StrategicInfluence5

July 1, 2021

1 Experimentation with Strategic Influence Network Model, Part 5

1.0.1 Comparative statics through variations in influence structure

James Yu
30 June 2021

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

This time, I have wrapped the utility for generating the optimal strategy into a "function" so I can test it with different inputs:

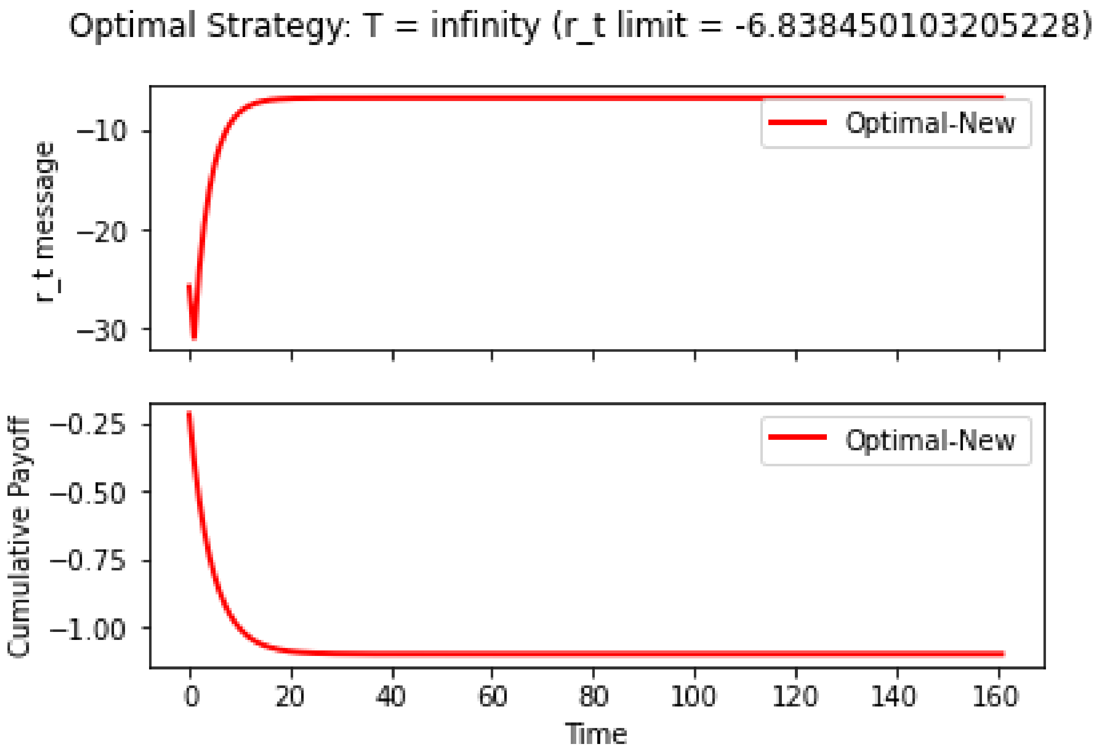
[2]:
```
def optimal_K(z, delta = 0.8):
    A = np.array([
      [0.217,    0.2022,   0.2358,   0.1256,   0.1403],
      [0.8988*0.2497,   0.8988*0.0107,   0.8988*0.2334,   0.8988*0.1282,   0.
↪8988*0.378],
      [0.1285,   0.0907,   0.3185,   0.2507,   0.2116],
      [0.1975,   0.0629,   0.2863,   0.2396,   0.2137],
      [0.1256,   0.0711,   0.0253,   0.2244,   0.5536],
    ], ndmin = 2)
    c = np.array([0, 0.1012, 0, 0, 0,], ndmin = 2).T
    A_tilde = np.concatenate((np.concatenate((A, c), axis = 1), # A c
        np.concatenate((np.zeros((1, 5)), np.array([1], ndmin = 2)), axis =␣
↪1)), # 0 1
        axis = 0)
    B = np.array([0.0791, 0, 0, 0, 0,], ndmin = 2).T
    B_tilde = np.concatenate((B, np.array([0], ndmin = 2)), axis = 0)
    x = np.array([-0.98, -4.62, 2.74, 4.67, 2.15,], ndmin = 2).T
    w_0 = np.concatenate((x, np.array([z], ndmin = 2)), axis = 0)
    Q = 0.2 * np.identity(5)
    Q_tilde = 0.2 * np.identity(6)
    Q_tilde[5, :] = 0

    def L(K_entry):
        return -1 * np.linalg.inv(B_tilde.T @ K_entry @ B_tilde) @ B_tilde.T @␣
↪K_entry @ A_tilde
```

```
    # first compute the sequence of optimal K_t matrices
    K = np.zeros((6, 6))
    K_t = [Q_tilde, K]
    K = Q_tilde
    current_difference = np.inf
    while abs(current_difference) != 0:
        K_new = delta * (A_tilde.T @ (K
                - (K @ B_tilde @ np.linalg.inv(B_tilde.T @ K @ B_tilde) @␣
↪B_tilde.T @ K))
                @ A_tilde) + Q_tilde
        K_t.insert(0, K_new)
        current_difference = np.max(np.abs(K - K_new))
        K = K_new

    # compute the Gamma matrix to use for later computations
    expr = A_tilde + B_tilde @ L(K_t[0])
    A_tilde_n = expr[:5, :5]
    c_nplus1 = np.array(expr[:5, 5], ndmin = 2).T
    x_t = x
    x_ts = [x]

    # compute the resulting sequence of x_t opinion vectors
    for K_ent in K_t:
        x_tp1 = A_tilde_n @ x_t + c_nplus1 * z
        x_ts.append(x_tp1)
        x_t = x_tp1

    # compute the sequence of r_t and cumulative costs
    payoff = 0
    payoffs = []
    r_ts = []
    i = 0
    for x_ent in x_ts:
        r_ts.append(L(K_t[0]) @ np.concatenate((x_ent, np.array([z], ndmin =␣
↪2)), axis = 0))
        payoff += (-1 * delta**i * (x_t.T @ Q @ x_t)).item() # account for␣
↪discounting
        payoffs.append(payoff)
        i += 1

    return r_ts, A_tilde, B_tilde, w_0, K_t, x_ts, payoffs
```

