The Kuramoto Model

Technical report for python numerical project

Paradis Enzo
Student at the university of Bourgogne Franche-Comté
Master CompuPhys - 1st year

Contents

I Introduction

The Kuramoto Model (Yoshki Kuramoto) is a mathematical model that describes the syncronization of coupled oscillators. This program is based on this model to describe a set of 2D-coupled oscillators in various cases. In particular to reproduce data obtained in the case of the coupled arrays of Josephson junctions [1]. This program was edited for the last time the 10/23/2020. It was written in python 3 under ubuntu. There are 6 files. The first one is the main file, `main.py`, it is the only one you have to execute, you could have to comment or uncomment some lines. Instead of that you just have to edit one file, the settings file, `settings.py` where you can find all the settings. Then you have the kuramoto file, `kuramoto.py`, where you can find the `class KuramotoModel` which contains all the functions related to the model or the verifications. The data file, `data.py`, contains the `class Data`, which is used to manage data. The graphs file, `graphs.py`, with the `class Graphs` manages the display of the results. And the integrator file, `integrator.py`, where you can find the `class Integration`, which contains the three integrators : Euler, RK2, RK4. There is also the parameters directory which is important to save data. You can find all these files on github.

II Functional requirement of the program

All the functions used for the computing or to display of the results are called in the main file. At first there are the functions which create the initial values in respect of the parameters, then these values are stored in data files (.dat) in a directory named "parameters". With these data files, we don't have to repeat the calculations every time we want to test something. Then you have the functions that compute the results (the phase of the oscillators, the complex mean average, and the Shannon entropy), wich are stored in data files in the same directory. And finally there are the functions that display the graphs using the module `matplotlib.pyplot`. In this section we will describe the functions that create the data files and their data. Firstly, initial data are created throught the `class Data` wich is in the data file. Finally the computing of the other values is in the `class KuramotoModel` wich is in the kuramoto file.

II.1 The initialisation of data

The initial data are created by the function: `data.init_data(state)`. You can choose if you want to create new initial data by changing the value of `newData` to `True`.

`data.init_data(state="random")`		
Description	Input	Output
This function is used to create the initial values, stored in data files, according to the value of the argument `state` set by default to `state="random"`. You can create data for random, chimera, inverse, or josephson states.	The argument `state` is a `string`. By default it takes the value `"random"` but you can give it these values: • `"random"` • `"chimera"` • `"inverse"` • `"josephson"`	This function will retrieve six data files in the parameters directory, computing in according to the `state` argument. The data files are: • `"omega.dat"` • `"theta0.dat"` • `"K.dat"` • `"eta.dat"` • `"alpha.dat"` • `"tau.dat"`

Table 1: function data.init_data()

Each `state` creates six variables, which are defined as follows:

- ω is a list of real of size N. It contains the natural oscillations of each oscillators
- θ_0 is a list of real of size N. It contains the intial phase of each oscillators. So at the time $t = 0$

- K is a matrix of real of size N×N. It contains the coupling coefficients between each oscillators depending on the nearest neighbours. The nearest neighbour of an oscillator are defined by M in the settings file. It is set as 30% of the total number of oscillators, you can change it
- α is a matrix of real of size N×N. It is the dephasing matrix of the coupling
- τ is a matrix of real of size N×N. It is the delay matrix
- η is a matrix of real of size N×T. It is the representations of external noises for each oscillators at each time $t \in]0, T[$

They are different for each state, you can see their definitions in the tables 2 and 3. The function `uniform()` is from the module `random`, and provide random real numbers with uniform distribution, in the range given. The function `randint()` do the same things with integers.

