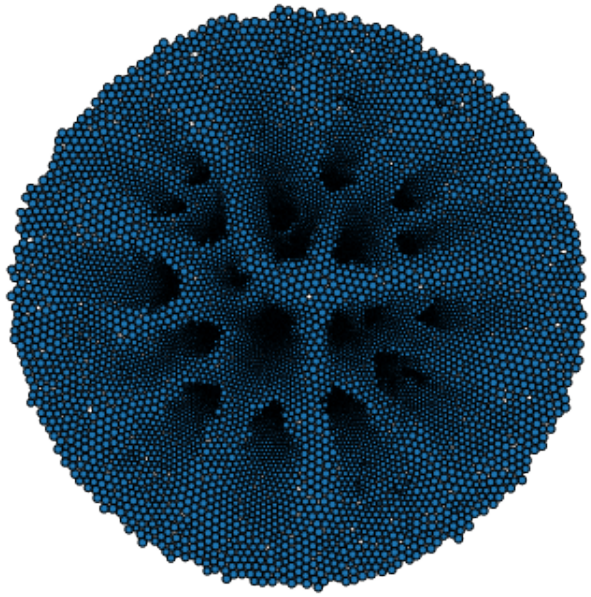
SiSyPHE

Antoine Diez

Nov 27, 2023

CONTENTS



The SiSyPHE library builds on recent advances in hardware and software for the efficient simulation of **large scale interacting particle systems**, both on the **GPU** and on the CPU. The implementation is based on recent libraries originally developed for machine learning purposes to significantly accelerate tensor (array) computations, namely the PyTorch package and the KeOps library. The **versatile object-oriented Python interface** is well suited to the comparison of new and classical many-particle models, enabling ambitious numerical experiments and leading to novel conjectures. The SiSyPHE library speeds up both traditional Python and low-level implementations by **one to three orders of magnitude** for systems with up to **several millions** of particles.

The project is hosted on GitHub, under the permissive MIT license.

CHAPTER

ONE

CITATION

If you use SiSyPHE in a research paper, please cite the JOSS publication :

```
@article{Diez2021,
doi = {10.21105/joss.03653},
url = {https://doi.org/10.21105/joss.03653},
year = {2021},
publisher = {The Open Journal},
volume = {6},
number = {65},
pages = {3653},
author = {Antoine Diez},
title = {`SiSyPHE`: A Python package for the Simulation of Systems of interacting mean-
↪field Particles with High Efficiency},
journal = {Journal of Open Source Software}}
```

Diez, A., (2021). SiSyPHE: A Python package for the Simulation of Systems of interacting mean-field Particles with High Efficiency. Journal of Open Source Software, 6(65), 3653, https://doi.org/10.21105/joss.03653

CHAPTER

TWO

CONTRIBUTING

Contributions to make SiSyPHE grow are warmly welcome! Examples of possible (and ongoing) developments include the following.

- The implementation of new models.
- The implementation of more complex boundary conditions and of models on *non-flat* manifolds.
- An improved visualization method (currently only basic visualization functions relying on Matplotlib are implemented).

Contributions can be made by opening an issue on the GitHub repository, via a pull request or by contacting directly the author.

CHAPTER

THREE

AUTHOR

Antoine Diez, Imperial College London

3.1 Acknowledgments

The development of this library would not have been possible without the help of Jean Feydy, his constant support and precious advice. This project was initiated by Pierre Degond and has grown out of many discussions with him.

CHAPTER

FOUR

TABLE OF CONTENTS

4.1 Background and motivation

4.1.1 Scope of application

Over the past decades, the study of systems of particles has become an important part of many research areas, from theoretical physics to applied biology and computational mathematics. Some current research trends include the following.

- Since the 80's, there is a joint effort from biologists, physicists, computer graphics scientists and more recently mathematicians to propose accurate models for large **self-organized** animal societies. While the motivations of the different communities are largely independent, a commmon goal is to be able to explain and reproduce the **emergence of complex patterns from simple interaction rules**. Typical examples of complex systems include flocks of birds, fish schools, herds of mammals or ant colonies. It seems reasonable to consider that none of these animals has an accute consciousness of the whole system organization but rather follows simple behavioral patterns: stay close to the group, do not collide with another individual, copy the attitude of the close neighbours etc. These models are often called **swarming models** or **collective dynamics** models [1, 4, 13, 41].
- The microscopic world if full of complex systems made of many simple entities. Colonies of bacteria [42] are a natural example reminiscent of the macroscopic animal societies described above. But complex patterns can also emerge from **systems of non-living particles**. Our body is actually a very complex self-organized assembly of cells which dictate every aspect of our life: our brain works thanks to the communications between billions of neurons [2, 21], the reproduction is based on the competition between millions of spermatozoa [9] and sometimes, death is sadly caused by the relentless division of cancer cells. Unlike the previous models, the communication between the particles cannot be based on their visual perception, but rather on chemical agents, on physical (geometrical) constraints etc.
- On a completely different side, there is an ever growing number of methods in computational mathematics which are based on the simulation of systems of particles. The pioneering Particle Swarm Optimization method [29] has shown how biologically-inspired artificial systems of particles can be used to solve tough optimization problems. Following these ideas, other **Swarm Intelligence** optimization algorithms have been proposed up to very recently [22, 33, 38, 39]. Since the beginning of the 2000's, particle systems are also at the core of filtering [10, 17, 18, 20, 28] and sampling methods [3, 8]. Recently, a particle-based interpretation [7, 12, 31, 34, 35] of the training task of neural networks has lead to new theoretical convergence results.

4.1.2 Why do we need to simulate particle systems?

Beyond the self-explanatory applications for particle-based algorithms in computational mathematics, the simulation of systems of particles is also a crucial **modelling tool** in Physics and Biology.

- On the one hand, field experiments are useful to collect data and trigger new modelling ideas. On the other hand, numerical simulations become necessary to test these ideas and to calibrate the models. **Numerical experiments** can be conducted to identify which mechanisms are able to produce a specific phenomena in a **controlled environment** [6].
- On a more theoretical side, the direct mathematical study of particle systems can rapidly become incredibly difficult. Inspired by the kinetic theory of gases, many **mesoscopic** and **macroscopic** models have been proposed to model the average statistical behavior of particle systems rather than the individual motion of each particle [4, 16, 37]. Within this framework, a particle system is rather seen as a **fluid** and it is decribed by Partial Differential Equations (PDE) reminiscent from the theory of fluid dynamics. PDE models are more easily theoretically and numerically tractable. However, **cheking the validity** of these models is not always easy and is sometimes only postulated based on phenomenological considerations. In order to design good models, it is often necessary to go back-and-forth between the PDE models and the numerical simulation of the underlying particle systems.

The development of the SiSyPHE library was initially motivated by the study of *body-oriented particles* [15]. The (formal) derivation of a macroscopic PDE model from the particle system has lead to a novel conjecture which postulates the existence of a class of so-called **bulk topological states** [14]. The quantitative comparison between this theoretical prediction and the numerical simulation of the particle system in a suitable regime (with more than 10^6 particles) has confirmed the existence of these new states of matter. The study of their physical properties which are observed in the numerical experiments but not readily explained by the PDE model is an ongoing work.

4.1.3 Mean-field particle systems

Currently, the models implemented in the SiSyPHE library belong to the family of **mean-field models**. It means that the motion of each particle is influenced by the average behavior of the whole system. The Vicsek model [16, 40] and the Cucker-Smale model [4, 11, 24] are two popular examples of mean-field models where each particle tries to move in the average direction of motion of its neighbours (it produces a so-called *flocking* behavior).

The mathematical point of view

From a mathematical point of view, a mean-field particle system with N particles is defined as a Markov process in $(\mathbb{R}^d)^N$ with a generator $\mathcal{L}_N$ whose action on a test function φ_N is of the form:

$$\mathcal{L}_N\varphi_N(x^1,\dots,x^N) = \sum_{i=1}^N L_{\mu_{\mathbf{x}^N}} \diamond_i \varphi_N(x^1,\dots,x^N),$$

where $\mathbf{x}^N = (x^1,\dots,x^N) \in (\mathbb{R}^d)^N$ and

$$\mu_{\mathbf{x}^N} := \frac{1}{N}\sum_{i=1}^N \delta_{x^i},$$

is the so-called **empirical measure**. The operator $L_{\mu_{\mathbf{x}_N}}$ depends on the empirical measure and acts on one-variable test functions on $\mathbb{R}^d$. Given an operator L and a test function φ_N, the notation $L \diamond_i \varphi_N$ denotes the function

$$L \diamond_i \varphi_N : (x^1,\dots,x^N) \in (\mathbb{R}^d)^N \mapsto L[x \mapsto \varphi_N(x_1,\dots,x^{i-1},x,x^{i+1},\dots,x^N)](x^i) \in \mathbb{R}.$$

The dynamics of the mean-field system depends on the operator $L_{\mu_{\mathbf{x}_N}}$ which is typically either a diffusion operator [16, 24] or a jump operator [2, 19]. In the first case, the mean-field particle systems can be alternatively defined by a system of N coupled Stochastic Differential Equations of the form:

$$\mathrm{d}X_t^i = b(X_t^i,\mu_{\mathcal{X}_t^N})\mathrm{d}t + \sigma(X_t^i,\mu_{\mathcal{X}_t^N})\mathrm{d}B_t^i, \quad i \in \{1,\dots,N\},$$

$$\mu_{\mathcal{X}_t^N} := \frac{1}{N}\sum_{i=1}^{N} \delta_{X_t^i},$$

where $(B_t^i)_t$ are N independent Brownian motions and the coefficients b and σ are called the *drift* and *diffusion* coefficients. In most cases, these coefficients are *linear* or *quasi-linear* which means that they are of the form

$$b(X_t^i, \mu_{\mathcal{X}_t^N}) = \tilde{b}\left(X_t^i, \frac{1}{N}\sum_{j=1}^{N} K(X_t^i, X_t^j)\right),$$

for two given functions $\tilde{b} : \mathbb{R}^d \times \mathbb{R}^n \to \mathbb{R}^d$ and $K : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}^n$.

When the particles are initially statistically independent, it can be shown that when $N \to +\infty$, each particle X_t^i converges towards an independent copy of the solution of the so-called **McKean-Vlasov** diffusion process [30, 32, 36] defined by the Stochastic Differential Equation

$$\mathrm{d}\overline{X}_t = b(\overline{X}_t, f_t)\mathrm{d}t + \sigma(\overline{X}_t, f_t)\mathrm{d}B_t,$$

where $(B_t)_t$ is a Brownian motion and f_t is the law of the process $\overline{X}_t$. It satisfies the **Fokker-Planck** Partial Differential Equation

$$\partial_t f_t = -\nabla \cdot (b(x, f_t)f_t) + \frac{1}{2}\sum_{i,j=1}^{N} \partial_{x_i}\partial_{x_j}(a_{ij}(x, f_t)f_t),$$

with $x = (x_1, \dots, x_d) \in \mathbb{R}^d$ and $a = \sigma\sigma^{\mathrm{T}}$. This phenomenon is called **propagation of chaos**, following the terminology introduced by Kac in the 50's [25, 27, 30, 32, 36].

Note: A popular example of mean-field particle system is the (stochastic) Cucker-Smale model [4, 11, 24]. Each particle is defined by its position $X_t^i \in \mathbb{R}^d$ and its velocity $V_t^i \in \mathbb{R}^d$ which evolve according to the system of $2N$ Stochastic Differential Equations:

$$\mathrm{d}X_t^i = V_t^i \mathrm{d}t, \quad \mathrm{d}V_t^i = \frac{1}{N}\sum_{i=1}^{N} K(|X_t^j - X_t^i|)(V_t^j - V_t^i)\mathrm{d}t + \mathrm{d}B_t^i,$$

where $K : [0, +\infty) \to [0, +\infty)$ is an **observation kernel** which models the visual perception of the particles. In this example, the function K is a smooth function vanishing at infinity and the communication between the particles is based on the distance between them. The motion of the particles follows the Newton's laws of motion with an additional stochastic term. The term $V_t^j - V_t^i$ is a relaxation force (for a quadratic potential) which tends to align the velocities of particle i and particle j.

Simulating mean-field particle systems

On a computer, the above examples which are time-continuous needs to be discretized, using for instance one of the classical numerical schemes for Stochastic Differential Equations. Then the difficulty lies in the evaluation of the empirical measure **at each time-step** of the numerical scheme.

In the example of the Cucker-Smale model, the force exerted on a particle is the sum of N small relaxation forces of order $1/N$. The total number of operations required is thus of order $\mathcal{O}(N)$ **for each particle**. Since there are N particles, the total time complexity of the algorithm is thus $\mathcal{O}(N^2 T)$ where T is the total number of iterations of the numerical scheme.

This **quadratic cost** is the main bottleneck in the simulation of mean-field particle systems. As explained in the documentation of the KeOps library, the evaluation of the N forces at time t

$$F_i(t) = \frac{1}{N}\sum_{i=1}^{N} K(|X_t^j - X_t^i|)(V_t^j - V_t^i), \quad i \in \{1, \dots, N\},$$

is called a **kernel operation** and can be understood as a discrete convolution operation (or matrix-vector product) between the matrix of distances $(K_{ij})_{i,j\in\{1\ldots,N\}}$ where $K_{ij} = K(|X_t^j - X_t^i|)$ and the vector of velocities $(V_t^j - V_t^i)_{j\in\{1,\ldots,N\}}$. When N is large (say $N > 10^4$), such operation is too costly even for array-based programming languages such as Matlab: the $N \times N$ kernel matrix $(K_{ij})_{i,j}$ would simply not fit into the memory. With lower level languages (Fortran, C), this operation can be implemented more efficiently with two nested loops but with a significantly higher global coding effort and with less versatility.

Over the past decades, several workarounds have been proposed. Popular methods include

- the low-rank decomposition of the kernel matrix,
- the fast-multipole methods [23] which to treat differently short- and long-range interactions,
- the Verlet list method which is based on a grid decompostion of the spatial domain to reduce the problem to only short-range interactions between subsets of the particle system,
- the Random Batch Method [26] which is based on a stochastic approximation where only interactions between randomly sampled subsets (*batches*) of the particle system are computed.

All these methods require an significant amount of work, either in terms of code or to justify the approximation procedures.

4.1.4 The SiSyPHE library

The present implementation is based on recent libraries originally developed for machine learning purposes to significantly accelerate such tensor (array) computations, namely the PyTorch package and the KeOps library [5]. Using the KeOps framework, the kernel matrix is a **symbolic matrix** defined by a mathematical formula and no approximation is required in the computation of the interactions (up to the time discretization). The SiSyPHE library speeds up both traditional Python and low-level CPU implementations by **one to three orders of magnitude** for systems with up to several millions of particles. Although the library is mainly intended to be used on a GPU, the implementation is fully functional on the CPU with a significant gain in efficiency.

Moreover, the **versatile object-oriented Python interface** is well suited to the comparison and study of new and classical many-particle models. This aspect is fundamental in applications in order to conduct ambitious numerical experiments in a systematic framework, even for particles with a complex structure and with a significantly reduced computational cost

4.1.5 References

4.2 Installation

4.2.1 Requirements

- **Python 3** with packages **NumPy** and **SciPy**
- **PyTorch** : version>= 1.5
- **PyKeops** : version>= 1.5

Note: In order to have a working version of PyKeOps, it may be necessary to install and upgrade Cmake to Cmake>=3.18 as discussed here.

4.2.2 Installation

Using pip

In a terminal, type:

```
pip install sisyphe
```

On Google Colab

The easiest way to get a working version of SiSyPHE is to use the free virtual machines provided by Google Colab.

1. On a new Colab notebook, navigate to Edit→Notebook Settings and select GPU from the Hardware Accelerator drop-down.
2. Install PyKeops with the Colab specifications **first** by typing

   ```
   !pip install pykeops[colab]
   ```

3. Install SiSyPHE by typing

   ```
   !pip install sisyphe
   ```

From source

Alternatively, you can clone the git repository at a location of your choice.

4.2.3 Testing the installation

The following test function will check the configuration and run the simulation of a system of body-oriented particles (see the example gallery). This simulation uses the main features of the SiSyPHE library: a complex system, nontrivial boundary conditions, a sampling-based interaction mechanism, the blocksparse reduction method and a nontrivial initial condition. Moreover, this model is theoretically well-understood which provides a theoretical baseline to check the accuracy of the output of the simulation.

Warning: This function is mainly intended to be runned on a GPU. Running this function on a CPU will take a long time! See below for a quick testing procedure.

In a Python terminal, type

```
import sisyphe
sisyphe.test_sisyphe()
```

On a fresh environment, it should return

```
Welcome! This test function will create a system of body-oriented particles in a␣
↪``milling configuration'' (cf. the example gallery). The test will be considered as␣
↪successful if the computed milling speed is within a 5% relative error range around␣
↪the theoretical value.
```

(continues on next page)

(continued from previous page)

```
 Running test, this may take a few minutes...

 Check configuration...
[pyKeOps] Initializing build folder for dtype=float32 and lang=torch in /root/.cache/
↪pykeops-1.5-cpython-37 ... done.
[pyKeOps] Compiling libKeOpstorch180bebcc11 in /root/.cache/pykeops-1.5-cpython-37:
       formula: Sum_Reduction(SqNorm2(x - y),1)
       aliases: x = Vi(0,3); y = Vj(1,3);
       dtype  : float32
...
[pyKeOps] Compiling pybind11 template libKeOps_template_574e4b20be in /root/.cache/
↪pykeops-1.5-cpython-37 ... done.
Done.

pyKeOps with torch bindings is working!

Done.

 Sample an initial condition...
Done.

 Create a model...
Done.

 Run the simulation...
[pyKeOps] Compiling libKeOpstorch269aaf150e in /root/.cache/pykeops-1.5-cpython-37:
       formula: Sum_Reduction((Step((Var(5,1,2) - Sum(Square((((Var(0,3,1) - Var(1,3,0))␣
↪+ (Step(((Minus(Var(2,3,2)) / Var(3,1,2)) - (Var(0,3,1) - Var(1,3,0)))) * Var(2,3,2)))␣
↪- (Step(((Var(0,3,1) - Var(1,3,0)) - (Var(2,3,2) / Var(4,1,2)))) * Var(2,3,2)))))))) *␣
↪Var(6,16,1)),0)
       aliases: Var(0,3,1); Var(1,3,0); Var(2,3,2); Var(3,1,2); Var(4,1,2); Var(5,1,2);␣
↪Var(6,16,1);
       dtype  : float32
...
Done.
Progress:100%Done.

 Check the result...
Done.

 SiSyPHE is working!
```

The core functionalities of the library are automatically and continuously tested through a GitHub workflow based on the module `sisyphe.test.quick_test`. The testing functions include basic computations on simple examples (computation of simple local averages in various situations) and small scales simulations. Note that unlike the function `sisyphe.test_sisyphe()`, these testing functions do not check the accuracy of the output of the simulations but only check that the code runs without errors. It is possible to use the Pytest package to run these tests manually: on a Python terminal, type

```
import pytest
from sisyphe.test import quick_test
retcode = pytest.main([quick_test.__file__,])
```

4.3 Benchmarks

We compare

- a Fortran implementation due to Sébastien Motsch using the Verlet list method with double precision float numbers, run on the NextGen compute cluster at Imperial College London.
- the CPU version of SiSyPHE with double precision tensors (float64) and the blocksparse-reduction method (BSR), run on an Intel MacBook Pro (2GHz Intel Core i5 processor with 8 Go of memory) ;
- the GPU version of SiSyPHE with single precision tensors (float32) with and without the *blocksparse-reduction method* (BSR), run on a GPU cluster at Imperial College London using an nVidia GTX 2080 Ti GPU ;
- the GPU version of SiSyPHE with double precision tensors (float64) with and without the *blocksparse-reduction method* (BSR), run on a GPU cluster at Imperial College London using an nVidia GTX 2080 Ti GPU.

Note: The choice of single or double precision tensors is automatically made according to the floating point precision of the input tensors.

We run a `Vicsek` model in a square periodic box with fixed parameters $L = 100$, $\nu = 5$, $\sigma = 1$, $c = R$, $dt = 0.01$ and various choices of N and R. The simulation is run for 10 units of time (i.e. 1000 iterations).

For each value of N, three values of R are tested which correspond to a dilute regime, a moderate regime and a dense mean-field regime. When the particles are uniformly scattered in the box, the average number of neighbours in the three regimes is respectively ~ 3, ~ 30 and ~ 300. The regime has a strong effect on the efficiency of the Verlet list and blocksparse reduction methods.

In the table below, the total computation times are given in seconds except when they exceed 12 hours. The best times are indicated in **bold** and the worst times in *italic*.

		Fortran	sisyphe64 CPU BSR	sisyphe32 GPU	sisyphe32 GPU BSR	sisyphe64 GPU	sisyphe64 GPU BSR
N = 10k	R = 1	19s	*26s*	**2.1s**	3.3s	13s	3.6s
	R = 3	*59s*	29s		3.4s		3.9s
	R = 10	*494s*	69s		3.4s		7.9s
N = 100k	R = 0.3	309s	*323s*	29s	**4.3s**	973s	9.3s
	R = 1	*1522s*	384s		**4.5s**		11s
	R = 3	*3286s*	796s		**4.9s**		28s
N = 1M	R = 0.1	*>12h*	6711s	2738s	**22s**	*>12h*	120s
	R = 0.3	*>12h*	6992s		**23s**		135s
	R = 1	*>12h*	9245s		**26s**		194s

The GPU implementation is at least 5 times faster in the dilute regime and outperform the other methods by three orders of magnitude in the mean-field regime with large values of N. Without the block-sparse reduction method, the GPU implementation does suffer from the quadratic complexity. The block-sparse reduction method is less sensitive to the density of particles than the traditional Verlet list method. It comes at the price of a higher memory cost (see *Tutorial 04: Block-sparse reduction*) but allows to run large scale simulations even on a CPU. On a CPU, the performances are not critically affected by the precision of the floating-point format so only simulations with double precision tensors are shown.

Note: The parameters of the blocksparse reduction method are chosen using the method

best_blocksparse_parameters() (see *Tutorial 04: Block-sparse reduction*). As a guideline, in this example and on the GPU, the best number of cells is respectively $\sim 15^2$, $\sim 42^2$ and $\sim 70^2$ for $N = 10^4$, $N = 10^5$ and $N = 10^6$ with sisyphe32 and $\sim 25^2$, $\sim 60^2$ and $\sim 110^2$ for sisyphe64. Note that the number of cells will be automatically capped at $(L/R)^2$. For the last case (sisyphe64 with $R = 1$), the maximal number of cells depends on the memory available.

As an example, the case sisyphe32 GPU BSR with $N = 10^5$ and $R = 1$ is obtained with the following script.

```
import time
import math
import torch
import sisyphe.models as models
from sisyphe.display import save

dtype = torch.cuda.FloatTensor

# Replace by the following line for double precision tensors.
# dtype = torch.cuda.DoubleTensor

N = 100000
L = 100.
dt = .01

nu = 5.
sigma = 1.

R = 1.
c = R

# The type of the initial condition will determine the floating point precision.

pos = L*torch.rand((N,2)).type(dtype)
vel = torch.randn(N,2).type(dtype)
vel = vel/torch.norm(vel,dim=1).reshape((N,1))

simu = models.Vicsek(
    pos = pos,
    vel = vel,
    v = R,
    sigma = sigma,
    nu = nu,
    interaction_radius = R,
    box_size = L,
    dt = dt,
    block_sparse_reduction = True,
    number_of_cells = 42**2)

simu.__next__() # GPU warmup

s = time.time()
data = save(simu, [10.], [], [])
e = time.time()
```

(continues on next page)

(continued from previous page)

```
print(e-s)
```

4.4 Tutorials

Basic features.

4.4.1 Tutorial 01: Particles and models

A particle system is an instance of one of the classes defined in the module *sisyphe.particles*.

Particles
The basic class *sisyphe.particles.Particles* defines a particle system by the positions.

Kinetic particles
The class *sisyphe.particles.KineticParticles* defines a particle system by the positions and the velocities.

Body-oriented particles.
The class *sisyphe.particles.BOParticles* defines a particle system in 3D by the positions and the body-orientations which are a rotation matrices in $SO(3)$ stored as quaternions.

A model is a subclass of a particle class. Several examples are defined in the module *sisyphe.models*. For example, let us create an instance of the Vicsek model *sisyphe.models.Vicsek* which is a subclass of *sisyphe.particles.KineticParticles*.

First, some standard imports...

```
import time
import torch
```

If CUDA is available, the computations will be done on the GPU and on the CPU otherwise. The type of the tensors (simple or double precision) are defined by the type of the initial conditions. Here and throughout the documentation, we work with single precision tensors.

```
use_cuda = torch.cuda.is_available()
dtype = torch.cuda.FloatTensor if use_cuda else torch.FloatTensor
```

We take initially N particles uniformly scattered in a box of size L with uniformly sampled directions of motion.

```
N = 10000
L = 100

pos = L*torch.rand((N,2)).type(dtype)
vel = torch.randn(N,2).type(dtype)
vel = vel/torch.norm(vel,dim=1).reshape((N,1))
```

Then we define the interaction radius R, the speed of the particles c and the drift and diffusion coefficients, respectively ν and σ.

```
R = 5.
c = 1.
nu = 3.
sigma = 1.
```

We take a small discretisation time step.

```
dt = .01
```

Finally, we define an instance of the Vicsek model with these parameters.

```
from sisyphe.models import Vicsek

simu = Vicsek(
    pos = pos,
    vel = vel,
    v = c,
    sigma = sigma,
    nu = nu,
    interaction_radius = R,
    box_size = L,
    dt = dt)
```

Note: The boundary conditions are periodic by default, see *Tutorial 03: Boundary conditions*.

So far, nothing has been computed. All the particles are implemented as Python iterators: in order to compute the next time step of the algorithm, we can call the method `__next__()`. This method increments the iteration counter by one and updates all the relevant quantities (positions and velocities) by calling the method *update()* which defines the model.

```
print("Current iteration: "+ str(simu.iteration))
simu.__next__()
print("Current iteration: "+ str(simu.iteration))
```

Out:

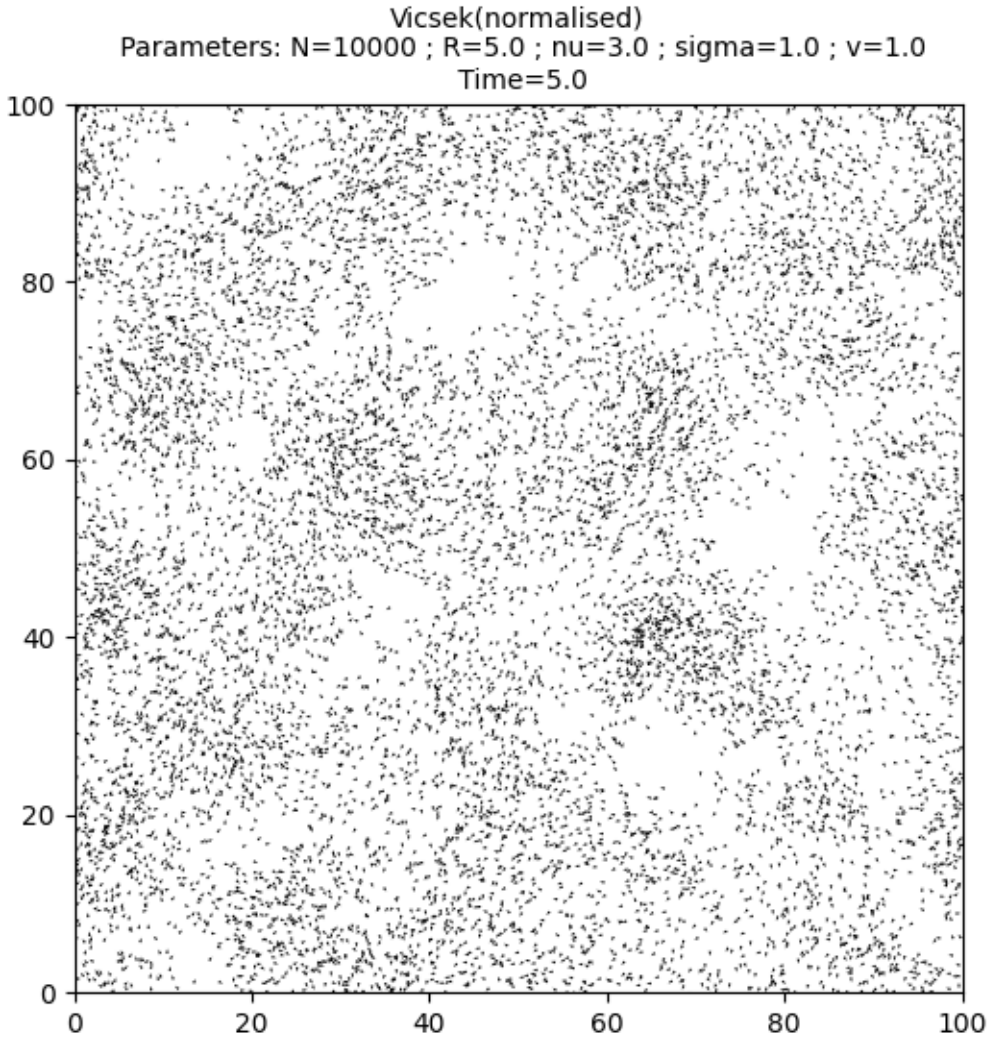
```
Current iteration: 0
Current iteration: 1
```

On a longer time interval, we can use the methods in the module *sisyphe.display*. For instance, let us fix a list of time frames.

