Homework2

December 21, 2020

1 Course : Network Dynamics and Learning

1.0.1 Author

Davide Bussone s276486

1.0.2 Final results compared with:

Antonio Dimitris Defonte

Alessandro Nicolì

Erich Malan

Filippo Cortese

Pedro Hernandez

2 EXERCISE 1

3 PRELIMINARY ELEMENTS

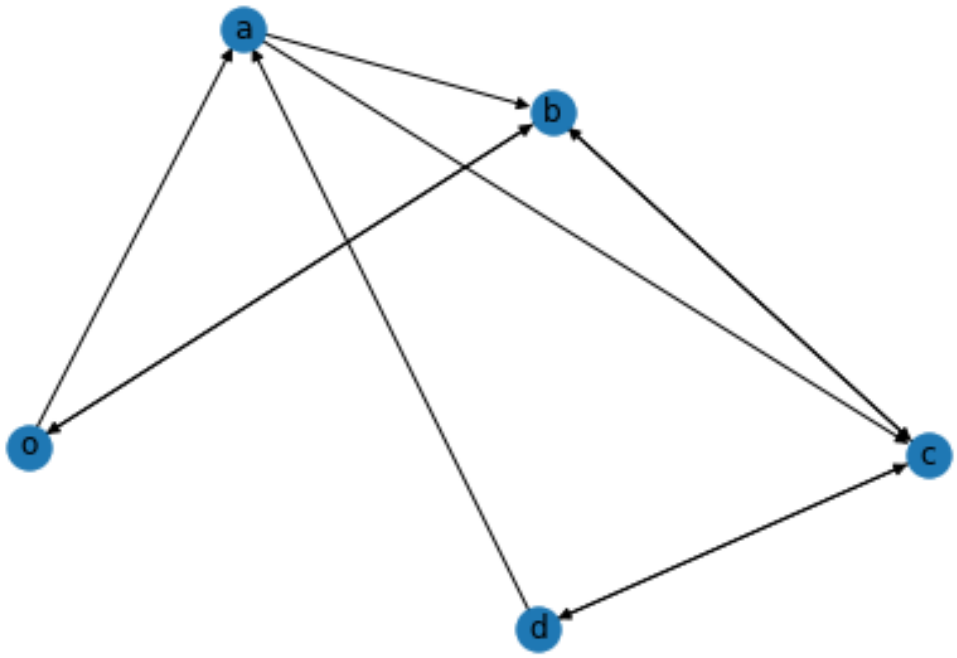
[90]:
```
import numpy as np
import matplotlib.pyplot as plt
import networkx as nx
from numpy.random import choice,random
```

[91]:
```
np.random.seed(0)

lamda = np.array([[0,2/5,1/5,0,0],
                  [0,0,3/4,1/4,0],
                  [1/2,0,0,1/2,0],
                  [0,0,1/3,0,2/3],
                  [0,1/3,0,1/3,0]])
lamda
```

[91]:
```
array([[0.        , 0.4       , 0.2       , 0.        , 0.        ],
       [0.        , 0.        , 0.75      , 0.25      , 0.        ],
       [0.5       , 0.        , 0.        , 0.5       , 0.        ],
       [0.        , 0.        , 0.33333333, 0.        , 0.66666667],
       [0.        , 0.33333333, 0.        , 0.33333333, 0.        ]])
```

[92]:
```
edges = []
nodes = [x for x in "oabcd"]
for r,row in enumerate(lamda):
    for c,col in enumerate(row):
        edges.append((nodes[r],nodes[c],col))
```

[93]:
```
G = nx.DiGraph()
G.add_nodes_from(nodes)
G.add_weighted_edges_from(edges)
edges_removed = []
for n, nbrs in G.adj.items():
    for nbr, eattr in nbrs.items():
        wt = eattr['weight']
        if not wt > 0.0:
            edges_removed.append((n,nbr,wt))
G.remove_edges_from(edges_removed)

pos = nx.layout.spring_layout(G)

nx.draw(G,pos,with_labels=True)
```



[94]:
```
W=lamda
degrees = np.sum(W,axis=1)
D = np.diag(degrees)
P = np.linalg.inv(D) @ W
w = np.sum(W, axis=1)
w_star = np.max(w)
Q = lamda/w_star
Q = Q + np.diag(np.ones(len(w))-np.sum(Q,axis=1))
Q_cum = np.cumsum(Q, axis=1)
```

4 DEFINITION OF RANDOM WALK AND AVERAGE COMPUTATION

1. RandomWalk's tasks

* Take the starting node
* Choose and save the successive node, according to probability matrix Q
* Store the time instants of all steps (t_next in the code)

2. averageTime's tasks

* Compute the random walk on a fixed number of steps

* Take the summation of the transition times
* Divide this sum by the number of steps

[101]:
```
def RandomWalk(G, start, end, Q, nodes, w):
    nodeSeq = [start]
    transition_times = [0,0]

    while True:

        index = np.argwhere(Q[nodes.index(start)] > np.random.rand())[0][0]
        start = nodes[index]

        t_next = - np.log(np.random.rand()) / w

        nodeSeq.append(start)

        t_next += transition_times[-1]

        transition_times.append(t_next)

        if start == end:
            return nodeSeq, transition_times
```

[102]:
```
def averageTime(start,end,num_steps,w):
    sum = 0

    for _ in range(num_steps):
        _, transition_times = RandomWalk(G,start,end,Q_cum, nodes, w)
        sum += transition_times[-1]

    return sum / num_steps # average return time
```

5 POINT A

The considered simulation has “a” acting as starting node and again “a” as ending node. The number of steps was set to 1000, a reasonable number to get an equilibrium between precision and computational time issues.

Approximatively, 6.55 is the result of this simulation.

[103]:
```
simulation=averageTime("a","a",1000,w_star)
print(simulation)
```

```
6.5495121658791025
```

6 POINT B

7 THEORIC EXPECTED RETURN TIME

Now, the previous obtained result is compared with the expected return time computed analytically: $E_i[T_a^+] = t_i, \forall i$

To do that, a linear system must be solved: $E_i[T_i^+] = \frac{1}{\omega_i} + \sum_{j \in V} P_{ij} t_j$ where, i is equal to "a" in this case, while j is the rest of node.

The code below represents this computation.

The resulting expected return time is 6.75, so I can say that the simulation is approximatively correct, since the error is quite low.

[104]:
```
n_nodes = G.number_of_nodes()

# Define the set S and the remaining nodes R
S = [1]
R = [node for node in range(n_nodes) if node not in S]

# Restrict P to R x R to obtain hat(P)
hatP = P[np.ix_(R, R)]
hatw= w[np.ix_(R)]
# np.linalg.solve solves a linear matrix equation given
hatx = np.linalg.solve((np.identity(n_nodes-len(S))-hatP),np.
↪ones(n_nodes-len(S))/hatw)

# define the hitting times to the set S
# hitting time is 0 if the starting node is in S
hitting_s = np.zeros(n_nodes)
# hitting time is hat(x) for nodes in R
hitting_s[R] = hatx
```

[105]:
```
expected_theoric = 1 + P[1,:] @ hitting_s
expected_theoric
```

[105]: 6.75

[106]:
```
print("Error simulation",abs(simulation - expected_theoric))
```

```
Error simulation 0.20048783412089755
```

8 POINT C

The considered simulation has "o" acting as starting node and "d" as ending node. The number of steps was set to 1000, a reasonable number to get an equilibrium between precision and

computational time issues.

Approximatively, 9.02 is the result of this simulation.

[107]:
```
simulation=averageTime("o","d",1000,w_star)
print(simulation)
```

```
9.021624406415983
```

9 POINT D

Now, same as before,the previous obtained result is compared with the expected return time computed analytically.

The code below represents this computation.

The resulting expected return time is 8.79, so I can say that the simulation is approximatively correct, since the error is quite low.

10 THEORIC EXPECTED RETURN TIME

[108]:
```
n_nodes = G.number_of_nodes()
S = [4]
R = [node for node in range(n_nodes) if node not in S]
hatP = P[np.ix_(R, R)]
hatw = w[np.ix_(R)]
hatx = np.linalg.solve((np.identity(n_nodes-len(S))-hatP),(np.
 ↪ones(n_nodes-len(S))/hatw))
hitting_s = np.zeros(n_nodes)
hitting_s[R] = hatx
expected_od=hitting_s[0]
print(expected_od)
```

```
8.785714285714285
```

[109]:
```
print("Error simulation",abs(simulation - expected_od))
```

```
Error simulation 0.23591012070169803
```

11 EXERCISE 2

12 PARTICLE PERSPECTIVE

MOVING PARTICLES INDIVIDUALLY Attachment of a Poisson clock to each of the particles and move them individually, as done for the previous section.

