jitcsim

Release 0.3

Abolfazl Ziaeemehr

Sep 17, 2022

CONTENTS:

CHAPTER

ONE

INTRODUCTION

JiTCSim is a python package for simulation of complex network dynamics, mainly based on JITC*DE packages for integration of ordinary/stochastic and delay differential equations.

1.1 What are JiTC*DE:

JiTCODE (just-in-time compilation for ordinary differential equations) is an extension of SciPy ODE (**scipy.integrate.ode**) or Solve IVP (**scipy.integrate.solve_ivp**). Where the latter take a Python function as an argument, JiTCODE takes an iterable (or generator function or dictionary) of symbolic expressions, which it translates to **C** code, compiles on the fly, and uses as the function to feed into SciPy ODE or Solve IVP. Symbolic expressions are mostly handled by SymEngine, SymPy's compiled-backend-to-be (see SymPy vs. SymEngine for details).

JiTCDDE (just-in-time compilation for delay differential equations) is a standalone Python implementation of the DDE integration method proposed by Shampine and Thompson [**?**], which in turn employs the Bogacki–Shampine Runge–Kutta pair [**?**].

JiTCSDE (just-in-time compilation for stochastic differential equations) is a standalone Python implementation of the adaptive integration method proposed by Rackauckas and Nie [**?**], which in turn employs Rößler-type stochastic Runge–Kutta methods [**?**]. It can handle both Itō and Stratonovich SDEs, converting the latter internally. JiTCSDE is designed in analogy to JiTCODE.

1.1.1 Indices and tables

- genindex
- modindex
- search

1.1.2 Control parameters

If you try to run a simulation multiple times and for each simulation change a parameter, it will be more efficient to avoid multiple compiling the model and use control parameter. In next example we consider the *coupling* as a control parameter:

The initial phase also could be changed in repeated simulations. The output is plotting the time average of the Kuramoto order parameter vs coupling. Only difference with respect to the previous examples is shown:

Listing 1.1: ../../jitcsim/examples/scripts/examples_kuramoto/02_ode_kuramoto_II_single_par

```
num_ensembles = 10
parameters = {
    'control': ['coupling'],
}
```

to make an instance of the model and measure the compilation time: (time module need to be imported)

```
sol = Kuramoto_II(parameters)
compile_time = time()
sol.compile()
print("Compile time : {:.3f} secondes.".format(time()-compile_time))
```

then define an array for the various coupling strenghts and an zero array to record the order parameter at different couplings:

```
couplings = np.arange(0, 0.8, 0.05) / (N-1)
orders = np.zeros((len(couplings), num_ensembles))
```

we need a loop over each coupling and repeat the simulation *num_ensembles* times. The initial state of the oscillators can also be changed for each simulation using `set_initial_state()`.

```
start_time = time()

for i in range(len(couplings)):
    for j in range(num_ensembles):

        controls = [couplings[i]]
        sol.set_initial_state(uniform(-pi, pi, N))
        data = sol.simulate(controls)
        x = data['x']
        t = data['t']
        orders[i, j] = np.mean(sol.order_parameter(x))
```

(continues on next page)

(continued from previous page)

```
print("Simulation time: {:.3f} seconds".format(time()-start_time))
```

and finally plot the time averaged order parameter vs coupling strength:

```
plot_order(couplings,
           np.mean(orders, axis=1),
           filename="data/02.png",
           ylabel="R",
           xlabel="coupling")
```

```
saving file to data/km.so
Compile time : 1.578 secondes.
Simulation time: 3.385 seconds
```

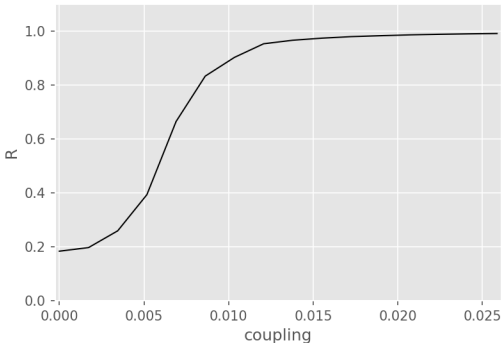

Fig. 1.1: Time average of the Kuramoto order parameter vs coupling strength.

1.1.3 SDE

Simulation of the Kuramoto model with noise.

$$\frac{d\theta_i}{dt} = \omega_i + \xi_i + \sum_{j=0}^{N-1} a_{i,j} \sin(y_j - y_i - \alpha)$$

where ξ_i is white noise for each osillator.

The control parameter of the model is coupling. The initial phase also could be changed in repeated simulations. The output is plotting the Kuramoto order parameter vs time.

Start with importing modules

Listing 1.2: ../../jitcsim/examples/scripts/examples_kuramoto/01_sde_kuramoto_II_single_par

```
import numpy as np
from numpy import pi
from numpy.random import uniform, normal
from jitcsim.visualization import plot_order
from jitcsim.models.kuramoto_sde import Kuramoto_II
from jitcsim.networks import make_network
```

define the amplitude of the noise with normal distribution, set the parameters and runnig the simulation:

```
np.random.seed(2)

N = 30
alpha0 = 0.0
sigma0 = 0.05
coupling0 = 0.5 / (N - 1)
omega0 = normal(0, 0.1, N)
initial_state = uniform(-pi, pi, N)

net = make_network()
adj = net.complete(N)

parameters = {
    'N': N,                                 # number of nodes
    'adj': adj,                             # adjacency matrix
    't_initial': 0.,                        # initial time of integration
    "t_final": 100,                         # final time of integration
    't_transition': 2.0,                    # transition time
    "interval": 1.0,                        # time interval for sampling

    "sigma": sigma0,                        # noise amplitude (normal␣
↪distribution)
    "alpha": alpha0,                        # frustration
    "omega": omega0,                        # initial angular frequencies
    'initial_state': initial_state,         # initial phase of oscillators

    'control': ['coupling'],                # control parameters

    "use_omp": False,                       # use OpenMP
    "output": "data",                       # output directory
}
```

(continues on next page)

(continued from previous page)

```
sol = Kuramoto_II(parameters)
sol.compile(); sol.set_seed(2)

controls = [coupling0]
data = sol.simulate(controls)
x = data['x']
t = data['t']
```

and finally plotting the order parameter:

```
# calculate the Kuramoto order parameter
order = sol.order_parameter(x)

# plot order parameter vs time
plot_order(t,
           order,
           filename="data/01_sde.png",
           xlabel="time",
           ylabel="r(t)")
```

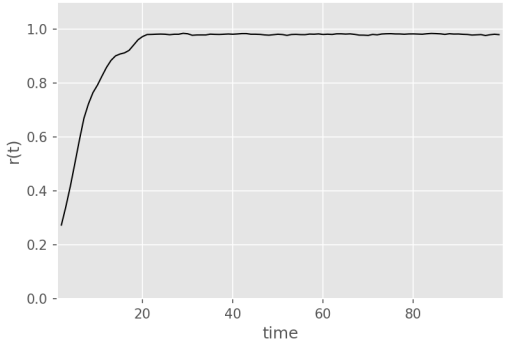

Fig. 1.2: Kuramoto order parameter vs time for complete network. The system of equations include noise.