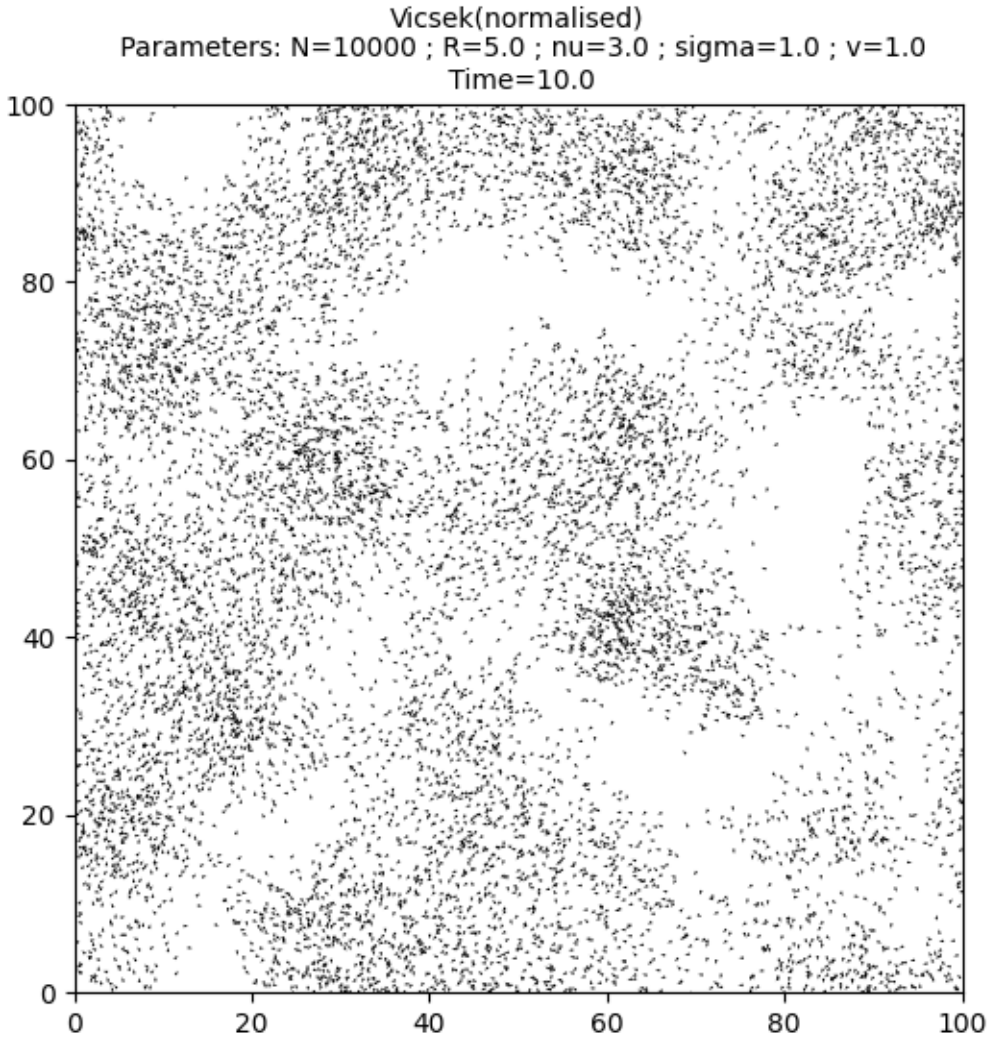
```
frames = [5., 10., 30., 50., 75., 100]
```

Using the method *sisyphe.display.display_kinetic_particles()*, the simulation will run until the last time in the list `frames`. The method also displays a scatter plot of the particle system at each of the times specified in the list and finally compute and plot the order parameter.

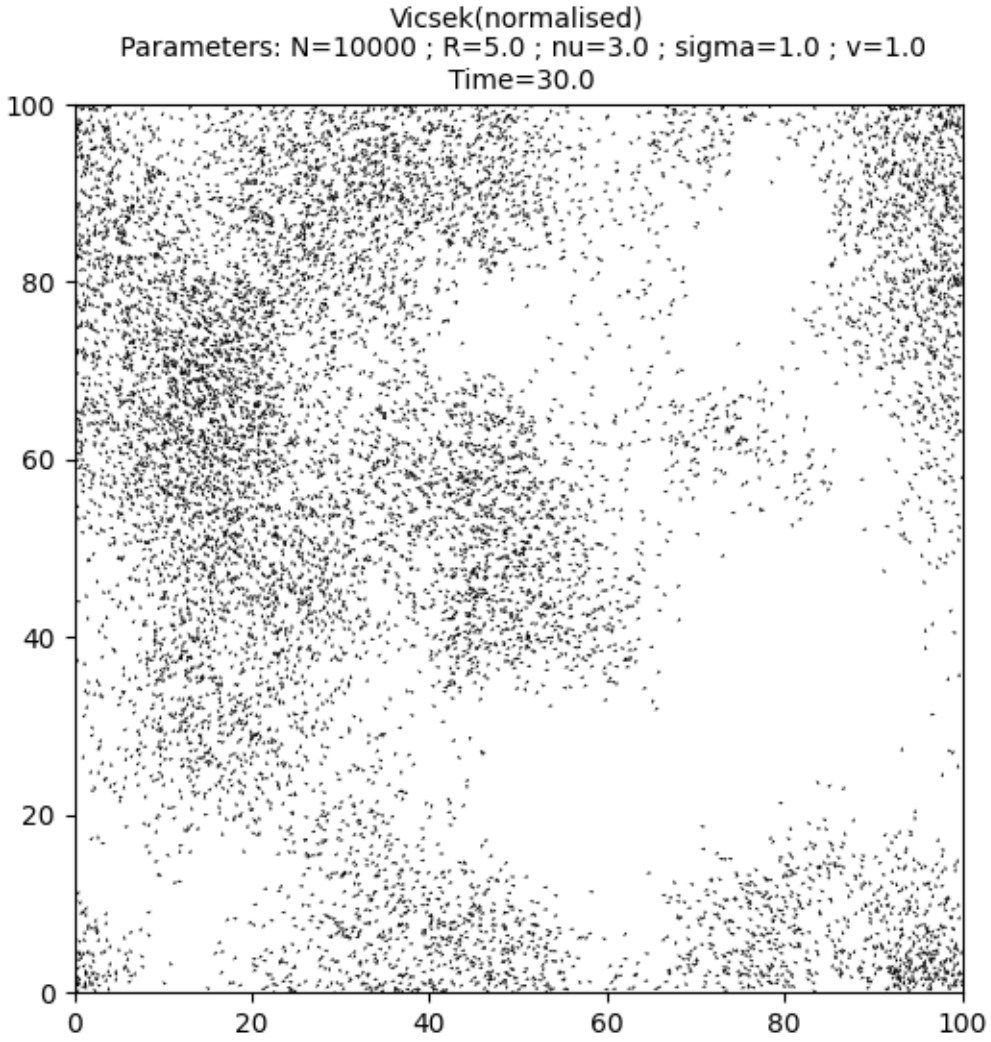
```
from sisyphe.display import display_kinetic_particles

s = time.time()
it, op = display_kinetic_particles(simu, frames, order=True)
e = time.time()
```


Vicsek(normalised)
Parameters: N=10000 ; R=5.0 ; nu=3.0 ; sigma=1.0 ; v=1.0
Time=5.0
100
80
60
40
20
0
0
20
40
60
80
100

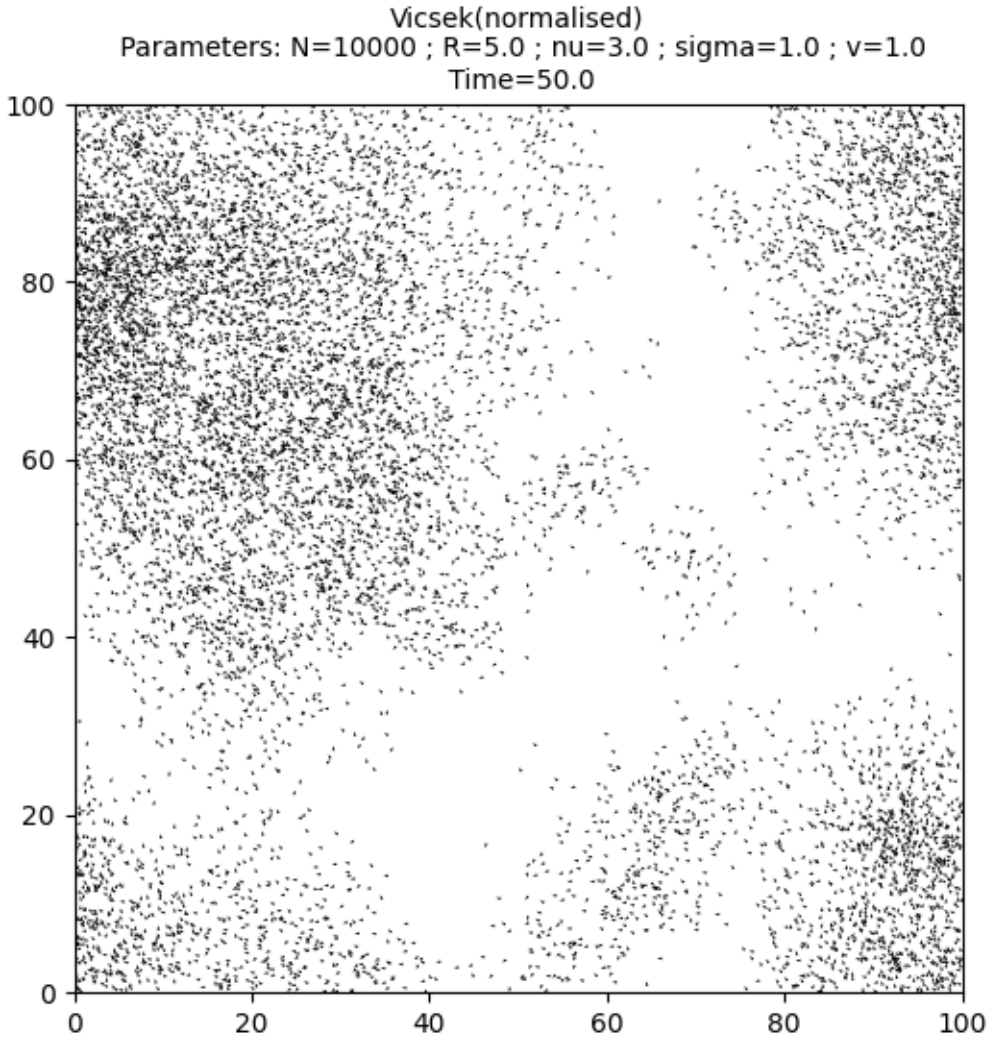
•


Vicsek(normalised)
Parameters: N=10000 ; R=5.0 ; nu=3.0 ; sigma=1.0 ; v=1.0
Time=10.0
100
80
60
40
20
0
0
20
40
60
80
100

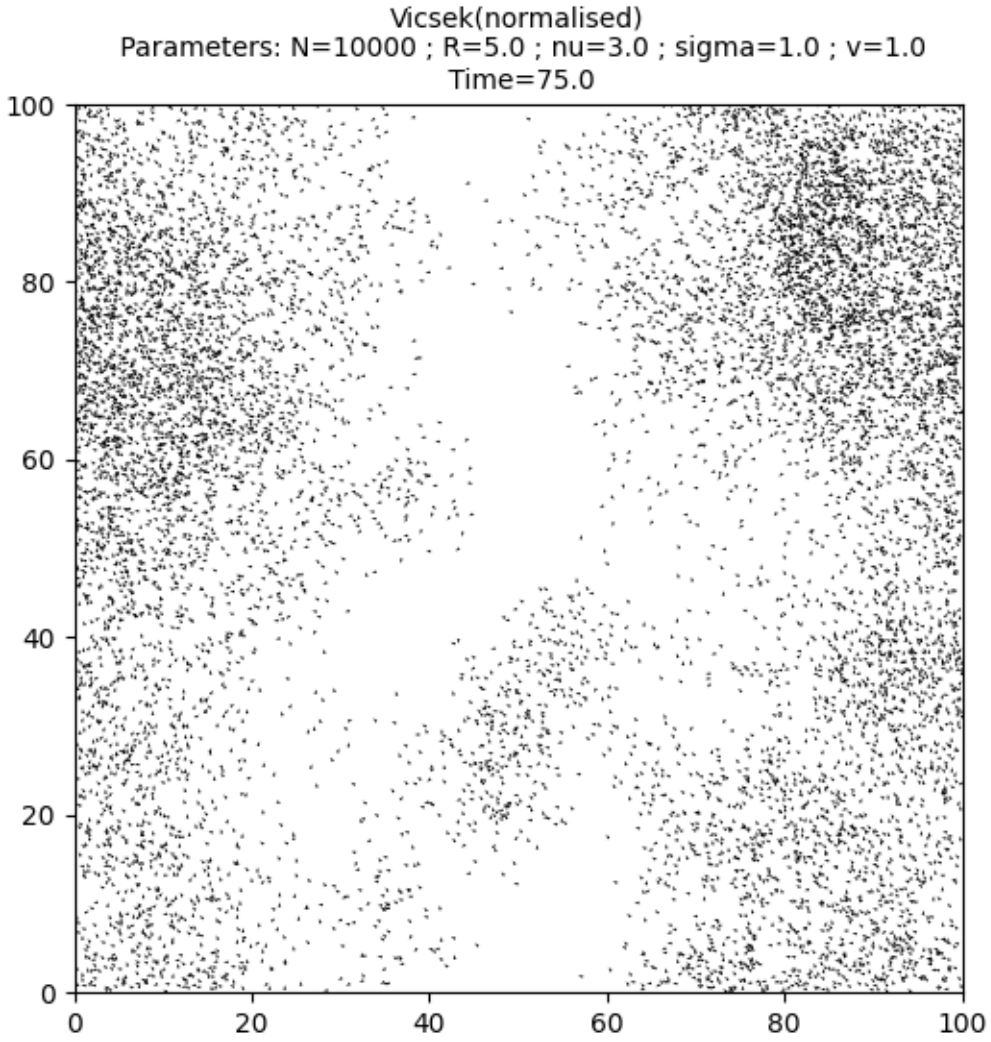
-


Vicsek(normalised)
Parameters: N=10000 ; R=5.0 ; nu=3.0 ; sigma=1.0 ; v=1.0
Time=30.0
100
80
60
40
20
0
0
20
40
60
80
100

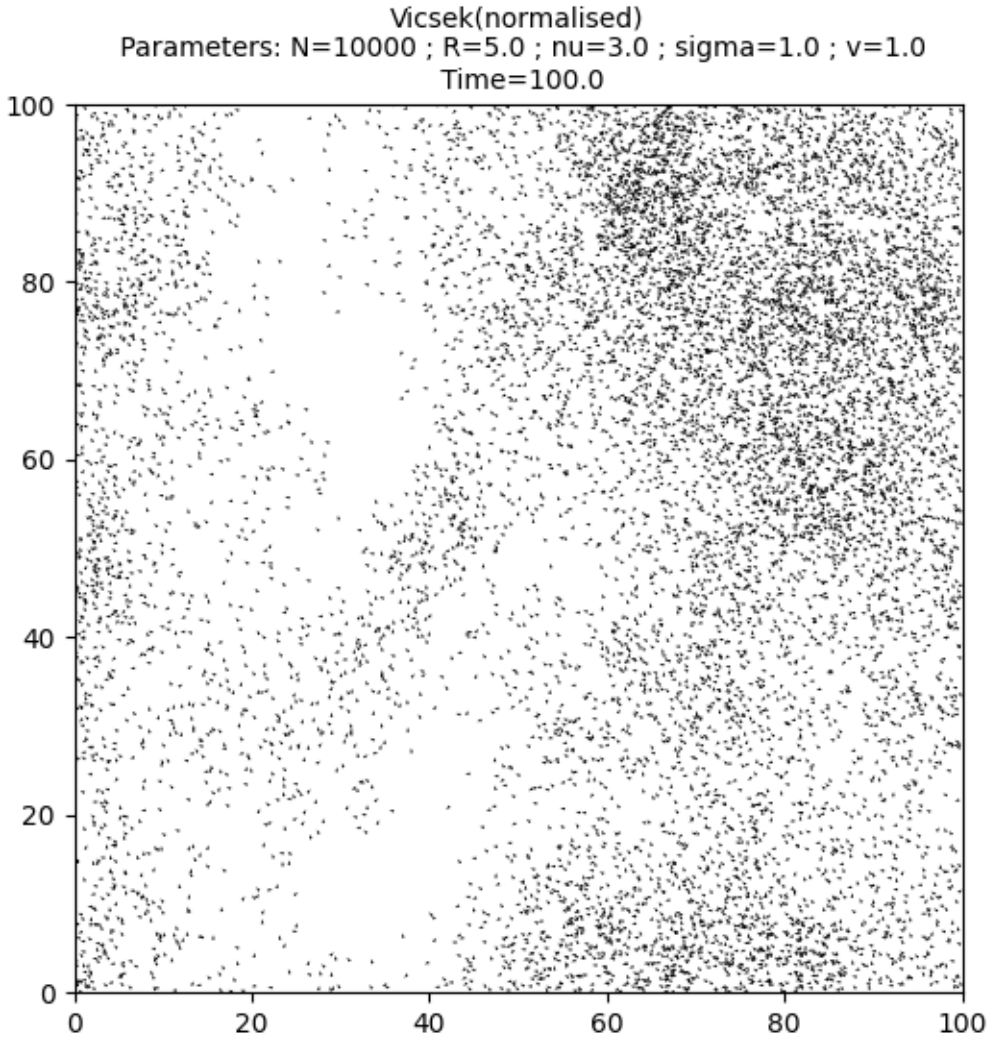
•

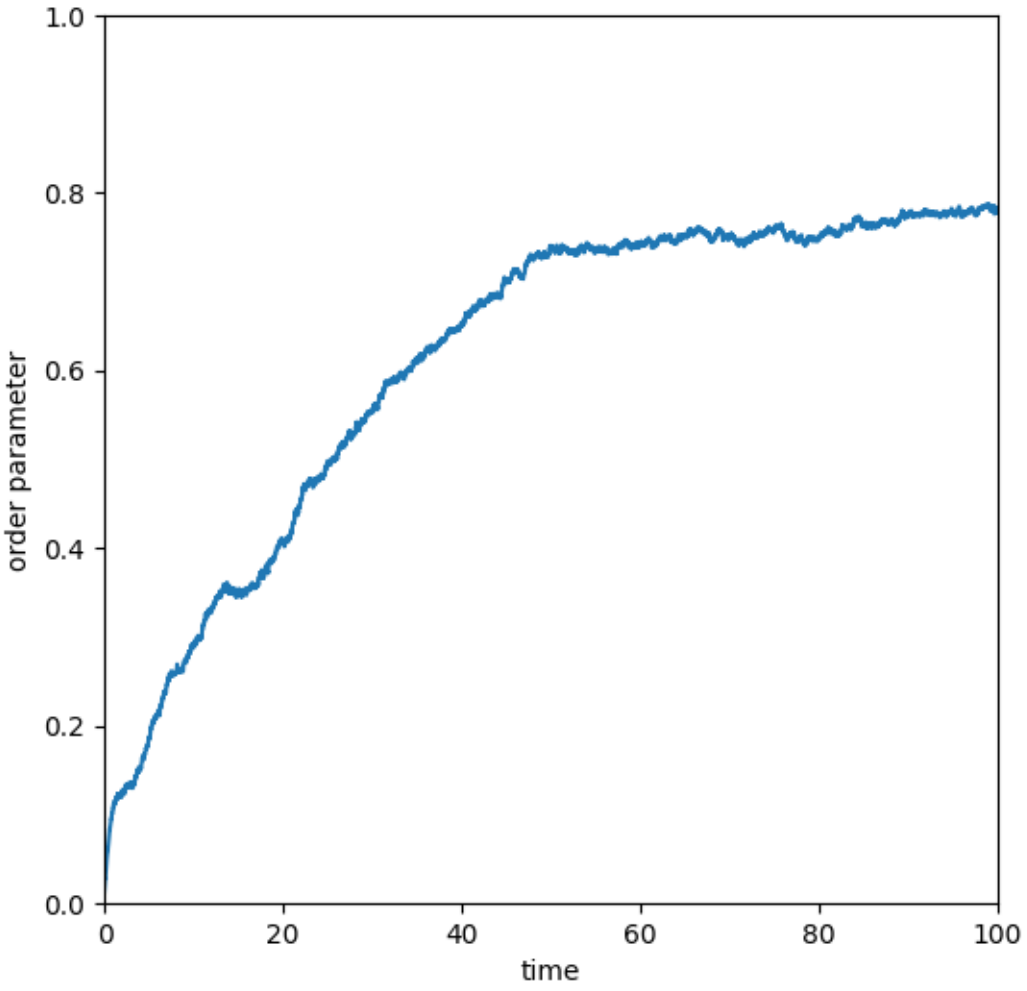

Vicsek(normalised)
Parameters: N=10000 ; R=5.0 ; nu=3.0 ; sigma=1.0 ; v=1.0
Time=50.0
100
80
60
40
20
0
0
20
40
60
80
100

-


Vicsek(normalised)
Parameters: N=10000 ; R=5.0 ; nu=3.0 ; sigma=1.0 ; v=1.0
Time=75.0
100
80
60
40
20
0
0
20
40
60
80
100

•


Vicsek(normalised)
Parameters: N=10000 ; R=5.0 ; nu=3.0 ; sigma=1.0 ; v=1.0
Time=100.0
100
80
60
40
20
0
0
20
40
60
80
100



-

Out:

```
Progress:0%
Progress:1%
Progress:2%
Progress:3%
Progress:4%
Progress:5%
Progress:6%
Progress:7%
Progress:8%
Progress:9%
Progress:10%
Progress:11%
Progress:12%
Progress:13%
Progress:14%
```

(continues on next page)

(continued from previous page)

```
Progress:15%
Progress:16%
Progress:17%
Progress:18%
Progress:19%
Progress:20%
Progress:21%
Progress:22%
Progress:23%
Progress:24%
Progress:25%
Progress:26%
Progress:27%
Progress:28%
Progress:29%
Progress:30%
Progress:31%
Progress:32%
Progress:33%
Progress:34%
Progress:35%
Progress:36%
Progress:37%
Progress:38%
Progress:39%
Progress:40%
Progress:41%
Progress:42%
Progress:43%
Progress:44%
Progress:45%
Progress:46%
Progress:47%
Progress:48%
Progress:49%
Progress:50%
Progress:51%
Progress:52%
Progress:53%
Progress:54%
Progress:55%
Progress:56%
Progress:57%
Progress:58%
Progress:59%
Progress:60%
Progress:61%
Progress:62%
Progress:63%
Progress:64%
Progress:65%
Progress:66%
```

(continues on next page)

(continued from previous page)

```
Progress:67%
Progress:68%
Progress:69%
Progress:70%
Progress:71%
Progress:72%
Progress:73%
Progress:74%
Progress:75%
Progress:76%
Progress:77%
Progress:78%
Progress:79%
Progress:80%
Progress:81%
Progress:82%
Progress:83%
Progress:84%
Progress:85%
Progress:86%
Progress:87%
Progress:88%
Progress:89%
Progress:90%
Progress:91%
Progress:92%
Progress:93%
Progress:94%
Progress:95%
Progress:96%
Progress:97%
Progress:98%
Progress:99%
```

Print the total simulation time and the average time per iteration.

```
print('Total time: '+str(e-s)+' seconds')
print('Average time per iteration: '+str((e-s)/simu.iteration)+' seconds')
```

Out:

```
Total time: 19.33805775642395 seconds
Average time per iteration: 0.0019336124144009549 seconds
```

Total running time of the script: (0 minutes 22.882 seconds)

4.4.2 Tutorial 02: Targets and options

Many useful **local averages** (see *Tutorial 05: Kernels and averages*) are pre-defined and may be directly used in the simulation of new or classical models. This tutorial showcases the basic usage of the **target methods** to simulate the variants of the Vicsek model.

An example: variants of the Vicsek model

In its most abstract form, the Vicsek model reads:

$$\mathrm{d}X_t^i = c_0 V_t^i \mathrm{d}t$$

$$\mathrm{d}V_t^i = \sigma \mathsf{P}(V_t^i) \circ (J_t^i \mathrm{d}t + \mathrm{d}B_t^i),$$

where c_0 is the speed, $\mathsf{P}(v)$ is the orthogonal projection on the orthogonal plane to v and σ is the diffusion coefficient. The drift coefficient J_t^i is called a **target**. In synchronous and asynchronous Vicsek models, the target is the center of the sampling distribution. A comparison of classical targets is shown below.

First, some standard imports…

```
import time
import torch
import pprint
from matplotlib import pyplot as plt
from sisyphe.models import Vicsek
from sisyphe.display import display_kinetic_particles

use_cuda = torch.cuda.is_available()
dtype = torch.cuda.FloatTensor if use_cuda else torch.FloatTensor
```

The classical non-normalised Vicsek model

The target is:

$$J_t^i = \kappa \frac{\sum_{j=1}^N K(|X_t^j - X_t^i|) V_t^j}{|\sum_{j=1}^N K(|X_t^j - X_t^i|) V_t^j|}.$$

The parameters of the model…

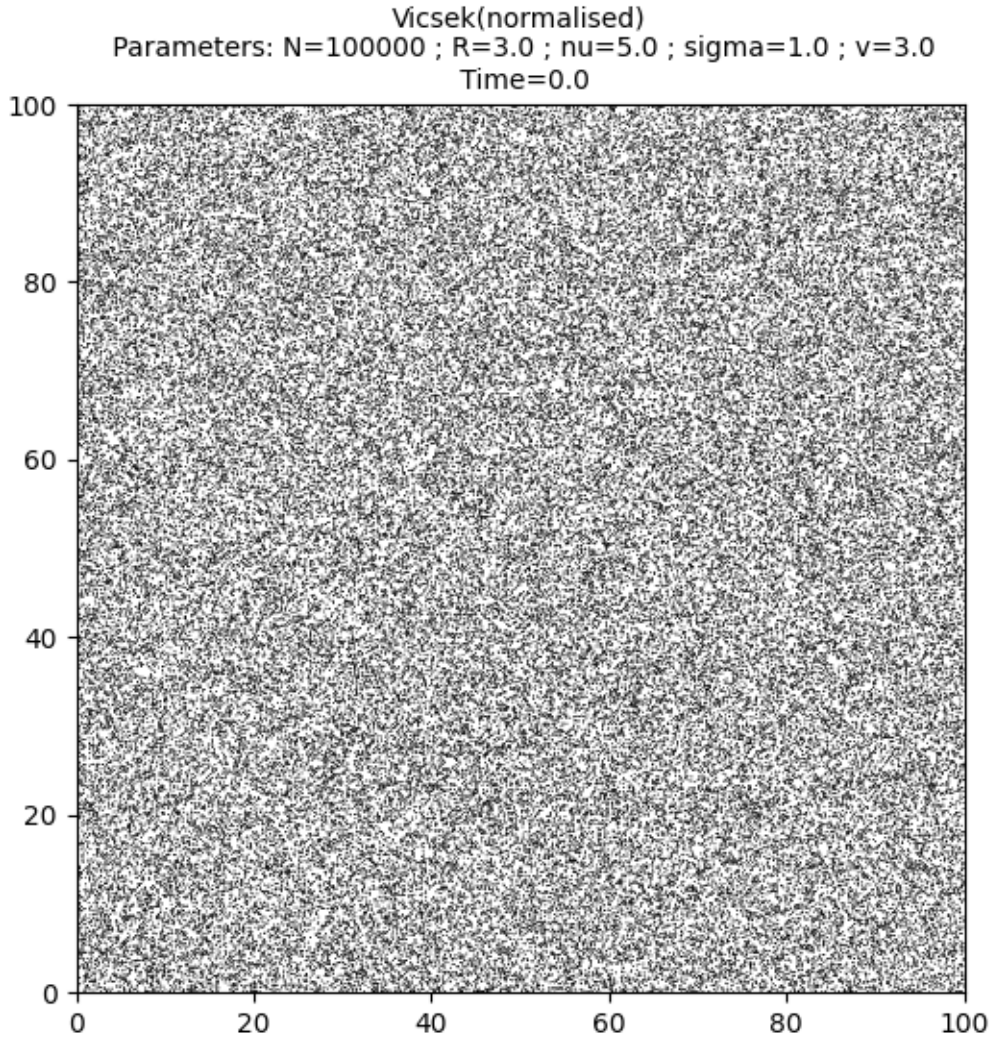
```
N = 100000
L = 100

pos = L*torch.rand((N,2)).type(dtype)
vel = torch.randn(N,2).type(dtype)
vel = vel/torch.norm(vel,dim=1).reshape((N,1))

R = 3.
c = 3.
nu = 5.
sigma = 1.

dt = .01
```

The choice of the target is implemented in the keyword argument `variant`. For the classical normalised target, it is given by the following dictionary.

