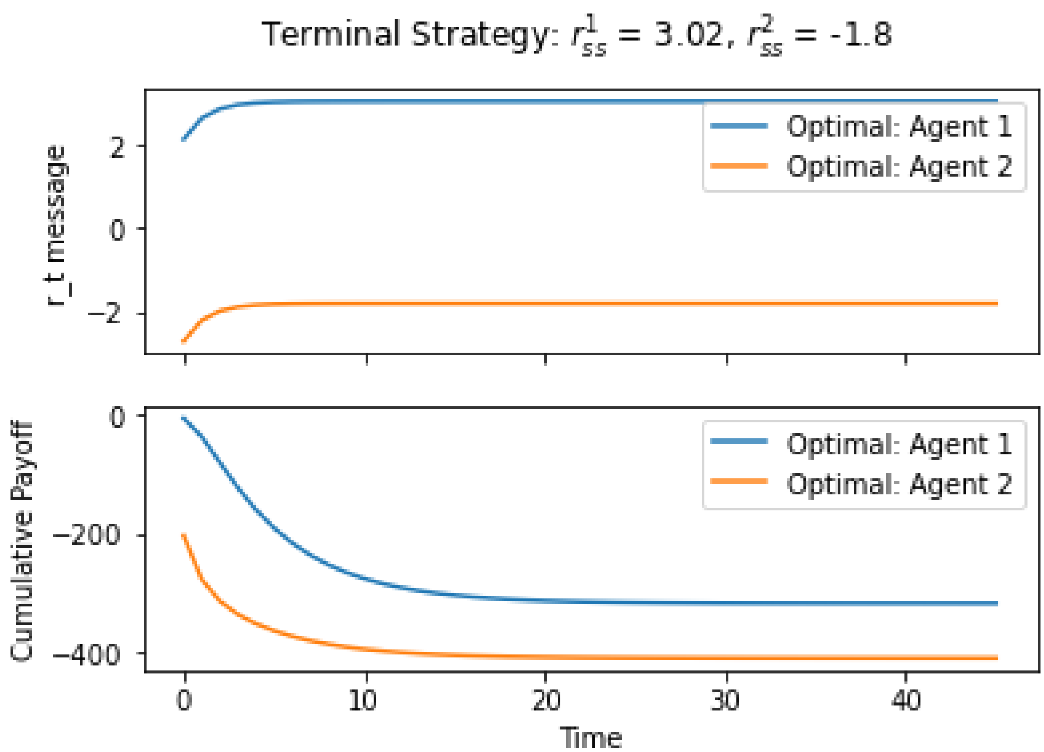
[10]: -433.77327120528565

[11]:
```
payoffs2[1][-1] # agent 2 with the -5
```

[11]: -212.00644422949424

1.2 Baseline, infinite time, different starting opinion of naive agent

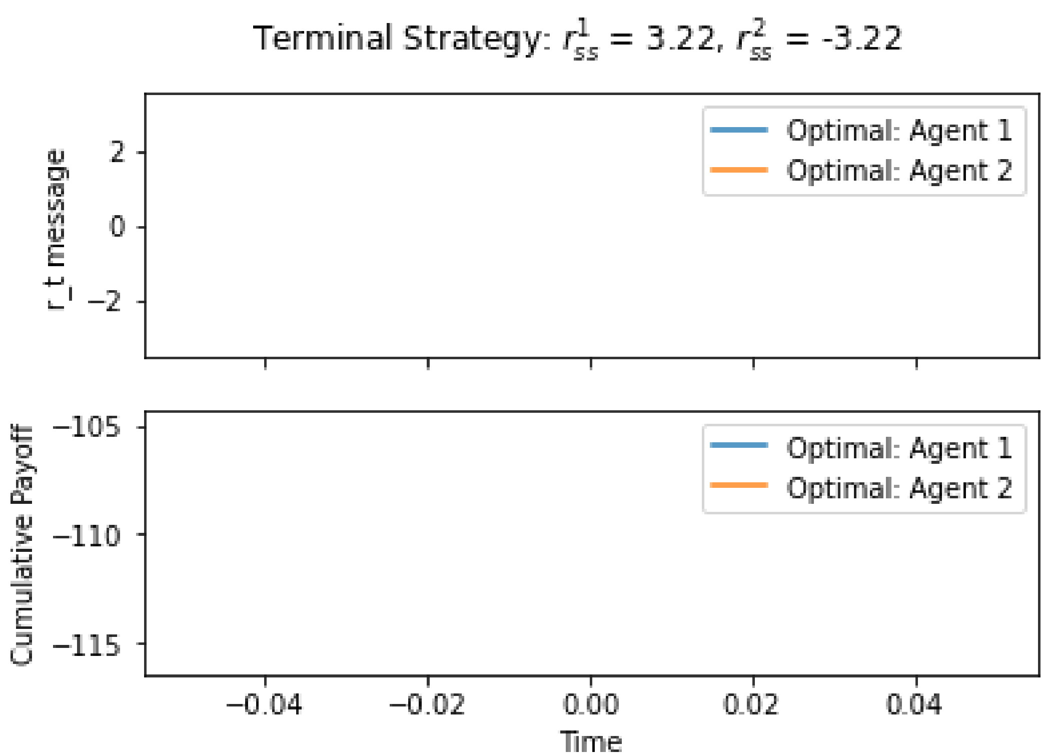
[12]:
```
X_0_1 = np.array([
  9 - 10,
], ndmin = 2).T

X_0_2 = np.array([
  9 + 5,
], ndmin = 2).T
X_0 = [X_0_1, X_0_2]

x = [10, -5] # here we have difference in agendas
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)]
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)
xs2, rs2, payoffs2 = optimal(X_0, historical_K, historical_k, historical_kappa)
do_plot(rs2, r, payoffs2, num_agents = 2)
```