[110]:
```
particles = np.array(range(100)).astype(np.int)


simulation=averageTime("a","a",1000 * len(particles),w_star)
print(simulation)
```

```
6.735474333664576
```

Since the analytical expected return time is the same as before, in this case I expect a higher precision, because in the system there are 100 particles, each one composed by a Poisson clock.

In fact, as expected, the error on the simulation is definitively lower than point 1a.

[112]:
```
print("Error simulation",abs(simulation - expected_theoric))
```

```
Error simulation 0.01452566633542407
```

USING GLOBAL RATE I consider a single system-wide Poisson clock with rate equal to the number of particles (in this case 100). This single particle is then moved to a neighbor node according to probability matrix Q.

Actually, the error is higher than before, but I gained in terms of computational speed.

[113]:
```
rate=len(particles)
Q = lamda/rate
Q = Q + np.diag(np.ones(len(w))-np.sum(Q,axis=1))
Q_cum = np.cumsum(Q, axis=1)
```

[121]:
```
simulation=averageTime("a","a",1000,rate)*rate
print(simulation)
```

```
6.4562654731085525
```

[122]:
```
print("Error simulation",abs(simulation - expected_theoric))
```

```
Error simulation 0.2937345268914475
```

13 NODE PERSPECTIVE

1. computeBegin's tasks

```
* Take the particles of a node in the list of nodes
* Extract a random number in [0,1] and compare it with the cumulative sums Q_cum.
* Pick the first (smallest) state for which
the cumulative sum is greater than the random number.
* Retrieve the node (the starting node for the simulation)
```

2. computeEnd's tasks

```
* Take the index of the neighbor node in which a particle should move, according to Q
* Retrieve the node associated to this index
```

3. makeSimulation's tasks

```
* Take the number of particles, the sequence of nodes
and transition times in the initial condition
* Select the starting and ending node, with the help of the previous functions
* Compute the time between two consecutive ticks and store it in the transition times list
* Remove a particle from the starting node and add a particle to the ending node
(if I start from "o" and I arrive to "a",
"o" will have 99 particles, while "a"'ll obtain the first  particle).
* I exit from the simulation, whene I exceed 60 units of time
```

4. computeAverage's tasks

```
* Perform the simulation on a fixed number of steps (100 in this case,
equal to the number of particles)
* For each node, sum the particles in it
* Divide the obtained result by this summation
```

[123]:
```
W=lamda
degrees = np.sum(W,axis=1)
D = np.diag(degrees)
P = np.linalg.inv(D) @ W
w = np.sum(W, axis=1)
w_star = np.max(w)
Q = lamda/w_star
Q = Q + np.diag(np.ones(len(w))-np.sum(Q,axis=1))
Q_cum = np.cumsum(Q, axis=1)
```

[124]:
```
initialPhase = {'o': 100, 'a': 0, 'b': 0, 'c': 0, 'd': 0}
```

[125]:
```
def computeBegin(G,nodes,tot_particles):
    num_particles = []

    for node in nodes:
        num_particles.append(G.nodes[node]['particles'])

    num_particles = np.array(num_particles)
    num_particles_cum = np.cumsum(num_particles)

    index = np.argwhere(num_particles_cum > np.random.rand() *␣
↪tot_particles)[0][0]

    starting_node = nodes[index]
```

```
    return starting_node
```

[126]:
```
def computeEnd (G,Q_cum,nodes,selected_node):
    index = np.argwhere(Q_cum[nodes.index(selected_node)] > np.random.
 ↪rand())[0][0]

    end_node = nodes[index]

    return end_node
```

[127]:
```
def makeSimulation(G,Q_cum,nodes,units,initialPhase):
    tot_particles = 0
    transition_times = [0.0]
    nodeSeq = {}

    for node in nodes:
        nodeSeq[node] = [initialPhase[node]]
        G.nodes[node]['particles'] = initialPhase[node]
        tot_particles += initialPhase[node]


    while True:

        starting_node = computeBegin(G,nodes,tot_particles)

        ending_node = computeEnd (G,Q_cum,nodes,starting_node)
        t_next = transition_times[-1] - np.log(np.random.rand()) / tot_particles

        transition_times.append(t_next)

        G.nodes[starting_node]['particles'] -= 1
        G.nodes[ending_node]['particles'] += 1

        for node in nodes:
            nodeSeq[node].append(G.nodes[node]['particles'])
        if t_next > units:
            break

    return nodeSeq,transition_times
```

[128]:
```
def computeAverage(G,Q_cum,nodes,steps,units,initialPhase):
    sum_particles = {}
```

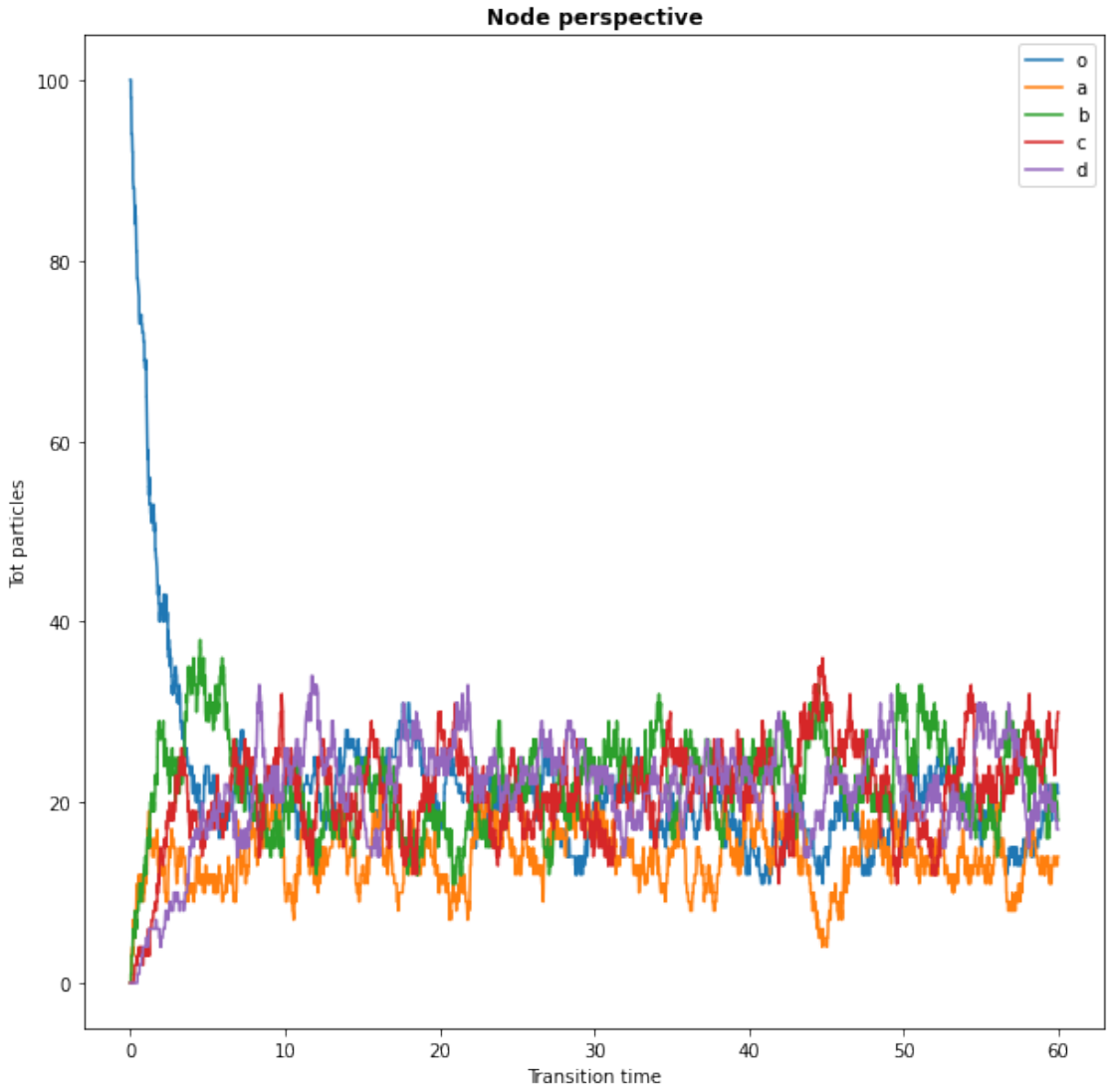
```
    for node in nodes:
        sum_particles[node] = 0
    for count in range(steps):
        nodeSeq,transition_times =␣
↪makeSimulation(G,Q_cum,nodes,units,initialPhase)

        for node in nodes:
            sum_particles[node] += nodeSeq[node][-1]
    avg_particles={}
    for node in nodes:
        avg_particles[node] = sum_particles[node] / steps

    return avg_particles, nodeSeq, transition_times
```