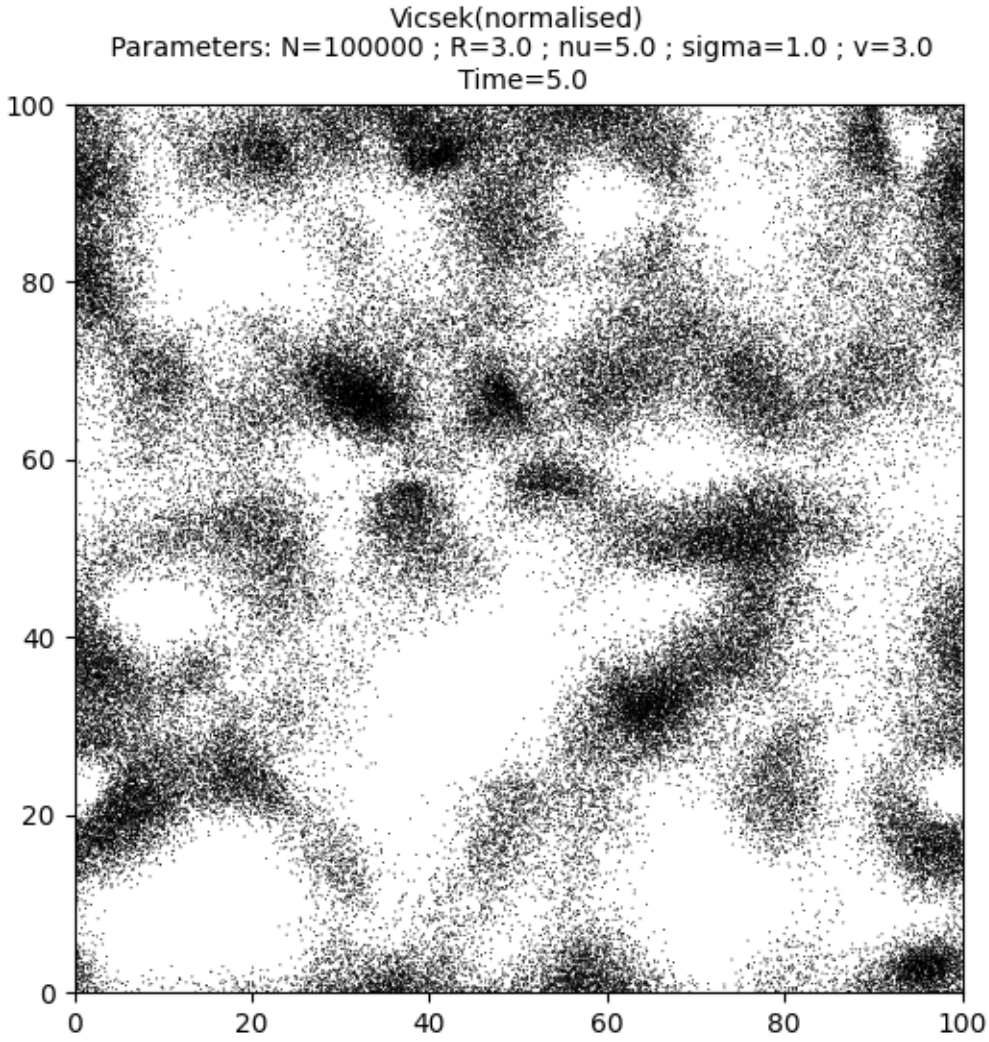
```
variant = {"name" : "normalised", "parameters" : {}}

simu = Vicsek(
    pos = pos.detach().clone(),
    vel = vel.detach().clone(),
    v = c,
    sigma = sigma,
    nu = nu,
    interaction_radius = R,
    box_size = L,
    dt = dt,
    variant = variant,
    block_sparse_reduction = True,
    number_of_cells = 40**2)
```

Finally run the simulation over 300 units of time....

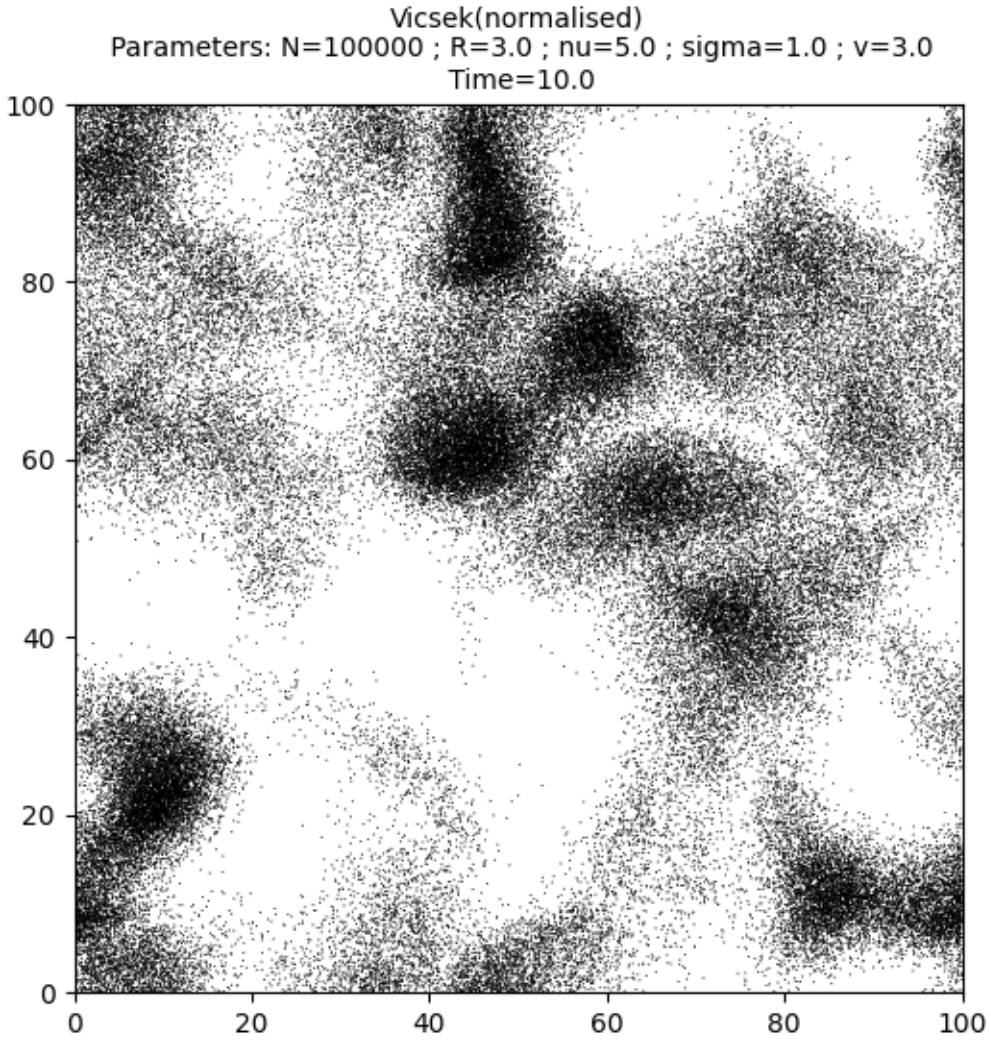
```
frames = [0., 5., 10., 30., 42., 71., 100, 124, 161, 206, 257, 300]

s = time.time()
it, op = display_kinetic_particles(simu, frames, order=True)
e = time.time()
```


Vicsek(normalised)
Parameters: N=100000 ; R=3.0 ; nu=5.0 ; sigma=1.0 ; v=3.0
Time=0.0
100
80
60
40
20
0
0
20
40
60
80
100

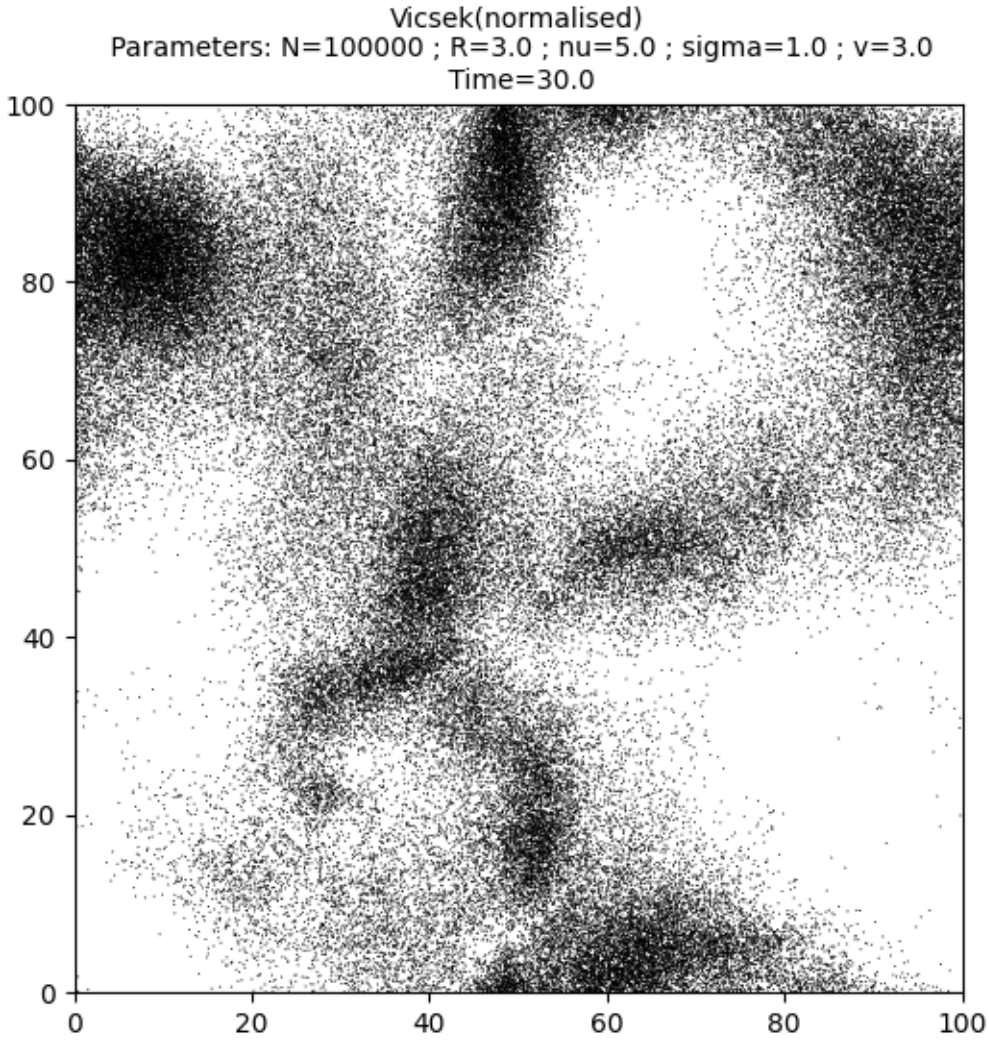
•


Vicsek(normalised)
Parameters: N=100000 ; R=3.0 ; nu=5.0 ; sigma=1.0 ; v=3.0
Time=5.0
100
80
60
40
20
0
0
20
40
60
80
100

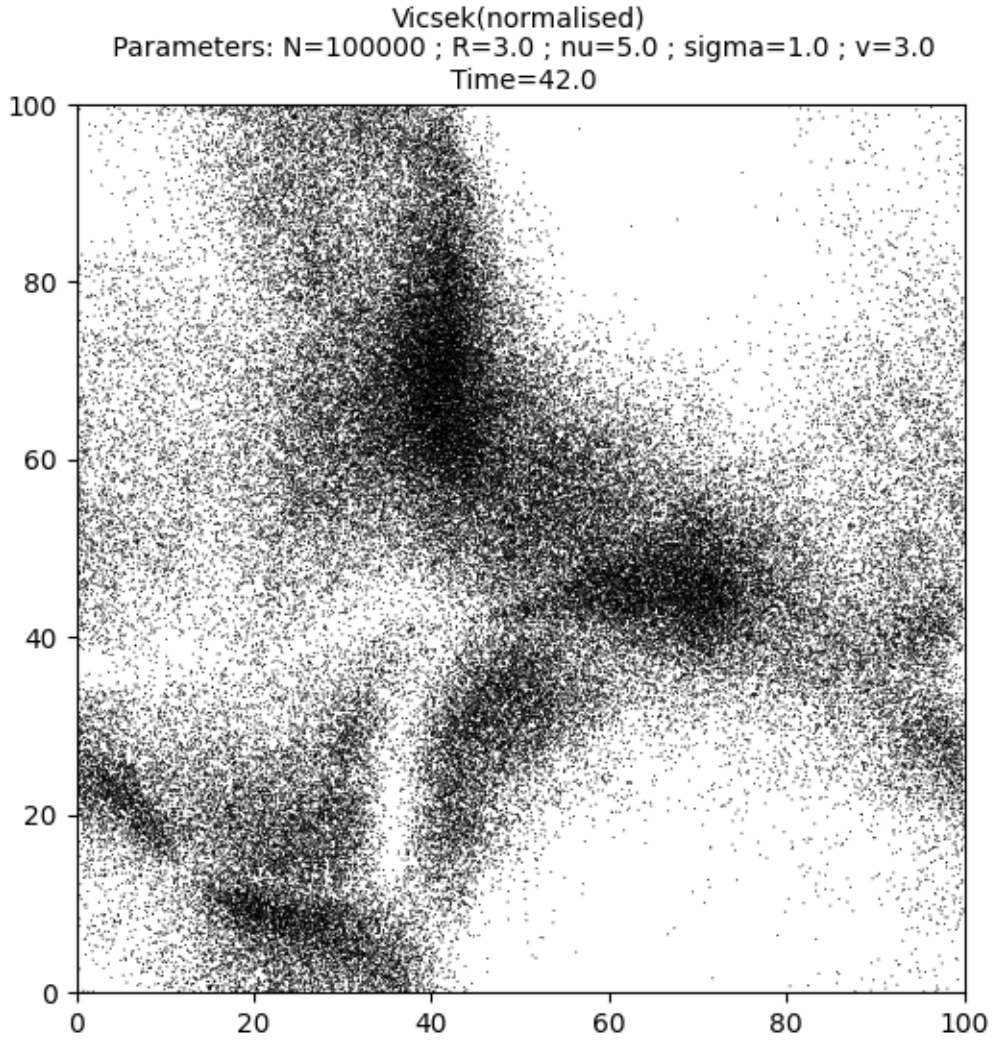
•

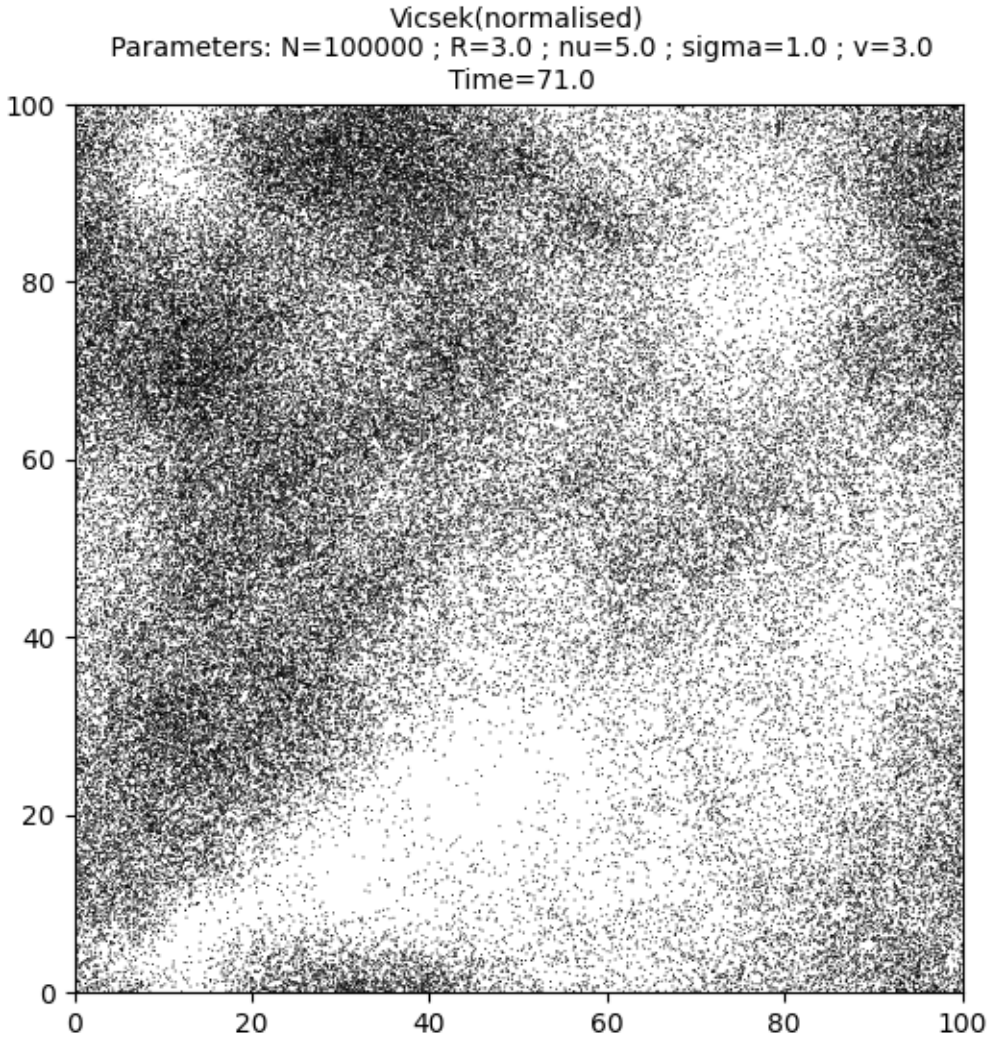

Vicsek(normalised)
Parameters: N=100000 ; R=3.0 ; nu=5.0 ; sigma=1.0 ; v=3.0
Time=10.0
100
80
60
40
20
0
0
20
40
60
80
100

•

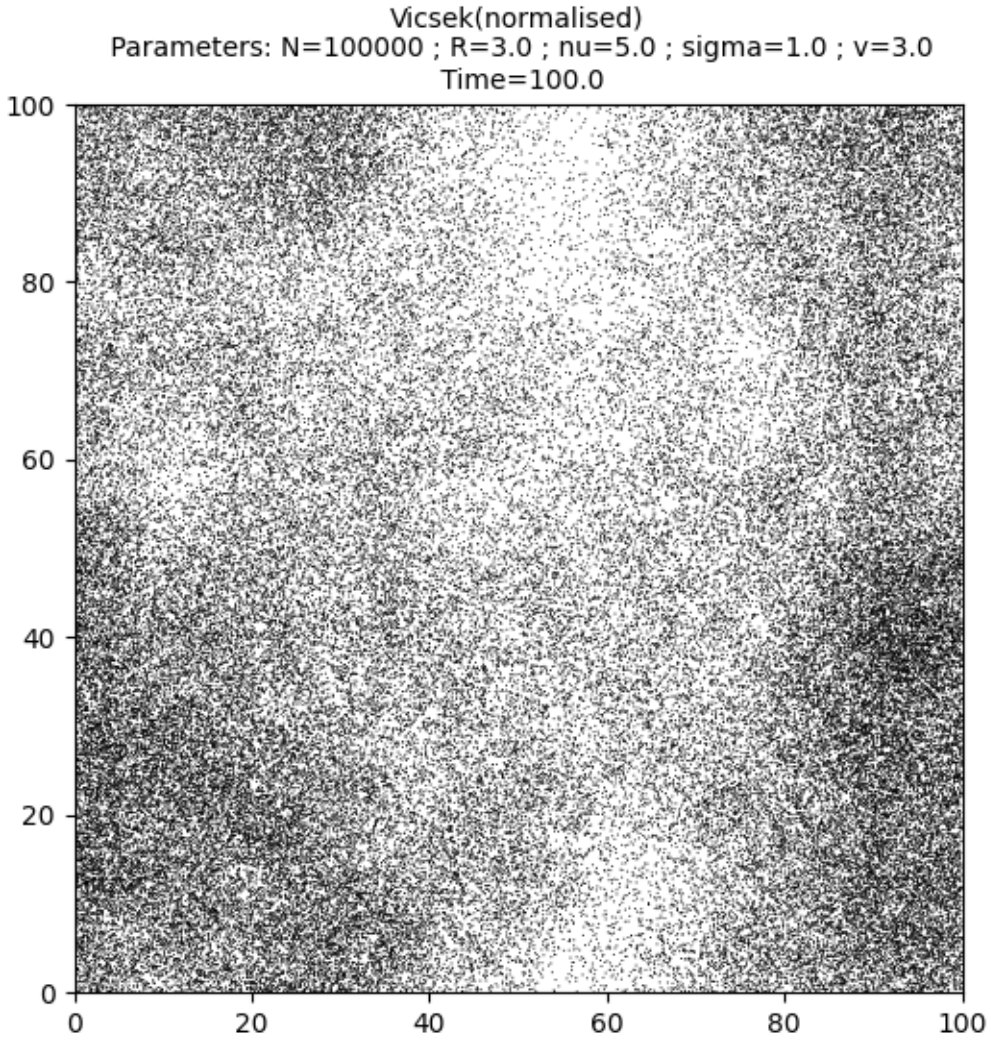

Vicsek(normalised)
Parameters: N=100000 ; R=3.0 ; nu=5.0 ; sigma=1.0 ; v=3.0
Time=30.0
100
80
60
40
20
0
0
20
40
60
80
100

•

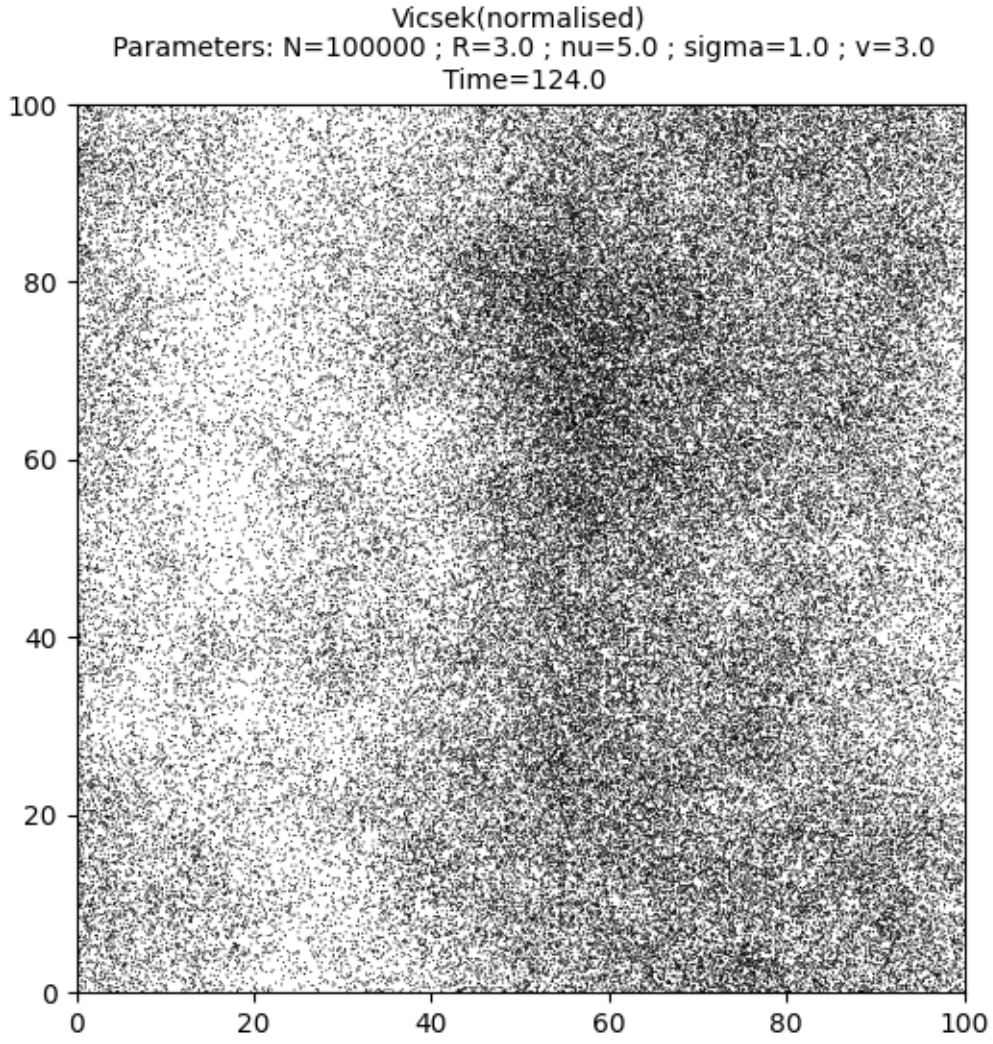

Vicsek(normalised)
Parameters: N=100000 ; R=3.0 ; nu=5.0 ; sigma=1.0 ; v=3.0
Time=42.0
100
80
60
40
20
0
0
20
40
60
80
100


Vicsek(normalised)
Parameters: N=100000 ; R=3.0 ; nu=5.0 ; sigma=1.0 ; v=3.0
Time=71.0
100
80
60
40
20
0
0
20
40
60
80
100

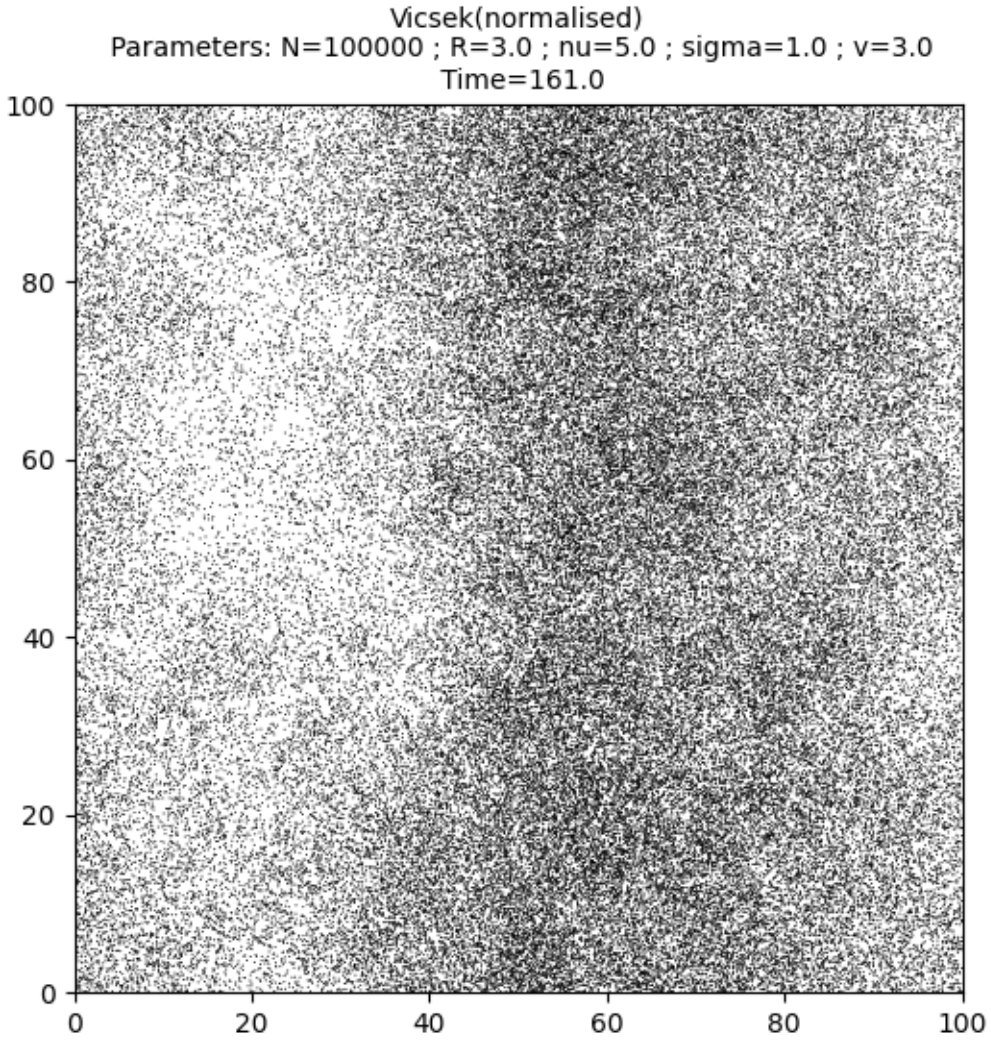
•


Vicsek(normalised)
Parameters: N=100000 ; R=3.0 ; nu=5.0 ; sigma=1.0 ; v=3.0
Time=100.0
100
80
60
40
20
0
0
20
40
60
80
100

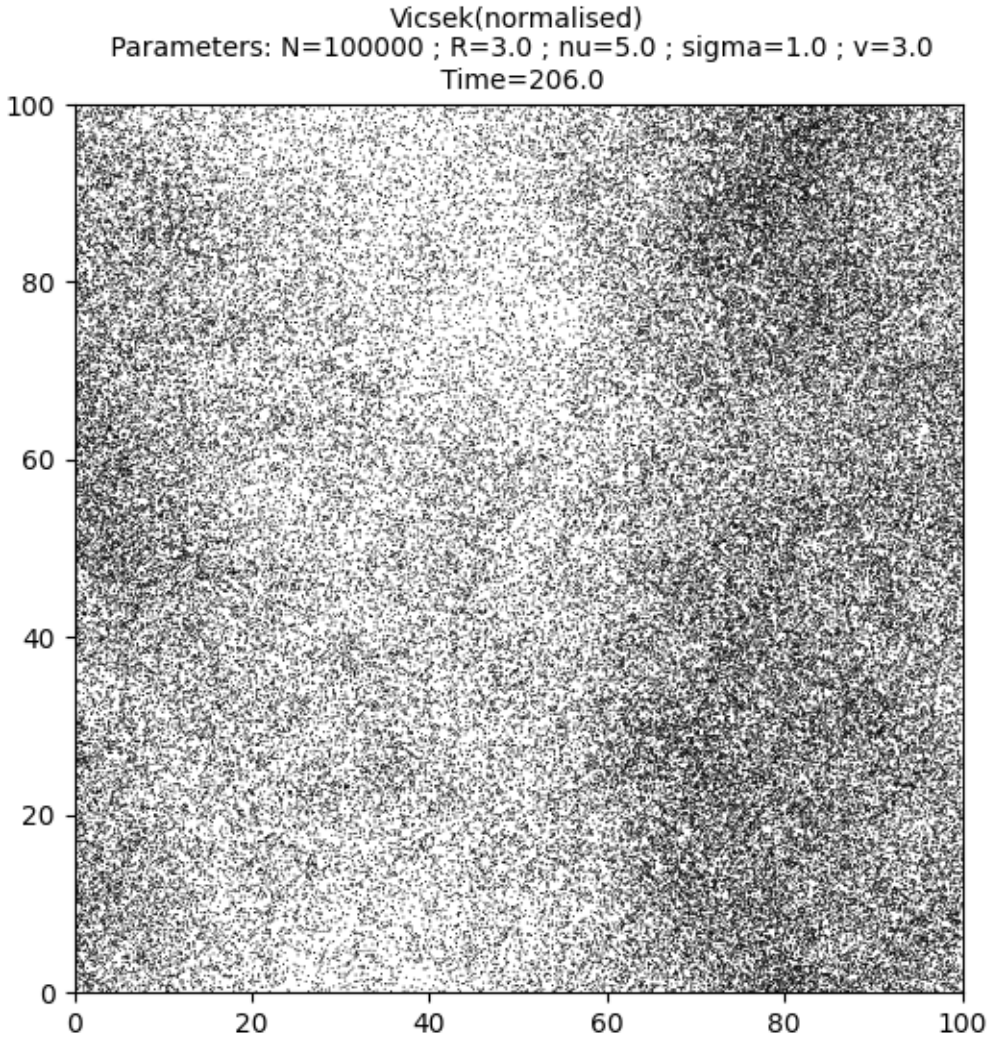
•


Vicsek(normalised)
Parameters: N=100000 ; R=3.0 ; nu=5.0 ; sigma=1.0 ; v=3.0
Time=124.0
100
80
60
40
20
0
0
20
40
60
80
100