The function can then be used as follows:

[3]:
```
rs, A_tilde_, B_tilde_, w_0_, Ks, xs, ps = optimal_K(10) # this means z = 10
fig, sub = plt.subplots(2, sharex=True)
fig.suptitle(f"Optimal Strategy: T = infinity (r_t limit = {rs[-1].item()})")
```

```
sub[0].plot(range(len(Ks)+1), [a.item() for a in rs], 'r', label =␣
↪"Optimal-New", linewidth=2)
sub[0].set(ylabel = "r_t message")
sub[1].plot(range(len(Ks)+1), ps, 'r', label = "Optimal-New", linewidth=2)
sub[1].set(xlabel = "Time", ylabel =  "Cumulative Payoff")

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



When checking for the amount by which the strategic agent influences naive agents in the long run, there is a specific procedure by which the $\tilde{A}$ matrix needs to be modified, which is based on the steady-state r_{ss} produced by the above function in the long run. For example, in the above graph this is about -6.84. Thus, I have set up another "function" to automatically take that value and use it to generate the resulting limit matrix necessary to do analysis.

[30]:
```
def l_matrix(r_ss, A_tilde, B_tilde, w_0):
    A_tilde_prime = np.concatenate((np.concatenate((A_tilde, B_tilde), axis =␣
↪1), # A c
                    np.concatenate((np.zeros((1, 6)), np.array([1], ndmin =␣
↪2)), axis = 1)), # 0 1
                    axis = 0)
    w_0_prime = np.concatenate((w_0, np.array([r_ss], ndmin = 2)), axis = 0)
    res_mat = np.linalg.matrix_power(A_tilde_prime, 1000000000000)
    res_vec = res_mat @ w_0_prime
    return res_mat, res_vec
```

[32]:
```
_, vec = l_matrix(rs[-1].item(), A_tilde_, B_tilde_, w_0_)
print(vec)
```

```
[[-0.41107847]
 [ 0.96122649]
 [ 0.06976123]
 [ 0.01239086]
 [ 0.04761921]
 [10.        ]
 [-6.8384501 ]]
```

[29]:
```
print(xs[-1])
```

```
[[-0.41107847]
 [ 0.96122649]
 [ 0.06976123]
 [ 0.01239086]
 [ 0.04761921]]
```

This acts as a confirmation benchmark. I can now try reducing z, say, to 8 from 10:

[33]:
```
rs, A_tilde_, B_tilde_, w_0_, _, _, _ = optimal_K(8)
_, vec = l_matrix(rs[-1].item(), A_tilde_, B_tilde_, w_0_)
print(vec)
```

```
[[-0.32886278]
 [ 0.76898119]
 [ 0.05580898]
 [ 0.00991269]
 [ 0.03809537]
 [ 8.        ]
 [-5.47076008]]
```

As expected the opinions became closer.

[34]:
```
rs, A_tilde_, B_tilde_, w_0_, _, _, _ = optimal_K(12)
_, vec = l_matrix(rs[-1].item(), A_tilde_, B_tilde_, w_0_)
print(vec)
```

```
[[-0.49329416]
 [ 1.15347179]
 [ 0.08371347]
 [ 0.01486904]
 [ 0.05714306]
 [12.        ]
 [-8.20614012]]
```

There is a seemingly uniform shift that occurs in the direction of the shift of the opinion of the bot. If for example I bring the bot's agenda all the way to zero, I get:

[35]:
```
rs, A_tilde_, B_tilde_, w_0_, _, _, _ = optimal_K(0)
_, vec = l_matrix(rs[-1].item(), A_tilde_, B_tilde_, w_0_)
print(vec)
```

```
[[-4.44513447e-21]
 [-3.85921245e-21]
 [-4.23983421e-21]
 [-4.26432917e-21]
 [-4.24928801e-21]
 [ 0.00000000e+00]
 [-7.18937766e-21]]
```

It is highly likely that these 10^{-21} values are actually zero (but there is no way to know for sure). Regardless, this is expected.

[36]:
```
rs, A_tilde_, B_tilde_, w_0_, _, _, _ = optimal_K(-10)
_, vec = l_matrix(rs[-1].item(), A_tilde_, B_tilde_, w_0_)
print(vec)
```

