Utrecht Summer School

Introduction to Complex Systems

Project Day 3

Networks

1 Disease Spreading

In this project, we will study a simple model of spreading on a network, which will illustrate how for example infectious diseases or ideas can spread through a population. The focus in this project will be on the effect of the network structure on the spreading through the network.

In this project's simplified model, each node of the network – representing e.g. a person in a population – can either be infected (1) or uninfected (0). When a node has become infected, it will stay infected. We initialize the network by selecting randomly one of the nodes to be infected. Then, at every timestep, we look at all the uninfected nodes that are neighbours of an infected node. Each of these uninfected nodes then becomes infected with probability p. Locate the files `infection.ipynb` and `infection_functions.py` and put them in the same folder. Again, only the main notebook needs to be opened and amended in this project.

NOTE: The files contain a function called `NetworkProperties` that can be called upon to visualise the network structure and to compute the degree of the nodes in the network when it is given an adjecency matrix A. It can be called by using `deg_av,degs=NetworkProperties(A)`. The variable `deg_av` gives the average degree of the nodes in the network and `degs` contains a list of the degree of the individual nodes.

1.1 Spreading on a lattice network

First, set-up the parameters and settings such that spread on a 16×16 lattice network is modelled. The network type can be changed by changing the variable `network_type`. Now, if you run the code, you will see a figure with the network structure in which all infected nodes will be marked in red, and in another panel the fraction of the infected nodes in the network. The programmed lattice network has periodic boundaries. Can you explain the visualisation of the network?

Run the code a couple of times. Why does the exact results change each time? Vary the infection probability p. How does this change the results? Does it take longer to fully infect the whole network. Try to quantify the behaviour for different values of p. To prevent randomness of a single run, it is better to take the average over multiple runs per p-value.

NOTE: you can speed up the simulations by taking a higher value for `plot_interval`, so plots are refreshed less often. You can also completely shut off the visualisations by setting `showPlot` and/or `showGraph` to false, which further speeds up the simulation.

1.2 Spreading on a random network

Instead of the very regular lattice network, now consider a random network that is created as follows. We start with a regular lattice network. Then, with probability q a link of the lattice network is

broken, and one of the connecting nodes of that link makes a new link with a randomly chosen node. How does this procedure change the network structure? Does the degree of nodes change? Does the average degree change? Run the infection model on such random networks (you only have to change the options for this). Investigate the effect of different q and p combinations. Compare the results on this random network to the results on the lattice network. Can you physically explain the differences between the two?

1.3 Spreading on a community network

Now, consider a community network that consists of two communities: that is, two random networks (created as before) with r links between these communities. (In the code, these inter-network links are created while simultaneously breaking the same number of intra-network links.) Determine how this procedure changes the network structure? Does the degree of nodes change? Does the average degree change?

Run the model for a community network of two communities of size 128 each (to have the same total number of nodes). Study what happens when you change r, q and p. Compare you results with your previous results on the lattice network and the random network.

1.4 Additional suggestions

If time permits, feel free to experiment with different network structures. For instance, you could try to increase the number of communities. Alternatively, you can construct a custom adjacency matrix to use in the simulations.

NOTE: if you are using your own adjacency matrix, be aware that the visualisation only works if the adjacency matrix is symmetric. That is, the visualisation only works properly for undirected networks. However, the internal code for the simulation should work even for directed networks.

2 Coupled Oscillators

In this part we will look at (an extension of) the Kuramoto model, which provides a mathematical description of synchronisation on a network. Specifically, given N different coupled oscillators that each have a natural frequency ω_i, the evolution of the phase θ_i of oscillator i is governed by

$$\frac{d\theta_i}{dt} = \omega_i + \frac{K}{N}\sum_j A_{ij}\sin(\theta_j - \theta_i). \tag{1}$$

Here K is the coupling strength (it does not necessarily need to have values between 0 and 1 for this system) and A_{ij} are the coefficients of an adjacency matrix A that indicate which oscillators are coupled.

2.1 Fully coupled network

The Kuramoto model corresponds to the situation where each oscillator is coupled to each other oscillator, that is $a_{ij} = 1$ if $i \neq j$. Locate the files `Kuramoto.ipynb` and `Kuramoto_functions.py` and put them in the same system folder (again only the main script or notebook needs to be opened and appended). In the simulations, every oscillator is given a natural frequencey ω_i randomly uniformly between 0 and 1, and initial phases $\theta_i(0)$ using a standard normal distribution. The default adjacency matrix for the Kuramoto model is

```
A = np.ones((N,N)) - np.eye(N).
```

Just for your information: in the code, the Kuramoto model is solved numerically using the Runge-Kutta 4 method.

By default, the coupling strength $K = 0$ and $N = 20$. Run the code for various coupling strengths and describe the amount of synchronization. You can also look at the evolution of the phases θ_i and the order parameter $r(t)$ that are plotted at the end of the simulation to get a feel of the amount of synchronisation. Feel free to also play around with the amount of oscillators N. Beware though that simulations get slower as N increases, and takes up more computer memory.

2.2 Order parameter

The (instantaneous) order parameter $r(t)$ has already been implemented in the code as follows

$$r(t) = \frac{1}{N} \left| \sum_{j=1}^{N} e^{i\theta_j(t)} \right|. \tag{2}$$

Does this value help you to accurately quantify the amount of synchronization? Why or why not?

It is possible to define a final order parameter $\bar{r}(t)$ by taking the long-time average over the instantaneous order parameter $r(t)$. Specifically, $\bar{r}(t)$ can be defined as

$$\bar{r} = \frac{1}{T} \lim_{T\to\infty} \int_{T_0}^{T_0+T} r(t)dt, \tag{3}$$

where T_0 is spin-up time period. That is, you ignore the value of the instantaneous order parameter $r(t)$ for times $t < T_0$. Why would we want something like this?

Now, implement the computation of $\bar{r}$ at the end of the script. To do so, you can just take the average of the order parameter $r(t)$ over the last part of your simulation (use the function `mean` or `np.mean` for this). Then, compute $\bar{r}$ for various coupling strengths and plot them as function of K. Do you see a phase transition?

Tip: if you set `updatePlot` and `plotEvolutions` to false, the code will run faster, as the visualisation steps are skipped.

2.3 The effect of the network structure on synchronization

So far, we have used an adjacency matrix in which $a_{ij} = 1$ if $i \neq j$. Investigate what happens for other adjacency matrices. Some suggestions:

- a 1D line of oscillators, with oscillators only interacting with direct neighbours. You can program this as follows:

```
A = np.diag( np.ones(N-1), 1) + np.diag( np.ones(N-1), -1)
```

- An Erdos-Renyi graph: a randomly constructed graph where any edge between two nodes is present with probability p. You can program this as follows:

```
p = 0.1
A = np.triu( np.random.rand(N,N) < p, 1)
A = A + A.transpose()
A = A.astype(int)
```

NOTE: the adjacency matrix A needs to be symmetric (i.e. an undirected network) for the graph visualisation to work. You can turn off these by setting `updatePlot` to false. The actual simulation will work for non-symmetric matrices A, as will the plots of the order parameter.