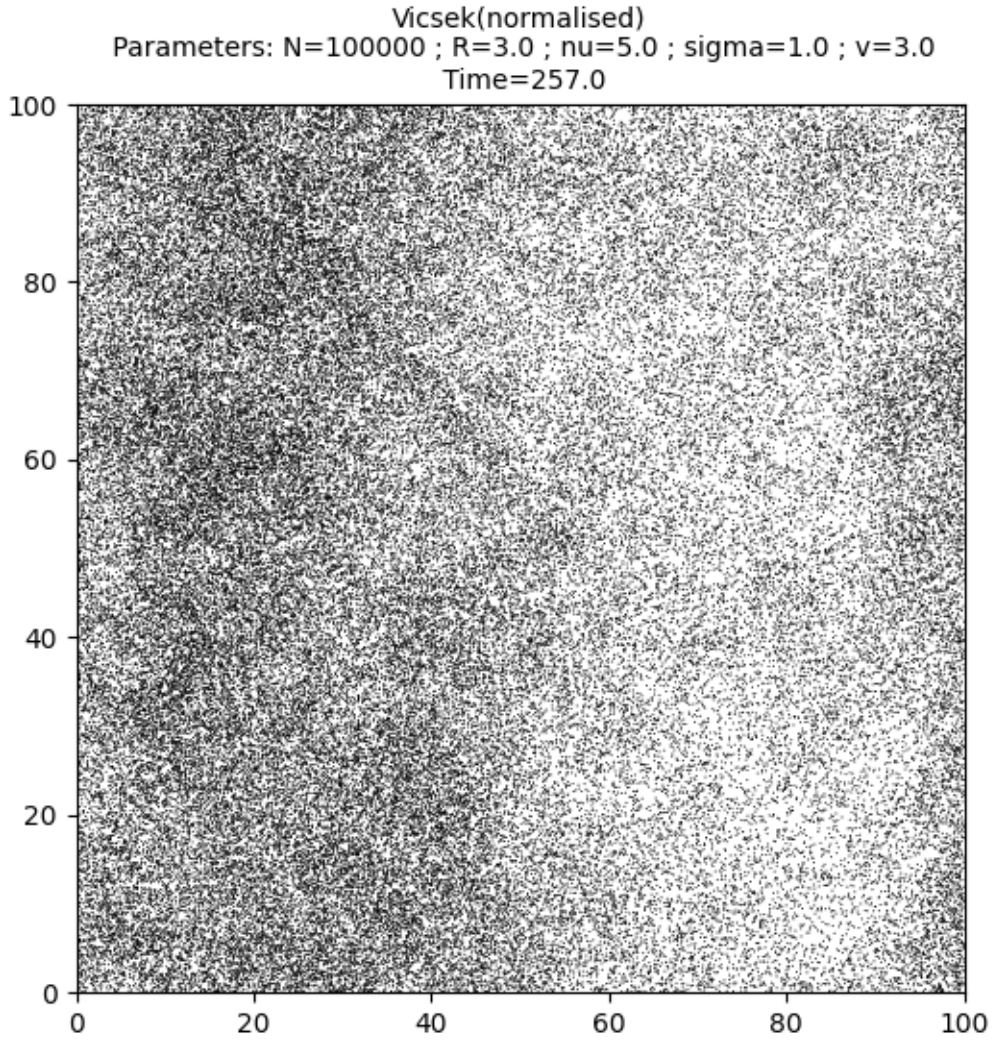
•


Vicsek(normalised)
Parameters: N=100000 ; R=3.0 ; nu=5.0 ; sigma=1.0 ; v=3.0
Time=161.0
100
80
60
40
20
0
0
20
40
60
80
100

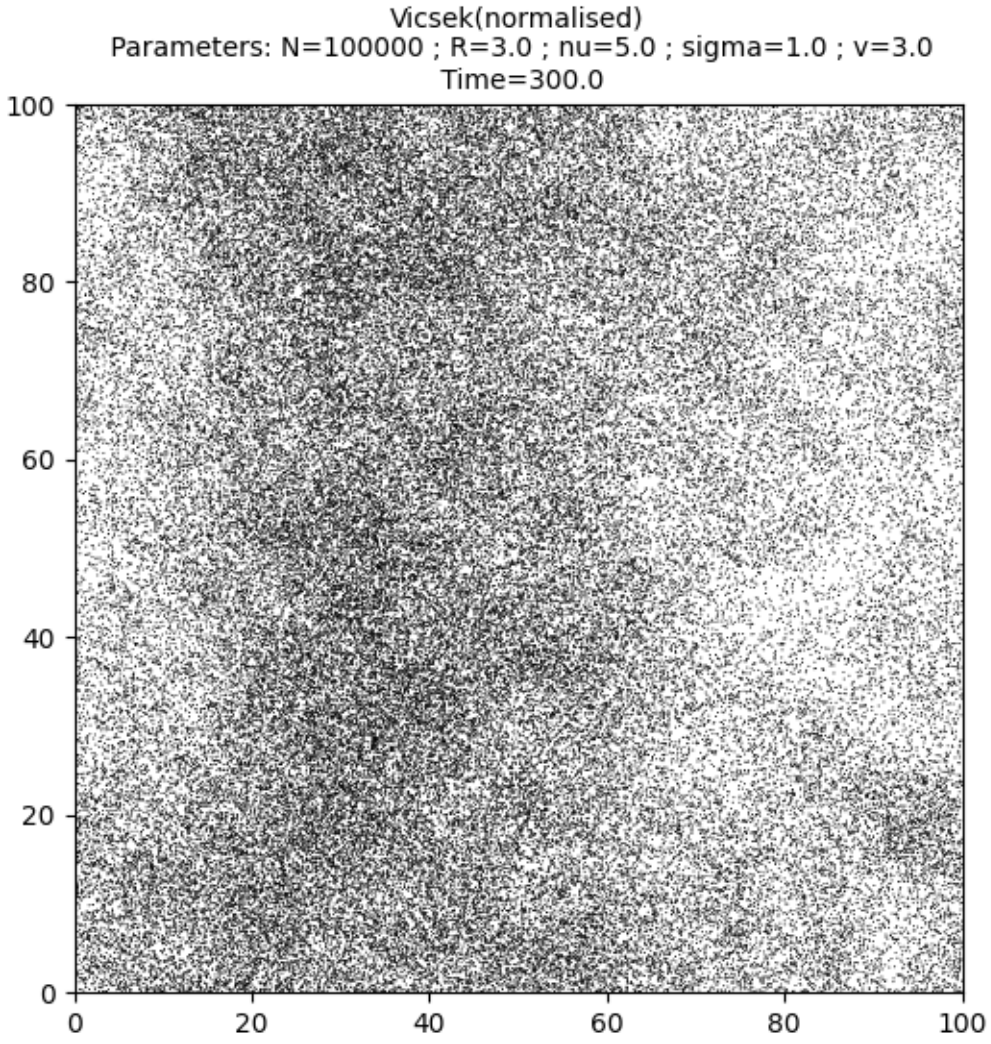
•


Vicsek(normalised)
Parameters: N=100000 ; R=3.0 ; nu=5.0 ; sigma=1.0 ; v=3.0
Time=206.0
100
80
60
40
20
0
0
20
40
60
80
100

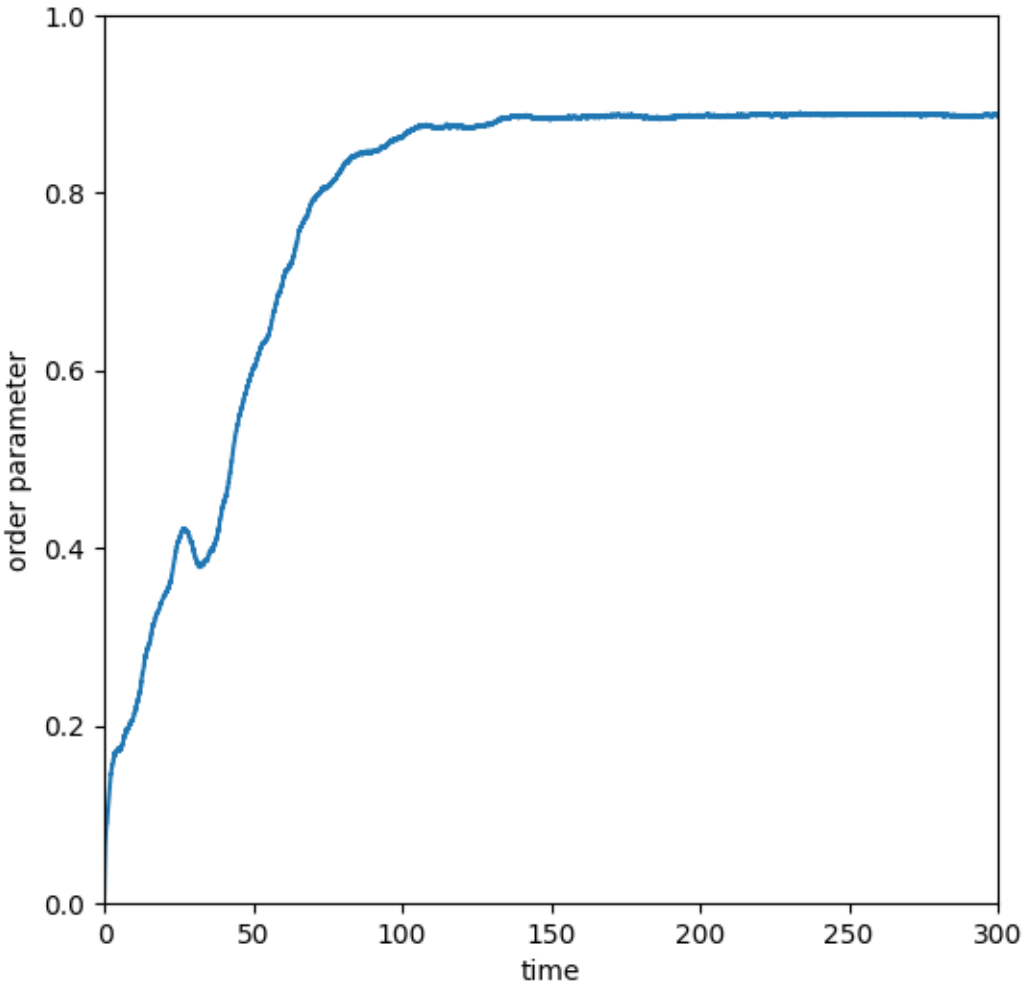
•


Vicsek(normalised)
Parameters: N=100000 ; R=3.0 ; nu=5.0 ; sigma=1.0 ; v=3.0
Time=257.0
100
80
60
40
20
0
0
20
40
60
80
100

•


Vicsek(normalised)
Parameters: N=100000 ; R=3.0 ; nu=5.0 ; sigma=1.0 ; v=3.0
Time=300.0
100
80
60
40
20
0
0
20
40
60
80
100

-



-

Out:

```
Progress:0%
Progress:1%
Progress:2%
Progress:3%
Progress:4%
Progress:5%
Progress:6%
Progress:7%
Progress:8%
Progress:9%
Progress:10%
Progress:11%
Progress:12%
Progress:13%
Progress:14%
```

(continues on next page)

(continued from previous page)

```
Progress:15%
Progress:16%
Progress:17%
Progress:18%
Progress:19%
Progress:20%
Progress:21%
Progress:22%
Progress:23%
Progress:24%
Progress:25%
Progress:26%
Progress:27%
Progress:28%
Progress:29%
Progress:30%
Progress:31%
Progress:32%
Progress:33%
Progress:34%
Progress:35%
Progress:36%
Progress:37%
Progress:38%
Progress:39%
Progress:40%
Progress:41%
Progress:42%
Progress:43%
Progress:44%
Progress:45%
Progress:46%
Progress:47%
Progress:48%
Progress:49%
Progress:50%
Progress:51%
Progress:52%
Progress:53%
Progress:54%
Progress:55%
Progress:56%
Progress:57%
Progress:58%
Progress:59%
Progress:60%
Progress:61%
Progress:62%
Progress:63%
Progress:64%
Progress:65%
Progress:66%
```

(continues on next page)

(continued from previous page)

```
Progress:67%
Progress:68%
Progress:69%
Progress:70%
Progress:71%
Progress:72%
Progress:73%
Progress:74%
Progress:75%
Progress:76%
Progress:77%
Progress:78%
Progress:79%
Progress:80%
Progress:81%
Progress:82%
Progress:83%
Progress:84%
Progress:85%
Progress:86%
Progress:87%
Progress:88%
Progress:89%
Progress:90%
Progress:91%
Progress:92%
Progress:93%
Progress:94%
Progress:95%
Progress:96%
Progress:97%
Progress:98%
Progress:99%
```

Print the total simulation time and the average time per iteration.

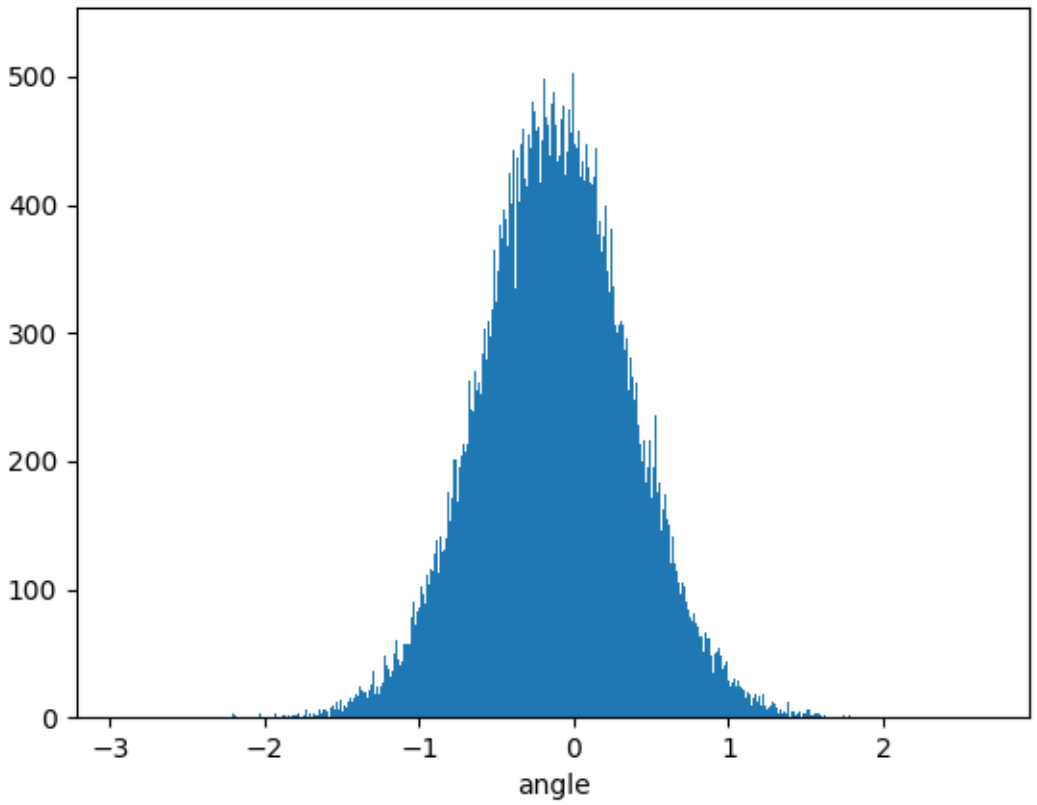
```
print('Total time: '+str(e-s)+' seconds')
print('Average time per iteration: '+str((e-s)/simu.iteration)+' seconds')
```

Out:

```
Total time: 145.67891550064087 seconds
Average time per iteration: 0.004855963850021362 seconds
```

Plot the histogram of the angles of the directions of motion.

```
angle = torch.atan2(simu.vel[:,1],simu.vel[:,0])
angle = angle.cpu().numpy()
h = plt.hist(angle, bins=1000)
plt.xlabel("angle")
plt.show()
```



After an initial clustering phase, the system self-organizes into a uniform flock.

Non-normalised Vicsek model

The target is:

$$J_t^i = \frac{\frac{1}{N}\sum_{j=1}^N K(|X_t^j - X_t^i|)V_t^j}{\frac{1}{\kappa} + \frac{1}{\kappa_0}|\frac{1}{N}\sum_{j=1}^N K(|X_t^j - X_t^i|)V_t^j|}.$$

Define the corresponding dictionary...

```
kappa_0 = 15.

variant = {"name" : "max_kappa", "parameters" : {"kappa_max" : kappa_0}}

simu = Vicsek(
    pos = pos.detach().clone(),
    vel = vel.detach().clone(),
    v = c,
    sigma = sigma,
    nu = nu,
    interaction_radius = R,
    box_size = L,
```

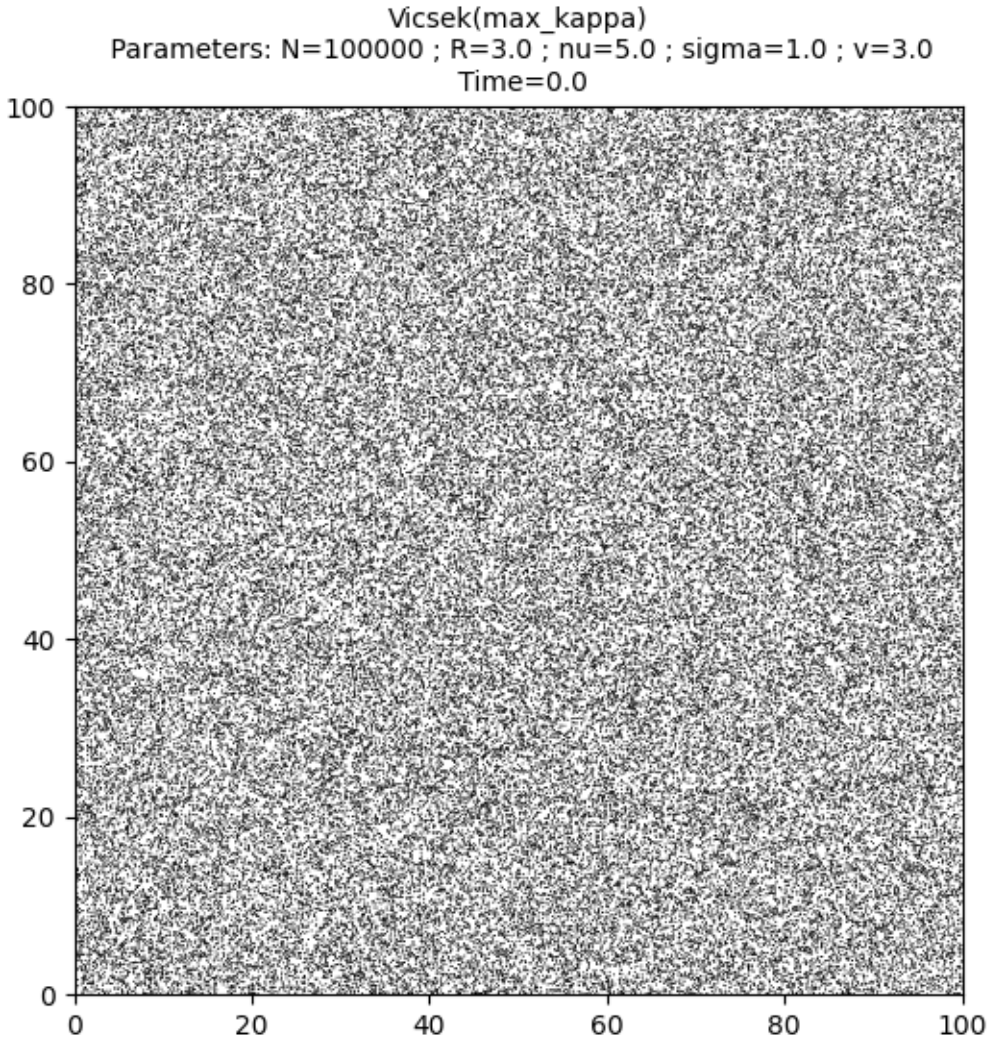
(continues on next page)

(continued from previous page)

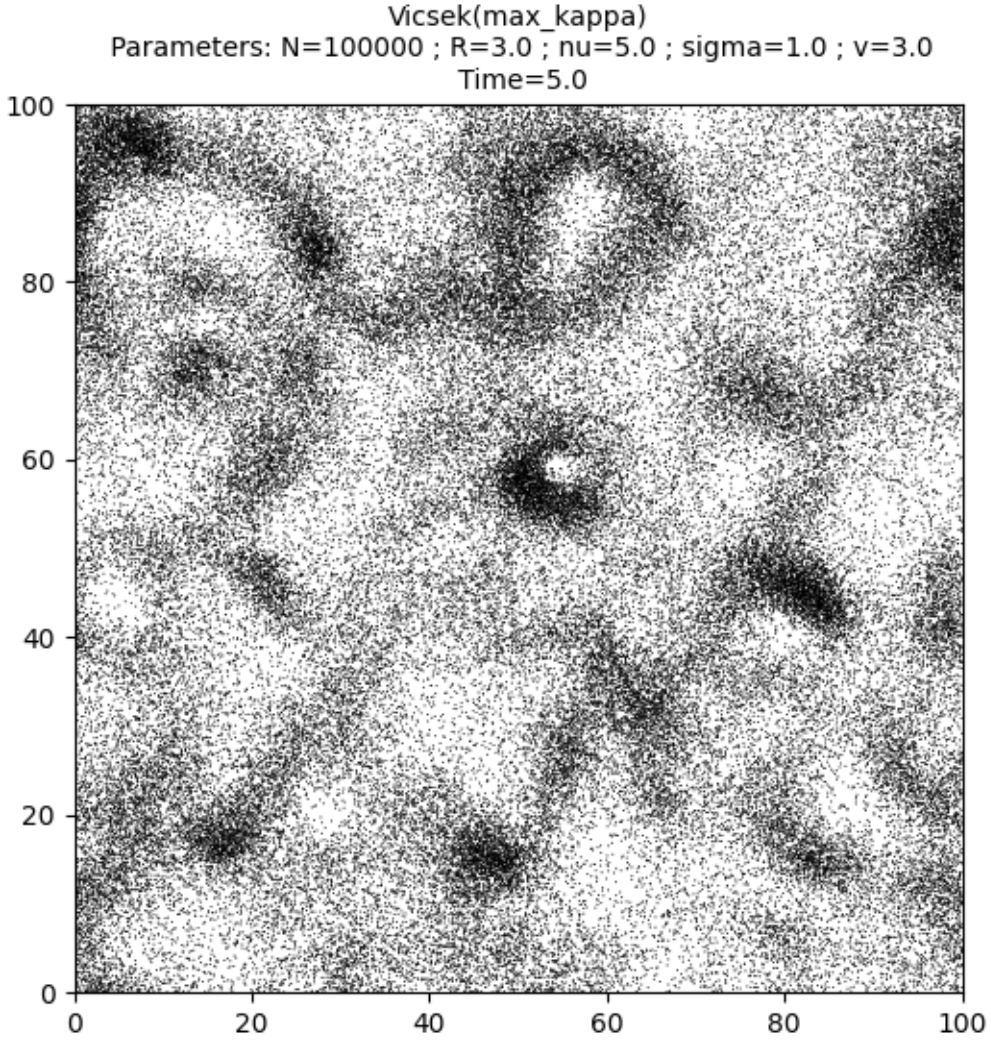
```
    dt = dt,
    variant = variant,
    block_sparse_reduction = True,
    number_of_cells = 40**2)
```

Finally run the simulation over 300 units of time....

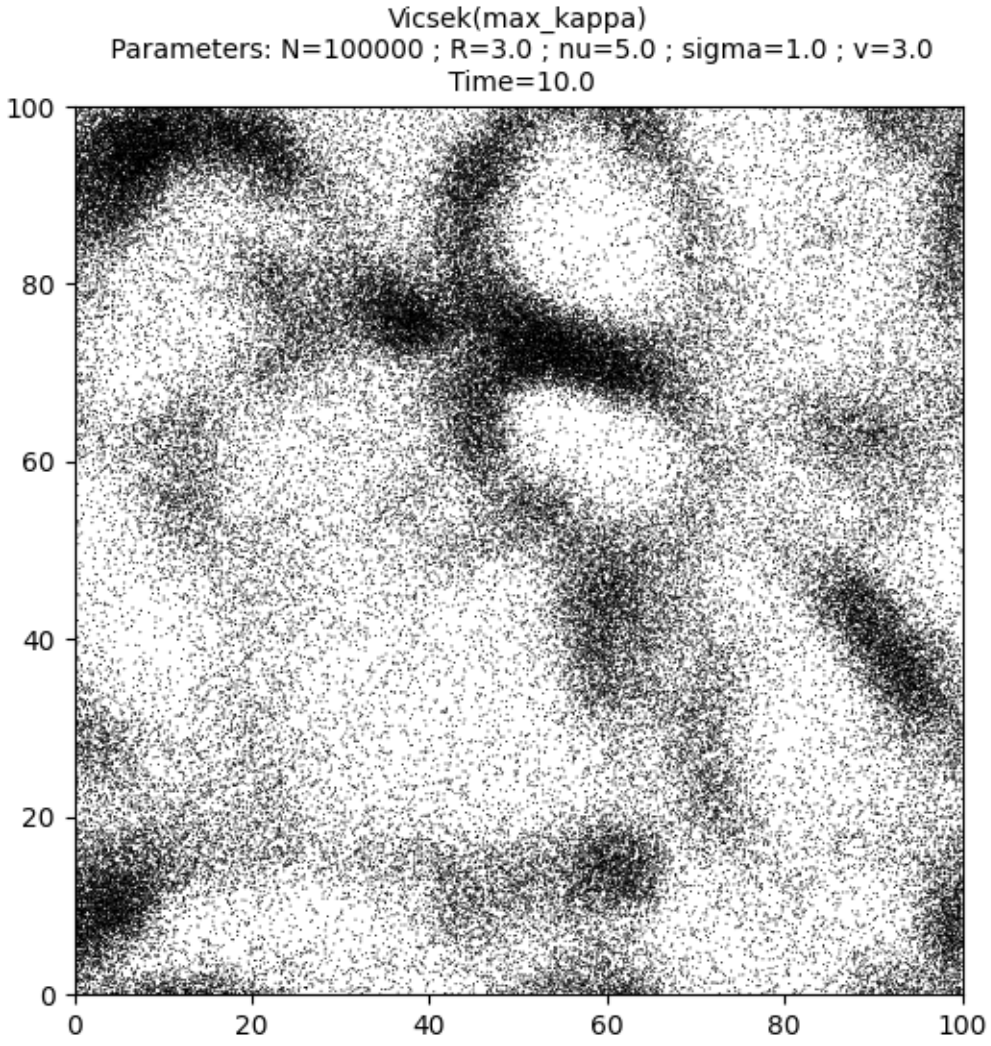
```
frames = [0., 5., 10., 30., 42., 71., 100, 124, 161, 206, 257, 300]

s = time.time()
it, op = display_kinetic_particles(simu, frames, order=True)
e = time.time()
```