[129]:
```
avg_particles, nodeSeq, transition_times = computeAverage(G, Q_cum,
        nodes, 100, 60, initialPhase)

print(avg_particles)
```

```
{'o': 19.05, 'a': 14.41, 'b': 22.77, 'c': 22.62, 'd': 21.15}
```

A line plot is the most suitable to represent an evolution during time. So, it was chosen to represent how the number of particles change over the transition times. Each different colour represents a node, in this way I have a clear idea of how many particles remain in a node. As expected, at the beginning node “o” has many particles, but then the averages will become very similar.

[130]:
```
fig, ax = plt.subplots(figsize=(10,10))
for node in nodes:
    ax.plot(transition_times, nodeSeq[node], label=node)


ax.set_title('Node perspective',fontweight='bold')
ax.set_xlabel('Transition time')
ax.set_ylabel('Tot particles')
ax.legend();
plt.savefig("ParticlesSimulation.png")
```



14 Exercise 3

15 Proportional rate

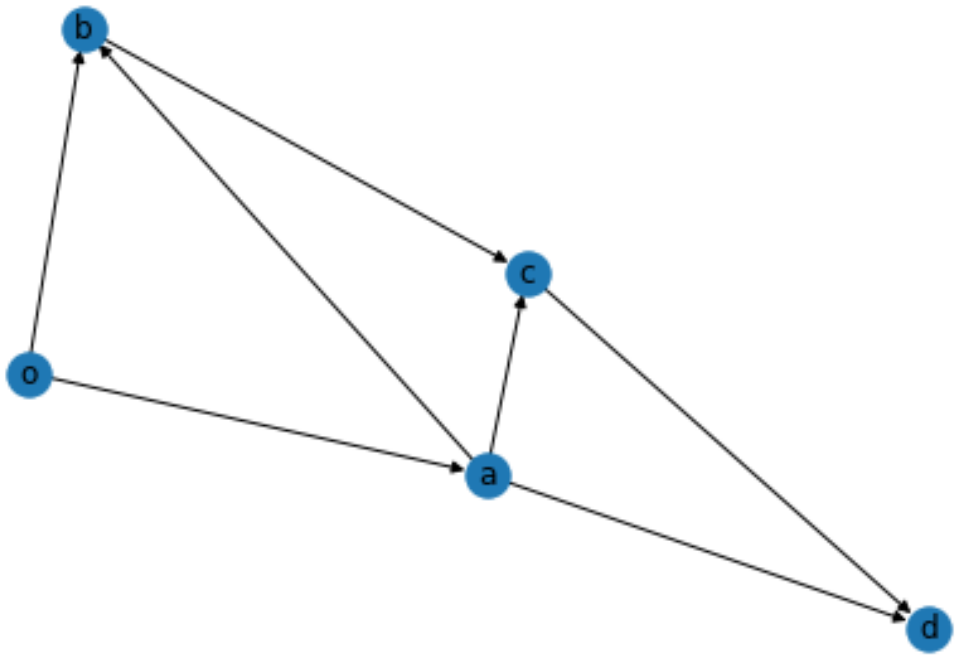
[136]:
```
np.random.seed(0)

lamda = np.array([[0,2/3,1/3,0,0],
                  [0,0,1/4,1/4,2/4],
                  [0,0,0,1,0],
                  [0,0,0,0,1],
                  [0,0,0,0,0]])
```

```
lamda
```

[136]:
```
array([[0.        , 0.66666667, 0.33333333, 0.        , 0.        ],
       [0.        , 0.        , 0.25      , 0.25      , 0.5       ],
       [0.        , 0.        , 0.        , 1.        , 0.        ],
       [0.        , 0.        , 0.        , 0.        , 1.        ],
       [0.        , 0.        , 0.        , 0.        , 0.        ]])
```

[137]:
```
edges = []
nodes = [x for x in "oabcd"]
for r,row in enumerate(lamda):
    for c,col in enumerate(row):
        edges.append((nodes[r],nodes[c],col))
```

[138]:
```
G = nx.DiGraph()
G.add_nodes_from(nodes)
G.add_weighted_edges_from(edges)
removed_edges = []
for n, nbrs in G.adj.items():
    for nbr, eattr in nbrs.items():
        wt = eattr['weight']
        if not wt > 0.0:
            removed_edges.append((n,nbr,wt))
G.remove_edges_from(removed_edges)

pos = nx.layout.spring_layout(G)

nx.draw(G,pos,with_labels=True)
```



[139]:
```
W=lamda
degrees = np.sum(W,axis=1)
D = np.diag(degrees)
w = np.sum(W, axis=1)
w_star = np.max(w)
Q = lamda/w_star
Q = Q + np.diag(np.ones(len(w))-np.sum(Q,axis=1))
Q_cum = np.cumsum(Q, axis=1)
```

[140]:
```
initialPhase = {'o': 0, 'a': 0, 'b': 0, 'c': 0, 'd': 0}
```

The functions are similar as previous section. In the proportional rate, each node will move across particles according to a Poisson process with a rate proportional to the number of particles, so the rate will change during the iterations and all functions keep in mind this concept.

Moreover, node 'd' does not have a node where to send its particles, so I handle this issue (for both proportional and fixed rate) by removing a particle, when the node 'd' is considered

[141]:
```
def computeBegin(G,nodes,rate):
    tot_particles = rate
    rate_in_node = [rate]

    for node in nodes:
        rate_in_node.append(G.nodes[node]['particles'])
```

```
        tot_particles += G.nodes[node]['particles']

    rate_in_node = np.array(rate_in_node)
    rate_in_node_cum = np.cumsum(rate_in_node)

    index = np.argwhere(rate_in_node_cum > np.random.rand() *␣
↪tot_particles)[0][0] - 1

    if index == -1:
        starting_node = -1
    else:
        starting_node = nodes[index]

    return starting_node, tot_particles
```

[142]:
```
def computeEnd (G,Q_cum,nodes,selected_node):

    index = np.argwhere(Q_cum[nodes.index(selected_node)] > np.random.
↪rand())[0][0]

    end_node = nodes[index]

    return end_node
```