```
[[  0.41107847]
 [ -0.96122649]
 [ -0.06976123]
 [ -0.01239086]
 [ -0.04761921]
 [-10.        ]
 [  6.8384501 ]]
```

[37]:
```
xs[-1]
```

[37]:
```
array([[-0.41107847],
       [ 0.96122649],
       [ 0.06976123],
       [ 0.01239086],
       [ 0.04761921]])
```

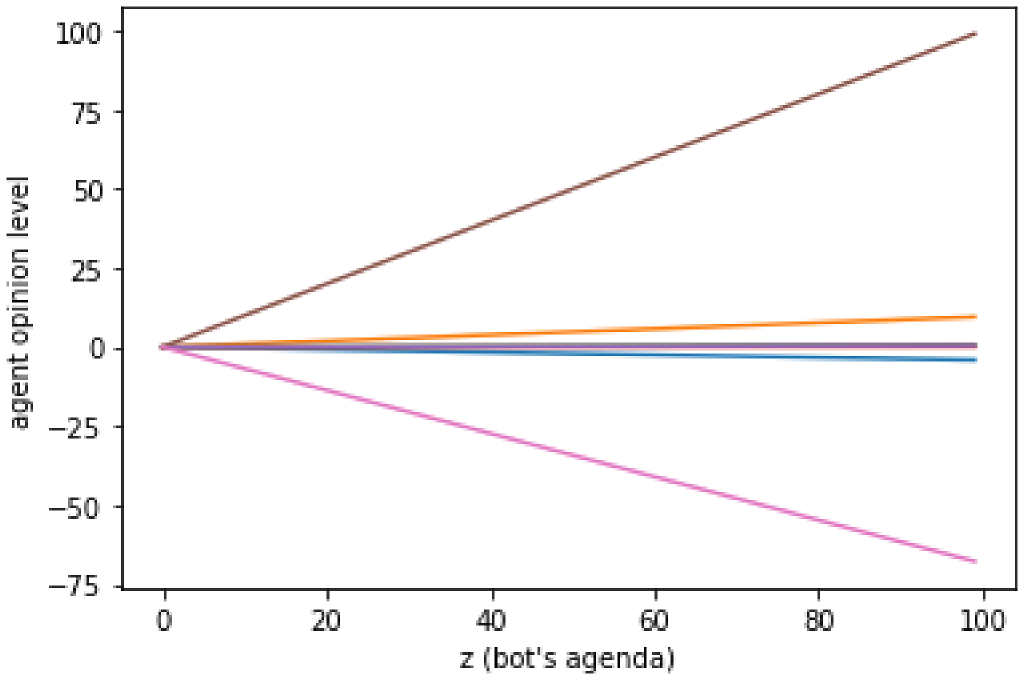
The above `xs[-1]` is that of the original $z = 10$. Note that it is the exact negative of the resulting steady-state opinion vector from the previous cell, meaning that the results are symmetric around zero (which is also expected).

The next question to answer is does there exist a closed form relation between the magnitude of z and the distance of each naive agents' opinions from zero (I predict this to be a different relation for each agent).

[38]:
```
vecs = []
for i in range(100):
    rs, A_tilde_, B_tilde_, w_0_, _, _, _ = optimal_K(i)
    _, vec = l_matrix(rs[-1].item(), A_tilde_, B_tilde_, w_0_)
    vecs.append(vec)
```

I now have a set of 100 samples, from $z = 0$ to $z = 100$. I can therefore individually plot them and compute a regression based on the resulting curve.

[40]:
```
for i in range(len(vecs[0])):
    plt.plot(range(100), [v[i] for v in vecs], )
plt.xlabel("z (bot's agenda)")
plt.ylabel("agent opinion level")
plt.show()
```



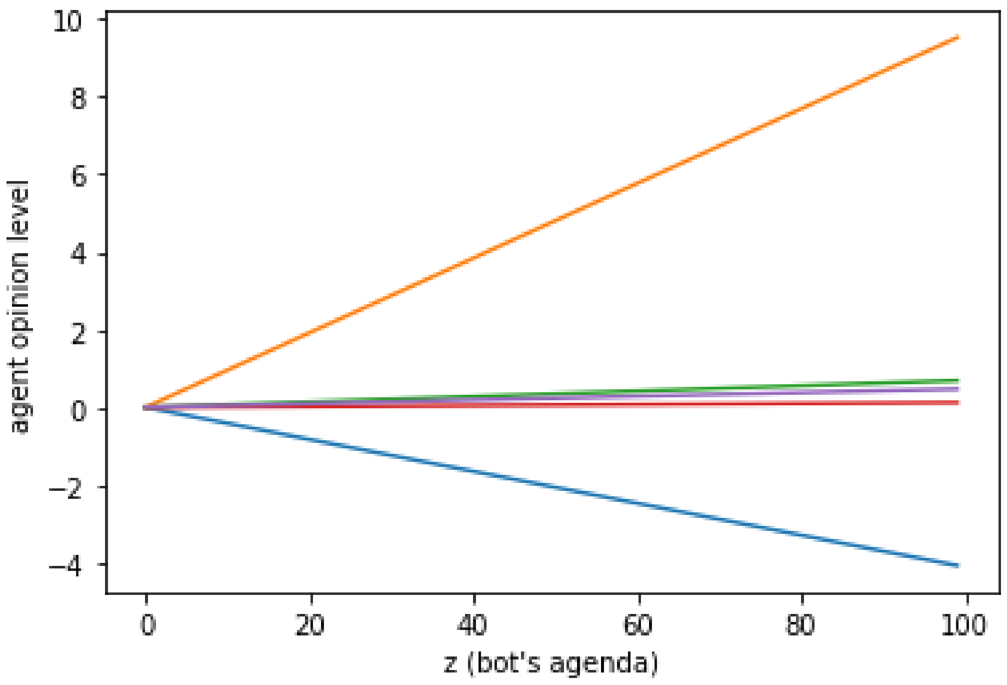
On quick glance they do appear to be linear. I can remove the ones that correspond to fixed agendas as follows:

[46]:
```
print(vecs[10])
```