"random"	"chimera"
This `state` represents the case with random values defined by: • $\omega^i = uniform(0, 3)$ • $\theta_0^i = uniform(0, \frac{2}{\pi})$ • $K_j^i = uniform(0, 1e10)$ • $\eta_j^i = uniform(0, 0.5)$ • $\alpha_j^i = uniform(0, \frac{2}{\pi})$ • $\tau_j^i = randint(0, N//2)$	This `state` represents the case of a quantum chimera state[2] defined by: • $\omega^i = 0.2 + i * 0.4 * sin(\frac{i^2 * \pi}{(2 * N^2)})$ • $\theta_0^i = uniform(0, \frac{2}{\pi})$ • $K_j^i = uniform(0, 1e10)$ • $\eta_j^i = uniform(0, 0.5)$ • $\alpha_j^i = 1.46$ • $\tau_j^i = randint(0, N//2)$

Table 2: `states`

"inverse"	"josephson"
This `state` represents the case with the coupling matrix defined by the inverse of the distance between two oscillators, and the delay matrix is proportionnal to this distance. • $\omega^i = uniform(0, 3)$ • $\theta_0^i = uniform(0, \frac{2}{\pi})$ • $K_j^i = \begin{cases} \frac{1}{\lvert i-j \rvert} \; if \; \lvert i-j \rvert \neq 0 \\ 1e20 \; otherwise \end{cases}$ • $\eta_j^i = uniform(0, 0.5)$ • $\alpha_j^i = 1.46$ • $\tau_j^i = \lvert i-j \rvert$	This `state` represents the case of the array of Josephson, data were drawn from this article [1]. Their data are defined by: • $\omega^i = 0.2 + 0.4i \times sin(\frac{i^2\pi}{(2N^2)})$ • $\theta_0^i = uniform(0, \frac{2}{\pi})$ • $K_j^i = \frac{Nr\omega^{i^2} 2\,\mathrm{e} / (\hbar r I_b) - \omega^i}{\sqrt{(L\omega^{i^2} - 1/C)^2 + \omega^{i^2}(R + rN)^2}}$ • $\eta_j^i = uniform(0, 0.5)$ • $cos(\alpha_j^i) = \frac{L\omega^{i^2} - 1/C}{\sqrt{(L\omega^{i^2} - 1/C)^2 + \omega^{i^2}(R + rN)^2}}$ • $\tau_j^i = randint(0, N//2)$

Table 3: `states`

For each `state` choosen you have to define in the settings file the parameters `Nr` and `Nc`, to define the geometry of the system. `Nr` define the number of rows and `Nc` the number of columms, so `N=Nr*Nc` is the number of oscillators. For example if you choose `N=12` in the geometry `Nr=3, Nc=4`, you will have this configuration:

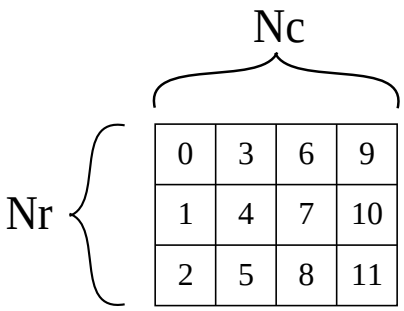


Figure 1: Configuration of the oscillators for `Nr=3, Nc=4`

The labels in the Figure are the labels of the oscillators. If you put a non-positive number the program will not work. So be careful to respect the physics of the system.

II.2 The computing functions

There are three computing functions imported from the `class KuramotoModel`, by the object `kuramoto`. You can choose if you want to compute new data by changing the value of `newData` to `True`.

`kuramoto.integrate(f, theta0, tf=100, integrator="RK4")`		
Description	Input	Output
This function integrates the function `f`, with the initial vector `theta0` during a time `tf`, set by default to `tf=100` with 1000 values(T), using the integrator, set by default to `integrator="RK4"`	• `f` is the function to integrate such that $f : \mathbb{R}^n \to \mathbb{R}^n$ • `theta0` is the initial vector, such that $\theta_0 \in \mathbb{R}^n$. It is a list of real of size N • `tf` is the duration of the integration. Set to 100 by default. It is an integer • `integrator` is the integrator that you want to choose. It can take only these values: `"Euler"`, `"RK2"`, `"RK4"`. Among the three integrators, RK4 is the one losing the least energy the longer the integration lasts, therefore it is the default value.	This function will retrieve two data files. • `"theta.dat"` : contains a matrix of real of size N×T. It is used like a list of vectors. Each vector contains the phase of each oscillators at one time, labeled in the same way as the Figure 1. So the matrix represent the evolution over time of this vector. • `"t.dat"` : contains a list of real of size T, defined by `linspace(0, tf, T)`. `linspace()` is a method from `numpy`.