1.1.4 DDE

Simulation of the Kuramoto model with delay.

The system of equations reads [**?**] :

$$\frac{d\theta_i}{dt} = \omega_i + \sum_{j=0}^{N-1} a_{i,j} \sin(y_j(t - \tau_{ij}) - y_i - \alpha)$$

The control parameter of the model is coupling. The output is plotting the Kuramoto order parameter vs time.

Start with importing required modules:

```
import numpy as np
from numpy import pi
from numpy.random import uniform
from jitcsim.visualization import plot_order
from jitcsim.models.kuramoto_dde import Kuramoto_II
from jitcsim.networks import make_network
```

setting the parameters of the model. The new item here is definition of delay matrix.

```
np.random.seed(2)

N = 12
alpha0 = 0.0
sigma0 = 0.05
coupling0 = 1.5 / (N - 1)
omega0 = [0.5*pi] * N
initial_state = uniform(-pi, pi, N)
net = make_network()
adj = net.complete(N)
delays = adj * 2.0

parameters = {
    'N': N,                             # number of nodes
    'adj': adj,                         # adjacency matrix
    'delays': delays,                   # matrix of delays
    't_initial': 0.,                    # initial time of integration
    "t_final": 100,                     # final time of integration
    't_transition': 10.0,               # transition time
    "interval": 0.2,                    # time interval for sampling

    "alpha": alpha0,                    # frustration
    "omega": omega0,                    # initial angular frequencies
    'initial_state': initial_state,     # initial phase of oscillators

    'control': ['coupling'],            # control parameters

    "use_omp": False,                   # use OpenMP
    "output": "data",                   # output directory
```

compiling and run the simulation. We can also determine type of Dealing with initial discontinuities. To suppress warning messages use `python -W ignore script.py`.

```
sol = Kuramoto_II(parameters)
sol.compile()

controls = [coupling0]
data = sol.simulate(controls, disc="blind", rtol=0, atol=1e-5)
x = data['x']
t = data['t']

order = sol.order_parameter(x)
plot_order(t,
           order,
           filename="data/01_dde.png",
           xlabel="time",
```

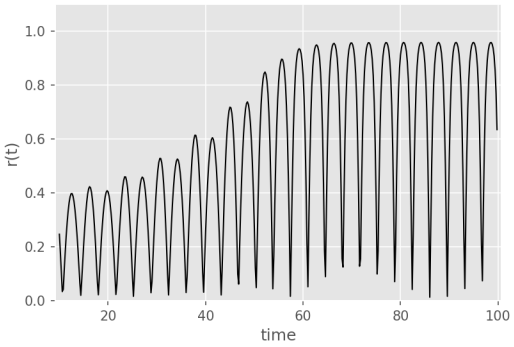

Fig. 1.3: Kuramoto order parameter vs time for a complete network. The system of equations include delay.

1.1.5 Parallel using Multiprocessing

In this example we use **multiprocessing** to speed up the computation by runnig the programm in parallel. The parameter of the model is coupling.

The output is plotting the time average of the Kuramoto order parameter vs coupling.

The only difference with respect to the previous examples is as follows:

Listing 1.3: ../../jitcsim/examples/scripts/examples_kuramoto/03_ode_kuramoto_II_single_par

```
from multiprocessing import Pool
```

```
N = 30
num_ensembles = 100
num_processes = 4                          # number of processes

def run_for_each(coupl):

    controls = [coupl]
    I = Kuramoto_II(parameters)
    I.set_initial_state(uniform(-pi, pi, N))
    data = I.simulate(controls)
    x = data['x']
    order = np.mean(I.order_parameter(x))

    return order
```

after compiling we need to provide the *par* and make a pool:

```
par = []
for i in range(len(couplings)):
    for j in range(num_ensembles):
        par.append(couplings[i])

with Pool(processes=num_processes) as pool:
    orders = (pool.map(run_for_each, par))
orders = np.reshape(orders, (len(couplings), num_ensembles))
```

1.1.6 Explosive synchronization and hysteresis loop

Consider the Kuramoto model on a Frequency Gap-conditioned (FGC) random network. To construct such a network which is proper to see explosive synchronization, we make a link from node i to j only if $|\omega_i - \omega_j| > \gamma$, where γ is frequency threshold [**?**] .

Start with importing modules

Listing 1.4: ../../jitcsim/examples/scripts/07_ode_explosive_synchronization.py

```
import sys
import numpy as np
import pylab as plt
from numpy import pi
from copy import copy
from time import time
from multiprocessing import Pool
```

(continues on next page)

(continued from previous page)

```
from numpy.random import uniform
from jitcsim.visualization import plot_order
from jitcsim.models.kuramoto_ode import Kuramoto_II
from jitcsim.utility import display_time
from jitcsim.networks import make_network
```

start with a low coupling strength, run the simulation and calculate the time average of order parameter and keep the last state of oscillators and use it for the next simulation with slightly increasing the coupling strength. We define a function to do this:

```
def simulateHalfLoop(direction):

    if direction == "backward":
        Couplings = copy(couplings[::-1])
    else:
        Couplings = copy(couplings)

    n = len(Couplings)
    orders = np.zeros(n)

    prev_phases = parameters['initial_state']

    for i in range(n):

        print("direction = {:10s}, coupling = {:10.6f}".format(
            direction, Couplings[i]))
        sys.stdout.flush()

        I = Kuramoto_II(parameters)
        I.set_initial_state(prev_phases)
        data = I.simulate([Couplings[i]])
        x = data['x']
        prev_phases = x[-1, :]
        orders[i] = np.mean(I.order_parameter(x))

    return orders
```

then we need to make our network

```
ki = 20
gamma = 0.45
alpha = 0.0
num_processes = 2
```