```
[[-0.41107847]
 [ 0.96122649]
 [ 0.06976123]
 [ 0.01239086]
 [ 0.04761921]
 [10.        ]
 [-6.8384501 ]]
```

Note there are otherwise 5 agents so I simply limit the number of graphs to the first 5 as follows:

[47]:
```
for i in range(5):
    plt.plot(range(100), [v[i] for v in vecs])
plt.xlabel("z (bot's agenda)")
plt.ylabel("agent opinion level")
plt.show()
```



Now I need a linear regression on each of the 5 plots since they appear to be linear. The easiest way to do this is a simple least-squares.

The method I use here is derived from https://numpy.org/doc/stable/reference/generated/numpy.linalg.lstsq.html. This estimates a solution x to the matrix equation $Ax = B$.

In this case, I need a linear regression, which is of the form $y = aX + b$. So not only do I need to solve for coefficients a, I need to solve for coefficients b as well. While these are most likely all going to be zero, recall that the computer presented them as something along the lines of 10^{-21}, so this acts as a fallback.

In this case I am solving for x of $Ax = B$, so x in that equation is actually going to be the a and b from the linear equation. In other words, x is a 1 by 2 matrix (two unknowns). A here contains what I do know, which is the X from the $y = aX + b$ (which is not the same x as $Ax = B$), and it also contains a 1 that comes from the fact that $y = aX + b * 1$.

In simpler terms, suppose I had the datapoints (8, 2) and (3, 4). Then I would need to solve $4 = a * 3 + b$ and $2 = a * 8 + b$ simultaneously.

Using matrix form, this becomes:

$$\begin{bmatrix} 1 & 8 \\ 1 & 3 \end{bmatrix} \begin{bmatrix} b \\ a \end{bmatrix} = \begin{bmatrix} 2 \\ 4 \end{bmatrix}$$

Note that when expanded out, this turns into $b * 1 + a * 8 = 2$ and $b * 1 + a * 3 = 4$ which are the equations I had originally.

Thus, on the computational side, I first need a matrix containing all the x-axis datapoints beside a column of 1s. (This column of 1s is the 1 of the $b * 1$ where b acts as the line's intercept).

[59]:
```
A = np.concatenate((np.ones((100, 1)), np.array(list(range(100)), ndmin = 2).
↪T), axis = 1)
```

Next I need a vector for all the y-axis datapoints. This will depend on the line being regressed, so this can be done in sequence along with the regression itself.

[71]:
```
pairs = []
for i in range(5):
    B = np.array([v[i] for v in vecs], ndmin = 2)
    X, _, _, _ = np.linalg.lstsq(A, B, rcond = None)
    print(f"FOR AGENT {i+1}:")
    print(f"a = {X[1][0]}")
    print(f"b = {X[0][0]}")
    print()
    pairs.append([X[1][0], X[0][0]])
```

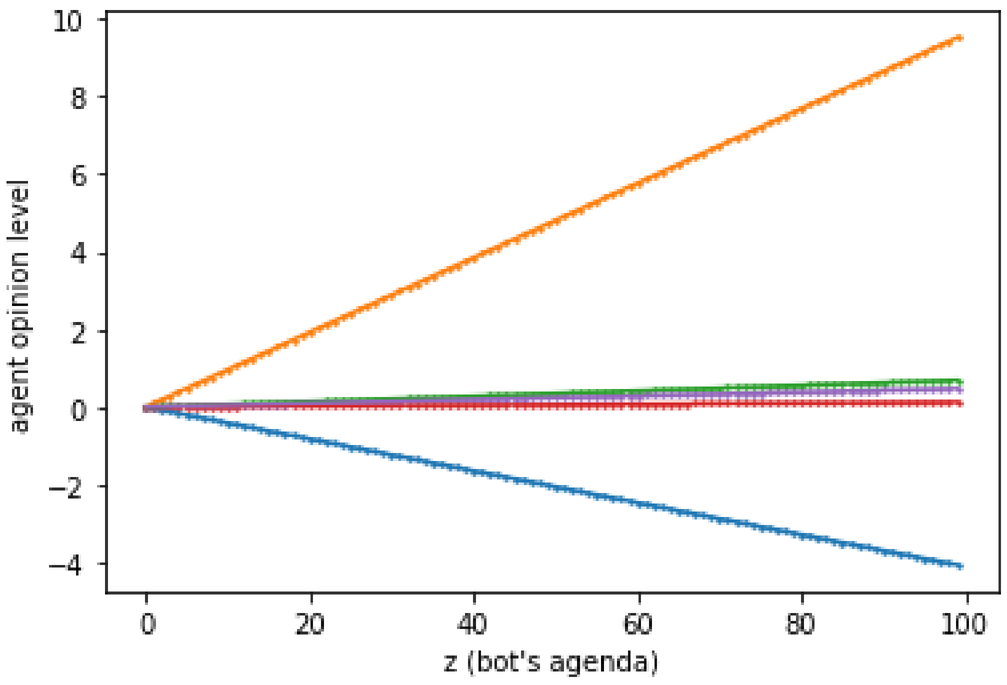
```
FOR AGENT 1:
a = -0.04110784688942589
b = -1.2311390133623404e-15