Table 4: function kuramoto.integrate()

`kuramoto.orders()`		
Description	Input	Output
This function computes the orders parameters R and Φ, of the θ^i over time, by the relation : $Re^{i\Phi} = \frac{1}{N}\sum_{i=1}^{N} e^{i\theta^i}$, for one time.	There are no arguments to this function because it takes only data files. It takes the theta file and the t file, see Table 4.	This function will retrieve two data files in the parameters directory. • `"R.dat"` : It contains a list of real of size T. `R` represents the complex mean of magnitudes of the θ^i over time. • `"phi.dat"` : It contains a list of real of size T. `phi` represents the complex mean of angles of the θ^i over time.

Table 5: function kuramoto.orders()

`kuramoto.shannon_entropies()`		
Description	Input	Output
This function computes the Shannon entropy over time of θ. So for each vector of the list of vectors `theta`	There are no arguments to this function because it takes only data files. It takes the theta file and the t file, see Table 4.	This function will retrieve two data files in the parameters directory. • `"S.dat"` : It contains a list of real of size T. `S` represents the Shannon entropy over time of each vector of `theta`

Table 6: function kuramoto.shannon_entropies()

To summarize in the main file there are 4 functions that are called for the computing. The first one is for initializing data according to the state choosed by the user. The following three are here to compute the main data by using the Kuramoto model. You can choose if you want to initialize new data (1) or do a new computing (4, 5 & 6) by modifying the logical variables (`newData`, `newComputing`) in the settings file. In this file you can modify the `state` variable (2 & 3). You can also modify the geometry of the system by changing `Nr` and `Nc`, don't forget to redo the initial data and the computing after that. Be careful not to set the number of rows or columns to a negative or zero number, it would not make any sense. If you want to save or use other files than the predifined ones you can modify it in the dictionnary `FILE`. You don't have to modify anything else that is not in the settings file. Be careful if you want to modify it, because you can break the program. For the rest of the main file there are 12 functions that are here to display the results. The last three are here to create gifs, they are the begining of the gold version. These displaying functions will be explained later.

III Internal structure of the program

III.1 Description of the physical model

In this program we had to describe 2D-coupled oscillators and tried to reproduce known results [1]. So we will use the Kuramoto Model. Proposed by Yoshiki Kuramoto this model is used to describe the syncronization of a large set of coupled oscillators. It is a mathematical model and we are going to use it in this form :

$$\dot{\theta^i}(t) = \omega_i + \frac{1}{N}\sum_{j=0}^{N-1} K_{ij}\, sin(\theta^j(t-\tau_{ij}) - \theta^i(t) + \alpha_{ij}) + \eta_i(t) \tag{III.1}$$

where :

- $\dot{\theta^i}(t)$ represents the varation of the phase of the i-th oscillator over time.
- ω_i represents the natural frequency of the i-th oscillator.
- N is the total number of oscillators in the system.
- K_{ij} is the coupling matrix. It represents the coupling between each oscillators depending on the inverse of the physical distance between them. Sometimes you can consider only the nearest neighbour.
- θ^i is the phase of the i-th oscillator.
- τ_{ij} is the delay matrix. It is to take into account the phase propagation of the oscillators which have an influence on the i-th oscillator.
- α_{ij} is the dephasing matrix. It represents the dephasing due of the coupled oscillators.
- $\eta_i(t)$ represents the influence of the environement, the possible external noises.

As you can see this model is pretty complete. The formula is the sum of all the influences due to the environment and the nearest coupled oscillators at a given time. That why it was chosen to build θ like a list of vectors.