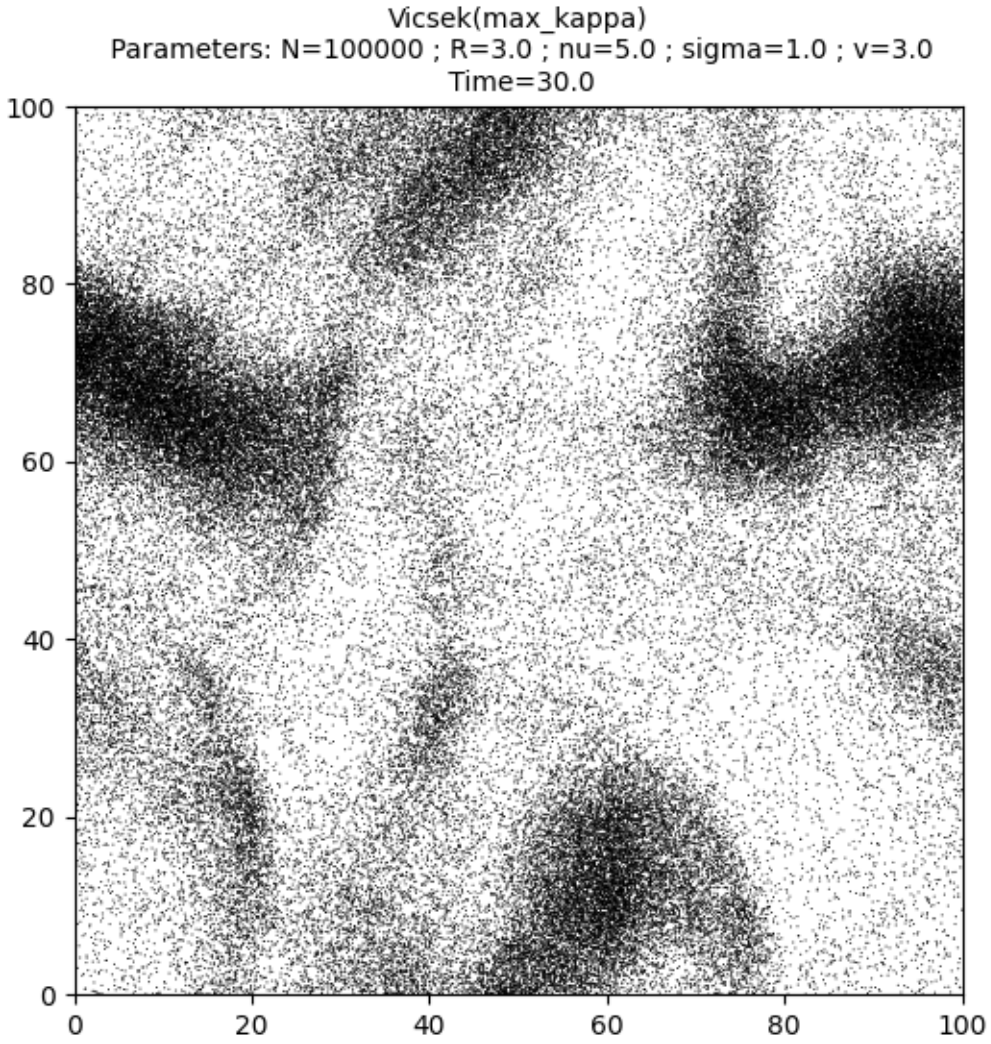


-

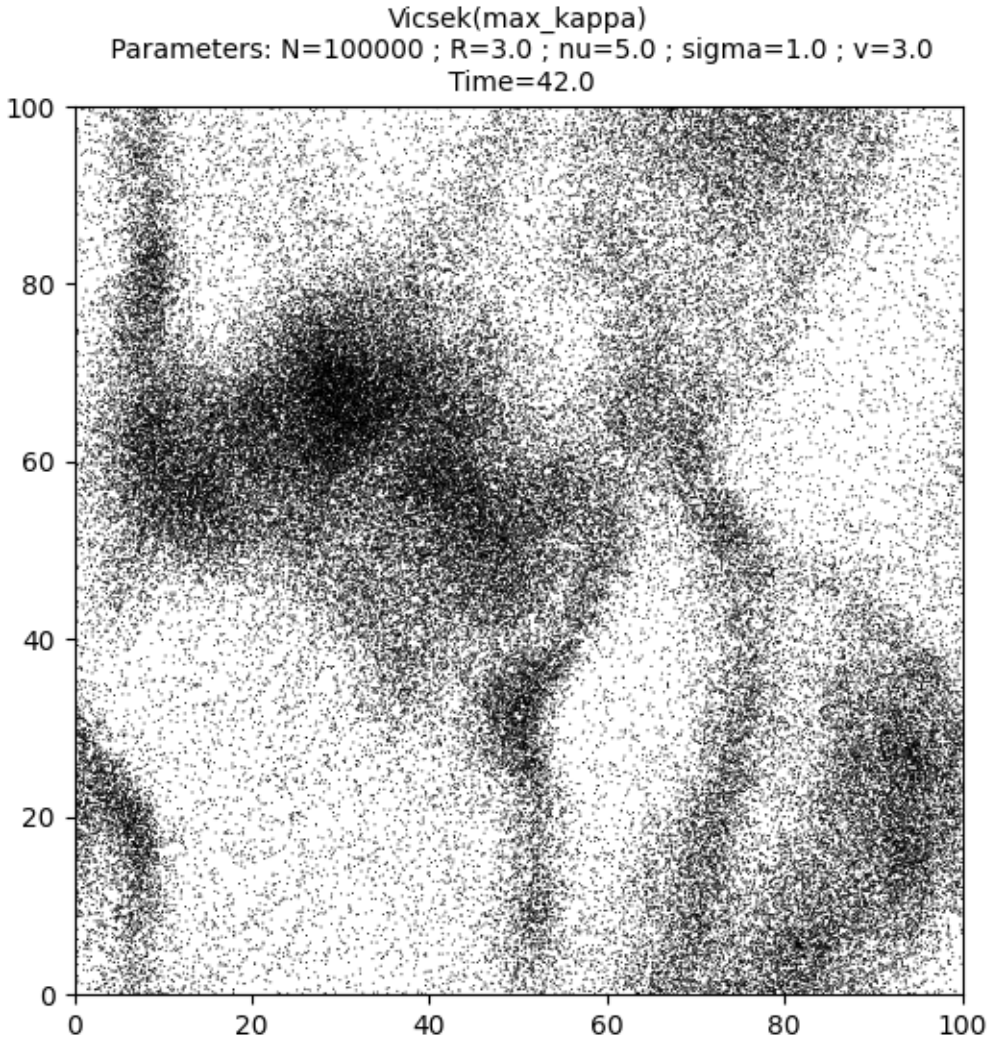

Vicsek(max_kappa)
Parameters: N=100000 ; R=3.0 ; nu=5.0 ; sigma=1.0 ; v=3.0
Time=5.0
100
80
60
40
20
0
0
20
40
60
80
100

•

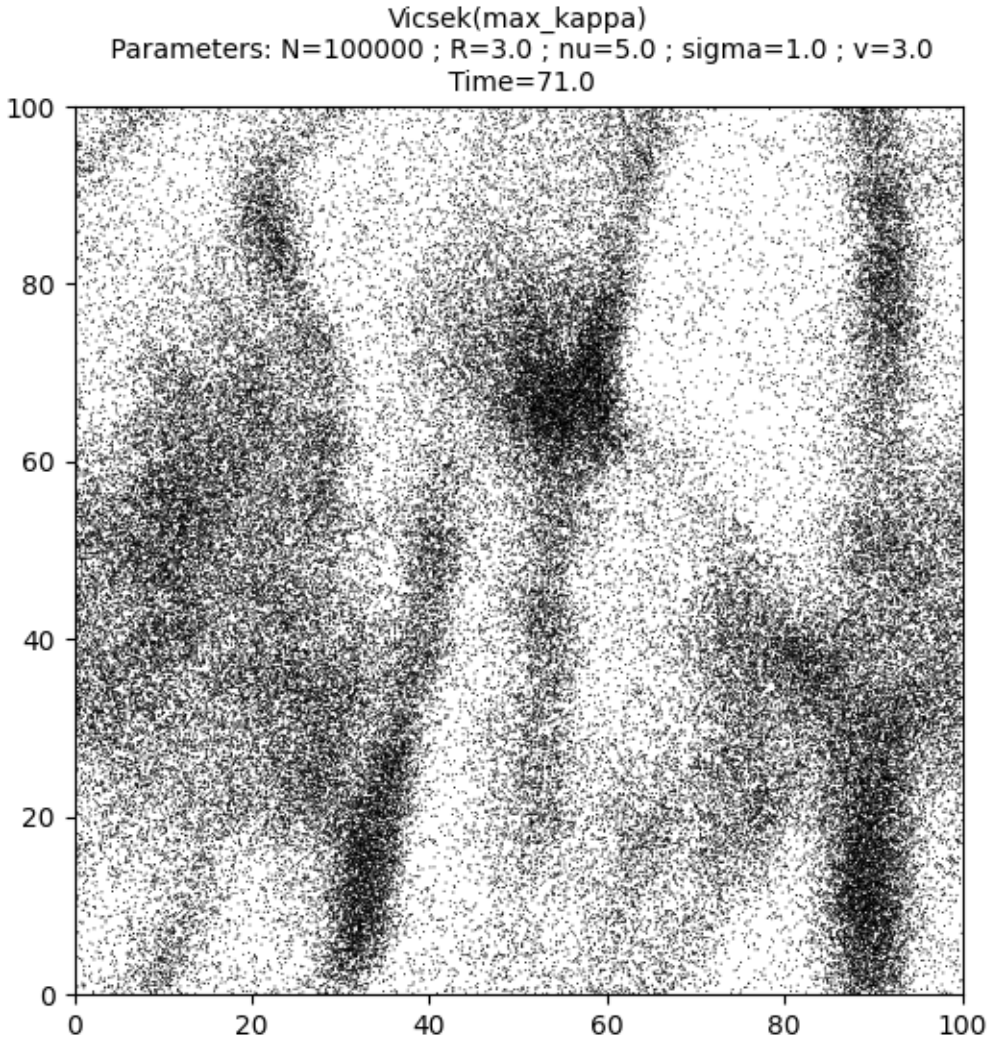

Vicsek(max_kappa)
Parameters: N=100000 ; R=3.0 ; nu=5.0 ; sigma=1.0 ; v=3.0
Time=10.0
100
80
60
40
20
0
0
20
40
60
80
100


Vicsek(max_kappa)
Parameters: N=100000 ; R=3.0 ; nu=5.0 ; sigma=1.0 ; v=3.0
Time=30.0
100
80
60
40
20
0
0
20
40
60
80
100

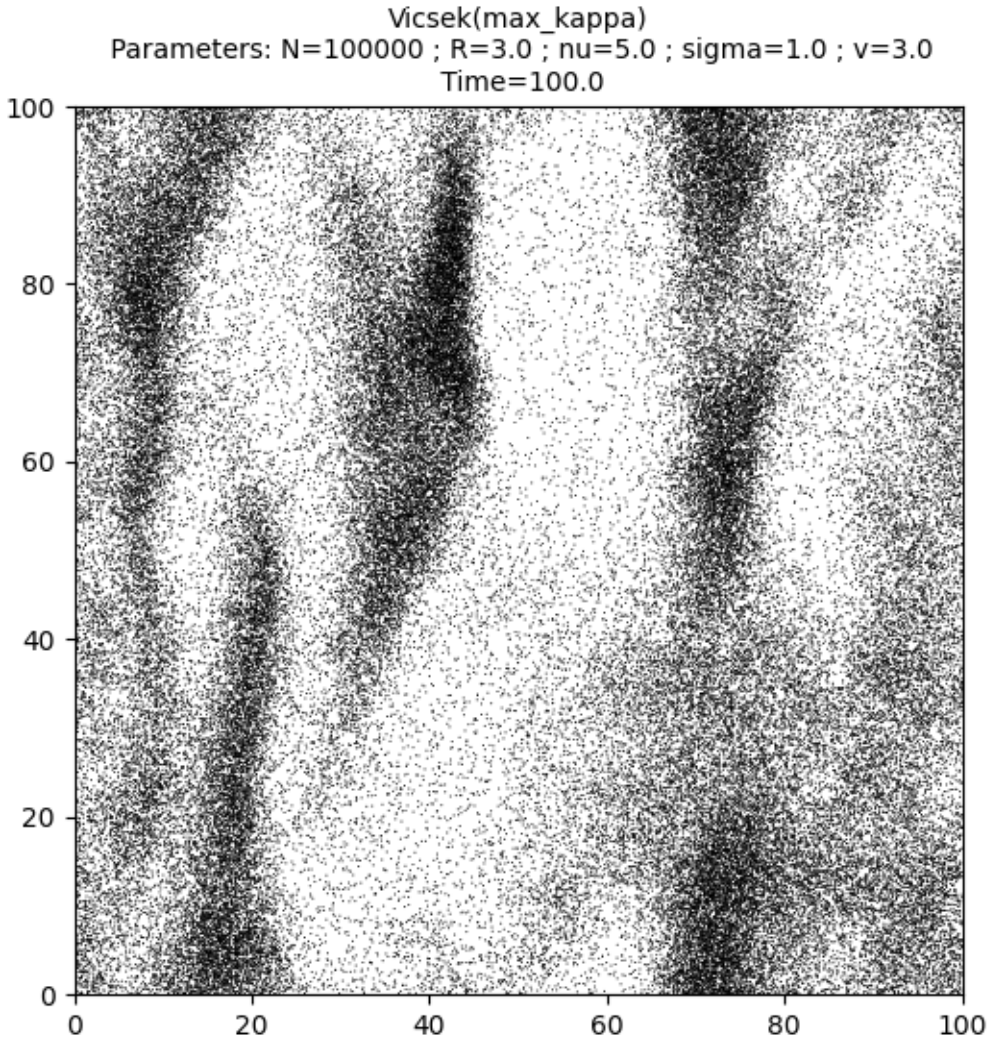
•

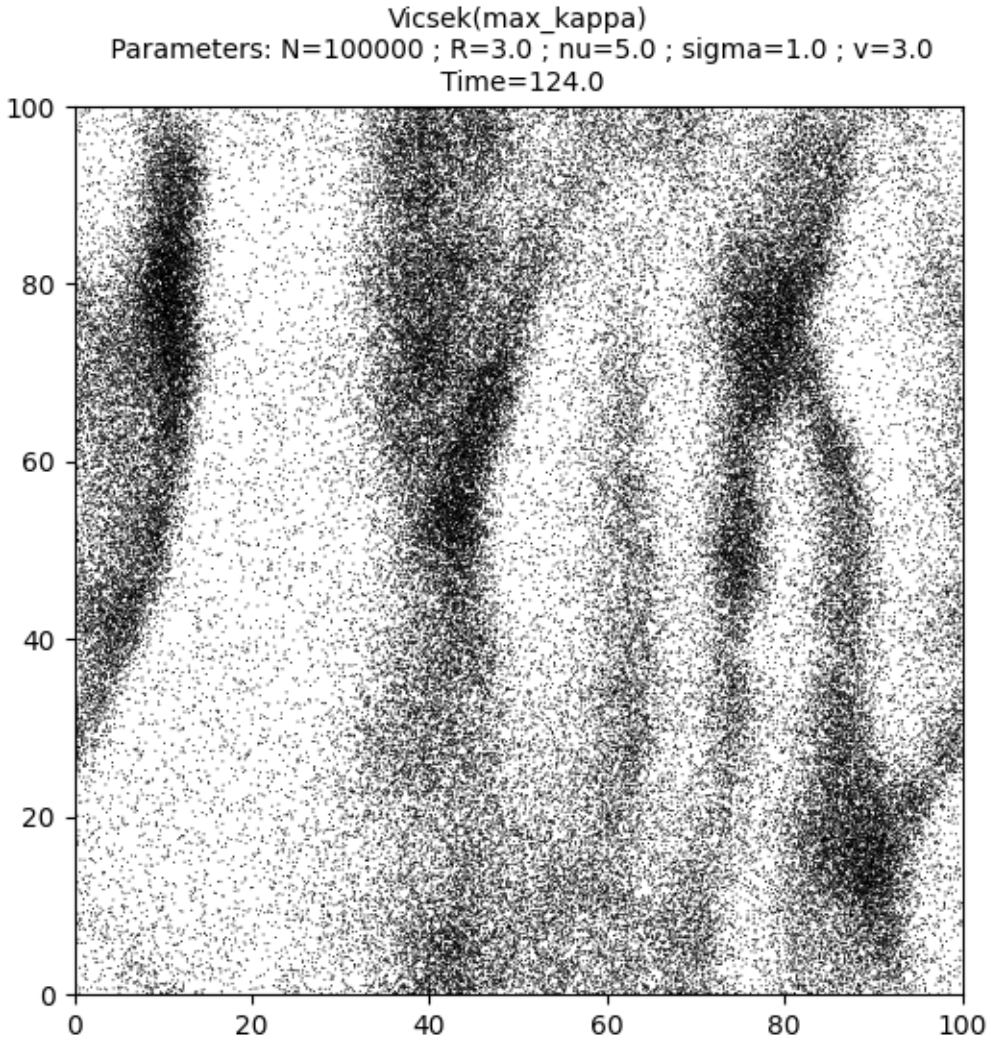

Vicsek(max_kappa)
Parameters: N=100000 ; R=3.0 ; nu=5.0 ; sigma=1.0 ; v=3.0
Time=42.0
100
80
60
40
20
0
0
20
40
60
80
100

•

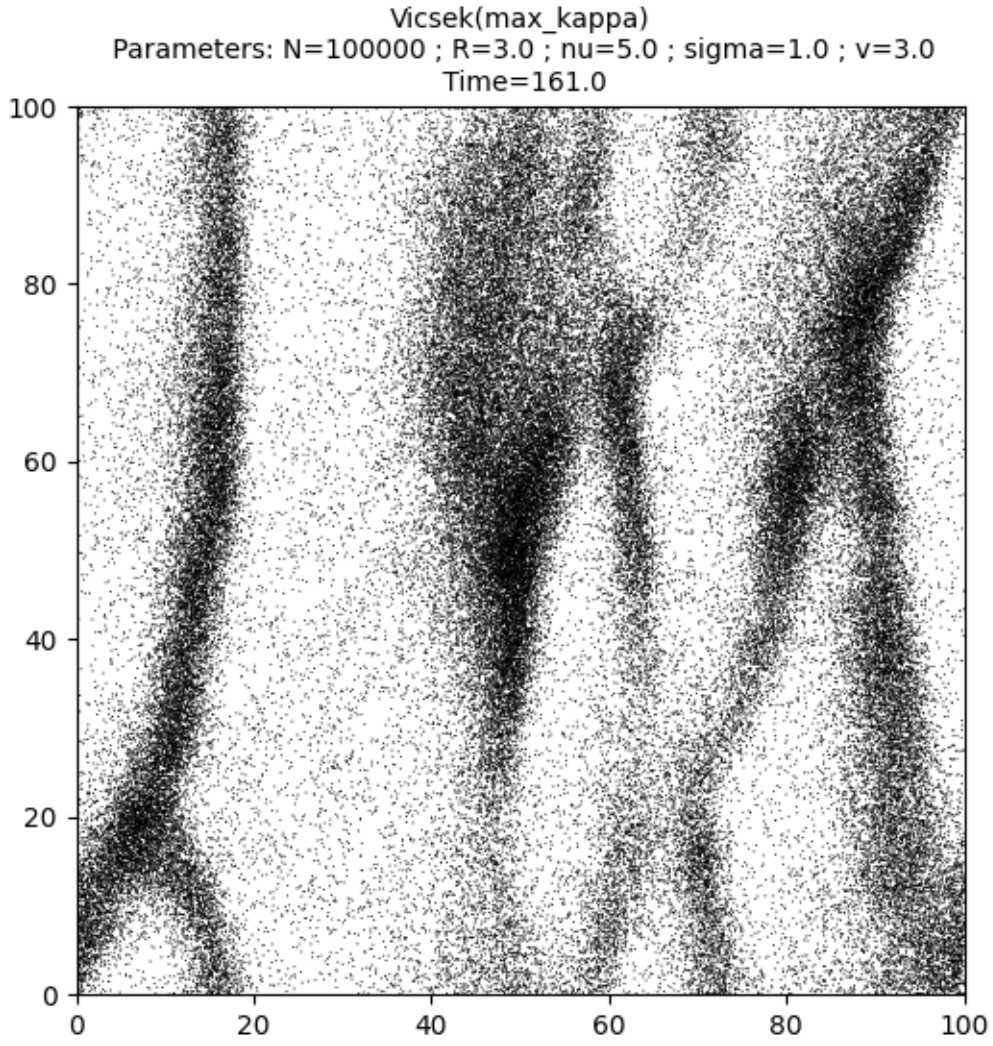

Vicsek(max_kappa)
Parameters: N=100000 ; R=3.0 ; nu=5.0 ; sigma=1.0 ; v=3.0
Time=71.0
100
80
60
40
20
0
0
20
40
60
80
100

•

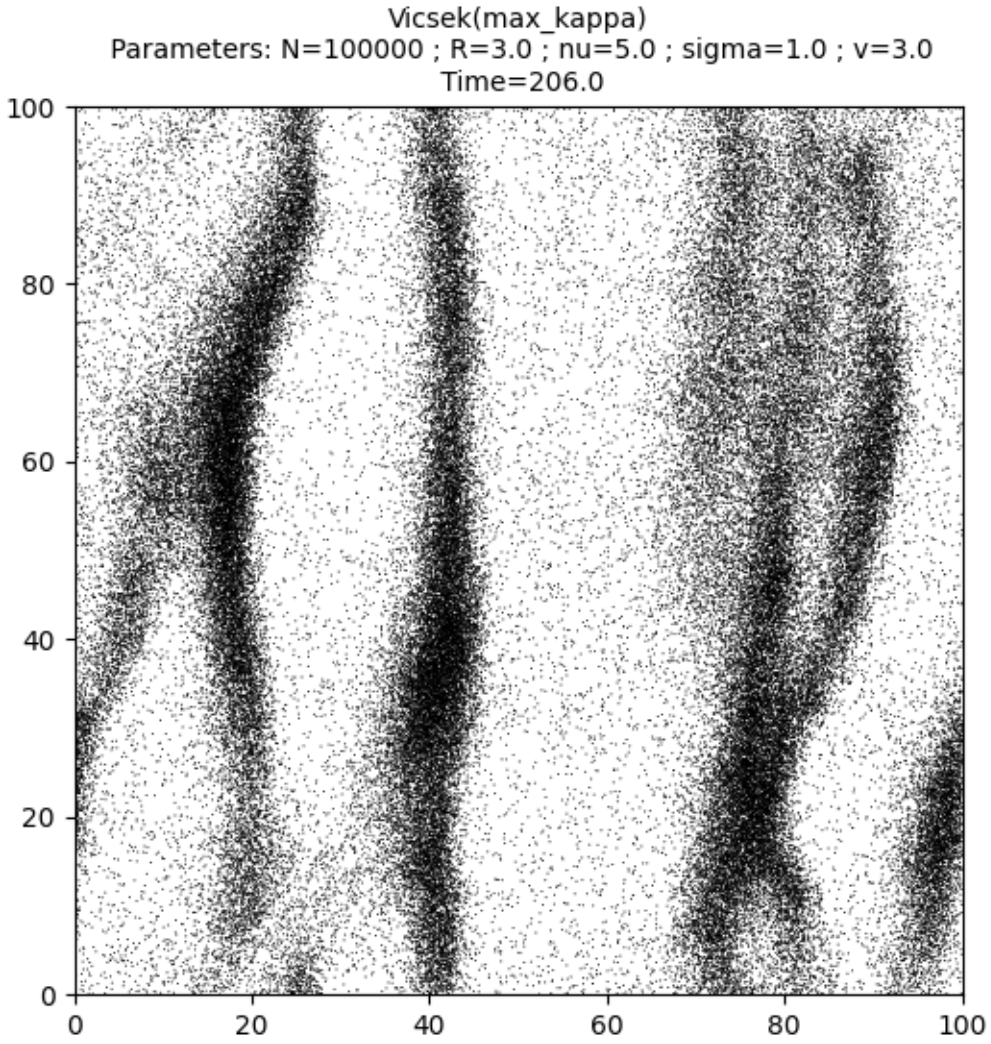

Vicsek(max_kappa)
Parameters: N=100000 ; R=3.0 ; nu=5.0 ; sigma=1.0 ; v=3.0
Time=100.0
100
80
60
40
20
0
0
20
40
60
80
100


Vicsek(max_kappa)
Parameters: N=100000 ; R=3.0 ; nu=5.0 ; sigma=1.0 ; v=3.0
Time=124.0
100
80
60
40
20
0
0
20
40
60
80
100

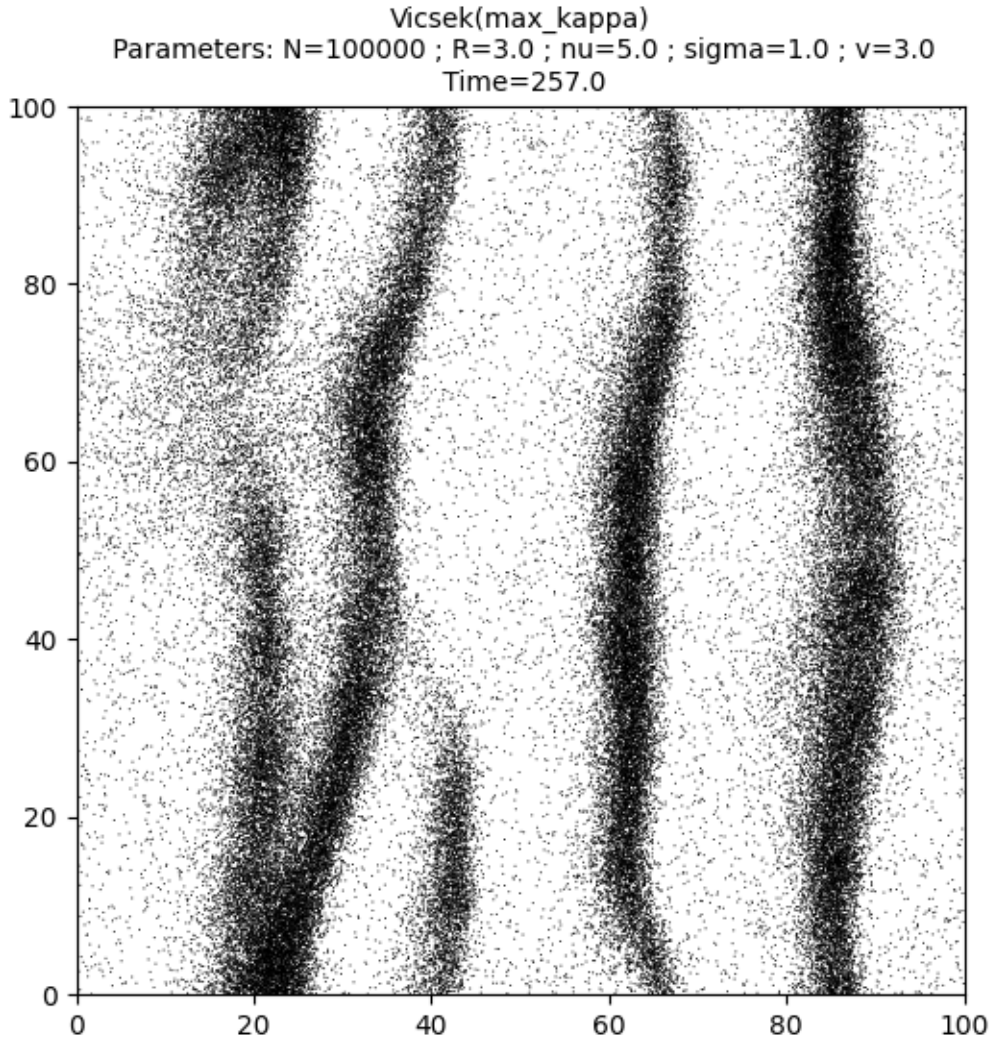
•


Vicsek(max_kappa)
Parameters: N=100000 ; R=3.0 ; nu=5.0 ; sigma=1.0 ; v=3.0
Time=161.0
100
80
60
40
20
0
0
20
40
60
80
100

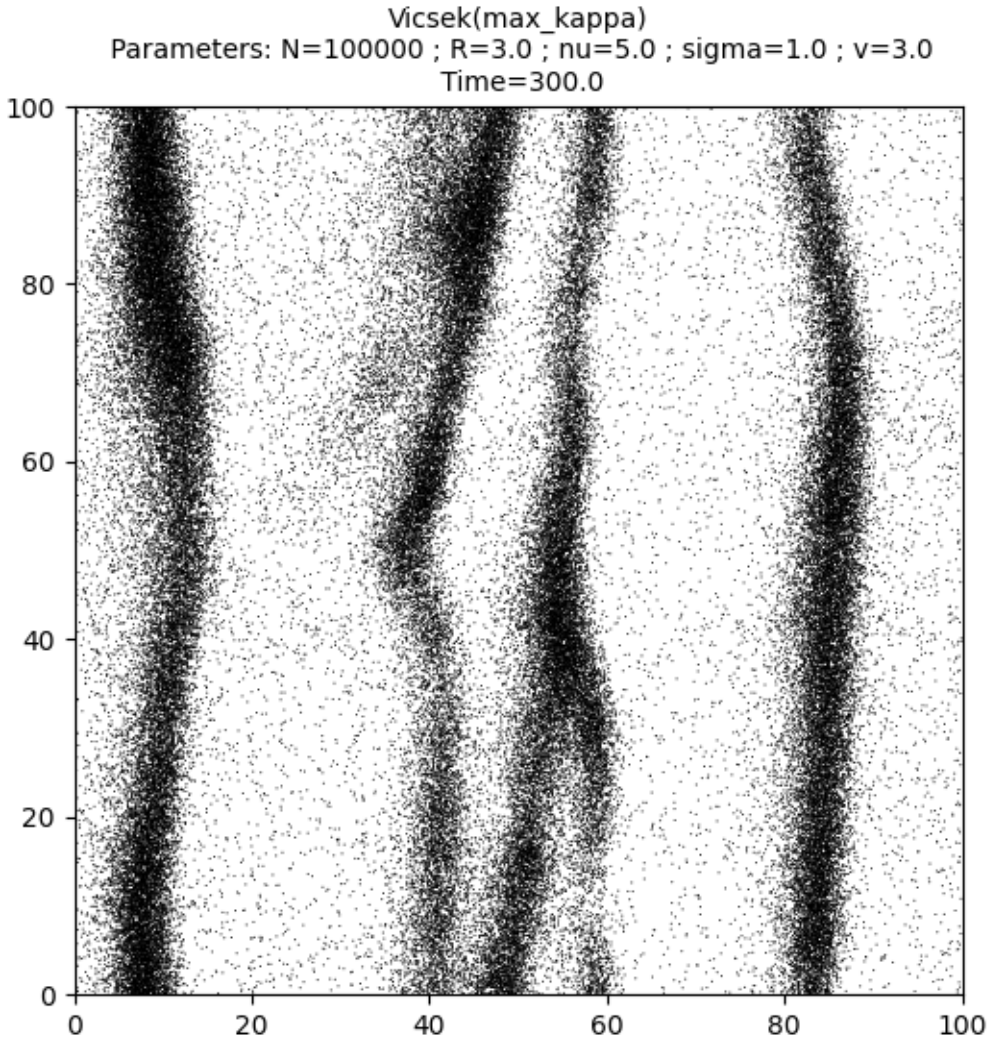
•


Vicsek(max_kappa)
Parameters: N=100000 ; R=3.0 ; nu=5.0 ; sigma=1.0 ; v=3.0
Time=206.0
100
80
60
40
20
0
0
20
40
60
80
100