FOR AGENT 2:
a = 0.09612264926448956
b = 1.923572055349328e-15

FOR AGENT 3:
a = 0.006976122527807187
b = -6.133568664303464e-16

FOR AGENT 4:
a = 0.0012390864725121673
b = -4.738693529791507e-16

FOR AGENT 5:
a = 0.004761921365991428
b = -7.2298061716050045e-16
```

It now occurs to me that the slope of the lines, given that they approximately have an intercept of zero, must be the results when $z = 1$. This actually ends up being:

[73]:
```
vecs[1]
```

[73]:
```
array([[-0.04110785],
       [ 0.09612265],
       [ 0.00697612],
       [ 0.00123909],
       [ 0.00476192],
       [ 1.        ],
        [-0.68384501]])
```

[76]:
```
for i in range(5):
    plt.plot(range(100), vecs[1][i] * np.array(range(100)))
    plt.scatter(range(100), [v[i] for v in vecs], s = 2)
plt.xlabel("z (bot's agenda)")
plt.ylabel("agent opinion level")
plt.show()
```



As can be seen in the diagram above, these are in fact linear. Thus, the distance of each agent's opinion from the opinion of the bot is linear in z with the slopes being the opinions when $z = 1$.

This intuitively makes sense; given any particular limit matrix, which has just two columns at the end, the agents' opinions at optimality would be a linear function of both the bot's agenda and the strategic agent's virtual (r_{ss}) agenda. Note:

[80]:
```
rs, A_tilde_, B_tilde_, w_0_, _, _, _ = optimal_K(10)
mat, vec = l_matrix(rs[-1].item(), A_tilde_, B_tilde_, w_0_)
print(np.round(mat, 5))
```

```
[[0.      0.      0.      0.      0.      0.38171 0.61829]
 [0.      0.      0.      0.      0.      0.46321 0.53679]
 [0.      0.      0.      0.      0.      0.41026 0.58974]
 [0.      0.      0.      0.      0.      0.40686 0.59314]
 [0.      0.      0.      0.      0.      0.40895 0.59105]
 [0.      0.      0.      0.      0.      1.      0.     ]
 [0.      0.      0.      0.      0.      0.      1.     ]]
```

So when I apply a w_0' where the last two entries are that of the bot and the strategic agent, I see that the steady-state opinions are simply linear combinations of the two columns of the above matrix based on those agenda entries.

This means that if I was to change the strategic agent's steady-state r_{ss} to something else, I would see a linear shift as well. This is thus a simple equation of two linear parameters.

Also note that the above matrix, which is the limit matrix, is not dependent on what the original opinions are. It is solely dependent on what the weights are for the A and other flow matrices, and so r_{ss} does not impact the fraction by which agents in the long run devote attention to the strategic agent or not.

What this means is that the distance from the strategic agent's agenda of the opinions of the naive agents is simply dependent linearly on what the long-run dominant agendas are. As an agenda moves in one direction, the opinions follow in that direction with a diminishing factor represented by the slope of the line which arises from:

[81]:
```
vecs[1]
```

[81]:
```
array([[-0.04110785],
       [ 0.09612265],
       [ 0.00697612],
       [ 0.00123909],
       [ 0.00476192],
       [ 1.        ],
       [-0.68384501]])
```

The next question is why these numbers arise.

The first thing to note is that the entirety of these results do not depend on the initial opinions x. Recall there is a simple explanation for this: in the long run, opinions are a function of exclusively the bot and the strategic agent. This means varying w_0 will have no impact on these values.

The next thing to note is that the reason these numbers are not precisely zero is because of how the bot's presence causes a "drown-out" of the strategic agent's broadcasts. In other words, because the bot is flooding the opinions of the naive agents, the broadcasts eventually become noise. So there is a precedent for nonzero opinions but the question becomes what function are they.

Suppose instead of using the setup that has been used for the past 4 notebooks, a setup like the following was used instead:

[120]:
```
A = np.array([
        [0.8988*0.2497,   0.8988*0.0107,   0.8988*0.2334,   0.8988*0.1282,   0.
 ↪8988*0.378],
        [0.8988*0.2497,   0.8988*0.0107,   0.8988*0.2334,   0.8988*0.1282,   0.
 ↪8988*0.378],
        [0.2497,   0.0107,   0.2334,   0.1282,   0.378],
        [0.2497,   0.0107,   0.2334,   0.1282,   0.378],
        [0.2497,   0.0107,   0.2334,   0.1282,   0.378],
    ], ndmin = 2)
c = np.array([0, 0.1012, 0, 0, 0,], ndmin = 2).T
B = np.array([0.1012, 0, 0, 0, 0,], ndmin = 2).T

def optimal_K_dynamic(A, c, B, z, delta = 0.8, tol = 10**(-18)):
    x = np.array([0, 0, 0, 0, 0], ndmin = 2).T
    A_tilde = np.concatenate((np.concatenate((A, c), axis = 1), # A c
        np.concatenate((np.zeros((1, 5)), np.array([1], ndmin = 2)), axis =␣
↪1)), # 0 1
        axis = 0)
    B_tilde = np.concatenate((B, np.array([0], ndmin = 2)), axis = 0)
    w_0 = np.concatenate((x, np.array([z], ndmin = 2)), axis = 0)
    Q = 0.2 * np.identity(5)
    Q_tilde = 0.2 * np.identity(6)
    Q_tilde[5, :] = 0

    def L(K_entry):
        return -1 * np.linalg.inv(B_tilde.T @ K_entry @ B_tilde) @ B_tilde.T @␣
↪K_entry @ A_tilde