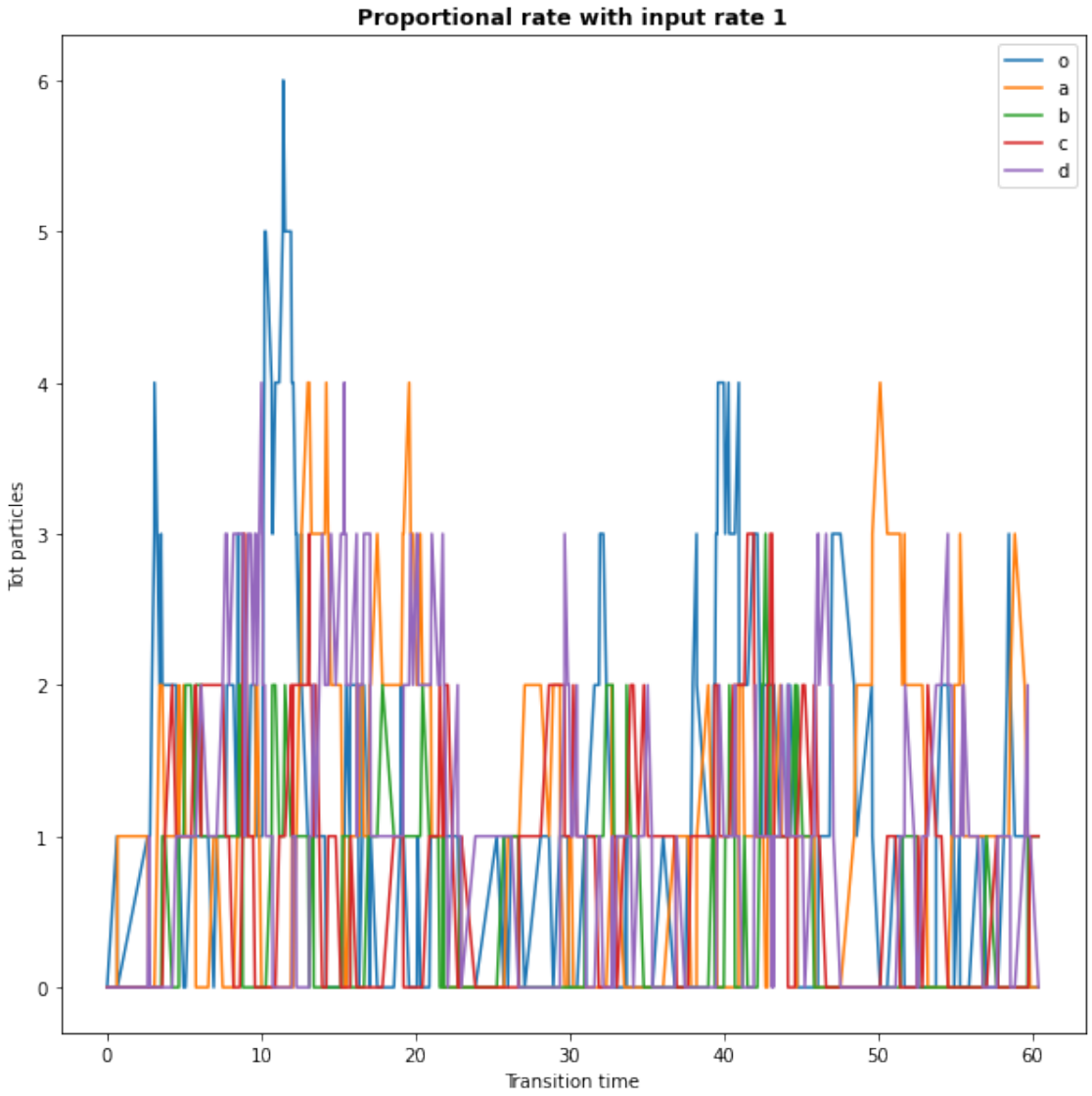
[144]:
```
def makeSimulation(G,Q_cum,nodes,rate,units,initialPhase):
    #tot_particles = 0
    transition_times = [0.0]
    nodeSeq = {}

    for node in nodes:
        nodeSeq[node] = [initialPhase[node]]
        G.nodes[node]['particles'] = initialPhase[node]
        #tot_particles += initialPhase[node]


    while True:

        starting_node,particles = computeBegin(G,nodes,rate)


        t_next = transition_times[-1] - np.log(np.random.rand()) / particles

        if starting_node == -1:
            G.nodes['o']['particles'] += 1 #A new particle enters the system␣
↪always in node "o"
        elif starting_node == 'd':
```

```
            G.nodes['d']['particles'] -= 1 #d does not have a node to send its␣
↪particles, so I have to decrease
            #the number of particles in the node by one
        else:
            end_node = computeEnd(G,Q_cum,nodes,starting_node) #Otherwise,␣
↪simply take the ending node

            G.nodes[starting_node]['particles'] -= 1 #Decrement the starting␣
↪node of 1 because it looses a particle
            G.nodes[end_node]['particles'] += 1 #increment the ending node of 1␣
↪because it takes a particle

        transition_times.append(t_next)

        for node in nodes:
            nodeSeq[node].append(G.nodes[node]['particles'])
        if t_next > units:
            break

    return nodeSeq,transition_times
```

[145]:
```
nodeSeq, transition_times =␣
↪makeSimulation(G,Q_cum,nodes,1,60,initialPhase)#Initial rate = 1 as requested
```

The system in this case is quite stable, but the line plot is quite irregular and so it is difficult to describe the behaviour.

[146]:
```
fig, ax = plt.subplots(figsize=(10,10))
for node in nodes:
    ax.plot(transition_times, nodeSeq[node], label=node)


ax.set_title('Proportional rate with input rate 1',fontweight='bold')
ax.set_xlabel('Transition time')
ax.set_ylabel('Tot particles')
ax.legend();
plt.savefig("ParticlesProportionalRate.png")
```