Terminal strategy is the same but the trajectory does differ because of the introduced shift in initial opinion.

1.3 Exactly opposite agendas, infinite time

[13]:
```
X_0_1 = np.array([
  0 - 10,
], ndmin = 2).T

X_0_2 = np.array([
  0 + 10,
], ndmin = 2).T
X_0 = [X_0_1, X_0_2]

x = [10, -10] # here we have difference in agendas
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)]
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)
xs2, rs2, payoffs2 = optimal(X_0, historical_K, historical_k, historical_kappa)
do_plot(rs2, r, payoffs2, num_agents = 2)
```



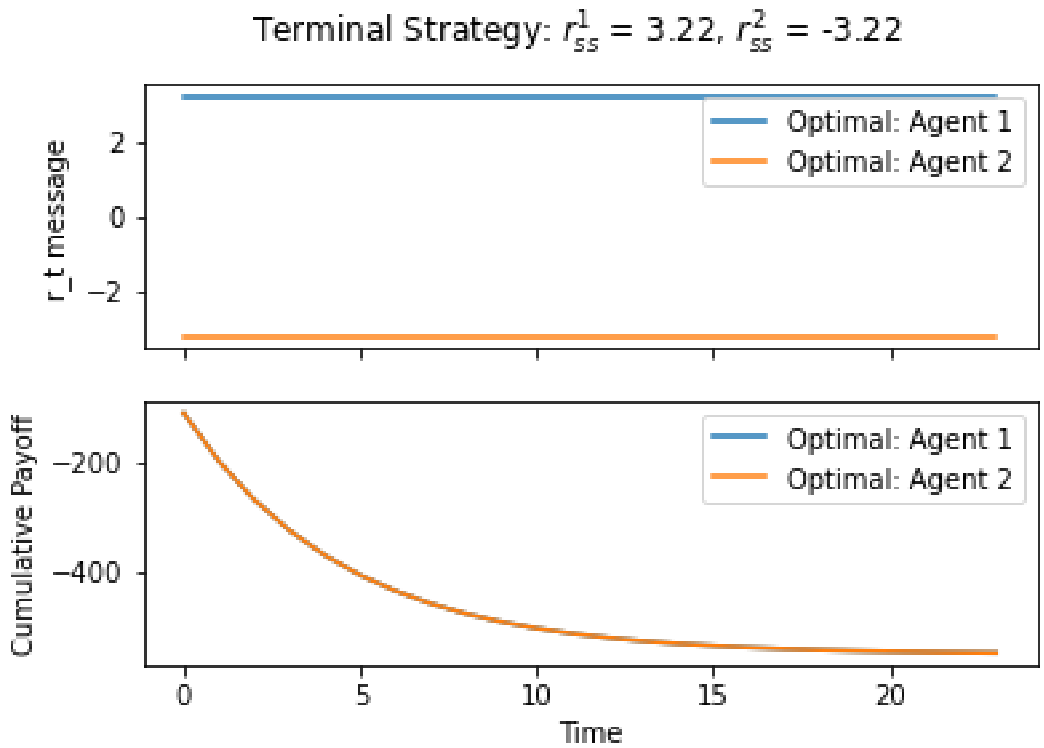
[14]:
```
rs2
```

[14]:
```
defaultdict(list, {0: [array([[3.21812075]])], 1: [array([[-3.21812075]])]})
```