    # first compute the sequence of optimal K_t matrices
    K = np.zeros((6, 6))
    K_t = [Q_tilde, K]
    K = Q_tilde
    current_difference = np.inf
    while abs(current_difference) > tol: # to avoid floating point error, don't␣
↪converge all the way to zero (this is standard)
        K_new = delta * (A_tilde.T @ (K
                - (K @ B_tilde @ np.linalg.inv(B_tilde.T @ K @ B_tilde) @␣
↪B_tilde.T @ K))
                @ A_tilde) + Q_tilde
        K_t.insert(0, K_new)
        current_difference = np.max(np.abs(K - K_new))
        K = K_new

    # compute the Gamma matrix to use for later computations
    expr = A_tilde + B_tilde @ L(K_t[0])
    A_tilde_n = expr[:5, :5]
    c_nplus1 = np.array(expr[:5, 5], ndmin = 2).T
    x_t = x
    x_ts = [x]

    # compute the resulting sequence of x_t opinion vectors
    for K_ent in K_t:
        x_tp1 = A_tilde_n @ x_t + c_nplus1 * z
        x_ts.append(x_tp1)
        x_t = x_tp1

    # compute the sequence of r_t and cumulative costs
    payoff = 0
    payoffs = []
    r_ts = []
    i = 0
    for x_ent in x_ts:
        r_ts.append(L(K_t[0]) @ np.concatenate((x_ent, np.array([z], ndmin =␣
↪2)), axis = 0))
        payoff += (-1 * delta**i * (x_t.T @ Q @ x_t)).item() # account for␣
↪discounting
        payoffs.append(payoff)
        i += 1

    return r_ts, A_tilde, B_tilde, w_0, K_t, x_ts, payoffs

rs, A_tilde_, B_tilde_, w_0_, Ks, xs, ps = optimal_K_dynamic(A, c, B, 1)
mat, vec = l_matrix(rs[-1].item(), A_tilde_, B_tilde_, w_0_)
print(np.round(mat, 7))
```

```
[[0.        0.        0.        0.        0.        0.0369323 0.9630677]
 [0.        0.        0.        0.        0.        0.1381323 0.8618677]
 [0.        0.        0.        0.        0.        0.0410906 0.9589094]
 [0.        0.        0.        0.        0.        0.0410906 0.9589094]
 [0.        0.        0.        0.        0.        0.0410906 0.9589094]
 [0.        0.        0.        0.        0.        1.        0.       ]
 [0.        0.        0.        0.        0.        0.        1.       ]]
```

The source of the long decimals in the above limit matrix is twofold. Firstly it comes from the numbers in the limit matrix being long decimals, and secondly it comes from r_{ss} being a long decimal.

[112]:
```
xs[-1]
```

[112]:
```
array([[-0.01852706],
       [ 0.08850065],
       [-0.01412922],
       [-0.01412922],
       [-0.01412922]])
```

With identical agents, I see identical opinions. However, the agent listening to the bot is much further away from zero than the agent listening to the strategic agent. The remaining three naive agents' opinions are actually the closest to zero.

Recall the long-run opinions are a linear combination, based on the fractions in the above matrix, of r_{ss} and z. Thus, opinions are based on weights assigned to the bot and strategic agent. This means the source of variability that would have the most impact is how the fractions come up.

If we use nicer numbers, we get:

[113]:
```
A = np.array([
      [0.2*0.5, 0.2*0.5, 0.2*0.5, 0.2*0.5, 0.2*0.5],
      [0.2*0.5, 0.2*0.5, 0.2*0.5, 0.2*0.5, 0.2*0.5],
      [0.2, 0.2, 0.2, 0.2, 0.2],
      [0.2, 0.2, 0.2, 0.2, 0.2],
      [0.2, 0.2, 0.2, 0.2, 0.2],
    ], ndmin = 2)
c = np.array([0, 0.5, 0, 0, 0,], ndmin = 2).T
B = np.array([0.5, 0, 0, 0, 0,], ndmin = 2).T
rs, A_tilde_, B_tilde_, w_0_, Ks, xs, ps = optimal_K_dynamic(A, c, B, 1)
mat, vec = l_matrix(rs[-1].item(), A_tilde_, B_tilde_, w_0_)
print(np.round(mat, 7))
```