(continues on next page)

(continued from previous page)

```
net = make_network()
adj = net.fgc(N=N, k=ki, omega=omega, gamma=gamma)
```

define a variable which determine the direction of movement toward larger or smaller couplings and run our *simulateHalfLoop* function with 2 processors.

```
args = ["forward", "backward"]

with Pool(processes=num_processes) as pool:
    orders = (pool.map(simulateHalfLoop, args))

r_forward, r_backward = orders
```

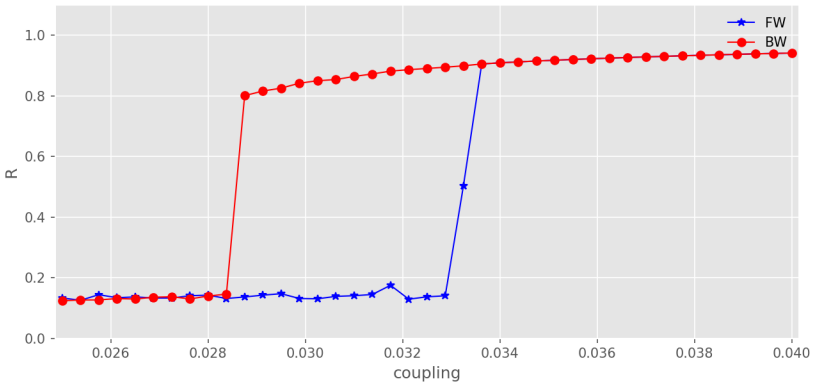

Fig. 1.4: Time average of the Kuramoto order parameter in forward and backward direction and appearence of the hysteresis loop in explosive synchronization.

There is also another exaple for calculation of explosive synchronization and hysteresis loop for the second order Kuramoto model [**?**]. Look at *papers/Kachhavah2017* for more details.

1.1.7 Lyapunov exponents

In this example we are going to calculate the Lyapunov exponent spectrum for the Kuramoto model on feed forward loop.

The output is Lyapunov exponent (LE) spectrum vs time.

Start with importing modules

Listing 1.5: ../../jitcsim/examples/scripts/08_ode_lyapunov_exponents.py

```
import numpy as np
from numpy import pi
import pylab as plt
from numpy.random import uniform, normal
from jitcsim.models.kuramoto_ode import Lyap_Kuramoto_II
from jitcsim.visualization import plot_lyaps
```

define feed forward loop adjacency matrix and set the parameters:

```
FeedForwardLoop = np.asarray([[0, 1, 1],
                              [0, 0, 1],
                              [0, 0, 0]])

parameters = {
    'N': N,                                 # number of nodes
    "adj": FeedForwardLoop,                 # adjacency matrix
    't_initial': 0.,                        # initial time of integration
    "t_final": 10000,                         # final time of integration
    't_transition': 0.0,                    # transition time
    "interval": 0.5,                        # time interval for sampling

    "alpha": alpha0,                        # frustration
    "omega": omega0,                        # initial angular frequencies
    'initial_state': initial_state,         # initial phase of oscillators

    'n_lyap': 3,                            # number of the Lyapunov␣
↪exponents to calculate

    'integration_method': 'RK45',           # integration method
    'control': ['coupling'],                # control parameters

    "use_omp": False,                       # use OpenMP
    "output": "data",                       # output directory
}
```

n_lyap is a new parameter which determine how many of the LEs need to be calculate. Usually we need 2 or 3 largets LE. Increasing the number of LEs make the compilation time extremly long for large networks and the simulation time also will be much longer.

Lyap_Kuramoto_II class is defined for calculation of LEs:

```
sol = Lyap_Kuramoto_II(parameters)
```

(continues on next page)

(continued from previous page)

```
sol.compile()

controls = [coupling0]
data = sol.simulate(controls)
x = data['lyap']
t = data['t']
```

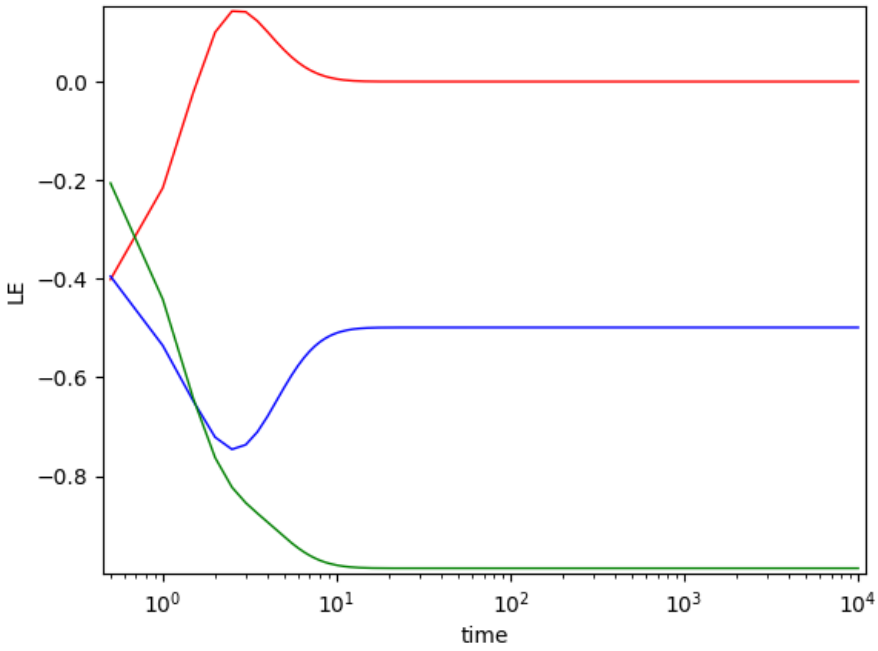

Fig. 1.5: The LEs of the Kuramoto oscillators on feed forward loop.

CHAPTER

TWO

ADDING NEW MODELS

One the main purpose of JiTCSim is flexiblity and ease of development for users with medium knowledge about Python. No exprience on C/C++ is required. Process of adding new model is strightforward as we see in the following. The system of equations need to be provided in Python syntax, considering requirements of JiTC*DE, and also using available models (using ODE/DDE/SDE sovlers) as template.

2.1 Damp oscillator

$$\frac{dx}{dt} = x - xy - ax^2$$
$$\frac{dy}{dt} = xy - y - by^2$$

To add a new damp oscillator model we need to define the model in *jitcsim/models/damp_oscillator.py* The file can be used as a template for any other ODE model. For other models, *rhs()* need to be redefined.