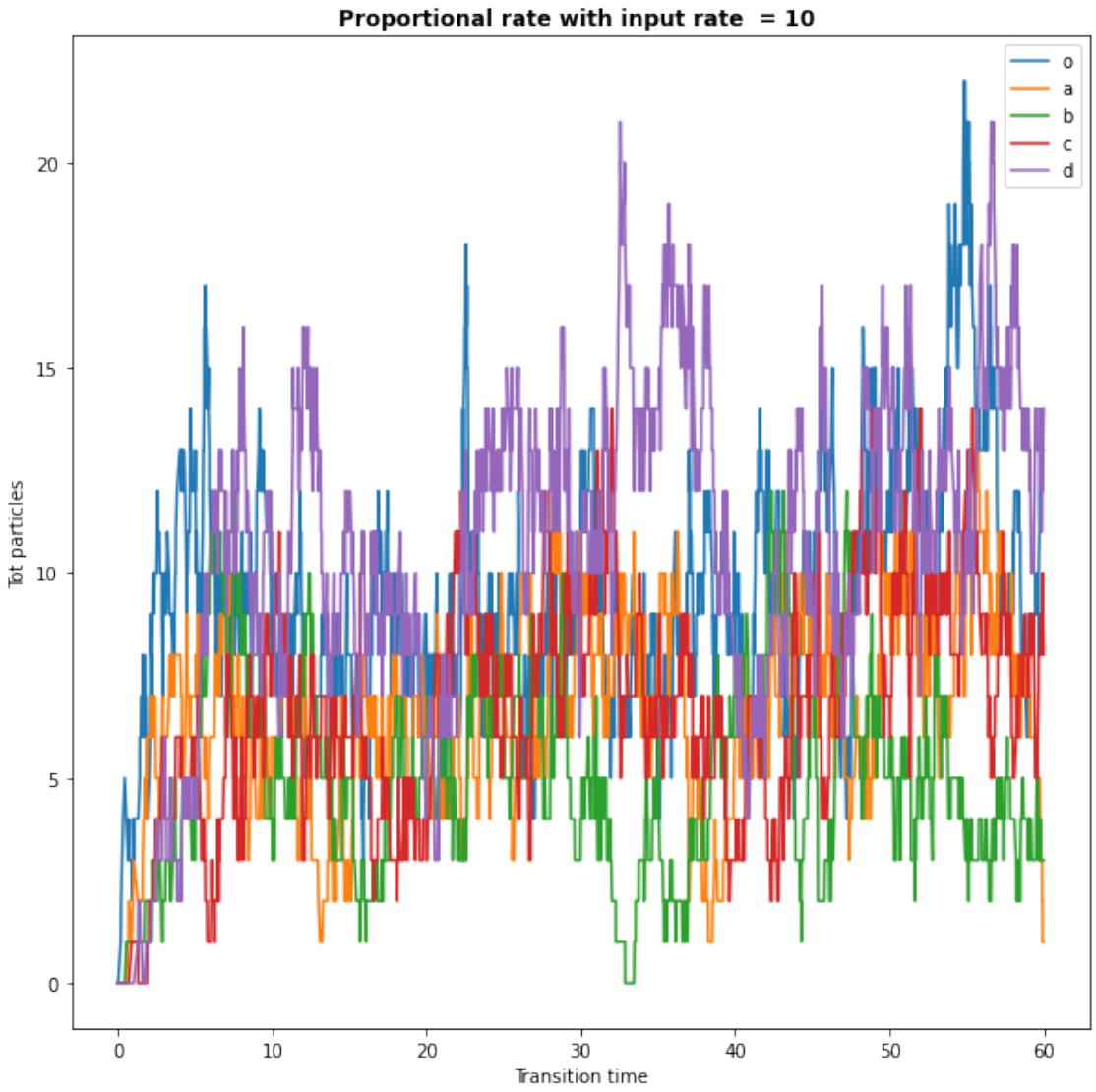
16 Control the blowing up

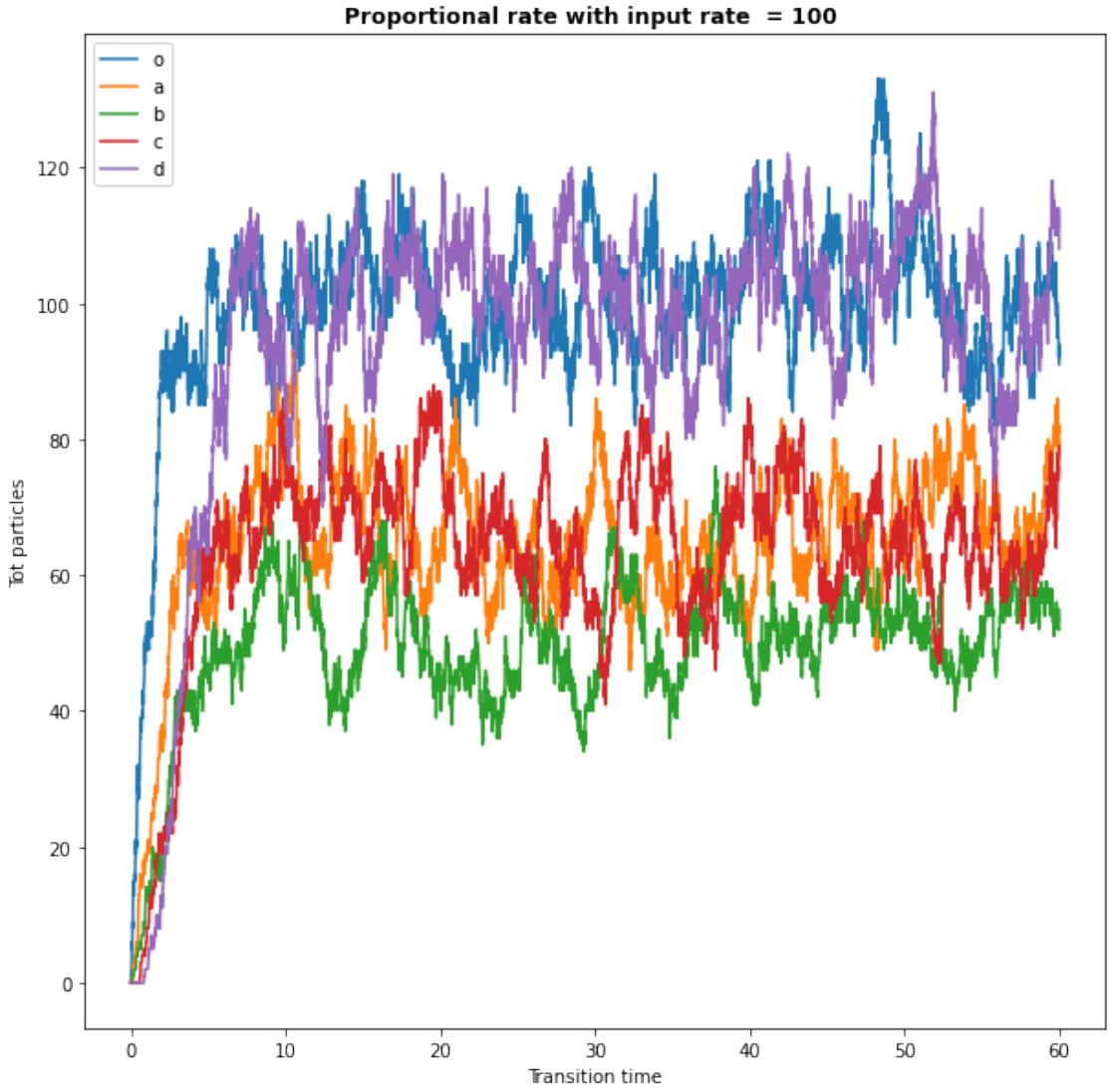
I tried values as 10^i , where i = {1,2,3,4,5}. As you can infer from the consecutive plots, also by increasing the rate, the system never blows up. More precisely:

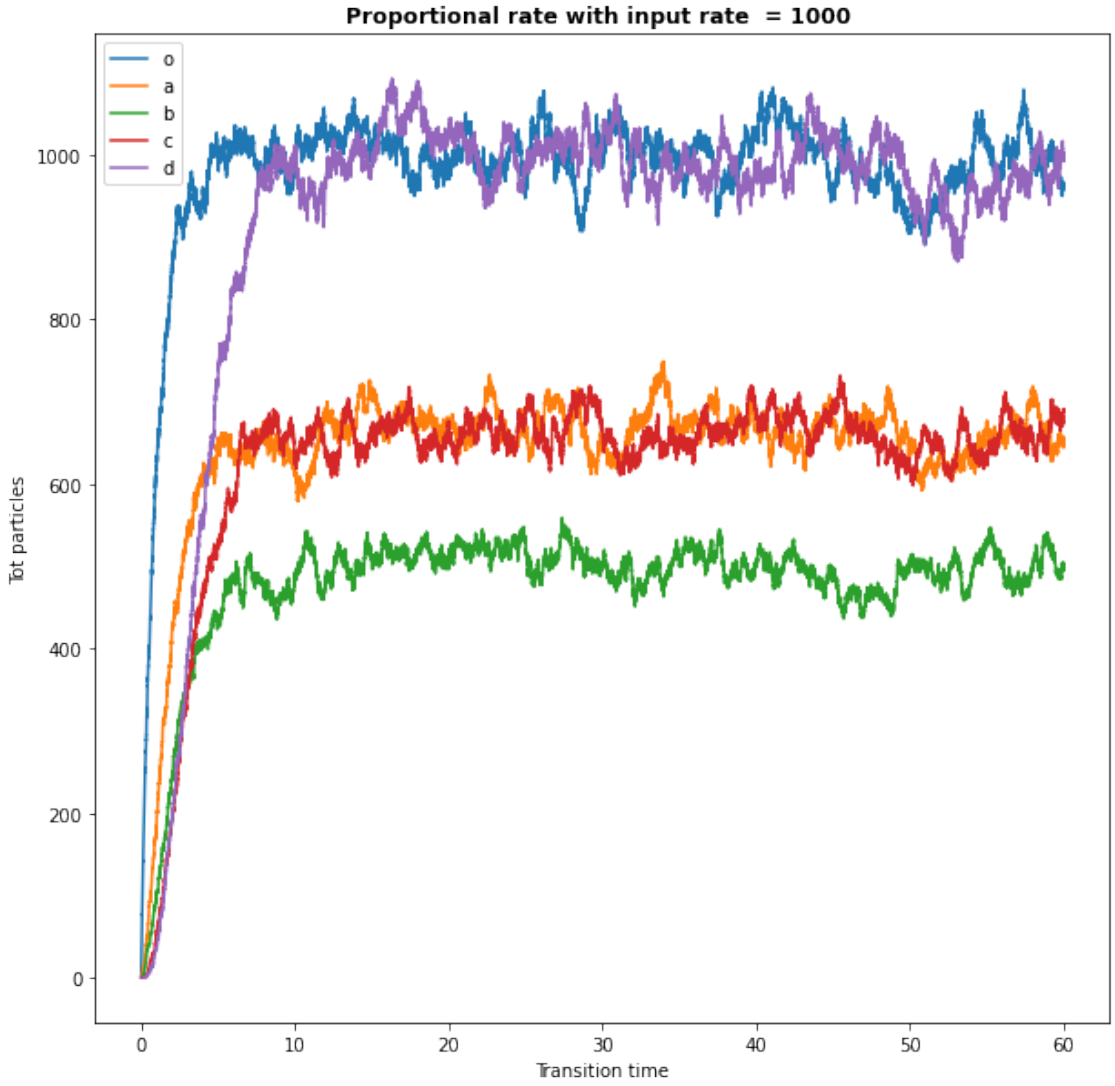
- Rate 10 -> Irregular distribution, but not blow up
- Rate 100 -> The number of particles seems reaching an average value, but there is too difference from upper and lower value
- Rate 1000 -> The average value for each node seems to be more defined, but the computational time is very high