III.2 Desciption of the used scientific computation algorithms

During the computation we used 3 algorithm for integration of the kuramoto model III.1, the Euler, RK2 and RK4 algorithm. But you can choose the one that you want to use. It is recommended to use the RK4 method because it loses less energy over time than the others, so it is more accurate. Otherwise if you want less computing time it is a good thing to try other. You can find all these integrators in the `class Integration`.
We are looking to integrate $\dot{\theta} = f(\theta)$.
Let $\theta \in \mathbb{R}^n$ and $f : \mathbb{R}^n \to \mathbb{R}^n$. We will note θ^i the i-th component of θ and θ_n the vector θ at the time t_n, $\theta_n \equiv \theta(t_n)$. So we have θ_n^i the i-th component at a time t_n of θ.
The Euler and Runge-Kutta methods are integrators of first-order differential equations and are based on the definition of the derivative:

$$\begin{aligned} \dot{\theta}(t_n) &= \frac{\theta(t_n + h) - \theta(t_n)}{h} \simeq \frac{\theta_{n+1} - \theta n}{\Delta t} \\ \iff \theta_{n+1} &= \theta_n + \dot{\theta}(t_n)\Delta t = \theta_n + f(\theta_n)\Delta t \end{aligned} \tag{III.2}$$

with $\Delta t = \frac{T}{N_t}$ the step of the simulation ($N_t \in \mathbb{N}$ the number of steps of the simulation and T the time of the simulation). We need Δt sufficiently small so N_t enought large and T sufficiently thin.

III.2.1 Euler's method

The Euler algorithm is directly an application of the Equation III.2:

$$\theta_{n+1} = \theta_n + f(\theta_n)\Delta t \tag{III.3}$$

This method will diverge quicker than the two other but the computing time is the shortest.

III.2.2 RK2 method

The Runge-Kutta algorithm (RK2) is like the Euler's method but with a double use of the Equation III.2. There is a middle step to tweak the estimation of f.

$$\theta_{n+1} = \theta_n + f(\theta_n + f(\theta_n)\frac{\Delta t}{2})\Delta t \tag{III.4}$$

This method will diverge quicker than the RK4 method and slower than the Euler's method. Its computing time will be longer than the one of the Euler's method but remains shorter than the one of RK4 method.

III.2.3 RK4 method

RK4, the fourth order Runge-kutta algorithm, is similar to the RK2 algorithm but with a tweak around the middle point.

$$\theta_{n+1} = \theta_n + (K_{1,n} + 2K_{2,n} + 2K_{3,n} + K_{4,n})\frac{\Delta t}{6} \tag{III.5}$$

with

$$\begin{aligned} K_{1,n} &= f(\theta_n) \\ K_{2,n} &= f\left(\theta_n + K_{1,n}\frac{\Delta t}{2}\right) \\ K_{3,n} &= f\left(\theta_n + K_{2,n}\frac{\Delta t}{2}\right) \\ K_{4,n} &= f\left(\theta_n + K_{3,n}\Delta t\right) \end{aligned} \tag{III.6}$$

Even if this is the slowest method of three orders of the Runge-Kutta algorithm, it is the most accurate one. You can find all this algorithm in the `class Integration`.

III.3 List of the constitutive elements of the program

The Program includes 4 classes. There are 3 that are used in the main file and 1 in the `class KuramotoModel`, see Figure 3.

III.3.1 The `class Integration`

This class doesn't have any attributes. It is just used to store the integrators. It can be used in an other program just by writting `import integrator` at the begining of you code, and create an object with the class. This class uses the module `numpy`. Here is a table of the class:

`class Integration`	
Attributes	Methods
∅	This class contains 3 methods, see subsection III.2: • `euler(f, x0,t)` • `RK2(f, x0,t)` • `RK4(f, x0,t)`

Table 7: class Integration

III.3.2 The `class Data`

This class is used to create the initial data using the value of the `state` variable. To work this class need the modules `numpy`, `random` and `time`. Here is a table of the class:

`class Data`	
Attributes	Methods
This class contains 8 attributes, see subsection II.1: • `M,type:int` • `N,type:int` • `K,type:ndarray,shape:(N,N)` • `alpha,type:ndarray,shape:(N,N)` • `eta,type:ndarray,shape:(N,T)` • `omega,type:ndarray,shape:(N,N)` • `tau,type:ndarray,shape:(N,N)` • `theta0,type:ndarray,shape:(N,N)`	This class contains 5 methods: • `init_data(state)`, see Table 1 • `chimera_states()`, see Table 2 • `inverse_states()`, see Table 3 • `random_states()`, see Table 2 • `josephson_matrice()`, see Table 3

Table 8: class Data

You can add a new method in this class to add new configurations. But don't forget to add a new possible value to the `state` variable in `init_data()`. As follows :

```
if state == "random":
    self.random_states()
elif state == "chimera":
    self.chimera_states()
elif state == "inverse":
    self.inverse_states()
elif state == "josephson":
    self.josephson_matrice()
elif state == "new method":
    self.new_method()
```

III.3.3 The `class Graphs`

This class is used to store the display functions and sfunctions to create gifs. It has no attributes and contains 13 methods:

- `kuramoto(t, show=True)` : plot $i \longmapsto \{\theta^i(t)\}_{i=0,\dots,N-1}$ at time t. The argument `show` is used by the animation function, it allows to not display the graph but save it in the directory "animation" (if `show==False`).
- `all_kuramoto()` : plot the N curves $t \longmapsto \theta^i(t)$
- `anim_kuramoto()` : using `kuramoto(t, show=False)` it takes all the `"kuramoto*.png"` in the directory "animation" and create a gif with them.
- `dens_kuramoto()` : plot $(i,t) \longmapsto \theta^i(t)$ using the `numpy.contourf()` method.
- `dens_kuramoto_coord(t, show=True)` : plot $(r,c) \longmapsto \theta^{i_{r,c}}(t)$ at time t using the `numpy.contourf()` method.
- `anim_dens_kuramoto_coord()` : using `dens_kuramoto_coord(t, show=False)` it create a gif.
- `orders()` : plot $t \longmapsto R(t)$ and $t \longmapsto \Phi(t)$.
- `shannon(t)` : plot $i \longmapsto S_i^{q,n}(t)$ at time t.
- `dens_shannon()` : plot $(i,t) \longmapsto S_i^{q,n}(t)$ using the `numpy.contourf()` method.

- `dens_shannon_coord(t, show=True)` : plot $(r,c) \longmapsto S_{i_{r,c}}^{q,n}(t)$ at time t using the `numpy.contourf()` method.
- `anim_dens_shannon_coord()` : using `dens_shannon_coord(t, show=False)` it create a gif.
- `connectivity(K, kmin)` : it is a static method that return a list with all the edges connected in the coupling matrix `K` using a criterion `kmin`.
- `graph_connectivity(kmin=0)` : using the static method `connectivity(K, kmin)` it creates a graph of the connectivity of the oscillators with the criterion `kmin`.

You need to have the modules `numpy`, `matplotlib.pyplot`, `networkx` and `PIL`, installed on your computer to use this class. The module `numpy` is used to recover data from the data files, `matplotlib.pyplot` is used to plot all the graphs, `networkx` is used to create the connectivity graph, and `PIL` is used to create the gifs with all the graphs.

III.3.4 The `class KuramotoModel`

This class is the main class of the program. To work this class need the modules `numpy` and `time`. Here is a table of the class:

`class KuramotoModel`	
Attributes	Methods
This class contains 9 attributes, see subsection II.1: • `N,type:int` • `omega,type:ndarray,shape:(N,N)` • `eta,type:ndarray,shape:(N,T)` • `K,type:ndarray,shape:(N,N)` • `alpha,type:ndarray,shape:(N,N)` • `tau,type:ndarray,shape:(N,N)` • `integr,type:Integration`, see Table 7 • `n,type:int`, number of nearest neighbour • `d_theta,type:ndarray,shape:(N,T)`, the left-side of the Equation III.1	This class contains 7 methods: • `__call__(theta, t)` : return `d_theta` using the kuramoto model, subsection III.1. • `coordinates_to_label(r,c)` : return the label of an oscillator using its coordinates in the array. • `label_to_coordinates(i)` : return coordinates of an oscillators in the array using its label. • `integrate(f, theta0, tf, integrator)`, see Table 4. • `orders()`, see Table 5. • `shannon_entropy(theta, i)` : return the computation of the Shannon entropy for the i-th oscillator. • `shannon_entropies()` : see Table 6

Table 9: class KuramotoModel

III.4 Functional diagram of the program

The program is cut out in 6 modules: main, graphs, kuramoto, data, integrator, and settings. You can see how they interact with each other in the following figure:

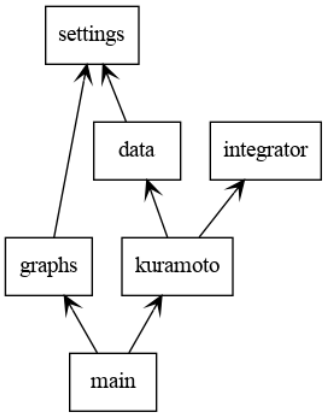


Figure 2: Diagram of the interactions between the modules.

In these modules data, integrator graphs and kuramoto contain a class. These classes are depending to each other like in the following diagram:

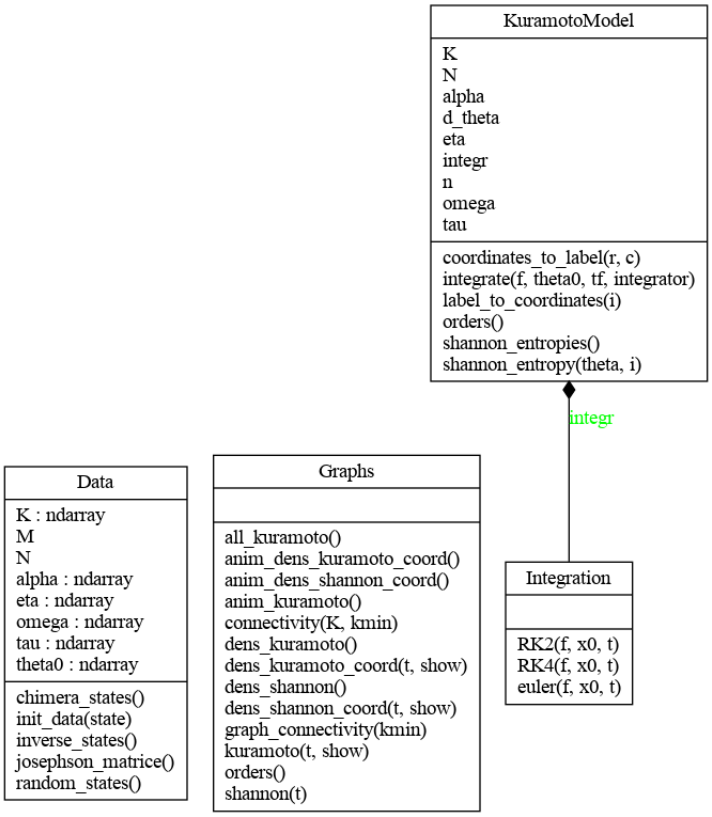


Figure 3: Diagram of the dependences of the classes.

The previous diagrams has been created using `pyreverse` in the directory with the programmation files. The full command used is : `pyreverse -o png main *.py`. That create two pngs, one for the connections between the classes and one for the inetractions between the modules.