Starting with a few imports

Listing 2.1: ../../jitcsim/models/damp_oscillator.py

```
import os
import os.path
import numpy as np
from os.path import join
from jitcode import jitcode, y
from symengine import Symbol
```

set compiler as *clang* which accelerate compiling the code for large system of equations, but this is optional. By default it uses *gcc*.

```
os.environ["CC"] = "clang"
```

choose a name for class start with Capital letter:

```
class DampOscillator:
```

define *__init__()* which you don't need to modify it usually.

define *rhs()* which is your system of equations:

```
def rhs(self):
    '''
    The right hand side of the system of equations : dy/dt = f(y, t)
    for a damp oscillator.
    '''

    yield y(0) - y(0) * y(1) - self.a * y(0) * y(0)
    yield y(0) * y(1) - y(1) - self.b * y(1) * y(1)
```

define *compile()*, *initial_state()* and *simulate()* functions. Again you don't need to modify these functions usually. Sometimes you need to modify *simulate* function if you need to process time series befor passing them as the result.

CHAPTER

THREE

AUTO GENERATED DOCUMENTATION

ODE Kuramoto model class
SDE Kuramoto model class
DDE Kuramoto model class

3.1 Kuramoto Model

The following are the main classes for the Kuramoto model including:

3.1.1 The main class for the ODE Kuramot model

class jitcsim.models.kuramoto_ode.**Kuramoto_Base**(*par*)

Base class for the Kuramoto model.

Parameters

- **N** (*int*) – number of nodes
- **adj** (*2d array*) – adjacency matrix
- **t_initial** (*float, int*) – initial time of integration
- **t_final** (*float, int*) – final time of integration
- **t_transition** (*float, int*) – transition time
- **interval** (*float*) – time interval for sampling
- **alpha** (*flaot*) – frustration
- **omega** (*float*) – initial angular frequencies
- **initial_state** (*array of size N*) – initial phase of oscillators
- **integration_method** (*str*) –

name of the integrator
One of the following (or a new method supported by either backend):

- *"dopri5"* - Dormand's and Prince's explicit fifth-order method via *ode*
- *"RK45"* - Dormand's and Prince's explicit fifth-order method via *solve_ivp*
- *"dop853"* - DoP853 (explicit) via *ode*
- *"DOP853"* - DoP853 (explicit) via *solve_ivp*
- *"RK23"* - Bogacki's and Shampine's explicit third-order method via *solve_ivp*
- *"BDF"* - Implicit backward-differentiation formula via *solve_ivp*
- *"lsoda"* - LSODA (implicit) via *ode*
- *"LSODA"* - LSODA (implicit) via *solve_ivp*
- *"Radau"* - The implicit Radau method via *solve_ivp*
- *"vode"* - VODE (implicit) via *ode*

The *solve_ivp* methods are usually slightly faster for large differential equations, but they come with a massive overhead that makes them considerably slower for small differential equations. Implicit solvers are slower than explicit ones, except for stiff problems. If you don't know what to choose, start with *"dopri5"*.

- **control** (*list of str*) – control parameters
- **use_omp** (*boolian*) – if *True* allow to use OpenMP
- **output** (*str*) – output directory
- **verbose** (*boolian*) – if *True* some information about the process will be desplayed.

compile(*modulename='km'*, ***kwargs*)

compile model and produce shared library. translates the derivative to C code using SymEngine's C-code printer.

Parameters
kwargs (*(key, value)*) –

used in generate_f_C including

- simplify : boolean or None
- **do_cse: boolian,**
  Whether SymPy's common-subexpression detection should be ap-

plied before translating to C code. It is almost always better to let the compiler do this (unless you want to set the compiler optimisation to *-O2* or lower): For simple differential equations this should not make any difference to the compiler's optimisations. For large ones, it may make a difference but also take long. As this requires all entries of f at once, it may void advantages gained from using generator functions as an input. Also, this feature uses SymPy and not SymEngine.

- **chunk_size: int**
  If the number of instructions in the final C code exceeds this number, it will be split into chunks of this size. See Handling very large differential equations on why this is useful and how to best choose this value. It also used for paralleling using OpenMP to determine task scheduling. If smaller than 1, no chunking will happen.

`local_order_parameter`(*phases*, *indices*)

calculate local Kuramoto order parameter for given node indices

Parameters

- **phases** (*float numpy array*) – phase of each oscillator.
- **indices** (*int numpy array*) – indices of given nodes.

`order_parameter`(*phases*)

calculate the Kuramoto order parameter

Parameters

phases (*2D numpy array [nstep by N]*) – phase of oscillators

`simulate`(*par*, ***integrator_params*)

integrate the system of equations and return the coordinates and times

Parameters

- **par** (*list*) – list of values for control parameters in order of appearence in control
- **Return** (*dict(t, x)*) –
  - t times
  - x coordinates.

`class jitcsim.models.kuramoto_ode.`**Kuramoto_II**(*par*)

Kuramoto model type II

$$\frac{d\theta_i}{dt} = \omega_i + \sum_{j=0}^{N-1} a_{i,j} \sin(y_j - y_i - \alpha)$$

Parameters

- **N** (*int*) – number of nodes
- **adj** (*2d array*) – adjacency matrix
- **t_initial** (*float, int*) – initial time of integration
- **t_final** (*float, int*) – final time of integration
- **t_transition** (*float, int*) – transition time
- **interval** (*float*) – time interval for sampling
- **alpha** (*flaot*) – frustration
- **omega** (*float*) – initial angular frequencies
- **initial_state** (*array of size N*) – initial phase of oscillators
- **integration_method** (*str*) –

  name of the integrator

  One of the following (or a new method supported by either backend):

  – *"dopri5"* - Dormand's and Prince's explicit fifth-order method via *ode*

  – *"RK45"* - Dormand's and Prince's explicit fifth-order method via *solve_ivp*

  – *"dop853"* - DoP853 (explicit) via *ode*

  – *"DOP853"* - DoP853 (explicit) via *solve_ivp*

  – *"RK23"* - Bogacki's and Shampine's explicit third-order method via *solve_ivp*

  – *"BDF"* - Implicit backward-differentiation formula via *solve_ivp*

  – *"lsoda"* - LSODA (implicit) via *ode*

  – *"LSODA"* - LSODA (implicit) via *solve_ivp*

  – *"Radau"* - The implicit Radau method via *solve_ivp*

  – *"vode"* - VODE (implicit) via *ode*

  The *solve_ivp* methods are usually slightly faster for large differential equations, but they come with a massive overhead that makes them considerably slower for small differential equations. Implicit solvers are slower than explicit ones, except for stiff problems. If you don't know what to choose, start with *"dopri5"*.