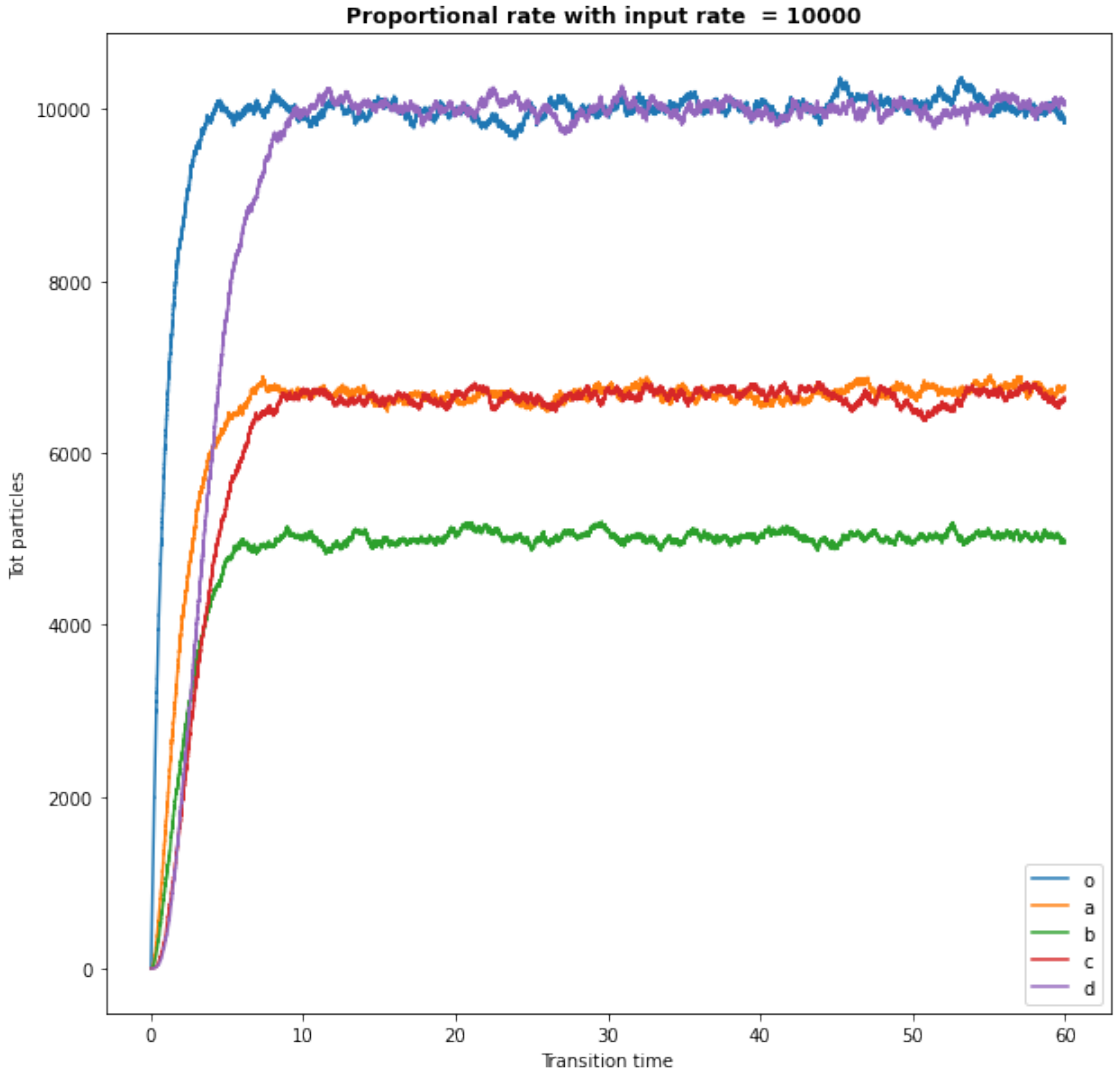
- Rate 10000 -> The average value reaches a very regular behaviour, despite computational time issues

[147]:
```
input_rates = [10, 100, 1000, 10000]
for i in input_rates:
    nodeSeq, transition_times = makeSimulation(G,Q_cum,nodes,i,60,initialPhase)

    fig, ax = plt.subplots(figsize=(10,10))
    for node in nodes:
        ax.plot(transition_times, nodeSeq[node], label=node)

    title = "Proportional rate with input rate  = %.f " % (i)
    ax.set_title(title ,fontweight='bold')
    ax.set_xlabel('Transition time')
    ax.set_ylabel('Tot particles')
    ax.legend();
```




Proportional rate with input rate = 100
o
a
b
c
d
Tot particles
Transition time
0
20
40
60
80
100
120
0
10
20
30
40
50
60


Proportional rate with input rate = 1000
o
a
b
c
d
Tot particles
Transition time
0
200
400
600
800
1000
0
10
20
30
40
50
60



17 Fixed rate

The functions are similar as previous section. In the fixed rate, each node will move across particles according to a Poisson process with a rate equal to 1, so the rate is always one during the iterations and all functions keep in mind this concept.

[148]:
```
def computeBegin(G,nodes,rate):
    tot_particles = rate
    rate_in_node = [rate] # rate of node
    #I have to not memorize the rate in each node, because it is fixed
```

```
    for node in nodes:
        rate_in_node.append(1)
        tot_particles += 1

    rate_in_node = np.array(rate_in_node)
    rate_in_node_cum = np.cumsum(rate_in_node)

    index = np.argwhere(rate_in_node_cum > np.random.rand() *␣
↪tot_particles)[0][0] - 1

    if index == -1:
        starting_node = -1
    else:
        starting_node = nodes[index]

    return starting_node, tot_particles
```

[149]:
```
def computeEnd (G,Q_cum,nodes,selected_node):

    index = np.argwhere(Q_cum[nodes.index(selected_node)] > np.random.
↪rand())[0][0]

    end_node = nodes[index]

    return end_node
```