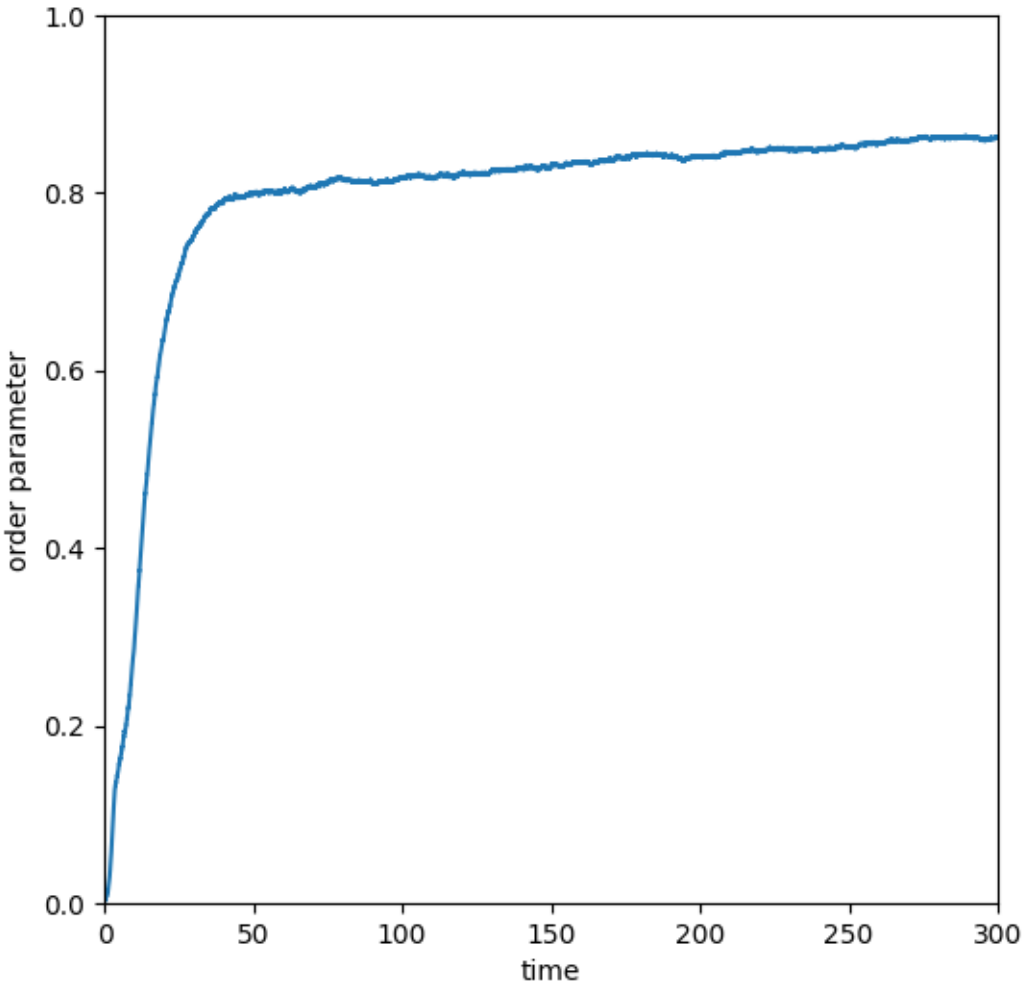
•


Vicsek(max_kappa)
Parameters: N=100000 ; R=3.0 ; nu=5.0 ; sigma=1.0 ; v=3.0
Time=257.0
100
80
60
40
20
0
0
20
40
60
80
100

•


Vicsek(max_kappa)
Parameters: N=100000 ; R=3.0 ; nu=5.0 ; sigma=1.0 ; v=3.0
Time=300.0
100
80
60
40
20
0
0
20
40
60
80
100

•



-

Out:

```
Progress:0%
Progress:1%
Progress:2%
Progress:3%
Progress:4%
Progress:5%
Progress:6%
Progress:7%
Progress:8%
Progress:9%
Progress:10%
Progress:11%
Progress:12%
Progress:13%
Progress:14%
```

(continues on next page)

(continued from previous page)

```
Progress:15%
Progress:16%
Progress:17%
Progress:18%
Progress:19%
Progress:20%
Progress:21%
Progress:22%
Progress:23%
Progress:24%
Progress:25%
Progress:26%
Progress:27%
Progress:28%
Progress:29%
Progress:30%
Progress:31%
Progress:32%
Progress:33%
Progress:34%
Progress:35%
Progress:36%
Progress:37%
Progress:38%
Progress:39%
Progress:40%
Progress:41%
Progress:42%
Progress:43%
Progress:44%
Progress:45%
Progress:46%
Progress:47%
Progress:48%
Progress:49%
Progress:50%
Progress:51%
Progress:52%
Progress:53%
Progress:54%
Progress:55%
Progress:56%
Progress:57%
Progress:58%
Progress:59%
Progress:60%
Progress:61%
Progress:62%
Progress:63%
Progress:64%
Progress:65%
Progress:66%
```

(continues on next page)

(continued from previous page)

```
Progress:67%
Progress:68%
Progress:69%
Progress:70%
Progress:71%
Progress:72%
Progress:73%
Progress:74%
Progress:75%
Progress:76%
Progress:77%
Progress:78%
Progress:79%
Progress:80%
Progress:81%
Progress:82%
Progress:83%
Progress:84%
Progress:85%
Progress:86%
Progress:87%
Progress:88%
Progress:89%
Progress:90%
Progress:91%
Progress:92%
Progress:93%
Progress:94%
Progress:95%
Progress:96%
Progress:97%
Progress:98%
Progress:99%
```

Print the total simulation time and the average time per iteration.

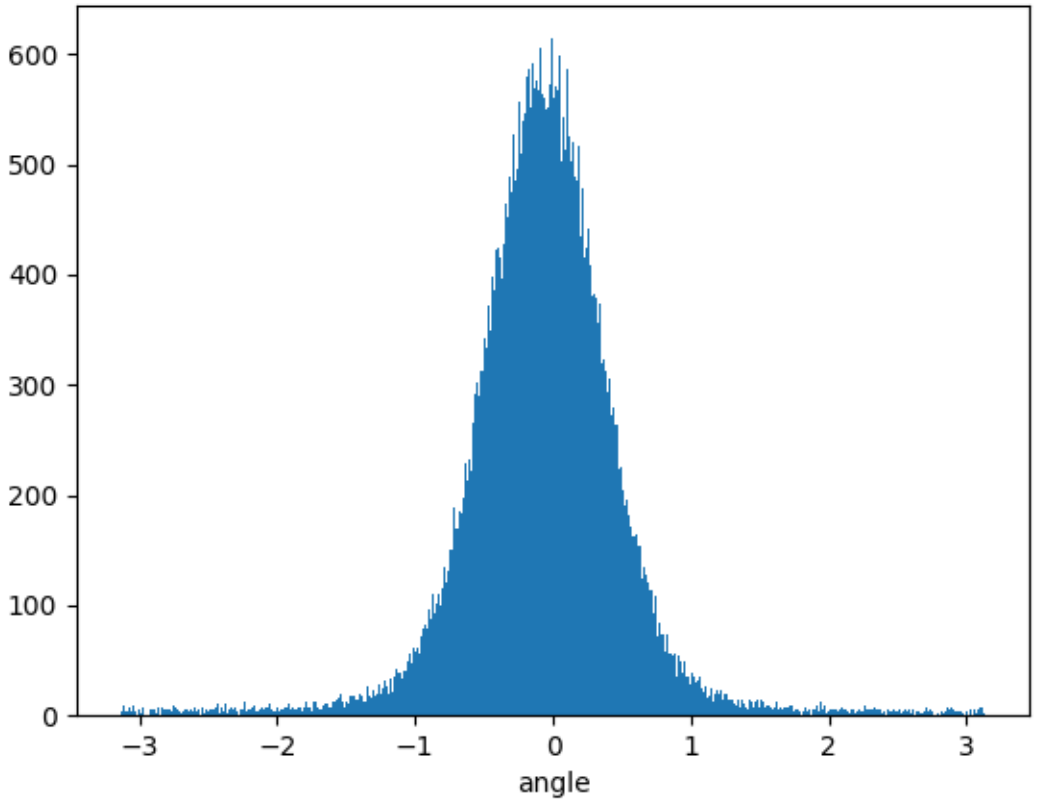
```
print('Total time: '+str(e-s)+' seconds')
print('Average time per iteration: '+str((e-s)/simu.iteration)+' seconds')
```

Out:

```
Total time: 156.8030505180359 seconds
Average time per iteration: 0.005226768350601196 seconds
```

Plot the histogram of the angles of the directions of motion.

```
angle = torch.atan2(simu.vel[:,1],simu.vel[:,0])
angle = angle.cpu().numpy()
h = plt.hist(angle, bins=1000)
plt.xlabel("angle")
plt.show()
```



The system self-organizes into a strongly clustered flock with band-like structures.

Nematic Vicsek model

The target is:

$$J_t^i = \kappa(V_t^i \cdot \overline{\Omega}_t^i)\overline{\Omega}_t^i,$$

where $\overline{\Omega}_t^i$ is any unit eigenvector associated to the maximal eigenvalue of the average Q-tensor:

$$Q_t^i = \frac{1}{N}\sum_{j=1}^{N} K(|X_t^j - X_t^i|)\left(V_t^j \otimes V_t^j - \frac{1}{d} I_d\right).$$

Define the corresponding dictionary…

```
variant = {"name" : "nematic", "parameters" : {}}

simu = Vicsek(
    pos = pos.detach().clone(),
    vel = vel.detach().clone(),
    v = c,
    sigma = sigma,
    nu = nu,
```

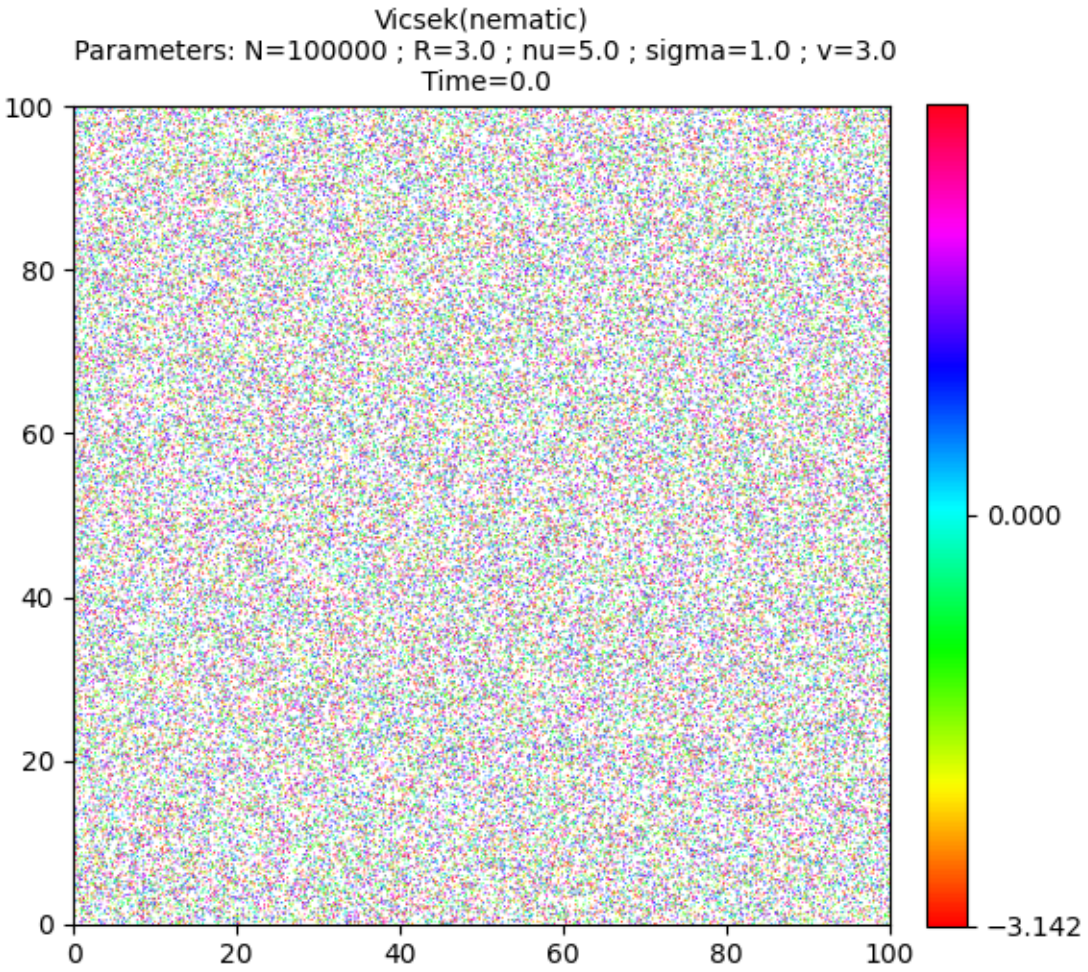
(continues on next page)

(continued from previous page)

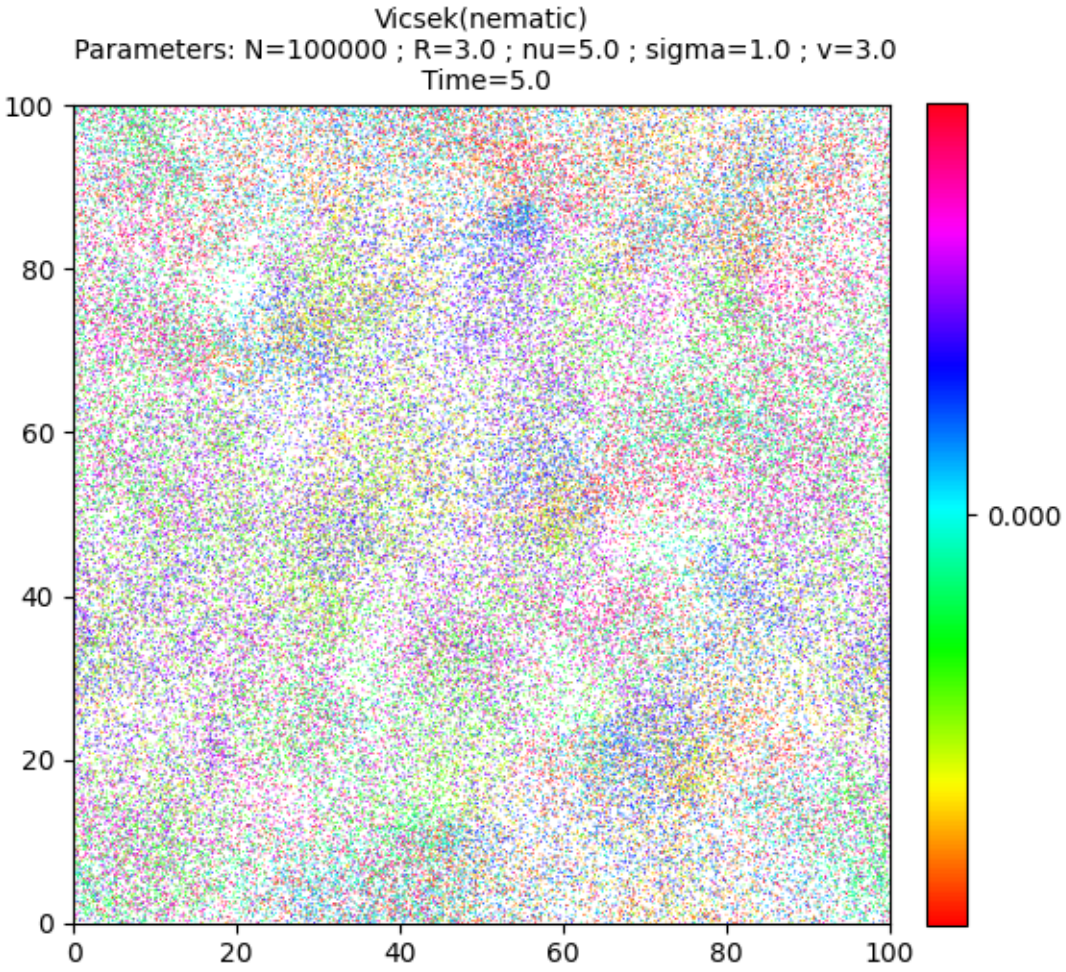
```
    interaction_radius = R,
    box_size = L,
    dt = dt,
    variant = variant,
    block_sparse_reduction = True,
    number_of_cells = 40**2)
```

Finally run the simulation over 100 units of time… The color code indicates the angle of the direction of motion between $-\pi$ and π.

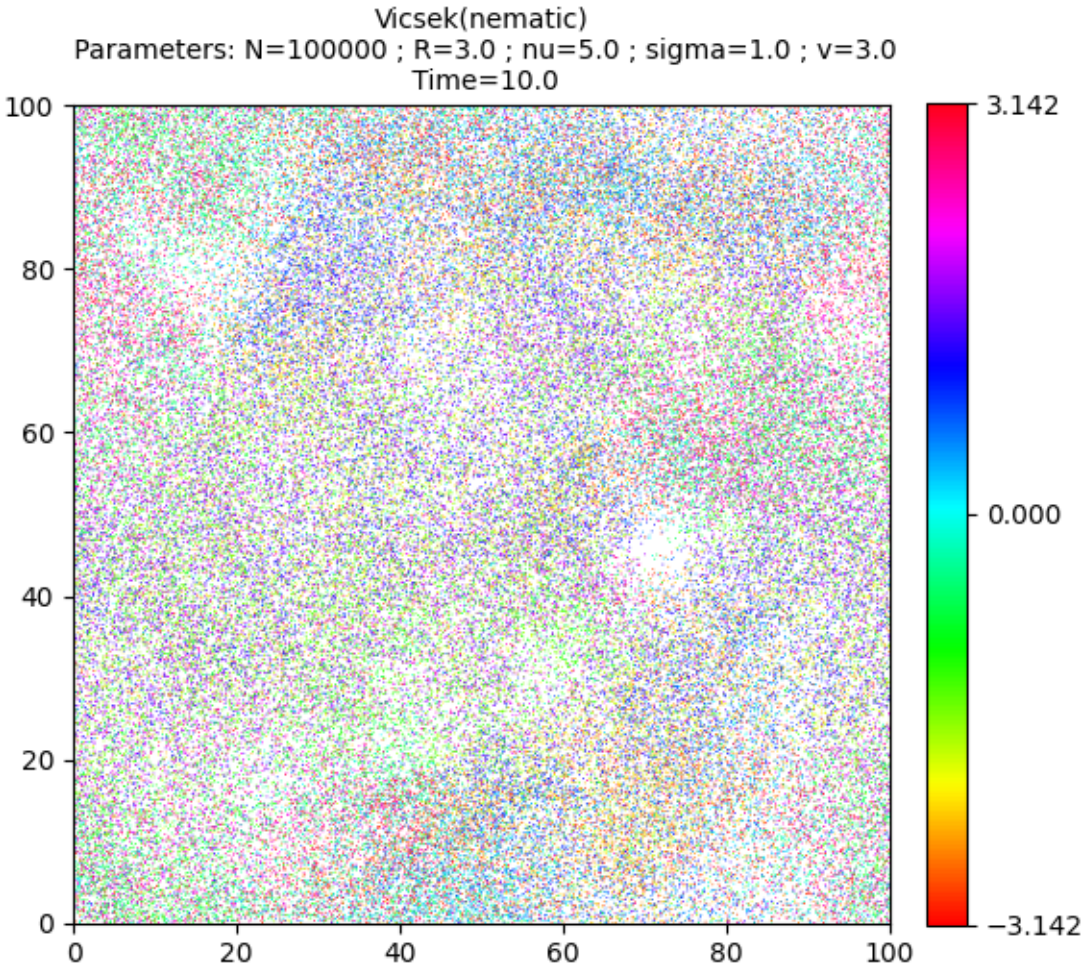
```
frames = [0., 5., 10., 30., 42., 71., 100]

s = time.time()
it, op = display_kinetic_particles(simu, frames, order=True, color=True)
e = time.time()
```



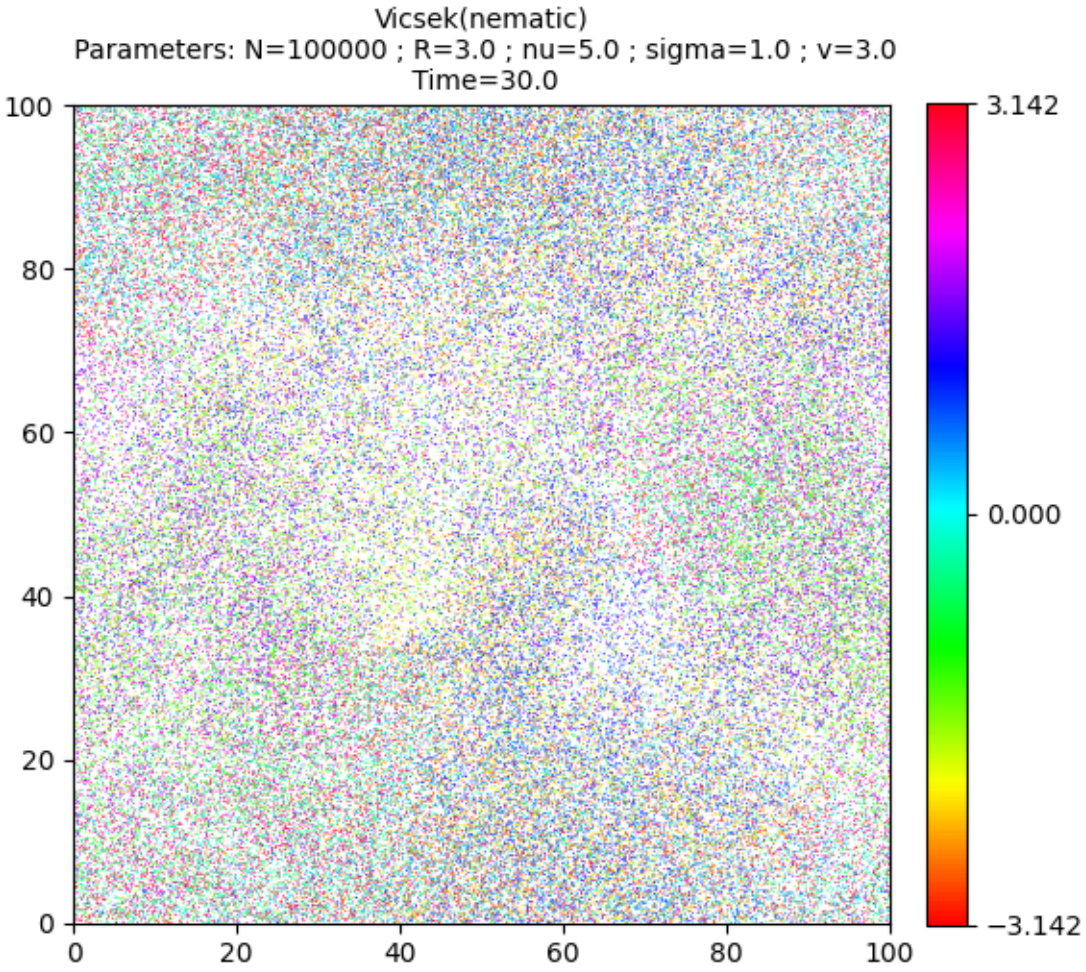
•


Vicsek(nematic)
Parameters: N=100000 ; R=3.0 ; nu=5.0 ; sigma=1.0 ; v=3.0
Time=5.0
100
80
60
40
20
0
0
20
40
60
80
100
0.000

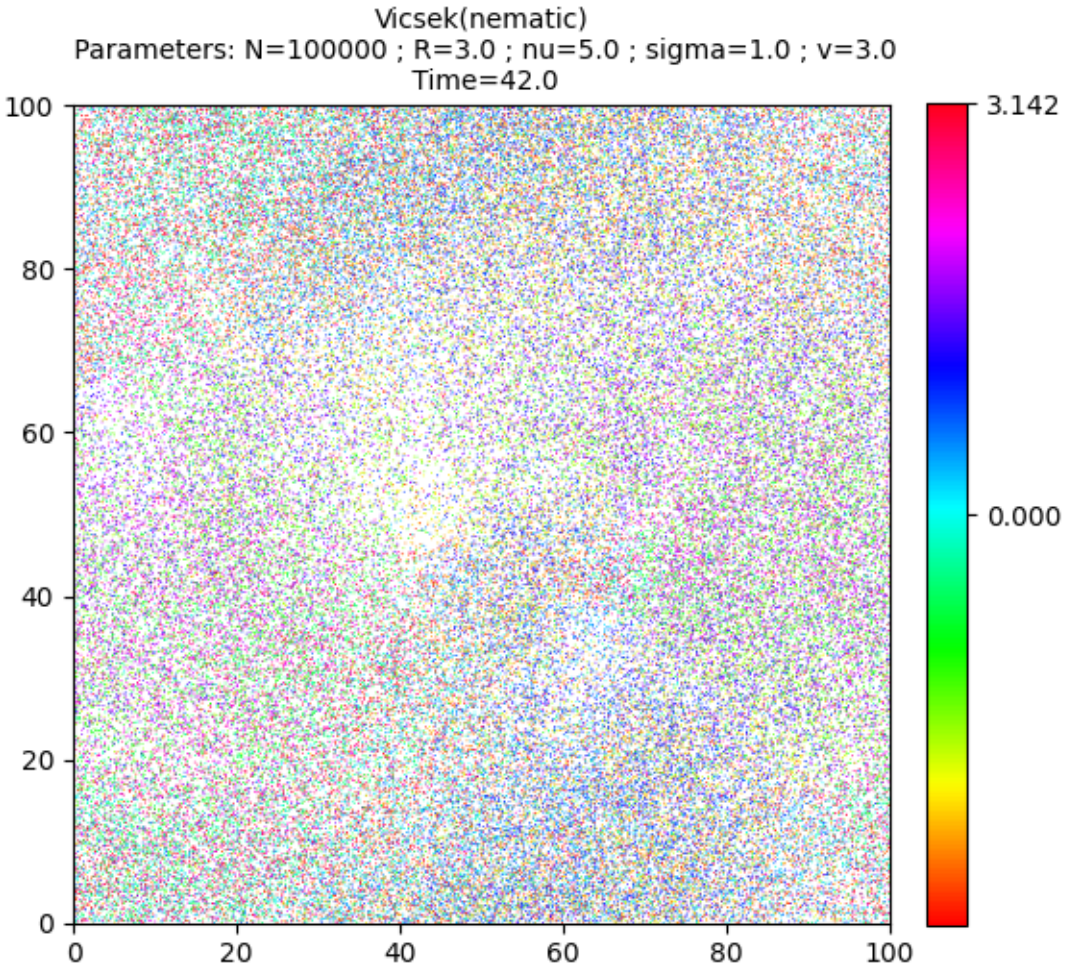
•


Vicsek(nematic)
Parameters: N=100000 ; R=3.0 ; nu=5.0 ; sigma=1.0 ; v=3.0
Time=10.0
100
80
60
40
20
0
0
20
40
60
80
100
3.142
0.000
−3.142

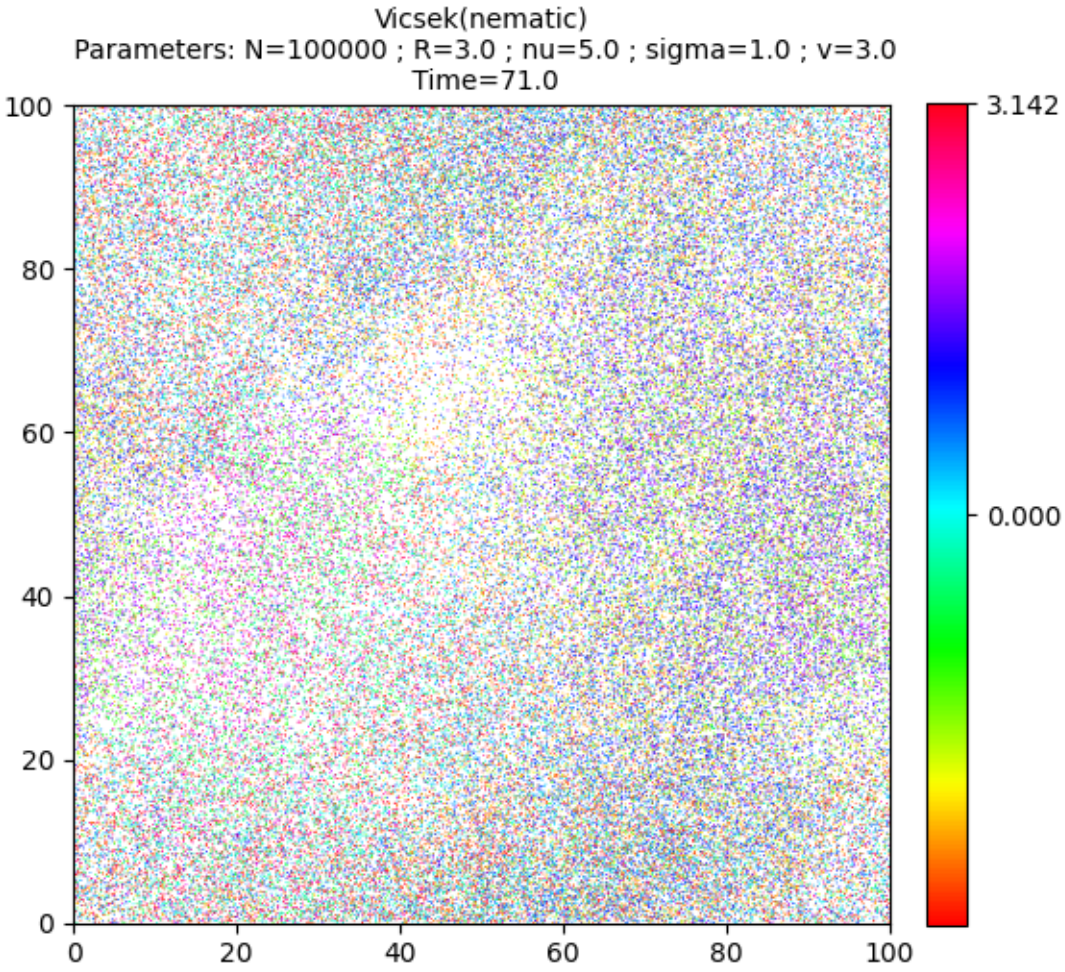
•


Vicsek(nematic)
Parameters: N=100000 ; R=3.0 ; nu=5.0 ; sigma=1.0 ; v=3.0
Time=30.0
100
80
60
40
20
0
0
20
40
60
80
100
3.142
0.000
−3.142