- **control** (*list of str*) – control parameters
- **use_omp** (*boolian*) – if *True* allow to use OpenMP
- **output** (*str*) – output directory
- **verbose** (*boolian*) – if *True* some information about the process will be desplayed.

compile(*modulename='km'*, ***kwargs*)

compile model and produce shared library. translates the derivative to C code using SymEngine's C-code printer.

Parameters

kwargs (*(key, value)*) –

used in generate_f_C including

- simplify : boolean or None
- **do_cse: boolian,**
  Whether SymPy's common-subexpression detection should be applied before translating to C code. It is almost always better to let the compiler do this (unless you want to set the compiler optimisation to *-O2* or lower): For simple differential equations this should not make any difference to the compiler's optimisations. For large ones, it may make a difference but also take long. As this requires all entries of f at once, it may void advantages gained from using generator functions as an input. Also, this feature uses SymPy and not SymEngine.
- **chunk_size: int**
  If the number of instructions in the final C code exceeds this number, it will be split into chunks of this size. See Handling very large differential equations on why this is useful and how to best choose this value. It also used for paralleling using OpenMP to determine task scheduling. If smaller than 1, no chunking will happen.

local_order_parameter(*phases*, *indices*)

calculate local Kuramoto order parameter for given node indices

Parameters

- **phases** (*float numpy array*) – phase of each oscillator.
- **indices** (*int numpy array*) – indices of given nodes.

order_parameter(*phases*)

calculate the Kuramoto order parameter

Parameters
phases (*2D numpy array [nstep by N]*) – phase of oscillators

rhs()

Kuramoto model of type II

$$\frac{d\theta_i}{dt} = \omega_i + \sum_{j=0}^{N-1} a_{i,j} \sin(y_j - y_i - \alpha)$$

simulate(*par*, ***integrator_params*)

integrate the system of equations and return the coordinates and times

Parameters

- **par** (*list*) – list of values for control parameters in order of appearence in control
- **Return** (*dict(t, x)*) –
  - t times
  - x coordinates.

---

class jitcsim.models.kuramoto_ode.**Kuramoto_I**(*par*)

Kuramot model type I

$$\frac{d\theta_i}{dt} = \omega_i + 0.5 * \sum_{j=0}^{N-1} a_{i,j} \Big(1 - \cos(y_j - y_i - \alpha)\Big)$$

Parameters

- **N** (*int*) – number of nodes
- **adj** (*2d array*) – adjacency matrix
- **t_initial** (*float, int*) – initial time of integration
- **t_final** (*float, int*) – final time of integration
- **t_transition** (*float, int*) – transition time
- **interval** (*float*) – time interval for sampling
- **alpha** (*flaot*) – frustration
- **omega** (*float*) – initial angular frequencies
- **initial_state** (*array of size N*) – initial phase of oscillators
- **integration_method** (*str*) –

name of the integrator

One of the following (or a new method supported by either backend):

- *"dopri5"* - Dormand's and Prince's explicit fifth-order method via *ode*
- *"RK45"* - Dormand's and Prince's explicit fifth-order method via *solve_ivp*
- *"dop853"* - DoP853 (explicit) via *ode*
- *"DOP853"* - DoP853 (explicit) via *solve_ivp*
- *"RK23"* - Bogacki's and Shampine's explicit third-order method via *solve_ivp*
- *"BDF"* - Implicit backward-differentiation formula via *solve_ivp*
- *"lsoda"* - LSODA (implicit) via *ode*
- *"LSODA"* - LSODA (implicit) via *solve_ivp*
- *"Radau"* - The implicit Radau method via *solve_ivp*
- *"vode"* - VODE (implicit) via *ode*

The *solve_ivp* methods are usually slightly faster for large differential equations, but they come with a massive overhead that makes them considerably slower for small differential equations. Implicit solvers are slower than explicit ones, except for stiff problems. If you don't know what to choose, start with *"dopri5"*.

- **control** (*list of str*) – control parameters
- **use_omp** (*boolian*) – if *True* allow to use OpenMP
- **output** (*str*) – output directory
- **verbose** (*boolian*) – if *True* some information about the process will be desplayed.

compile(*modulename='km'*, ***kwargs*)

compile model and produce shared library. translates the derivative to C code using SymEngine's C-code printer.

Parameters

kwargs (*(key, value)*) –

used in generate_f_C including

- simplify : boolean or None

- **do_cse: boolian,**
  Whether SymPy's common-subexpression detection should be applied before translating to C code. It is almost always better to let the compiler do this (unless you want to set the compiler optimisation to *-O2* or lower): For simple differential equations this should not make any difference to the compiler's optimisations. For large ones, it may make a difference but also take long. As this requires all entries of f at once, it may void advantages gained from using generator functions as an input. Also, this feature uses SymPy and not SymEngine.
- **chunk_size: int**
  If the number of instructions in the final C code exceeds this number, it will be split into chunks of this size. See Handling very large differential equations on why this is useful and how to best choose this value. It also used for paralleling using OpenMP to determine task scheduling. If smaller than 1, no chunking will happen.

`local_order_parameter`(*phases*, *indices*)

calculate local Kuramoto order parameter for given node indices

Parameters

- **phases** (*float numpy array*) – phase of each oscillator.
- **indices** (*int numpy array*) – indices of given nodes.

`order_parameter`(*phases*)

calculate the Kuramoto order parameter

Parameters

phases (*2D numpy array [nstep by N]*) – phase of oscillators

`rhs`()

Kuramoto model of type I

$$\frac{d\theta_i}{dt} = \omega_i + 0.5 * \sum_{j=0}^{N-1} a_{i,j}\Big(1 - \cos(y_j - y_i - \alpha)\Big)$$

Return :

right hand side of the Kuramoto model

`simulate`(*par*, ***integrator_params*)

integrate the system of equations and return the coordinates and times

Parameters

- **par** (*list*) – list of values for control parameters in order of appearence in control

- **Return** (*dict(t, x)*) –
  - t times
  - x coordinates.