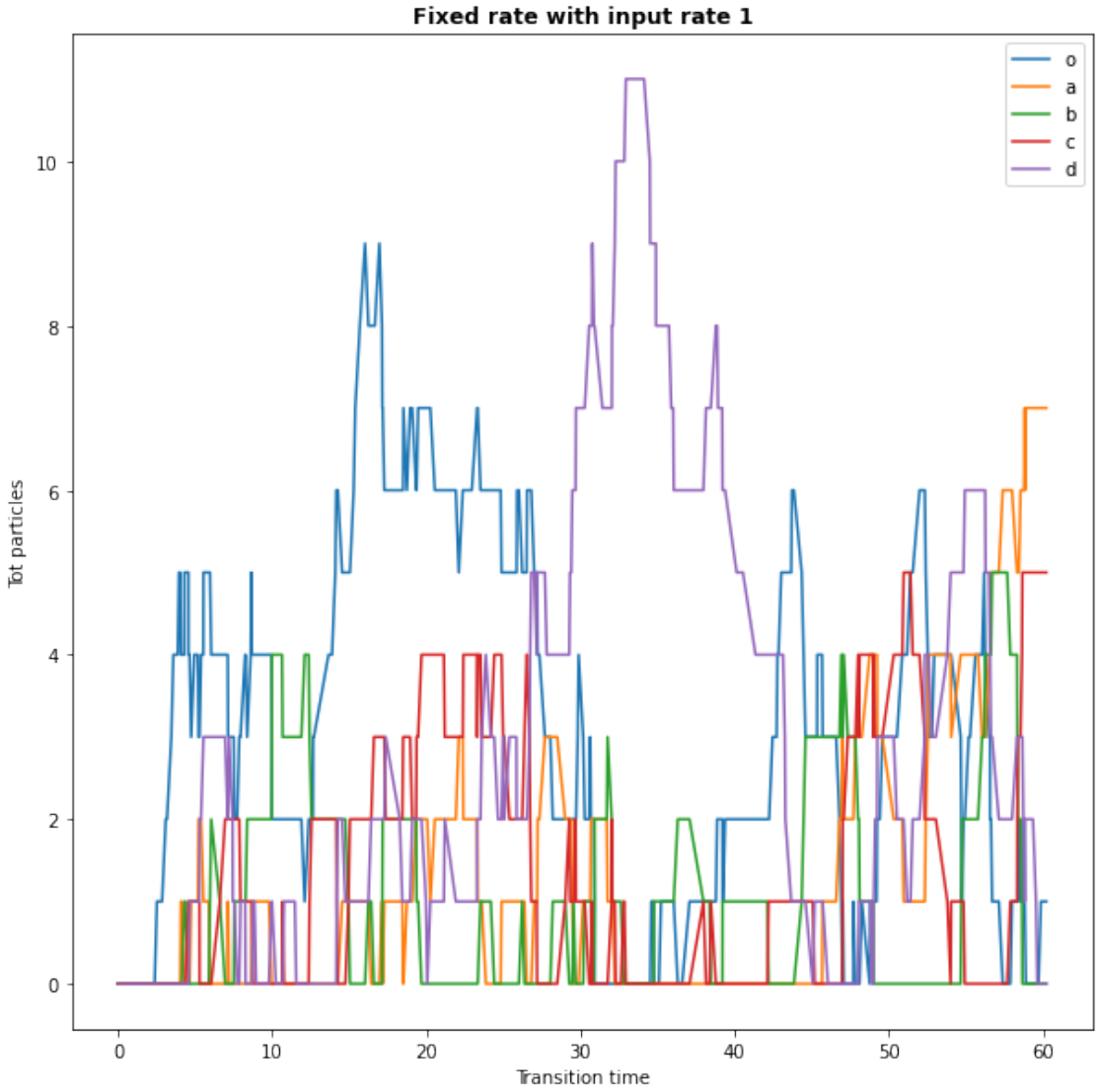
[150]:
```
def makeSimulation(G,Q_cum,nodes,rate,units,initialPhase):
    #tot_particles = 0
    transition_times = [0.0]
    nodeSeq = {}

    for node in nodes:
        nodeSeq[node] = [initialPhase[node]]
        G.nodes[node]['particles'] = initialPhase[node]
        #tot_particles += initialPhase[node]


    while True:

        starting_node,particles = computeBegin(G,nodes,rate)

        #ending_node = computeEnd (G,Q_cum,nodes,starting_node)
        t_next = transition_times[-1] - np.log(np.random.rand()) / particles

        if starting_node == -1:
            G.nodes['o']['particles'] += 1 #A new particle enters the system␣
↪always in node "o"
```

```
        elif G.nodes[starting_node]['particles'] == 0:
            pass
        elif starting_node == 'd':
            G.nodes['d']['particles'] -= 1 #d does not have a node to send its␣
↪particles, so I have to decrease
            #the number of particles in the node by one
        else:
            end_node = computeEnd(G,Q_cum,nodes,starting_node) #Otherwise,␣
↪simply take the ending node

            G.nodes[starting_node]['particles'] -= 1 #Decrement the starting␣
↪node of 1 because it looses a particle
            G.nodes[end_node]['particles'] += 1 #increment the ending node of 1␣
↪because it takes a particle

        transition_times.append(t_next)

        for node in nodes:
            nodeSeq[node].append(G.nodes[node]['particles'])
        if t_next > units:
            break

    return nodeSeq,transition_times
```

[151]:
```
nodeSeq, transition_times = makeSimulation(G,Q_cum,nodes,1,60,initialPhase)
```

The system in this case is quite stable, but the line plot is quite irregular and so it is difficult to describe the behaviour. There is not blow up, but there is a high disparity in terms of number of particles among the nodes.

[152]:
```
fig, ax = plt.subplots(figsize=(10,10))
for node in nodes:
    ax.plot(transition_times, nodeSeq[node], label=node)
ax.set_xlabel('Transition time')
ax.set_ylabel('Tot particles')
ax.set_title('Fixed rate with input rate 1',fontweight='bold')
ax.legend();
plt.savefig("ParticlesFixedRate.png")
```