```
[[0.   0.   0.   0.   0.   0.25 0.75]
 [0.   0.   0.   0.   0.   0.75 0.25]
 [0.   0.   0.   0.   0.   0.5  0.5 ]
 [0.   0.   0.   0.   0.   0.5  0.5 ]
 [0.   0.   0.   0.   0.   0.5  0.5 ]
 [0.   0.   0.   0.   0.   1.   0.  ]
 [0.   0.   0.   0.   0.   0.   1.  ]]
```

This is now easier to parse. I made all the agents identical, and made them listen to themselves identically. I then had half of agents one and two dedicated to their respective static-agenda agents (the bot and the strategic agent). This gives nicer numbers in the limit matrix.

For the agents that listen to nobody, there is equal weighting to the bot and the strategic agent because there is equal weighting to the agents that feed this agent. This provides a possible explanation to the numbers found previously. For the first two agents, there is a 3/4 split in favour of whomever the agent listens to. Note that 0.5 * 0.5 = 0.25 representing the shift of half the agent's influence. If I change this to not be half and rather be 1/4, we get:

[114]:
```
A = np.array([
        [0.2*0.75, 0.2*0.75, 0.2*0.75, 0.2*0.75, 0.2*0.75],
        [0.2*0.75, 0.2*0.75, 0.2*0.75, 0.2*0.75, 0.2*0.75],
        [0.2, 0.2, 0.2, 0.2, 0.2],
        [0.2, 0.2, 0.2, 0.2, 0.2],
        [0.2, 0.2, 0.2, 0.2, 0.2],
    ], ndmin = 2)
c = np.array([0, 0.25, 0, 0, 0,], ndmin = 2).T
B = np.array([0.25, 0, 0, 0, 0,], ndmin = 2).T
rs, A_tilde_, B_tilde_, w_0_, Ks, xs, ps = optimal_K_dynamic(A, c, B, 1)
mat, vec = l_matrix(rs[-1].item(), A_tilde_, B_tilde_, w_0_)
print(np.round(mat, 7))
```

```
[[0.    0.    0.    0.    0.    0.375 0.625]
 [0.    0.    0.    0.    0.    0.625 0.375]
 [0.    0.    0.    0.    0.    0.5   0.5  ]
 [0.    0.    0.    0.    0.    0.5   0.5  ]
 [0.    0.    0.    0.    0.    0.5   0.5  ]
 [0.    0.    0.    0.    0.    1.    0.   ]
 [0.    0.    0.    0.    0.    0.    1.   ]]
```

Now the weight assigned to the "representative broadcaster" is 1 - (0.75 * 0.5) = 0.625. This is equal to 0.5 * 1.25 and 1.25 is bigger than any number on here meaning the driving force is the 0.75 * 0.5 = 0.375 dedicated to not-the-broadcaster. I make this more apparent as follows:

[115]:
```
A = np.array([
      [0.2, 0.2, 0.2, 0.2, 0],
      [0.2, 0.2, 0.2, 0.2, 0],
      [0.2, 0.2, 0.2, 0.2, 0.2],
      [0.2, 0.2, 0.2, 0.2, 0.2],
      [0.2, 0.2, 0.2, 0.2, 0.2],
    ], ndmin = 2)
c = np.array([0, 0.2, 0, 0, 0,], ndmin = 2).T
B = np.array([0.2, 0, 0, 0, 0,], ndmin = 2).T
rs, A_tilde_, B_tilde_, w_0_, Ks, xs, ps = optimal_K_dynamic(A, c, B, 1)
mat, vec = l_matrix(rs[-1].item(), A_tilde_, B_tilde_, w_0_)
print(np.round(mat, 7))
```

```
[[0.  0.  0.  0.  0.  0.4 0.6]
 [0.  0.  0.  0.  0.  0.6 0.4]
 [0.  0.  0.  0.  0.  0.5 0.5]
 [0.  0.  0.  0.  0.  0.5 0.5]
 [0.  0.  0.  0.  0.  0.5 0.5]
 [0.  0.  0.  0.  0.  1.  0. ]
 [0.  0.  0.  0.  0.  0.  1. ]]
```

Now the first two agents trade off listening to the fifth agent for listening to a broadcaster. The split becomes 0.4 vs 0.6.

[117]:
```
A = np.array([
      [0.2, 0.2, 0.2, 0.1, 0],
      [0.2, 0.2, 0.2, 0.1, 0],
      [0.2, 0.2, 0.2, 0.2, 0.2],
      [0.2, 0.2, 0.2, 0.2, 0.2],
      [0.2, 0.2, 0.2, 0.2, 0.2],
    ], ndmin = 2)
c = np.array([0, 0.3, 0, 0, 0,], ndmin = 2).T
B = np.array([0.3, 0, 0, 0, 0,], ndmin = 2).T
rs, A_tilde_, B_tilde_, w_0_, Ks, xs, ps = optimal_K_dynamic(A, c, B, 1)
mat, vec = l_matrix(rs[-1].item(), A_tilde_, B_tilde_, w_0_)
print(np.round(mat, 7))
```

```
[[0.   0.   0.   0.   0.   0.35 0.65]
 [0.   0.   0.   0.   0.   0.65 0.35]
 [0.   0.   0.   0.   0.   0.5  0.5 ]
 [0.   0.   0.   0.   0.   0.5  0.5 ]
 [0.   0.   0.   0.   0.   0.5  0.5 ]
 [0.   0.   0.   0.   0.   1.   0.  ]
 [0.   0.   0.   0.   0.   0.   1.  ]]
```

[118]:
```
A = np.array([
      [0.2, 0.2, 0.2, 0.3, 0],
      [0.2, 0.2, 0.2, 0.3, 0],
        [0.2, 0.2, 0.2, 0.2, 0.2],
        [0.2, 0.2, 0.2, 0.2, 0.2],
        [0.2, 0.2, 0.2, 0.2, 0.2],
    ], ndmin = 2)
c = np.array([0, 0.1, 0, 0, 0,], ndmin = 2).T
B = np.array([0.1, 0, 0, 0, 0,], ndmin = 2).T
rs, A_tilde_, B_tilde_, w_0_, Ks, xs, ps = optimal_K_dynamic(A, c, B, 1)
mat, vec = l_matrix(rs[-1].item(), A_tilde_, B_tilde_, w_0_)
print(np.round(mat, 7))
```