---

class jitcsim.models.kuramoto_ode.**SOKM_SingleLayer**(*par*)

Second order Kuramoto Model for single layer network

$$m\frac{d^2\theta_i(t)}{dt^2} + \frac{d\theta_i(t)}{dt} = \omega_i + \frac{\lambda}{\langle k \rangle}\sum_{j=1}^{N}\sin\left[\theta_j(t) - \theta_i(t)\right]$$

Reference:

Kachhvah, A.D. and Jalan, S., 2017. Multiplexing induced explosive synchronization in Kuramoto oscillators with inertia. EPL (Europhysics Letters), 119(6), p.60005.

compile(***kwargs*)

compile model and produce shared library. translates the derivative to C code using SymEngine's C-code printer.

Parameters

kwargs (*(key, value)*) –

used in generate_f_C including

- simplify : boolean or None
- **do_cse: boolian,**
  Whether SymPy's common-subexpression detection should be applied before translating to C code. It is almost always better to let the compiler do this (unless you want to set the compiler optimisation to *-O2* or lower): For simple differential equations this should not make any difference to the compiler's optimisations. For large ones, it may make a difference but also take long. As this requires all entries of f at once, it may void advantages gained from using generator functions as an input. Also, this feature uses SymPy and not SymEngine.
- **chunk_size: int**
  If the number of instructions in the final C code exceeds this number, it will be split into chunks of this size. See Handling very large differential equations on why this is useful and how to best choose this value. It also used for paralleling using OpenMP to determine task scheduling. If smaller than 1, no chunking will happen.

local_order_parameter(*phases*, *indices*)

calculate local Kuramoto order parameter for given node indices

Parameters

- **phases** (*float numpy array*) – phase of each oscillator.
- **indices** (*int numpy array*) – indices of given nodes.

order_parameter(*phases*)

calculate the Kuramoto order parameter

Parameters

phases (*2D numpy array [nstep by N]*) – phase of oscillators

simulate(*par*, ***integrator_params*)

Integrate the system of equations and return the coordinates and times

Return

[dict(t, x)]

- **t** times
- **x** coordinates.

3.1.2 The main class for the SDE Kuramot model

class jitcsim.models.kuramoto_sde.**Kuramoto_Base**(*par*)

Base class for the Kuramoto model.

Parameters

- **N** (*int*) – number of nodes
- **adj** (*2d array*) – adjacency matrix
- **t_initial** (*float, int*) – initial time of integration
- **t_final** (*float, int*) – final time of integration
- **t_transition** (*float, int*) – transition time
- **interval** (*float*) – time interval for sampling
- **sigma** (*float*) – noise aplitude of normal distribution
- **alpha** (*flaot*) – frustration
- **omega** (*float*) – initial angular frequencies
- **initial_state** (*array of size N*) – initial phase of oscillators
- **control** (*list of str*) – control parameters

- **use_omp** (*boolian*) – if *True* allow to use OpenMP
- **output** (*str*) – output directory
- **verbose** (*boolian*) – if *True* some information about the process will be desplayed.

set_integrator_parameters(*atol=1e-05*, *rtol=0.01*, *min_step=1e-05*, *max_step=10.0*)

! set properties for integrator

simulate(*par=[]*, *mode_2pi=True*)

integrate the system of equations and return the coordinates and times

Parameters

- **par** (*list*) – list of values for control parameters in order of appearence in control
- **Return** (*dict(t, x)*) –
  - t times
  - x coordinates.

---

class jitcsim.models.kuramoto_sde.**Kuramoto_II**(*par*)

Kuramoto model with noise.

$$\frac{d\theta_i}{dt} = \omega_i + \xi_i + \sum_{j=0}^{N-1} a_{i,j} \sin(y_j - y_i - \alpha)$$

Parameters

- **N** (*int*) – number of nodes
- **adj** (*2d array*) – adjacency matrix
- **t_initial** (*float, int*) – initial time of integration
- **t_final** (*float, int*) – final time of integration
- **t_transition** (*float, int*) – transition time
- **interval** (*float*) – time interval for sampling
- **sigma** (*float*) – noise aplitude of normal distribution
- **alpha** (*flaot*) – frustration
- **omega** (*float*) – initial angular frequencies
- **initial_state** (*array of size N*) – initial phase of oscillators

- **control** (*list of str*) – control parameters
- **use_omp** (*boolian*) – if *True* allow to use OpenMP
- **output** (*str*) – output directory
- **verbose** (*boolian*) – if *True* some information about the process will be desplayed.

f_sym()

Kuramoto model of type II

$$\frac{d\theta_i}{dt} = \omega_i + \xi_i + \sum_{j=0}^{N-1} a_{i,j} \sin(y_j - y_i - \alpha)$$

g_sym()

to do.

set_integrator_parameters(*atol=1e-05*, *rtol=0.01*, *min_step=1e-05*, *max_step=10.0*)

! set properties for integrator

simulate(*par=[]*, *mode_2pi=True*)

integrate the system of equations and return the coordinates and times

Parameters

- **par** (*list*) – list of values for control parameters in order of appearence in control
- **Return** (*dict(t, x)*) –
  - t times
  - x coordinates.

3.2 Tutorial

Considering the Kuramoto model. The following examples simulate a network of coupled Kuramoto oscillators with frustration, delay and noise.

3.2.1 ODE

Simulation of the Kuramoto model on complete network.

$$\frac{d\theta_i}{dt} = \omega_i + \sum_{j=0}^{N-1} a_{i,j} \sin(y_j - y_i - \alpha)$$

where θ_i is phase of oscillator i, ω_i angular frequency of oscillator i, α frustration, N number of oscillators, $a_{i,j}$ is an element of the adjacency matrix, $a_{i,j} = 1$ if there is a directed link from the node j to i; otherwise $a_{i,j} = 0$.

The output of the example is plotting the Kuramoto order parameter vs time.