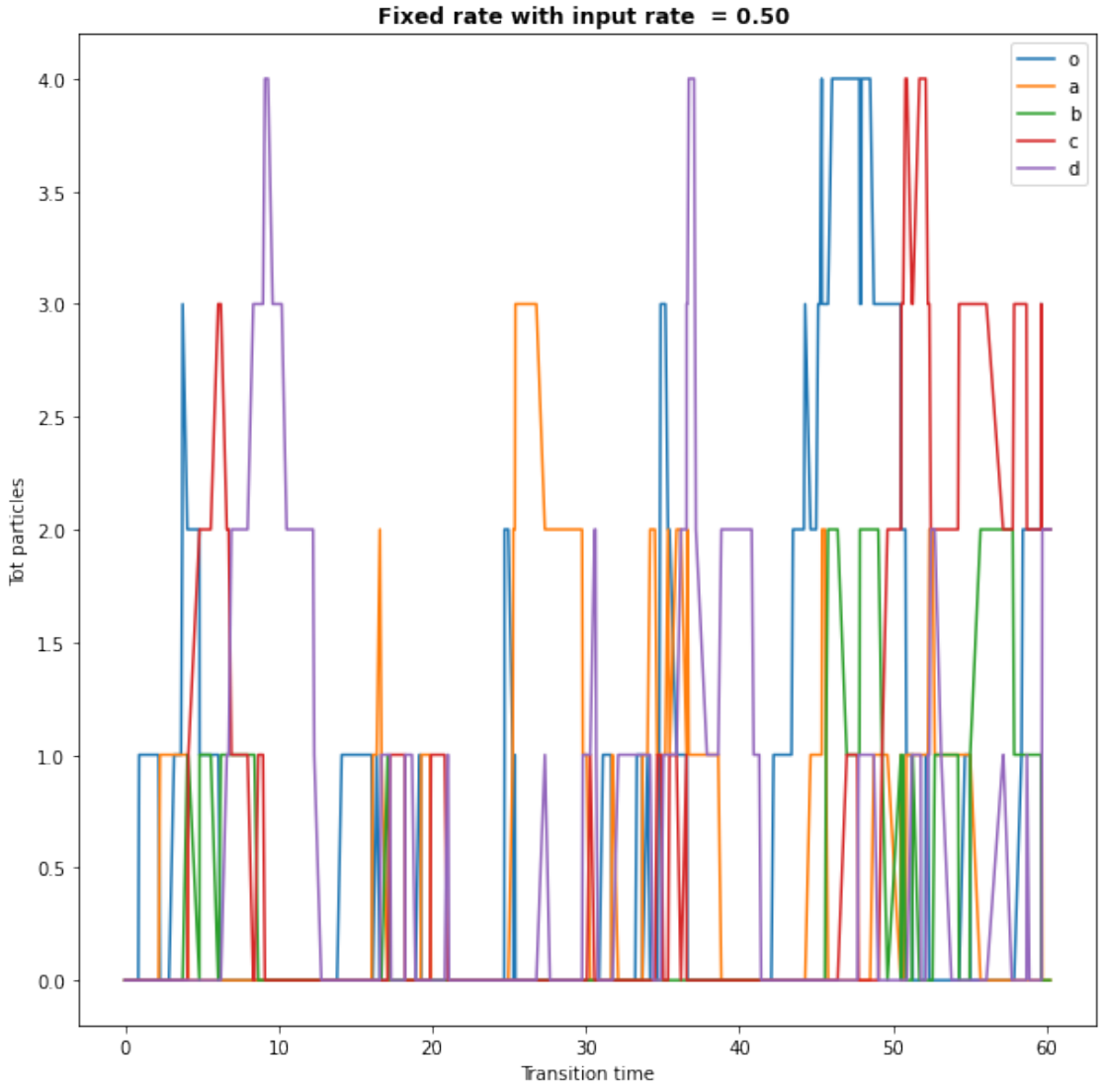
18 Control the blowing up

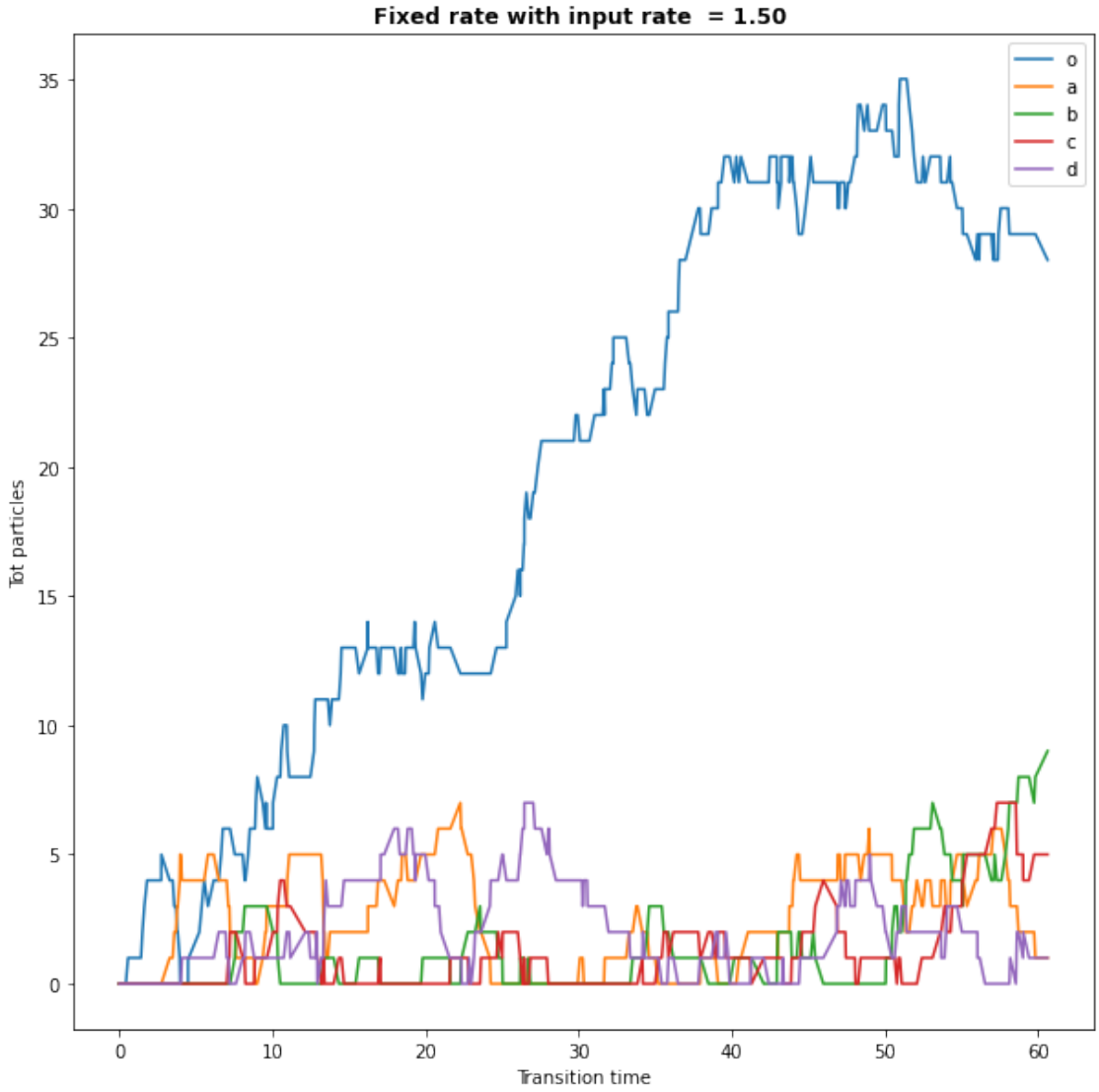
I tried values next to 1. As you can infer from the consecutive plots, by increasing the rate, the system risks to blow up. I saw the opposite behaviour, by decreasing it. More precisely:

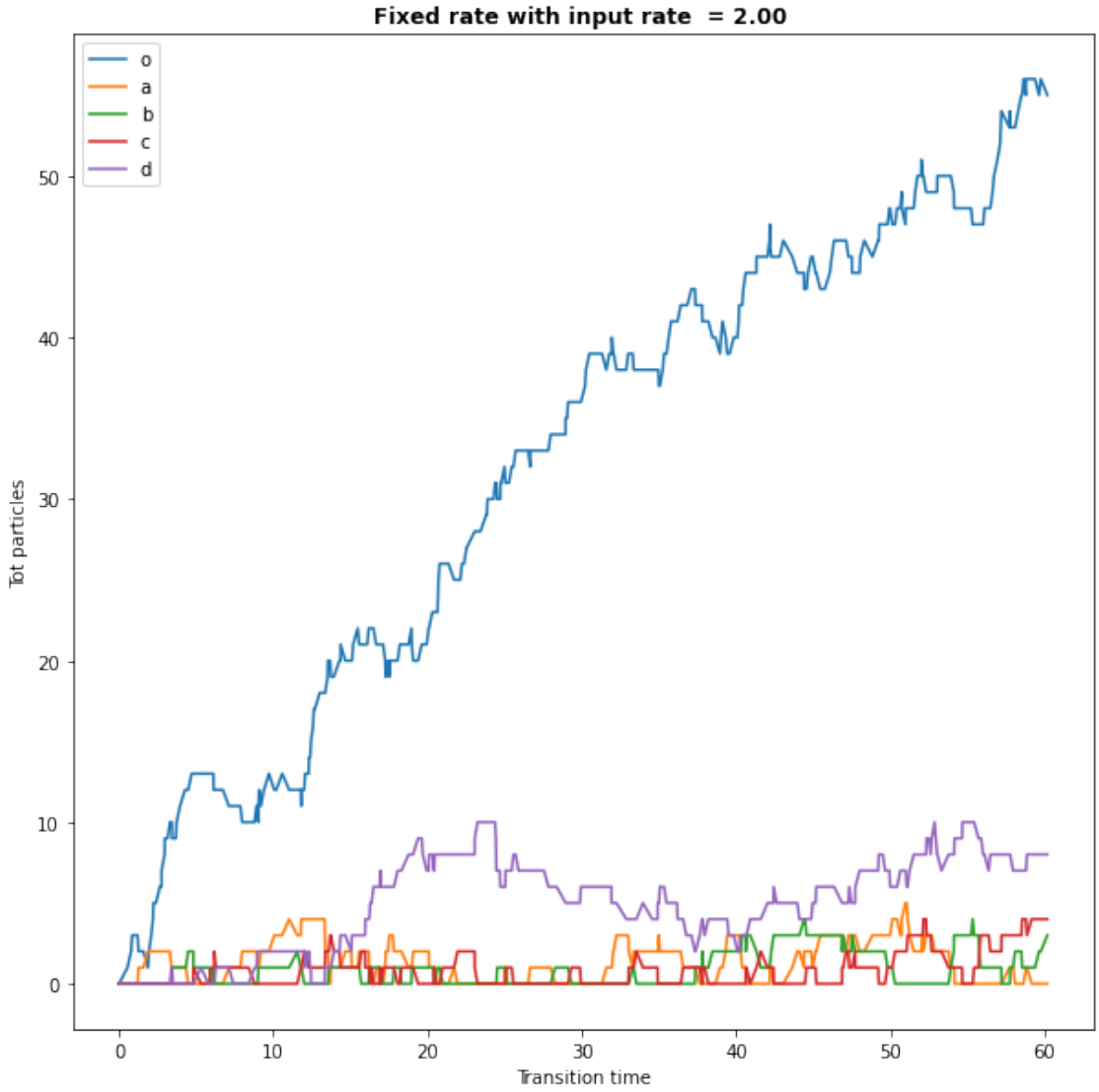
- Rate 0.5 -> The system does not blow up and there is a lower disparity, in terms of number of particles, among nodes
- Rate 1.5 -> The system blows up, as you can infer there is a divergence
- Rate 2 -> The system blows up, quicker compared to rate 1.5.

[153]:
```
input_rates = [0.5,1.5,2]
for i in input_rates:
```

```
nodeSeq, transition_times = makeSimulation(G,Q_cum,nodes,i,60,initialPhase)

fig, ax = plt.subplots(figsize=(10,10))
for node in nodes:
    ax.plot(transition_times, nodeSeq[node], label=node)

title = "Fixed rate with input rate  = %.2f " % (i)
ax.set_title(title ,fontweight='bold')
ax.set_xlabel('Transition time')
ax.set_ylabel('Tot particles')
ax.legend();
```




Fixed rate with input rate = 1.50
o
a
b
c
d
Tot particles
Transition time
0
5
10
15
20
25
30
35
0
10
20
30
40
50
60


Fixed rate with input rate = 2.00
o
a
b
c
d
Tot particles
Transition time
0
10
20
30
40
50
60