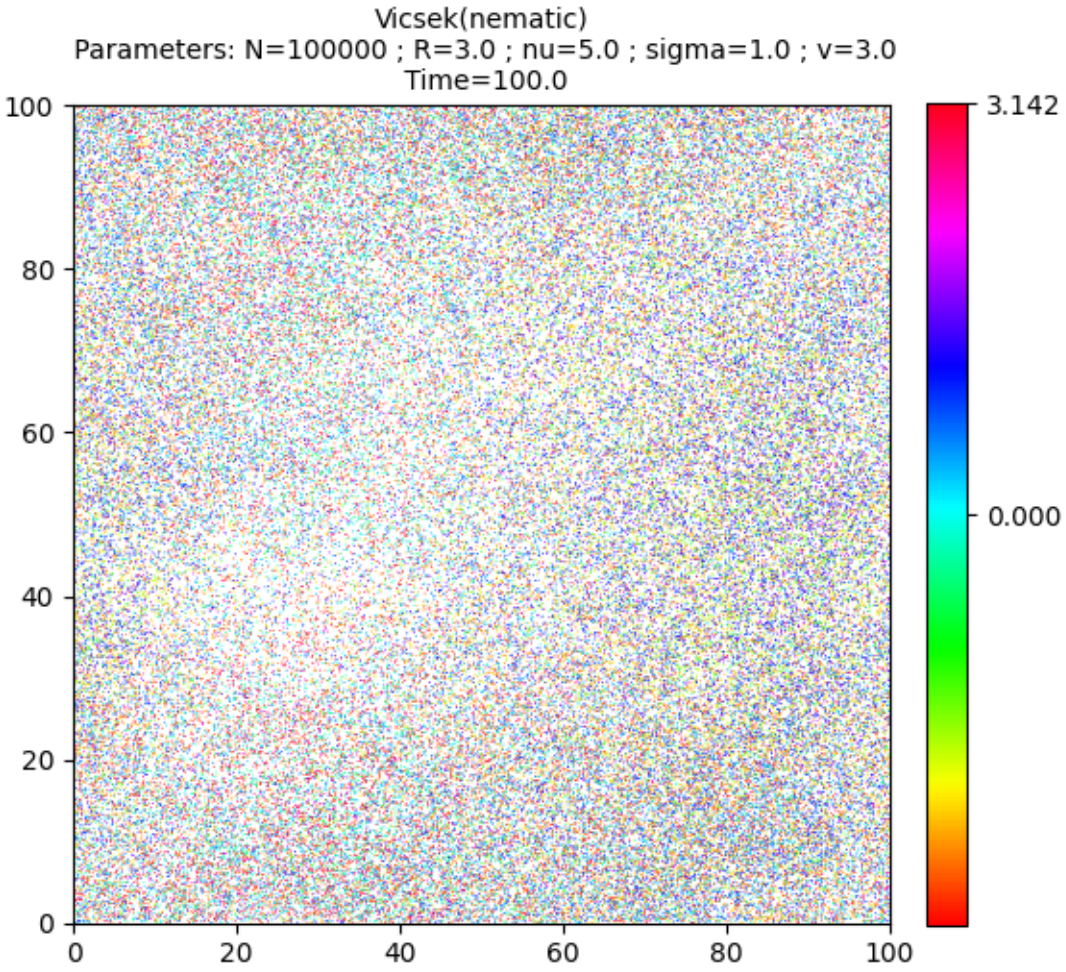
•


Vicsek(nematic)
Parameters: N=100000 ; R=3.0 ; nu=5.0 ; sigma=1.0 ; v=3.0
Time=42.0
100
80
60
40
20
0
0
20
40
60
80
100
3.142
0.000

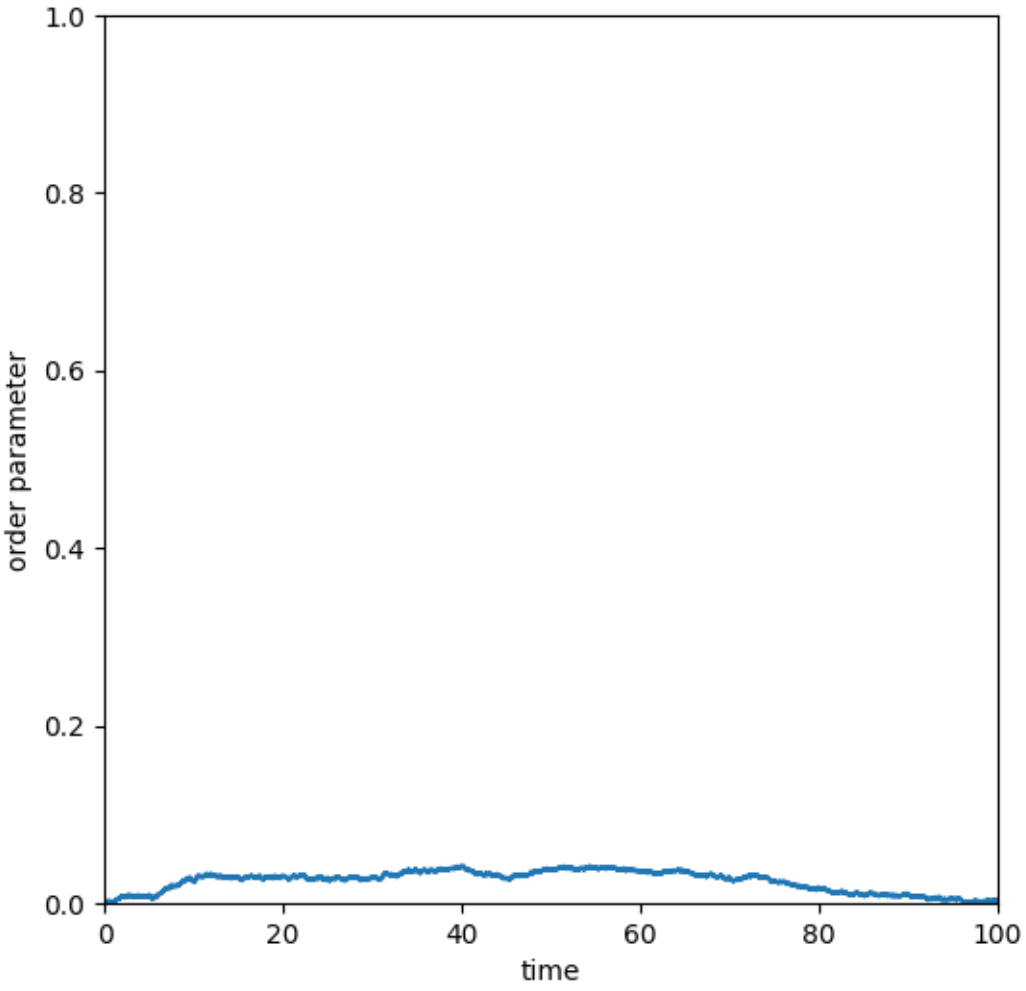
•


Vicsek(nematic)
Parameters: N=100000 ; R=3.0 ; nu=5.0 ; sigma=1.0 ; v=3.0
Time=71.0
100
80
60
40
20
0
0
20
40
60
80
100
3.142
0.000

•


Vicsek(nematic)
Parameters: N=100000 ; R=3.0 ; nu=5.0 ; sigma=1.0 ; v=3.0
Time=100.0
100
80
60
40
20
0
0
20
40
60
80
100
3.142
0.000

•



Out:

```
Progress:0%
Progress:1%
Progress:2%
Progress:3%
Progress:4%
Progress:5%
Progress:6%
Progress:7%
Progress:8%
Progress:9%
Progress:10%
Progress:11%
Progress:12%
Progress:13%
Progress:14%
```

(continues on next page)

(continued from previous page)

```
Progress:15%
Progress:16%
Progress:17%
Progress:18%
Progress:19%
Progress:20%
Progress:21%
Progress:22%
Progress:23%
Progress:24%
Progress:25%
Progress:26%
Progress:27%
Progress:28%
Progress:29%
Progress:30%
Progress:31%
Progress:32%
Progress:33%
Progress:34%
Progress:35%
Progress:36%
Progress:37%
Progress:38%
Progress:39%
Progress:40%
Progress:41%
Progress:42%
Progress:43%
Progress:44%
Progress:45%
Progress:46%
Progress:47%
Progress:48%
Progress:49%
Progress:50%
Progress:51%
Progress:52%
Progress:53%
Progress:54%
Progress:55%
Progress:56%
Progress:57%
Progress:58%
Progress:59%
Progress:60%
Progress:61%
Progress:62%
Progress:63%
Progress:64%
Progress:65%
Progress:66%
```

(continues on next page)

(continued from previous page)

```
Progress:67%
Progress:68%
Progress:69%
Progress:70%
Progress:71%
Progress:72%
Progress:73%
Progress:74%
Progress:75%
Progress:76%
Progress:77%
Progress:78%
Progress:79%
Progress:80%
Progress:81%
Progress:82%
Progress:83%
Progress:84%
Progress:85%
Progress:86%
Progress:87%
Progress:88%
Progress:89%
Progress:90%
Progress:91%
Progress:92%
Progress:93%
Progress:94%
Progress:95%
Progress:96%
Progress:97%
Progress:98%
Progress:99%
```

Print the total simulation time and the average time per iteration.

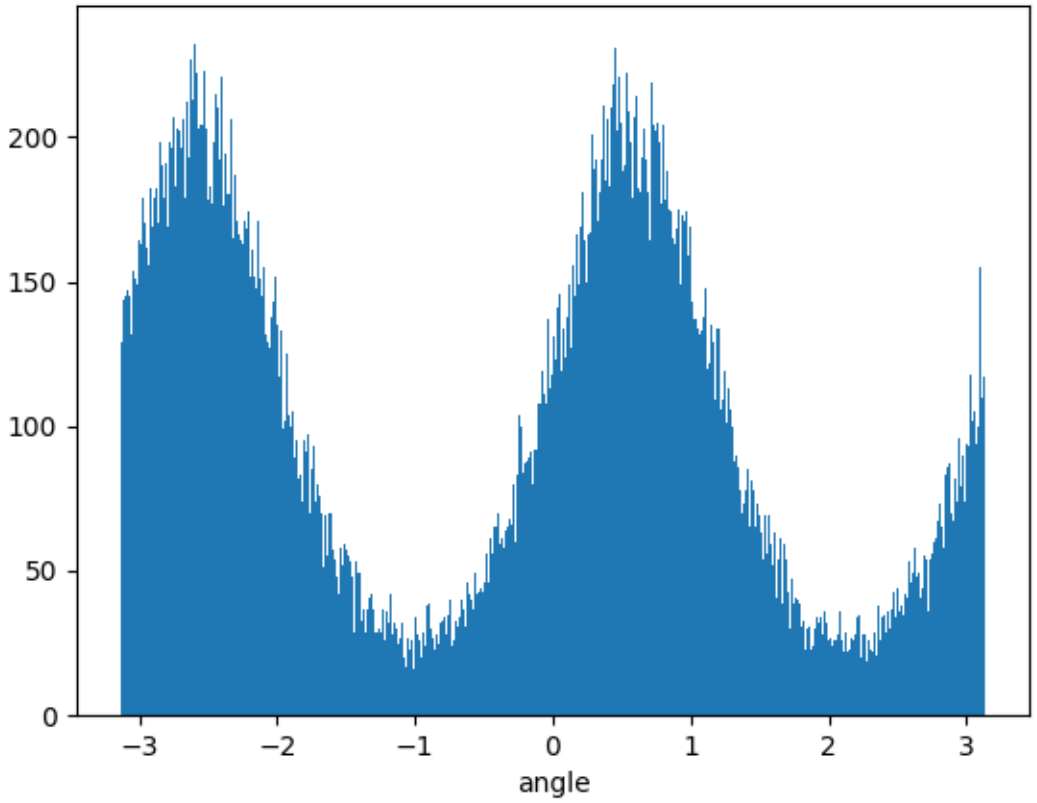
```
print('Total time: '+str(e-s)+' seconds')
print('Average time per iteration: '+str((e-s)/simu.iteration)+' seconds')
```

Out:

```
Total time: 56.330119132995605 seconds
Average time per iteration: 0.00563301191329956 seconds
```

Plot the histogram of the angles of the directions of motion.

```
# sphinx_gallery_thumbnail_number = -1

angle = torch.atan2(simu.vel[:,1],simu.vel[:,0])
angle = angle.cpu().numpy()
h = plt.hist(angle, bins=1000)
plt.xlabel("angle")
plt.show()
```



There are two modes separated by an angle π which indicates that two groups of equal size are moving in opposite direction. This a *nematic flock*.

The target dictionary

Several other targets are implemented. The complete list of available targets can be found in the **dictionary of targets** *target_method*.

```
pprint.pprint(simu.target_method)
```

Out:

```
{'max_kappa': <bound method KineticParticles.max_kappa of <sisyphe.models.Vicsek object␣
↪at 0x7efb4e14e970>>,
 'mean_field': <bound method KineticParticles.mean_field of <sisyphe.models.Vicsek␣
↪object at 0x7efb4e14e970>>,
 'morse': <bound method Particles.morse_target of <sisyphe.models.Vicsek object at␣
↪0x7efb4e14e970>>,
 'motsch_tadmor': <bound method KineticParticles.motsch_tadmor of <sisyphe.models.Vicsek␣
↪object at 0x7efb4e14e970>>,
 'nematic': <bound method KineticParticles.nematic of <sisyphe.models.Vicsek object at␣
↪0x7efb4e14e970>>,
 'normalised': <bound method KineticParticles.normalised of <sisyphe.models.Vicsek␣
```

(continues on next page)

(continued from previous page)

```
↪object at 0x7efb4e14e970>>,
 'overlapping_repulsion': <bound method Particles.overlapping_repulsion_target of
↪<sisyphe.models.Vicsek object at 0x7efb4e14e970>>,
 'quadratic_potential': <bound method Particles.quadratic_potential_target of <sisyphe.
↪models.Vicsek object at 0x7efb4e14e970>>}
```

Custom targets can be added to the dictionary of targets using the method *add_target_method()*. Then a target can be readily used in a simulation using the method *compute_target()*.

Options

Customized targets can also be defined by applying an **option** which modifies an existing target. See *Mills*.

Total running time of the script: (6 minutes 25.941 seconds)

4.4.3 Tutorial 03: Boundary conditions

The particle systems are defined in a rectangular box whose dimensions are specified by the attribute L.

The boundary conditions are specified by the attribute bc which can be one of the following.

- A list of size d containing for each dimension either 0 (periodic) or 1 (wall with reflecting boundary conditions).
- The string "open" : no boundary conditions.
- The string "periodic" : periodic boundary conditions.
- The string "spherical" : reflecting boundary conditions on the sphere of diameter L enclosed in the square domain $[0, L]^d$.

For instance, let us simulate the Vicsek model in an elongated rectangular domain $[0, L_x] \times [0, L_y]$ with periodic boundary conditions in the x-dimension and reflecting boundary conditions in the y-dimension.

First, some standard imports…

```
import time
import torch
from sisyphe.models import Vicsek
from sisyphe.display import display_kinetic_particles

use_cuda = torch.cuda.is_available()
dtype = torch.cuda.FloatTensor if use_cuda else torch.FloatTensor
```

The parameters of the model

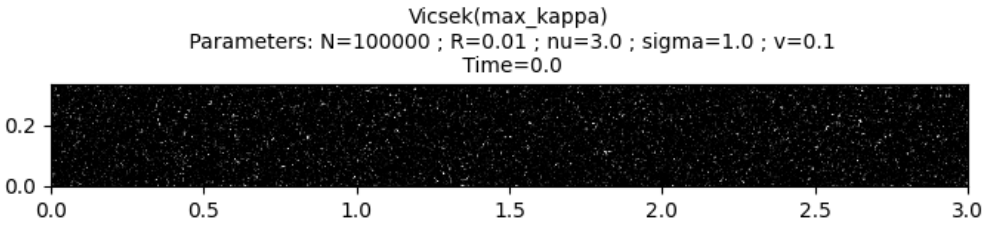
```
N = 100000

R = .01
c = .1
nu = 3.
sigma = 1.

dt = .01

variant = {"name" : "max_kappa", "parameters" : {"kappa_max" : 10.}}
```

The spatial domain, the boundary conditions and the initial conditions…

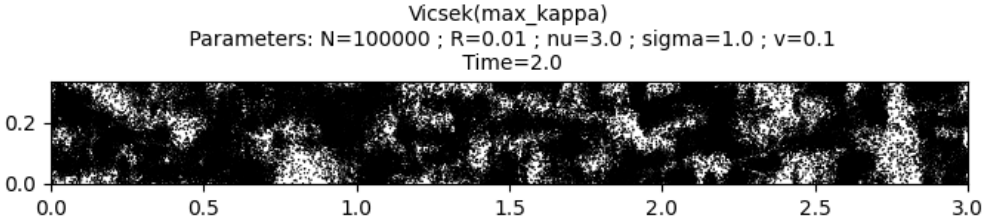
```
Lx = 3.
Ly = 1./3.
L = [Lx, Ly]
bc = [0,1]

pos = torch.rand((N,2)).type(dtype)
pos[:,0] = L[0]*pos[:,0]
pos[:,1] = L[1]*pos[:,1]
vel = torch.randn(N,2).type(dtype)
vel = vel/torch.norm(vel,dim=1).reshape((N,1))

simu = Vicsek(
    pos = pos.detach().clone(),
    vel = vel.detach().clone(),
    v = c,
    sigma = sigma,
    nu = nu,
    interaction_radius = R,
    box_size = L,
    boundary_conditions=bc,
    dt = dt,
    variant = variant,
    block_sparse_reduction = True,
    number_of_cells = 100**2)
```

Finally run the simulation over 300 units of time.…

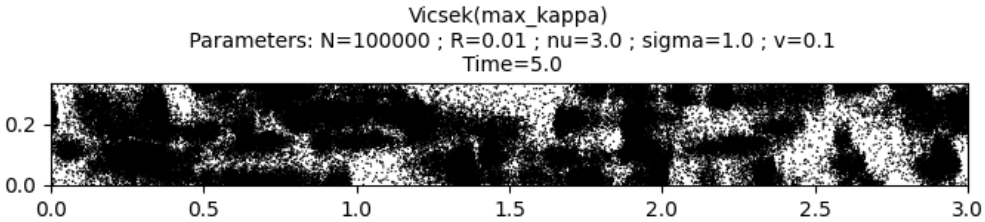
```
# sphinx_gallery_thumbnail_number = 15

frames = [0., 2., 5., 10., 30., 42., 71., 100, 123, 141, 182, 203, 256, 272, 300]

s = time.time()
it, op = display_kinetic_particles(simu, frames, order=True, figsize=(8,3))
e = time.time()
```



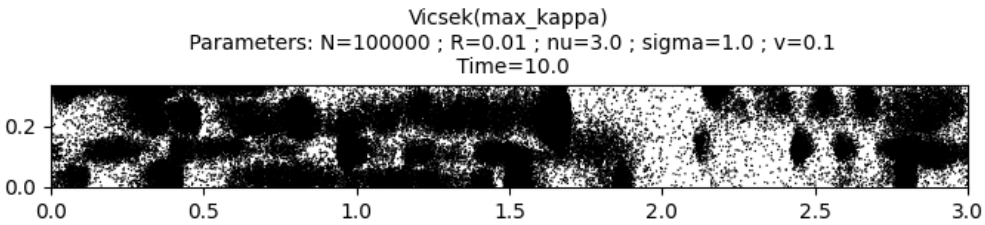
-


Vicsek(max_kappa)
Parameters: N=100000 ; R=0.01 ; nu=3.0 ; sigma=1.0 ; v=0.1
Time=2.0
0.2
0.0
0.0
0.5
1.0
1.5
2.0
2.5
3.0

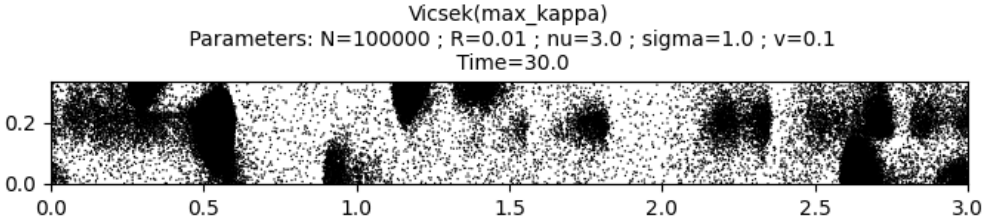
-


Vicsek(max_kappa)
Parameters: N=100000 ; R=0.01 ; nu=3.0 ; sigma=1.0 ; v=0.1
Time=5.0
0.2
0.0
0.0
0.5
1.0
1.5
2.0
2.5
3.0

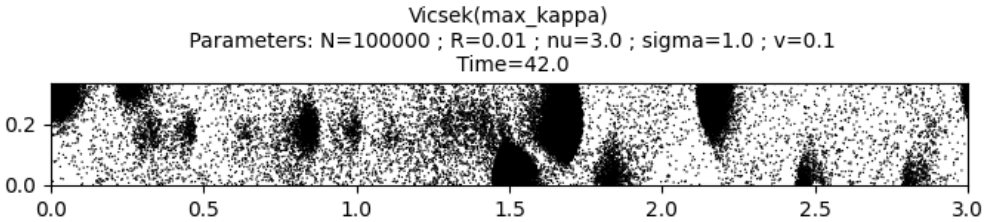
-


Vicsek(max_kappa)
Parameters: N=100000 ; R=0.01 ; nu=3.0 ; sigma=1.0 ; v=0.1
Time=10.0
0.2
0.0
0.0
0.5
1.0
1.5
2.0
2.5
3.0

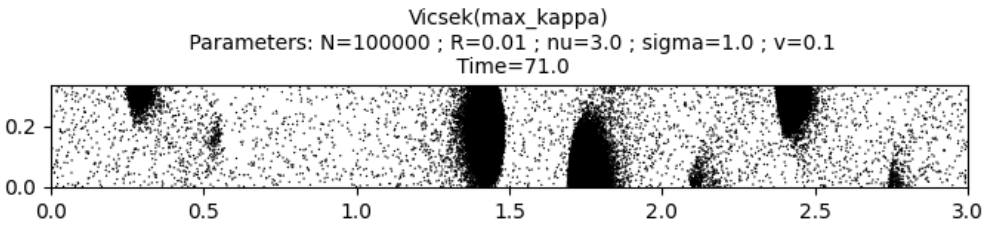
-


Vicsek(max_kappa)
Parameters: N=100000 ; R=0.01 ; nu=3.0 ; sigma=1.0 ; v=0.1
Time=30.0
0.2
0.0
0.0
0.5
1.0
1.5
2.0
2.5
3.0

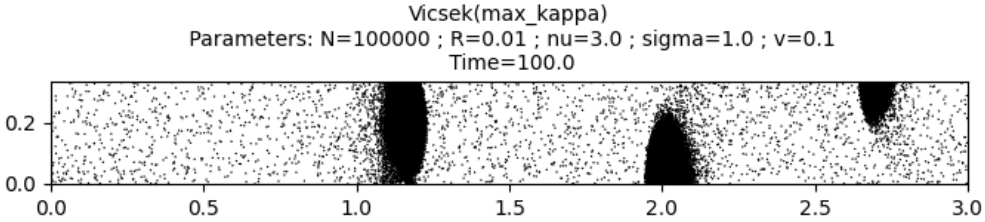
-


Vicsek(max_kappa)
Parameters: N=100000 ; R=0.01 ; nu=3.0 ; sigma=1.0 ; v=0.1
Time=42.0
0.2
0.0
0.0
0.5
1.0
1.5
2.0
2.5
3.0

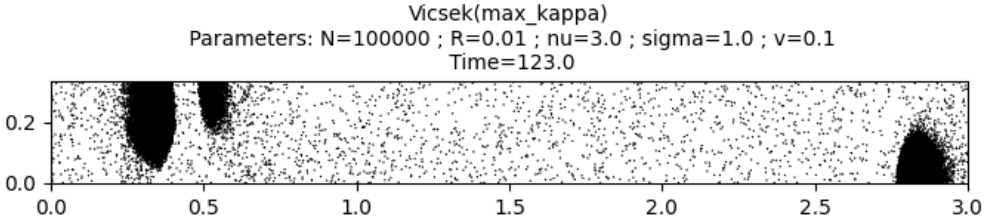
-


Vicsek(max_kappa)
Parameters: N=100000 ; R=0.01 ; nu=3.0 ; sigma=1.0 ; v=0.1
Time=71.0
0.2
0.0
0.0
0.5
1.0
1.5
2.0
2.5
3.0

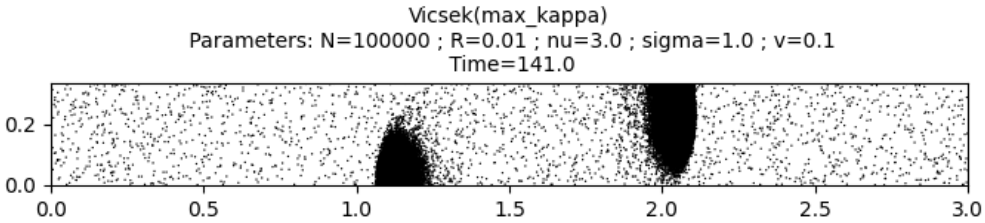
-


Vicsek(max_kappa)
Parameters: N=100000 ; R=0.01 ; nu=3.0 ; sigma=1.0 ; v=0.1
Time=100.0
0.2
0.0
0.0
0.5
1.0
1.5
2.0
2.5
3.0

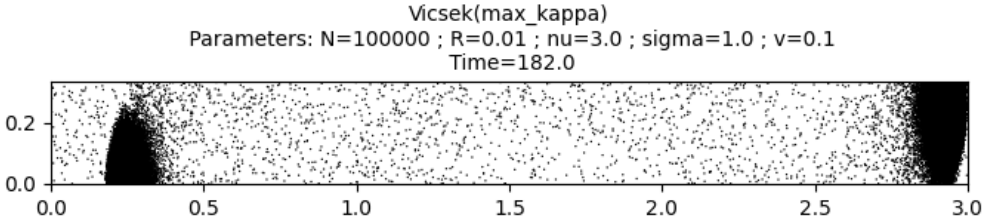
-


Vicsek(max_kappa)
Parameters: N=100000 ; R=0.01 ; nu=3.0 ; sigma=1.0 ; v=0.1
Time=123.0
0.2
0.0
0.0
0.5
1.0
1.5
2.0
2.5
3.0

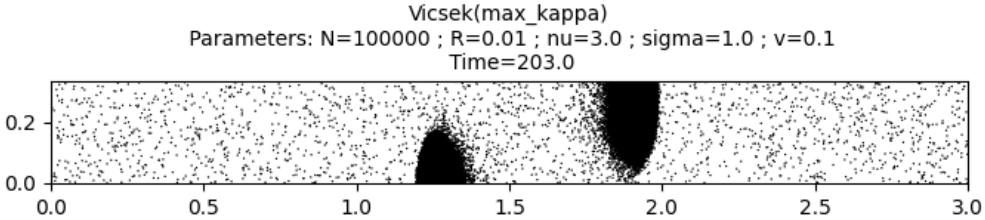
-


Vicsek(max_kappa)
Parameters: N=100000 ; R=0.01 ; nu=3.0 ; sigma=1.0 ; v=0.1
Time=141.0
0.2
0.0
0.0
0.5
1.0
1.5
2.0
2.5
3.0

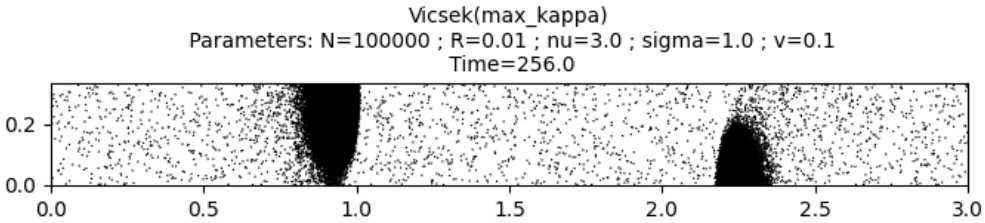
-


Vicsek(max_kappa)
Parameters: N=100000 ; R=0.01 ; nu=3.0 ; sigma=1.0 ; v=0.1
Time=182.0
0.2
0.0
0.0
0.5
1.0
1.5
2.0
2.5
3.0

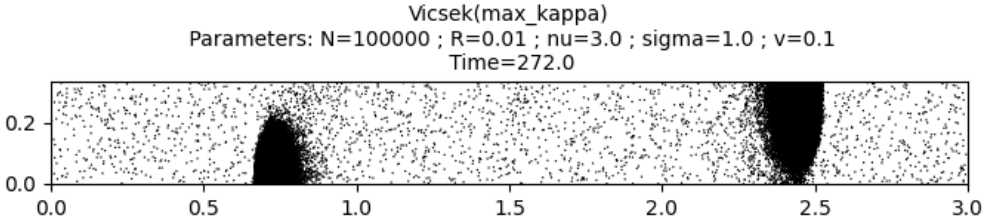
-


Vicsek(max_kappa)
Parameters: N=100000 ; R=0.01 ; nu=3.0 ; sigma=1.0 ; v=0.1
Time=203.0
0.2
0.0
0.0
0.5
1.0
1.5
2.0
2.5
3.0

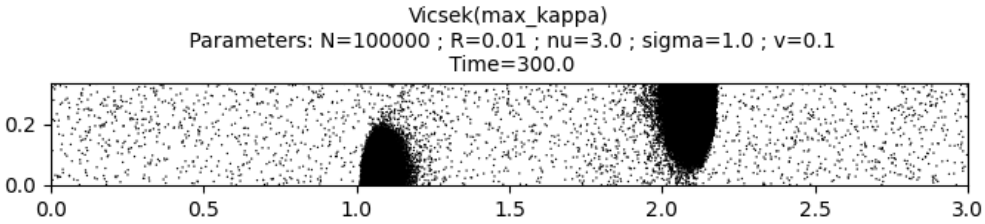
-


Vicsek(max_kappa)
Parameters: N=100000 ; R=0.01 ; nu=3.0 ; sigma=1.0 ; v=0.1
Time=256.0
0.2
0.0
0.0
0.5
1.0
1.5
2.0
2.5
3.0