Starting with a few imports

Listing 3.1: ../../jitcsim/examples/scripts/examples_kuramoto/00_ode_kuramoto_II.py

```
import numpy as np
from numpy import pi
from numpy.random import uniform, normal
from jitcsim.models.kuramoto_ode import Kuramoto_II
from jitcsim import plot_order
from jitcsim import make_network
```

then we set the parameters of the system. We consier a complete network with N oscillators.

```
np.random.seed(1)
N = 30
omega0 = normal(0, 0.1, N)
initial_state = uniform(-pi, pi, N)

# make complete network
net = make_network()
adj = net.complete(N)

parameters = {
    'N': N,                                 # number of nodes
    'adj': adj,                             # adjacency matrix
    't_initial': 0.,                        # initial time of integration
    "t_final": 100,                         # final time of integration
    't_transition': 2.0,                    # transition time
    "interval": 1.0,                        # time interval for sampling

    "coupling": 0.5 / (N - 1),              # coupling strength
    "alpha": 0.0,                           # frustration
    "omega": omega0,                        # initial angular frequencies
```

(continues on next page)

(continued from previous page)

```
    'initial_state': initial_state,     # initial phase of oscillators

    'integration_method': 'dopri5',     # integration method
    'control': [],                      # control parameters

    "use_omp": False,                   # use OpenMP
    "output": "data",                   # output directory
}
```

make an instance of the model and pass the parameters to it. Then compile the model.

```
sol = Kuramoto_II(parameters)
sol.compile()
```

and run the simulation

```
data = sol.simulate([])
x = data['x']
t = data['t']
```

to calculate and plot the order parameter of the system vs time:

```
order = sol.order_parameter(x)

plot_order(t,
           order,
           filename="data/00.png",
           xlabel="time",
           ylabel="r(t)")
```

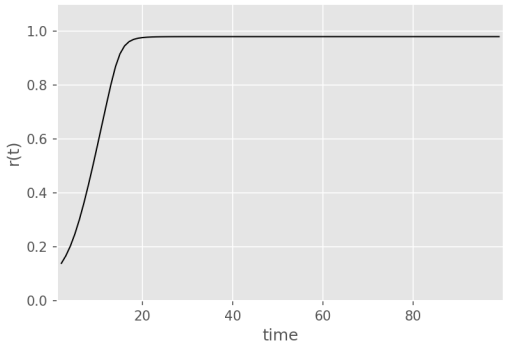

Fig. 3.1: Kuramoto order parameter vs time, for a complete network at a fixed coupling strength.

3.2.2 The main class for the DDE Kuramot model

class jitcsim.models.kuramoto_dde.**Kuramoto_Base**(*par*)

Base class for the Kuramoto model.

Parameters

- **N** (*int*) – number of nodes
- **adj** (*2d array*) – adjacency matrix
- **delays** (*2d array*) – delay matrix
- **maxdelay** (*float*) – maximum value of delays matrix
- **t_initial** (*float, int*) – initial time of integration
- **t_final** (*float, int*) – final time of integration
- **t_transition** (*float, int*) – transition time
- **interval** (*float*) – time interval for sampling
- **alpha** (*flaot*) – frustration
- **omega** (*float*) – initial angular frequencies
- **initial_state** (*array of size N*) – initial phase of oscillators
- **control** (*list of str*) – control parameters
- **use_omp** (*boolian*) – if *True* allow to use OpenMP
- **output** (*str*) – output directory
- **verbose** (*boolian*) – if *True* some information about the process will be desplayed.

simulate(*par*, *disc='blind'*, *step=0.1*, *propagations=1*, *min_distance=1e-05*, *max_step=None*, *shift_ratio=0.0001*, *mode_2pi=True*, ***integrator_params*)

integrate the system of equations and return the computed state of the system after integration and times

Parameters

- **par** (*list*) – values of control parameters in order of appearance in *control*
- **disc** (*str*) –

  type of discontinuities handling. The default value is blind

  - step_on [step_on_discontinuities]
  - blind [integrate_blindly]

  - adjust [adjust_diff]

- **step** (*float*) – argument for integrate_blindly aspired step size. The actual step size may be slightly adapted to make it divide the integration time. If *None*, *0*, or otherwise falsy, the maximum step size as set with *max_step* of *set_integration_parameters* is used.
- **propagations** (*int*) – argument for step_on_discontinuities: how often the discontinuity has to propagate to before it's considered smoothed.
- **min_distance** (*float*) – argument for step_on_discontinuities: If two required steps are closer than this, they will be treated as one.
- **max_step** (*float*) – argument for step_on_discontinuities: Retired parameter. Steps are now automatically adapted.
- **shift_ratio** (*float*) – argument for adjust_diff. Performs a zero-amplitude (backwards) *jump* whose *width* is *shift_ratio* times the distance to the previous anchor into the past. See the documentation of *jump* for the caveats of this and see *discontinuities* for more information on why you almost certainly need to use this or an alternative way to address initial discontinuities.
- **Return** (*dict(t, x)*) –
  - **t** times
  - **x** coordinates.

---

class jitcsim.models.kuramoto_dde.**Kuramoto_II**(*par*)

rhs()

Kuramoto model of type II

$$\frac{d\theta_i}{dt} = \omega_i + \sum_{j=0}^{N-1} a_{i,j} \sin(y_j(t - \tau_{ij}) - y_i - \alpha)$$

simulate(*par*, *disc='blind'*, *step=0.1*, *propagations=1*, *min_distance=1e-05*, *max_step=None*, *shift_ratio=0.0001*, *mode_2pi=True*, ***integrator_params*)

integrate the system of equations and return the computed state of the system after integration and times

Parameters

- **par** (*list*) – values of control parameters in order of appearance in *control*

- **disc** (*str*) –

  type of discontinuities handling. The default value is blind

  - step_on [step_on_discontinuities]
  - blind [integrate_blindly]
  - adjust [adjust_diff]

- **step** (*float*) – argument for integrate_blindly aspired step size. The actual step size may be slightly adapted to make it divide the integration time. If *None*, *0*, or otherwise falsy, the maximum step size as set with *max_step* of *set_integration_parameters* is used.
- **propagations** (*int*) – argument for step_on_discontinuities: how often the discontinuity has to propagate to before it's considered smoothed.
- **min_distance** (*float*) – argument for step_on_discontinuities: If two required steps are closer than this, they will be treated as one.
- **max_step** (*float*) – argument for step_on_discontinuities: Retired parameter. Steps are now automatically adapted.
- **shift_ratio** (*float*) – argument for adjust_diff. Performs a zero-amplitude (backwards) *jump* whose *width* is *shift_ratio* times the distance to the previous anchor into the past. See the documentation of *jump* for the caveats of this and see *discontinuities* for more information on why you almost certainly need to use this or an alternative way to address initial discontinuities.
- **Return** (*dict(t, x)*) –
  - **t** times
  - **x** coordinates.

3.2.3 Visualization

jitcsim.visualization.**plot_lyaps**(*x*, *y*, *filename='f.png'*, *color='k'*, *xlabel=None*, *ylabel=None*, *label=None*, *xlog=False*, *ylog=False*, *xlim=None*, *ylim=None*, *ax=None*, *close_fig=True*, ***kwargs*)

plot y vs x with given x and y labels

jitcsim.visualization.**plot_matrix**(*A*, *ax=None*, *ticks=None*, *vmax=None*, *vmin=None*, *labelsize=14*, *colorbar=True*, *aspect='equal'*, *cmap='seismic'*, *filename='F.png'*)

Plot given matrix using imshow module.

