Exercise Sheet #4

Deadline: 20.11.2023, 12:00h

Problem 1 (*Barabási-Albert Model in Python*) (10 points)

The goal of this problem is to implement a generator for scale-free graphs in Python using the method by Barabási and Albert. The method comprises of sequentially adding new nodes to a graph with initially N_0 nodes and connecting each of the newly added nodes with m existing nodes, where the connecting probability is proportional to the degree of the node. This process is repeated until there are N nodes in the graph.

(a) Implement a Python function that generates a graph using the method by Barabási and Albert.

Hint: You may use the graph class given on the next page to store your graph.

Hint: You may use `random.choice(seq)` to randomly sample an element from a sequence `seq`.

(b) Compute the degree distribution of a graph with $N = 1000$ nodes generated with your algorithm from (a) and use `matplotlib` to plot the degree distribution in a log-log plot. What do you observe?

Hint: The graph class given on the next page can help you with computing the degree distribution.

```python
import random
from collections import Counter

class Graph():
    def __init__(self):
        # Graph stored as dictionary. Keys = node IDs, values =
            set of neighbors
        self.neighbors = dict()

    def number_of_nodes(self):
        return len(self.neighbors)

    def nodes(self):
        return list(self.neighbors)

    def add_node(self):
        new_node_id = len(self.neighbors)
        self.neighbors[new_node_id] = set()
        return new_node_id

    def add_edge(self, u, v):
        if v in self.neighbors[u] or u in self.neighbors[v]:
            raise Exception("Edge already in graph!")
        self.neighbors[u].add(v)
        self.neighbors[v].add(u)

    def degree_of(self, n):
        # Get the degree of the node with ID n
        if n not in self.neighbors:
            raise Exception("Node not in graph!")
        return len(self.neighbors[n])

    def degree_distribution(self):
        # Generate and return a tuple with degrees in the graph
            and their
        # respective probability of appearance
        degrees = [self.degree_of(n) for n in self.neighbors]
        c = Counter(degrees)
        count = [cnt / self.number_of_nodes for cnt in
            c.values()]
        return list(c.keys()), count

if __name__ == "__main__":
    # Example usage
    G = graph()
    for _ in range(3):
        G.add_node()
    G.add_edge(0, 1)
```

Problem 2 (*Harmonic Oscillator*) (10 points)

The dynamics of the classical harmonic oscillator is determined by the second order ordinary differential equation (ODE)

$$\ddot{x}(t) = -\omega_0^2 x(t) - \alpha \dot{x}(t), \tag{1}$$

where $\omega_0 > 0$ is the natural frequency of the oscillator and α is the damping factor. Some prominent cases of operating regimes for this system are:

- Strong damping: $\frac{\alpha}{2} > \omega_0$.
- Weak damping: $0 < \frac{\alpha}{2} < \omega_0$.
- No damping: $\alpha = 0$.
- Negative damping, i.e. energy uptake: $\alpha < 0$.

(a) Solve Eq. 1 for the initial conditions $x(t_0) = x_0, \dot{x}(t_0) = v_0$ analytically. Then sketch the system's long time evolution, i.e. $x(t)$ for $t \gg t_0$, for each of the four cases mentioned above.

(b) Now investigate the harmonic oscillator as a dynamical system.

- Re-write the second order ODE (Eq. 1) as a system of first order ODEs in x, y where $y = \dot{x}$.
- Find the fixed point of the re-written ODE.
- Sketch the flow of the system in the phase space for each of the four cases mentioned above and indicate the fixed point in the sketches. For which case is it (un)stable?

(c) Compare the analytic solution of Eq. 1 you calculated in (a) with the dynamical systems' point of view obtained in (b).