IV Quality approach, reliability of the program

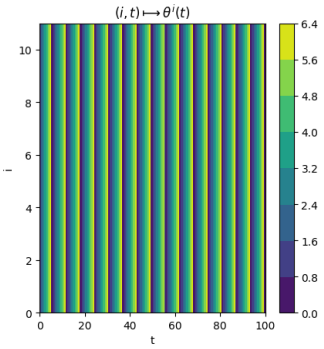
IV.1 Optimization of the code

In a process to limit the memory usage by heavy arrays and reduce computing time, it has been decided to store these arrays in data files (for example Table 1) while waiting for their use. So arrays doesn't takes up memory space between their use. In a quality and time approach, 3 integrators have been used and you can choose the one that you want depending on the quality and the time you want, see subsection III.2. During the execution of the program you can see the running time of the computing functions, see the following example (`N=3*4, state="chimera"`):

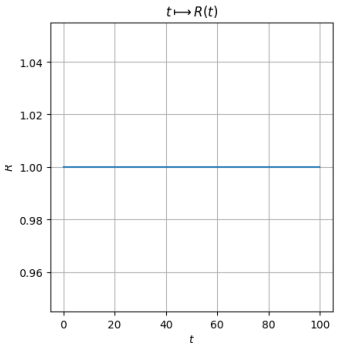
```
----- init data -----
running time : 0.0090s
----- integrate RK4 -----
running time : 8.7955s
----- orders -----
running time : 0.0463s
----- Shannon entropy -----
running time : 0.9747s
```

IV.2 Code verifications

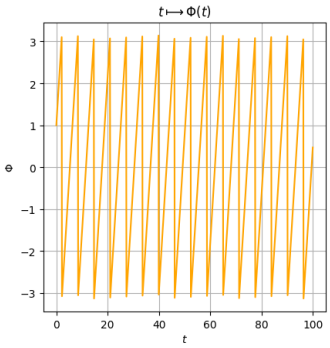
Code verifications has been made for the three computing fuctions, subsection II.2. We have plot graphs corresponding and compare them with the expected results in the choosen configuration. The `function dens_kuramoto()`, the `function orders()` and the `function shannon_entropies()` were also used to verify the resullts. We have tested the case of N independant identical harmonic oscillators (abbreviate in Niiho), i.e. $\omega^i = 1, \theta_0^i = 1, \eta = 0$ and $K_j^i = 0$. We have also used the `function graph_connectivity()` to see connections between oscillators. So after computing we have these graphs :



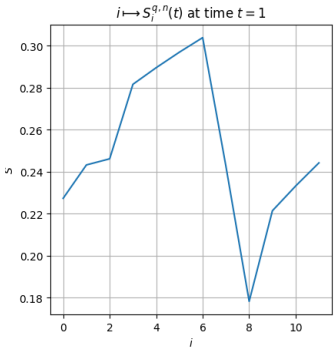
(a) `function dens_kuramoto()`



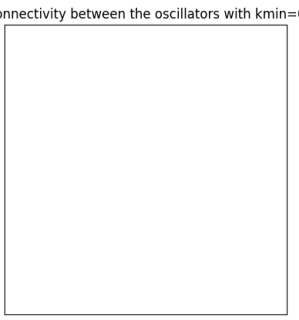
(b) Complex mean of magnitude



(c) Complex mean of phase



(d) Shannon entropy



(e) Connectivity

Figure 4: Niiho case

We can see in the Figure 4a phases are totaly independent, identical and act periodically. The Figure 4b and Figure 4c show us that oscillators are identical and harmonic. And the Figure 4d and Figure 4e prove the oscillators are independant. With the simple case of Niiho we can see the program retrives results corresponding to the expected results, so the program works correctly.

V Outcome examples

In this section we will see two outcome examples with `state="chimera"` and `state="josephson"`. The graphs of the two other will be adding in the appendix of this document. The graphs, the gifs and their data could be find in the corresponding directory in the ”outcomes” directory.

V.1 Chimera

References

[1] Steven H. Strogatz Kurt Wiesenfeld, Pere Colet. Frequency locking in josephson arrays: Connection with the kuramoto model. 57(2), February 1998.

[2] David Viennot. Chaos et chimères quantiques. http://perso.utinam.cnrs.fr/ viennot/publi/chaos.pdf. En ligne; dernière visite : 31 octobre 2020.