```
[[0.   0.   0.   0.   0.   0.45 0.55]
 [0.   0.   0.   0.   0.   0.55 0.45]
 [0.   0.   0.   0.   0.   0.5  0.5 ]
 [0.   0.   0.   0.   0.   0.5  0.5 ]
 [0.   0.   0.   0.   0.   0.5  0.5 ]
 [0.   0.   0.   0.   0.   1.   0.  ]
 [0.   0.   0.   0.   0.   0.   1.  ]]
```

From observation the magnitude of the entries in the upper-right corner appear to shift by 0.05 for every 0.1 shift in the share of influence dedicated to the broadcasters. This is consistent with what would happen if there were no share, which would result in a balanced 0.45+0.05 = 0.5, and 0.55 - 0.05 = 0.5.

Thus, there is some sort of a relationship between long-run share of influence (limit matrix) and short-run share of influence (fractions in A, c and B).

[124]:
```
A = np.array([
        [0.2, 0.2, 0.2, 0.2, 0],
        [0.2, 0.2, 0.2, 0.2, 0],
        [0.6, 0.4, 0, 0, 0],
        [0.4, 0.6, 0, 0, 0],
        [0.6, 0.4, 0, 0, 0],
    ], ndmin = 2)
c = np.array([0, 0.2, 0, 0, 0,], ndmin = 2).T
B = np.array([0.2, 0, 0, 0, 0,], ndmin = 2).T
rs, A_tilde_, B_tilde_, w_0_, Ks, xs, ps = optimal_K_dynamic(A, c, B, 1, tol =␣
 ↪10**(-14))
mat, vec = l_matrix(rs[-1].item(), A_tilde_, B_tilde_, w_0_)
print(np.round(mat, 7))
```

```
[[0.   0.   0.   0.   0.   0.4  0.6 ]
 [0.   0.   0.   0.   0.   0.6  0.4 ]
 [0.   0.   0.   0.   0.   0.48 0.52]
 [0.   0.   0.   0.   0.   0.52 0.48]
 [0.   0.   0.   0.   0.   0.48 0.52]
 [0.   0.   0.   0.   0.   1.   0.  ]
 [0.   0.   0.   0.   0.   0.   1.  ]]
```

[126]:
```
A = np.array([
      [0.2, 0.2, 0.1, 0.1, 0.1],
      [0.2, 0.2, 0.1, 0.1, 0.1],
      [0.6, 0.4, 0, 0, 0],
      [0.4, 0.6, 0, 0, 0],
      [0.6, 0.4, 0, 0, 0],
    ], ndmin = 2)
c = np.array([0, 0.3, 0, 0, 0,], ndmin = 2).T
B = np.array([0.3, 0, 0, 0, 0,], ndmin = 2).T
rs, A_tilde_, B_tilde_, w_0_, Ks, xs, ps = optimal_K_dynamic(A, c, B, 1, tol =␣
 ↪10**(-14))
mat, vec = l_matrix(rs[-1].item(), A_tilde_, B_tilde_, w_0_)
print(np.round(mat, 7))
```

```
[[0.   0.   0.   0.   0.   0.34 0.66]
 [0.   0.   0.   0.   0.   0.64 0.36]
 [0.   0.   0.   0.   0.   0.46 0.54]
 [0.   0.   0.   0.   0.   0.52 0.48]
 [0.   0.   0.   0.   0.   0.46 0.54]
 [0.   0.   0.   0.   0.   1.   0.  ]
 [0.   0.   0.   0.   0.   0.   1.  ]]
```

[128]:
```
A = np.array([
      [0.2, 0.2, 0.3, 0, 0],
      [0.2, 0.2, 0, 0.3, 0],
      [0.6, 0.4, 0, 0, 0],
      [0.4, 0.6, 0, 0, 0],
      [0.6, 0.4, 0, 0, 0],
    ], ndmin = 2)
c = np.array([0, 0.3, 0, 0, 0,], ndmin = 2).T
B = np.array([0.3, 0, 0, 0, 0,], ndmin = 2).T
rs, A_tilde_, B_tilde_, w_0_, Ks, xs, ps = optimal_K_dynamic(A, c, B, 1, tol =␣
 ↪10**(-14))
mat, vec = l_matrix(rs[-1].item(), A_tilde_, B_tilde_, w_0_)
print(np.round(mat, 7))
```

```
[[0.        0.        0.        0.        0.        0.3404255 0.6595745]
 [0.        0.        0.        0.        0.        0.6595745 0.3404255]
 [0.        0.        0.        0.        0.        0.4680851 0.5319149]
 [0.        0.        0.        0.        0.        0.5319149 0.4680851]
 [0.        0.        0.        0.        0.        0.4680851 0.5319149]
 [0.        0.        0.        0.        0.        1.        0.       ]
 [0.        0.        0.        0.        0.        0.        1.       ]]
```

There is some effect as to having agents 1 and 2 listen strongly to the other naive agents, as can be seen above.