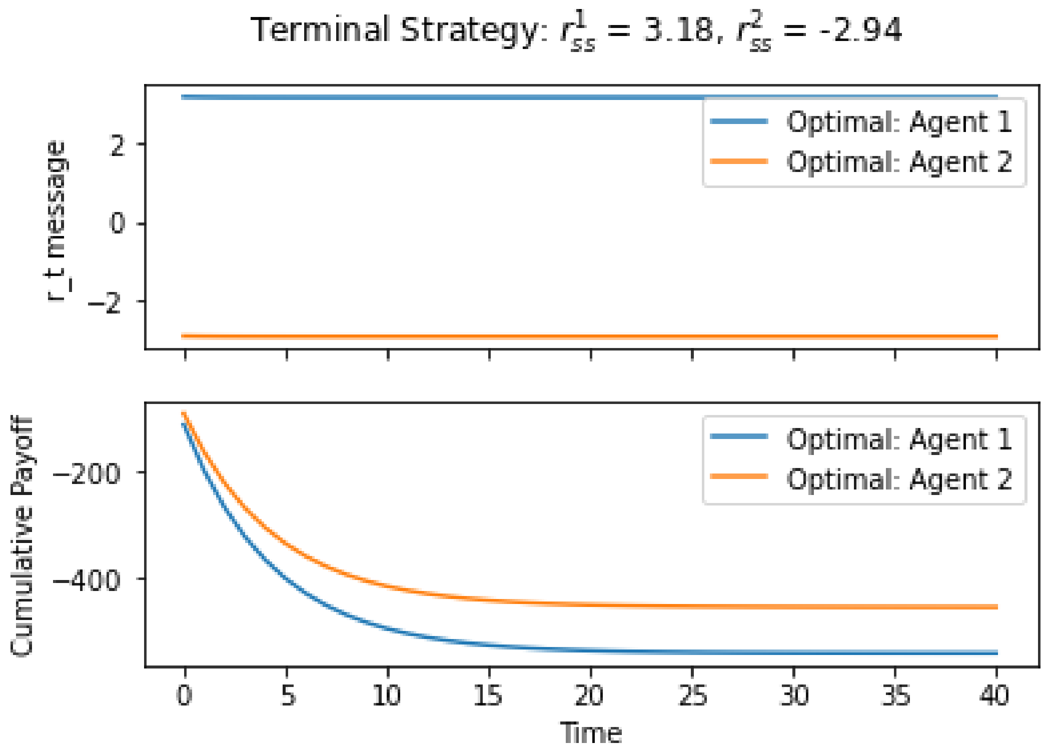
This is a standstill case and the blank graph is not an error. Neither agent has more influence than the other, so the opinion of the naive agent doesn't change, and convergence is immediate.

1.4 Slight deviations from exactly opposite agendas, infinite time

[15]:
```
X_0_1 = np.array([
  0 - 10,
], ndmin = 2).T

X_0_2 = np.array([
  0 + 9.999999,
], ndmin = 2).T
X_0 = [X_0_1, X_0_2]

x = [10, -9.999999]
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)]
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)
xs2, rs2, payoffs2 = optimal(X_0, historical_K, historical_k, historical_kappa)
do_plot(rs2, r, payoffs2, num_agents = 2)
```



[16]:
```
X_0_1 = np.array([
  0 - 10,
], ndmin = 2).T

X_0_2 = np.array([
  0 + 9,
], ndmin = 2).T
X_0 = [X_0_1, X_0_2]

x = [10, -9]
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)]
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)
xs2, rs2, payoffs2 = optimal(X_0, historical_K, historical_k, historical_kappa)
do_plot(rs2, r, payoffs2, num_agents = 2)
```



When shifting one agenda closer to zero, the strategy follows moving closer to zero, and the strategy of the non-shifting agent also moves closer to zero but by a lesser amount.

1.5 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, with the correct pairings:
    base = [[(B[l_prime].T @ K[l_prime] @ B[l]).item() for l in range(L)] 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 l == L - 1 and 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()
```