Parameters

- **A** (*2D numpy array*) –
- **ax** (*None or matplotlib axis object*) –
- **ticks** (*None or list of int or float*) –
- **vmax** (*None, float or int*) –
- **vmin** (*None, float or int*) –
- **labelsize** (*int*) –
- **colorbar** (*boolian,*) –
- **aspect** (*{'equal', 'auto'} or float, default: rcParams["image.aspect"] (default: 'equal')*) –
- **Colormap** (*cmap cmapstr or*) – The Colormap instance or registered colormap name used to map scalar data to colors. This parameter is ignored for RGB(A) data.
- **default** (*rcParams["image.cmap"] (default: 'seismic')*) – The Colormap instance or registered colormap name used to map scalar data to colors. This parameter is ignored for RGB(A) data.
- **filename** (*str, (default: F.png)*) – filename of the figure to be stored.

jitcsim.visualization.**plot_order**(*x*, *y*, *filename='f.png'*, *color='k'*, *xlabel=None*, *ylabel=None*, *label=None*, *ax=None*, *close_fig=True*, *figsize=(6, 4)*, ***kwargs*)

plot y vs x with given x and y labels

Parameters

- **x** (*list or array*) – values of x axis
- **y** (*list or array*) – values of y axis
- **filename** (*name*) – filename for figure, need to end with .png, .jpg , ...
- **color** (*str*) – line color
- **xlabel** (*None or str*) – label of x axis
- **ylabel** (*None or str*) – label of y axis
- **label** (*None or str*) – label of the curve

- **ax** (*None or matplotlib axis object*) –
- **close_fig** (*boolian*) – if *True* the figure object will be closed.

3.2.4 Networks

class jitcsim.networks.**make_network**(*seed=None*)

make different graphs ans return their adjacency matrices as a 1 dimensional double vector in stl library

barabasi(*N*, *m*)

Return random network using Barabási-Albert preferential attachment model. A graph of n nodes is grown by attaching new nodes each with m edges that are preferentially attached to existing nodes with high degree.

This is *networkx.barabasi_albert_graph* module.

Parameters

- **n** (*int*) – Number of nodes
- **m** (*int*) – Number of edges to attach from a new node to existing nodes

Returns
A – adjacency matrix

Return type
2D int numpy array

complete(*N*)

make complete all to all adjacency matrix

Parameters
N (*int*) – number of nodes

Returns
A – adjacency matrix

Return type
2D int numpy array

erdos_renyi(*N*, *p*, *directed=False*)

make Erdos Renyi network

Parameters

- **N** (*int*) – number of nodes
- **p** (*float*) – Probability for edge creation.

- **directed** (*(bool, optional (default=False))*) – If *True*, this function returns a directed adjacency matrix.

Returns
A – adjacency matrix

Return type
2D int numpy array

fgc(*N*, *k*, *omega*, *gamma=0.4*)

Frequency Gap-conditioned (FGC) network

Parameters

- **N** (*int*) – the number of oscillators in the system
- **k** (*int*) – degree of the network
- **gamma** (*float*) – minimal frequency gap

Returns
A – adjacency matrix

Return type
2D int numpy array

3.2.5 Utility

jitcsim.utility.**display_time**(*time*, *message=''*)

display elapsed time in hours, minutes, seconds :param time: elaspsed time in seconds :type time: float

jitcsim.utility.**flatten**(*t*)

flatten a list of list

Parameters

- **t** (*list of list*) –
- **Return** – flattend list

jitcsim.utility.**kuramoto_correlation**(*x*)

Calculate the Kuramoto correlation between phase of nodes

Parameters
x (*numpy array, float*) – input phase of oscillators

Returns
cor – The correlation matrix.

Return type
2D numpy array

jitcsim.utility.**local_order_parameter**(*phases*, *indices*)

calculate the local order parameter of given indices

Parameters

- **phases** (*numpy 2D array (num_time_step by num_nodes)*) – phase of each node
- **indices** (*array(list) of int*) – indices of nodes to measure their order parameters;

Returns
r – Kuramoto order parameter.

Return type
float

jitcsim.utility.**order_parameter**(*phases*)

calculate the Kuramoto order parameter.

Parameters
phases (*numpy 2D array (num_time_step by num_nodes)*) – phase of oscillators

Returns
r – Kuramotoorder parameter.

Return type
float

jitcsim.utility.**timer**(*func*)

decorator to measure elapsed time :param func: function to be decorated :type func: function

BIBLIOGRAPHY

[Shampine2001] Shampine, L.F. and Thompson, S., 2001. Solving ddes in matlab. Applied Numerical Mathematics, 37(4), pp.441-458., 10.1016/S0168-9274(00)00055-6.

[Bogacki1989] Bogacki, P. and Shampine, L.F., 1989. A 3 (2) pair of Runge-Kutta formulas. Applied Mathematics Letters, 2(4), pp.321-325. 10.1016/0893-9659(89)90079-7.

[Rackauckas2017] Rackauckas, C. and Nie, Q., 2017. Adaptive methods for stochastic differential equations via natural embeddings and rejection sampling with memory. Discrete and continuous dynamical systems. Series B, 22(7), p.2731, 10.3934/dcdsb.2017133.

[Robler2010] Rößler, A., 2010. Runge–Kutta methods for the strong approximation of solutions of stochastic differential equations. SIAM Journal on Numerical Analysis, 48(3), pp.922-952. 10.1137/09076636X.

[Yeung1999] Yeung, M.S. and Strogatz, S.H., 1999. Time delay in the Kuramoto model of coupled oscillators. Physical Review Letters, 82(3), p.648. Figure 3.

[Leyva2013] Leyva, I., Navas, A., Sendina-Nadal, I., Almendral, J.A., Buldu, J.M., Zanin, M., Papo, D. and Boccaletti, S., 2013. Explosive transitions to synchronization in networks of phase oscillators. Scientific reports, 3(1), pp.1-5.

[Kachhvah] Kachhvah, A.D. and Jalan, S., 2017. Multiplexing induced explosive synchronization in Kuramoto oscillators with inertia. EPL (Europhysics Letters), 119(6), p.60005.

PYTHON MODULE INDEX

j