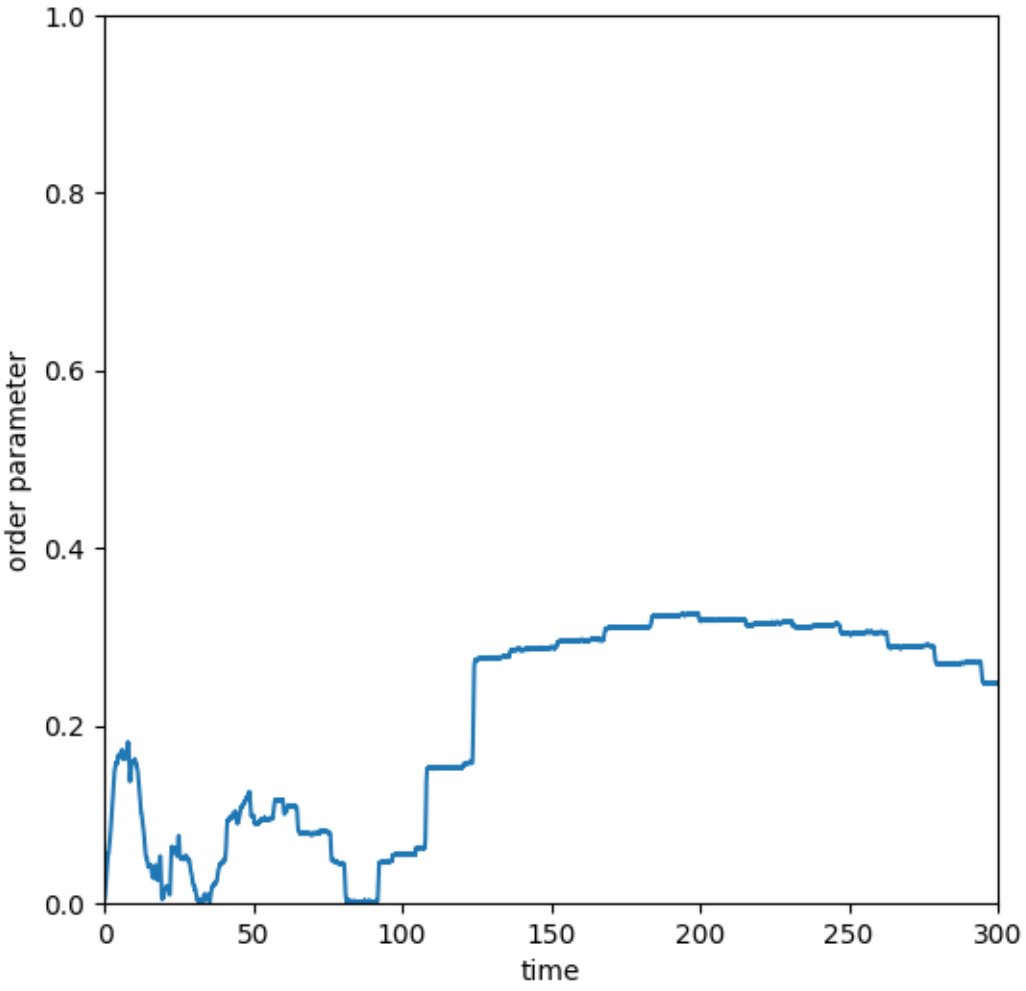
-


Vicsek(max_kappa)
Parameters: N=100000 ; R=0.01 ; nu=3.0 ; sigma=1.0 ; v=0.1
Time=272.0
0.2
0.0
0.0
0.5
1.0
1.5
2.0
2.5
3.0

-


Vicsek(max_kappa)
Parameters: N=100000 ; R=0.01 ; nu=3.0 ; sigma=1.0 ; v=0.1
Time=300.0
0.2
0.0
0.0
0.5
1.0
1.5
2.0
2.5
3.0

-



-

Out:

```
Progress:0%
Progress:1%
Progress:2%
Progress:3%
Progress:4%
Progress:5%
Progress:6%
Progress:7%
Progress:8%
Progress:9%
Progress:10%
Progress:11%
Progress:12%
Progress:13%
Progress:14%
```

(continues on next page)

(continued from previous page)

```
Progress:15%
Progress:16%
Progress:17%
Progress:18%
Progress:19%
Progress:20%
Progress:21%
Progress:22%
Progress:23%
Progress:24%
Progress:25%
Progress:26%
Progress:27%
Progress:28%
Progress:29%
Progress:30%
Progress:31%
Progress:32%
Progress:33%
Progress:34%
Progress:35%
Progress:36%
Progress:37%
Progress:38%
Progress:39%
Progress:40%
Progress:41%
Progress:42%
Progress:43%
Progress:44%
Progress:45%
Progress:46%
Progress:47%
Progress:48%
Progress:49%
Progress:50%
Progress:51%
Progress:52%
Progress:53%
Progress:54%
Progress:55%
Progress:56%
Progress:57%
Progress:58%
Progress:59%
Progress:60%
Progress:61%
Progress:62%
Progress:63%
Progress:64%
Progress:65%
Progress:66%
```

(continues on next page)

(continued from previous page)

```
Progress:67%
Progress:68%
Progress:69%
Progress:70%
Progress:71%
Progress:72%
Progress:73%
Progress:74%
Progress:75%
Progress:76%
Progress:77%
Progress:78%
Progress:79%
Progress:80%
Progress:81%
Progress:82%
Progress:83%
Progress:84%
Progress:85%
Progress:86%
Progress:87%
Progress:88%
Progress:89%
Progress:90%
Progress:91%
Progress:92%
Progress:93%
Progress:94%
Progress:95%
Progress:96%
Progress:97%
Progress:98%
Progress:99%
```

Print the total simulation time and the average time per iteration.

```
print('Total time: '+str(e-s)+' seconds')
print('Average time per iteration: '+str((e-s)/simu.iteration)+' seconds')
```

Out:

```
Total time: 523.2031211853027 seconds
Average time per iteration: 0.01744010403951009 seconds
```

The simulation produces small clusters moving from left to right or from right to left. Each “step” in the order parameter corresponds to a collision between two clusters moving in opposite directions.

Total running time of the script: (8 minutes 55.368 seconds)

4.4.4 Tutorial 04: Block-sparse reduction

In many cases, the interaction radius R is much smaller than the size of the domain. Consequently, the sums in the local averages (see *Tutorial 05: Kernels and averages*) contain only a small fraction of non zero terms. To gain in efficiency, we can follow the classical strategy:

- Subdivide the domain into a fixed number of cells of size at least R.
- For a particle in a given cell, only look at the contiguous cells to compute the local averages. In dimension d, there are 3^d contiguous cells (including the cell itself).

A practical implementation is called the *Verlet list method*. However, the implementation below is different than the classical one. It is adapted from the block-sparse reduction method implemented in the KeOps library.

We illustrate the gain in efficency for the Vicsek model.

Note: The method is sub-optimal for moderate numbers of particles. As a rule of thumb, the block-sparse reduction method becomes useful for systems with at least 10^4 particles.

Set up and benchmarks

First, some standard imports…

```
import copy
import time
import torch
from matplotlib import pyplot as plt

use_cuda = torch.cuda.is_available()
dtype = torch.cuda.FloatTensor if use_cuda else torch.FloatTensor
```

Let the N particles be uniformly scattered in a box of size L with interaction radius R and uniformly sampled velocities.

```
from sisyphe.models import Vicsek

N = 100000
L = 100.
R = 1.

pos = L*torch.rand((N,2)).type(dtype)
vel = torch.randn(N,2).type(dtype)
vel = vel/torch.norm(vel,dim=1).reshape((N,1))

simu=Vicsek(pos=pos,vel=vel,
            v=1.,
            sigma=1.,nu=3.,
            interaction_radius=R,
            box_size=L)

simu.__next__() #GPU warmup...
```

Out:

```
{'position': tensor([[31.6564,  6.3145],
        [61.3271, 90.9929],
        [88.1392,  1.8941],
        ...,
        [98.8685, 90.9011],
        [28.4884, 29.0810],
        [15.1606, 66.4232]], device='cuda:0'), 'velocity': tensor([[ 0.9581, -0.2866],
        [-0.2292, -0.9734],
        [-0.8279, -0.5608],
        ...,
        [-0.6027, -0.7979],
        [ 0.9594, -0.2821],
        [-0.4047, -0.9145]], device='cuda:0')}
```

Without block-sparse reduction, let us compute the simulation time of 100 iterations.

```
simu_copy = copy.deepcopy(simu) # Make a new deepcopy
s = time.time()
for k in range(100):
    simu_copy.__next__()
e = time.time()

simulation_time = e-s

print("Average simulation time without block-sparse reduction: " + str(simulation_time)␣
↪+ " seconds.")
```

Out:

```
Average simulation time without block-sparse reduction: 2.9741082191467285 seconds.
```

Then with block-sparse reduction… First, turn on the attribute `blocksparse`.

```
simu.blocksparse = True
```

Then, we need to define the maximum number of cells. This can be set by the keyword argument `number_of_cells` when an instance of the class *sisyphe.particles.Particles* is created. The number of cells has a strong influence on the efficiency of the method and should be chosen wisely. When the optimal value is not known a priori, it is recommanded to use the method *best_blocksparse_parameters()* which will time 100 iterations of the simulation for various numbers of cells and automatically choose the best one. Below, we test all the numbers of cells which are powers of the dimension (here $d = 2$) between 10^2 and 70^2.

```
ncell_min = 10
ncell_max = 70
fastest, nb_cells, average_simu_time, simulation_time = simu.best_blocksparse_
↪parameters(ncell_min, ncell_max, step=1, nb_calls=100)
```

Out:

```
Progress:0.0%
Progress:1.67%
Progress:3.33%
Progress:5.0%
Progress:6.67%
```

(continues on next page)

(continued from previous page)

```
Progress:8.33%
Progress:10.0%
Progress:11.67%
Progress:13.33%
Progress:15.0%
Progress:16.67%
Progress:18.33%
Progress:20.0%
Progress:21.67%
Progress:23.33%
Progress:25.0%
Progress:26.67%
Progress:28.33%
Progress:30.0%
Progress:31.67%
Progress:33.33%
Progress:35.0%
Progress:36.67%
Progress:38.33%
Progress:40.0%
Progress:41.67%
Progress:43.33%
Progress:45.0%
Progress:46.67%
Progress:48.33%
Progress:50.0%
Progress:51.67%
Progress:53.33%
Progress:55.0%
Progress:56.67%
Progress:58.33%
Progress:60.0%
Progress:61.67%
Progress:63.33%
Progress:65.0%
Progress:66.67%
Progress:68.33%
Progress:70.0%
Progress:71.67%
Progress:73.33%
Progress:75.0%
Progress:76.67%
Progress:78.33%
Progress:80.0%
Progress:81.67%
Progress:83.33%
Progress:85.0%
Progress:86.67%
Progress:88.33%
Progress:90.0%
Progress:91.67%
Progress:93.33%
```

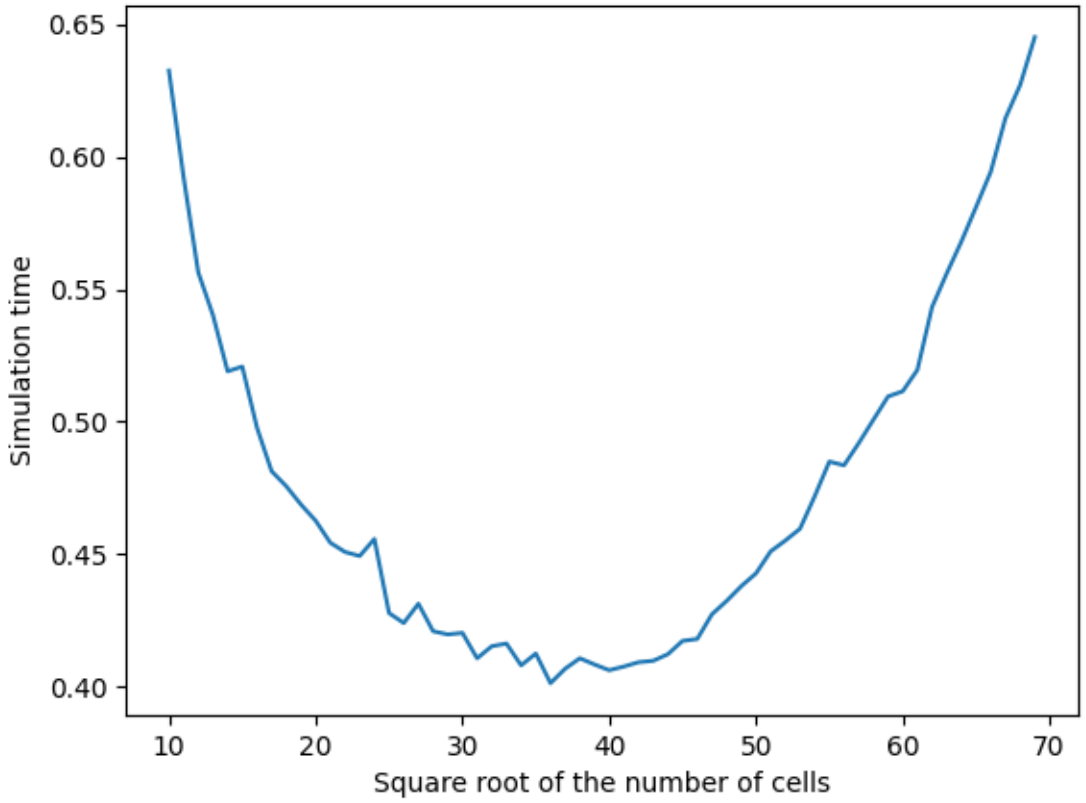
(continues on next page)

(continued from previous page)

```
Progress:95.0%
Progress:96.67%
Progress:98.33%
```

We plot the average simulation time as a function of the square root of the number of cells and print the best.

```
plt.plot(nb_cells,average_simu_time)
plt.xlabel("Square root of the number of cells")
plt.ylabel("Simulation time")

print("Average simulation time with block-sparse reduction: " + str(average_simu_time.
↪min()) + " seconds.")
```



Out:

```
Average simulation time with block-sparse reduction: 0.4012424945831299 seconds.
```

Same experiment with one million particles.

```
N = 1000000
L = 100.
R = 1.
```

(continues on next page)

(continued from previous page)

```
pos = L*torch.rand((N,2)).type(dtype)
vel = torch.randn(N,2).type(dtype)
vel = vel/torch.norm(vel,dim=1).reshape((N,1))


simu=Vicsek(pos=pos,vel=vel,
            v=1.,
            sigma=1.,nu=3.,
            interaction_radius=R,
            box_size=L,
            block_sparse_reduction=False)

simu_copy = copy.deepcopy(simu) # Make a new deepcopy
s = time.time()
for k in range(100):
    simu_copy.__next__()
e = time.time()

simulation_time = e-s

print("Average simulation time without block-sparse reduction: " + str(simulation_time)␣
↪+ " seconds.")
```

Out:

```
Average simulation time without block-sparse reduction: 274.7271876335144 seconds.
```

With block-sparse reduction…

```
simu.blocksparse = True

fastest, nb_cells, average_simu_time, simulation_time = simu.best_blocksparse_
↪parameters(30, 100, nb_calls=100)
```

Out:

```
Progress:0.0%
Progress:1.43%
Progress:2.86%
Progress:4.29%
Progress:5.71%
Progress:7.14%
Progress:8.57%
Progress:10.0%
Progress:11.43%
Progress:12.86%
Progress:14.29%
Progress:15.71%
Progress:17.14%
Progress:18.57%
Progress:20.0%
Progress:21.43%
Progress:22.86%
```

(continues on next page)

(continued from previous page)

```
Progress:24.29%
Progress:25.71%
Progress:27.14%
Progress:28.57%
Progress:30.0%
Progress:31.43%
Progress:32.86%
Progress:34.29%
Progress:35.71%
Progress:37.14%
Progress:38.57%
Progress:40.0%
Progress:41.43%
Progress:42.86%
Progress:44.29%
Progress:45.71%
Progress:47.14%
Progress:48.57%
Progress:50.0%
Progress:51.43%
Progress:52.86%
Progress:54.29%
Progress:55.71%
Progress:57.14%
Progress:58.57%
Progress:60.0%
Progress:61.43%
Progress:62.86%
Progress:64.29%
Progress:65.71%
Progress:67.14%
Progress:68.57%
Progress:70.0%
Progress:71.43%
Progress:72.86%
Progress:74.29%
Progress:75.71%
Progress:77.14%
Progress:78.57%
Progress:80.0%
Progress:81.43%
Progress:82.86%
Progress:84.29%
Progress:85.71%
Progress:87.14%
Progress:88.57%
Progress:90.0%
Progress:91.43%
Progress:92.86%
Progress:94.29%
Progress:95.71%
Progress:97.14%
```

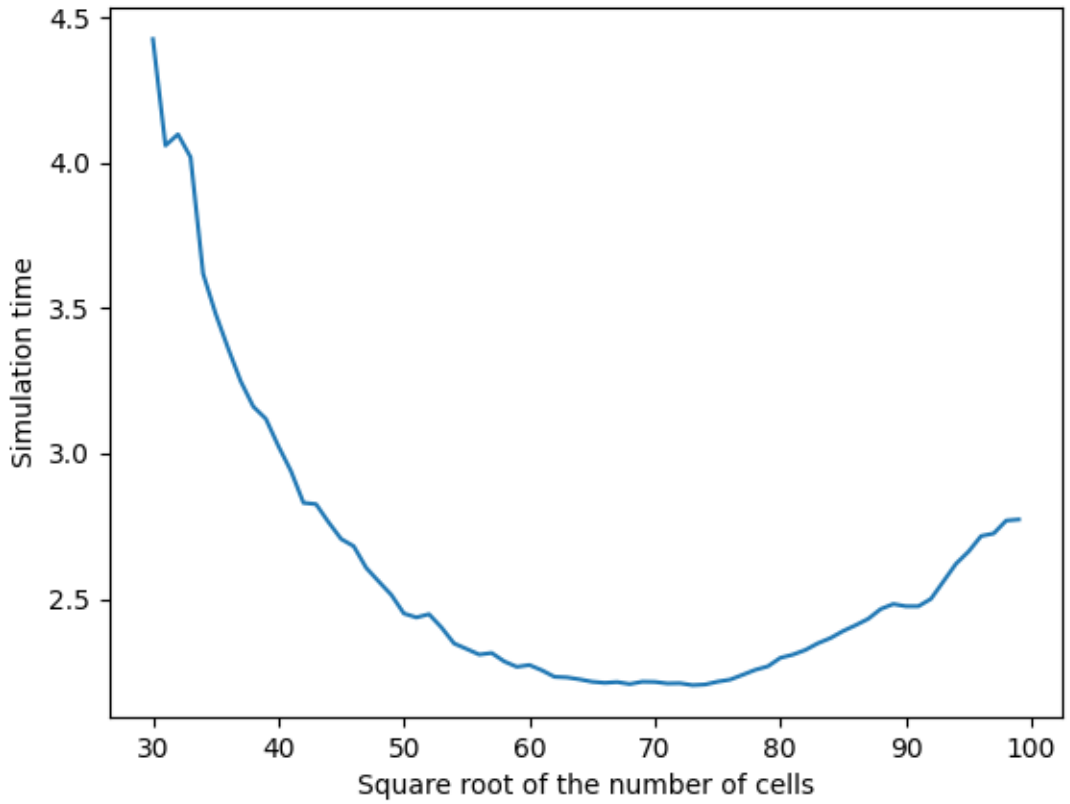
(continues on next page)

(continued from previous page)

```
Progress:98.57%
```

We plot the average simulation time as a function of the square root of the number of cells and print the best.

```
plt.plot(nb_cells,average_simu_time)
plt.xlabel("Square root of the number of cells")
plt.ylabel("Simulation time")

print("Average simulation time with block-sparse reduction: " + str(average_simu_time.
↪min()) + " seconds.")
```



Out:

```
Average simulation time with block-sparse reduction: 2.2031195163726807 seconds.
```

Note: The optimal parameters chosen initially may not stay optimal in the course of the simulation. This may be the case in particular if there is a strong concentration of particles.

How does it work

Cell size and number of cells

The cells have a rectangular shape. The length of the cells along each dimension cannot be smaller than the interaction radius R. The maximum number of cells is thus equal to:

$$n_{\max} = \prod_{k=1}^{d} \left\lfloor \frac{L_k}{R} \right\rfloor,$$

where L_k is the length of the (rectangular) domain along dimension k. This corresponds to rectangular cells with a length along dimension k equal to:

$$\varepsilon_k = \frac{L_k}{\left\lfloor \frac{L_k}{R} \right\rfloor}.$$

If the number of cells demanded n_0 exceeds $n_{\max}$, this will be the chosen value. Otherwise, we first compute the typical length:

$$\varepsilon_0 = \left(\frac{\prod_{k=1}^{d} L_k}{n_0} \right)^{1/d}$$

Then the length of the cells along dimension k is set to

$$\varepsilon_k = \frac{L_k}{\left\lfloor \frac{L_k}{\varepsilon_0} \right\rfloor}.$$

In particular, in a square domain $L_k = L$ for all k and when n_0 is a power of d, then there are exactly n_0 square cells with length $L/n_0^{1/d}$.

The block-sparse parameters

The initialisation or the method *best_blocksparse_parameters()* define three attributes which are used to speed up the computations. Given a number of cells, they are computed by the method *compute_blocksparse_parameters()*.

- `centroids` : the coordinates of the centers of the cells.
- `keep` : a square BoolTensor which indicates whether two cells are contiguous.
- `eps` : the length of the cells along each dimension.

The particles are clustered into the cells using the method *uniform_grid_separation()*.

Note: A drawback of the method is the high memory cost needed to store the boolean mask `keep`. As a consequence, unlike the classical Verlet list method, the optimal number of cells is often **not** the maximum one. In the examples presented in this documentation, the optimal number of cells is always smaller than 10^4.

Total running time of the script: (8 minutes 9.001 seconds)

4.4.5 Tutorial 05: Kernels and averages

Simulating swarming models requires expensive mean-field convolution operations of the form:

$$J^i = \frac{1}{N}\sum_{j=1}^{N} K(|X^j - X^i|)U^j,$$

for $1 \leq i \leq N$, where $(X^i)_{1\leq i\leq N}$ are the positions of the particles, $(U^j)_{1\leq j\leq N}$ are given vectors and K is an **observation kernel**. Typically, $K(|X^i - X^j|)$ is equal to 1 if X^i and X^j are at distance smaller than a fixed interaction distance and 0 otherwise. Other kernels are defined in the module *sisyphe.kernels*. Below, we show a simple application case.

Linear local averages

First, some standard imports…

```
import time
import math
import torch
from matplotlib import pyplot as plt

use_cuda = torch.cuda.is_available()
dtype = torch.cuda.FloatTensor if use_cuda else torch.FloatTensor
```

Let the N particles be uniformly scattered in a box of size L with interaction radius R.

```
N = 100000
L = 1.
R = .15

pos = L*torch.rand((N,2)).type(dtype)
```

We can also assume that the particles have a bounded cone of vision around an axis (defined by a unit vector). The default behaviour is a full vision angle equal to 2π in which case the axis is a `None` object. Here we take a cone of vision with angle $\pi/2$ around an axis which is sampled uniformly. For the *sisyphe.particles.KineticParticles*, the default axis is the velocity.

```
angle = math.pi/2
axis = torch.randn(N,2).type(dtype)
axis = axis/torch.norm(axis,dim=1).reshape((N,1))
```

Let us create an instance of a particle system with these parameters.

```
from sisyphe.particles import Particles

particles = Particles(
    pos = pos,
    interaction_radius = R,
    box_size = L,
    vision_angle = angle,
    axis = axis)
```

Note: By default, the system and the operations below are defined with periodic boundary conditions.

As a simple application, we can compute the number of neighbours of each particle and print the number of neighbours of the first particle. This operation is already implemented in the method *number_of_neighbours()*. It simply corresponds to the average:

$$N^i_{\text{neigh}} = \sum_{j=1}^{N} K(|X^j - X^i|).$$

```
Nneigh = particles.number_of_neighbours()

Nneigh0 = int(Nneigh[0].item())

print("The first particle sees " + str(Nneigh0) + " other particles.")
```

Out:

```
The first particle sees 1754 other particles.
```

For custom objects, the mean-field average can be computed using the method *linear_local_average()*. As an example, let us compute the center of mass of the neighbours of each particle. First we define the quantity U that we want to average. Here, since we are working on a torus, there are two: the sine and the cosine of the spatial coordinates.

```
cos_pos = torch.cos((2*math.pi / L) * particles.pos)
sin_pos = torch.sin((2*math.pi / L) * particles.pos)
```

Then we compute the two mean field averages, i.e. the standard convolution over the N particles. The center of mass along each dimension is the argument of the complex number whose coordinates are the average cosine and sine.

```
average_cos, average_sin = particles.linear_local_average(cos_pos, sin_pos)
center_x = torch.atan2(average_sin[:,0], average_cos[:,0])
center_x = (L / (2*math.pi)) * torch.remainder(center_x, 2*math.pi)
center_y = torch.atan2(average_sin[:,1], average_cos[:,1])
center_y = (L / (2*math.pi)) * torch.remainder(center_y, 2*math.pi)

center_of_mass = torch.cat((center_x.reshape((N,1)), center_y.reshape((N,1))),
                           dim=1)
```

Out:

```
[pyKeOps] Compiling libKeOpstorch3dcd0c2195 in /data/and18/.cache/pykeops-1.5-cpython-38:
       formula: Sum_Reduction(((Step((Var(5,1,2) - Sum(Square((((Var(0,2,1) - Var(1,2,
↪0)) + (Step(((Minus(Var(2,2,2)) / Var(3,1,2)) - (Var(0,2,1) - Var(1,2,0)))) * Var(2,2,
↪2))) - (Step(((Var(0,2,1) - Var(1,2,0)) - (Var(2,2,2) / Var(4,1,2)))) * Var(2,2,
↪2)))))))) * Step((Var(7,1,2) + (Sum(((((Var(0,2,1) - Var(1,2,0)) + (Step(((Minus(Var(2,
↪2,2)) / Var(3,1,2)) - (Var(0,2,1) - Var(1,2,0)))) * Var(2,2,2))) - (Step(((Var(0,2,1) -
↪ Var(1,2,0)) - (Var(2,2,2) / Var(4,1,2)))) * Var(2,2,2))) * Var(6,2,0))) /␣
↪Sqrt(Sum(Square((((Var(0,2,1) - Var(1,2,0)) + (Step(((Minus(Var(2,2,2)) / Var(3,1,2)) -
↪ (Var(0,2,1) - Var(1,2,0)))) * Var(2,2,2))) - (Step(((Var(0,2,1) - Var(1,2,0)) -␣
↪(Var(2,2,2) / Var(4,1,2)))) * Var(2,2,2)))))))))) * Var(8,4,1)),0)
       aliases: Var(0,2,1); Var(1,2,0); Var(2,2,2); Var(3,1,2); Var(4,1,2); Var(5,1,2);␣
↪Var(6,2,0); Var(7,1,2); Var(8,4,1);
       dtype  : float32
...
Done.
```

In the method *linear_local_average()*, the default observation kernel is a `LazyTensor` of size (N, N) whose (i, j) component is equal to 1 when particle j belongs to the cone of vision of particle i and 0 otherwise. To retrieve the indexes of the particles which belong to the cone of vision of the first particle, we can use the K-nearest-neighbours reduction provided by the KeOps library.

```
from sisyphe.kernels import lazy_interaction_kernel

interaction_kernel = lazy_interaction_kernel(
    particles.pos,
    particles.pos,
    particles.R,
    particles.L,
    boundary_conditions = particles.bc,
    vision_angle = particles.angle,
    axis = particles.axis)

K_ij = 1. - interaction_kernel

neigh0 = K_ij.argKmin(Nneigh0, dim=1)[0]

print("The indexes of the neighbours of the first particles are: ")
print(neigh0)
```

Out:

```
[pyKeOps] Compiling libKeOpstorchaf7a666555 in /data/and18/.cache/pykeops-1.5-cpython-38:
       formula: ArgKMin_Reduction((Var(8,1,2) - (Step((Var(5,1,2) - Sum(Square((((Var(0,
↪2,1) - Var(1,2,0)) + (Step(((Minus(Var(2,2,2)) / Var(3,1,2)) - (Var(0,2,1) - Var(1,2,
↪0)))) * Var(2,2,2))) - (Step(((Var(0,2,1) - Var(1,2,0)) - (Var(2,2,2) / Var(4,1,2))))␣
↪* Var(2,2,2)))))) * Step((Var(7,1,2) + (Sum(((((Var(0,2,1) - Var(1,2,0)) +␣
↪(Step(((Minus(Var(2,2,2)) / Var(3,1,2)) - (Var(0,2,1) - Var(1,2,0)))) * Var(2,2,2))) -␣
↪(Step(((Var(0,2,1) - Var(1,2,0)) - (Var(2,2,2) / Var(4,1,2)))) * Var(2,2,2))) * Var(6,
↪2,0))) / Sqrt(Sum(Square((((Var(0,2,1) - Var(1,2,0)) + (Step(((Minus(Var(2,2,2)) /␣
↪Var(3,1,2)) - (Var(0,2,1) - Var(1,2,0)))) * Var(2,2,2))) - (Step(((Var(0,2,1) - Var(1,
↪2,0)) - (Var(2,2,2) / Var(4,1,2)))) * Var(2,2,2))))))))))),1754,0)
       aliases: Var(0,2,1); Var(1,2,0); Var(2,2,2); Var(3,1,2); Var(4,1,2); Var(5,1,2);␣
↪Var(6,2,0); Var(7,1,2); Var(8,1,2);
       dtype  : float32
...
Done.
The indexes of the neighbours of the first particles are:
tensor([    0,    29,   130,  ..., 99811, 99868, 99982], device='cuda:0')
```

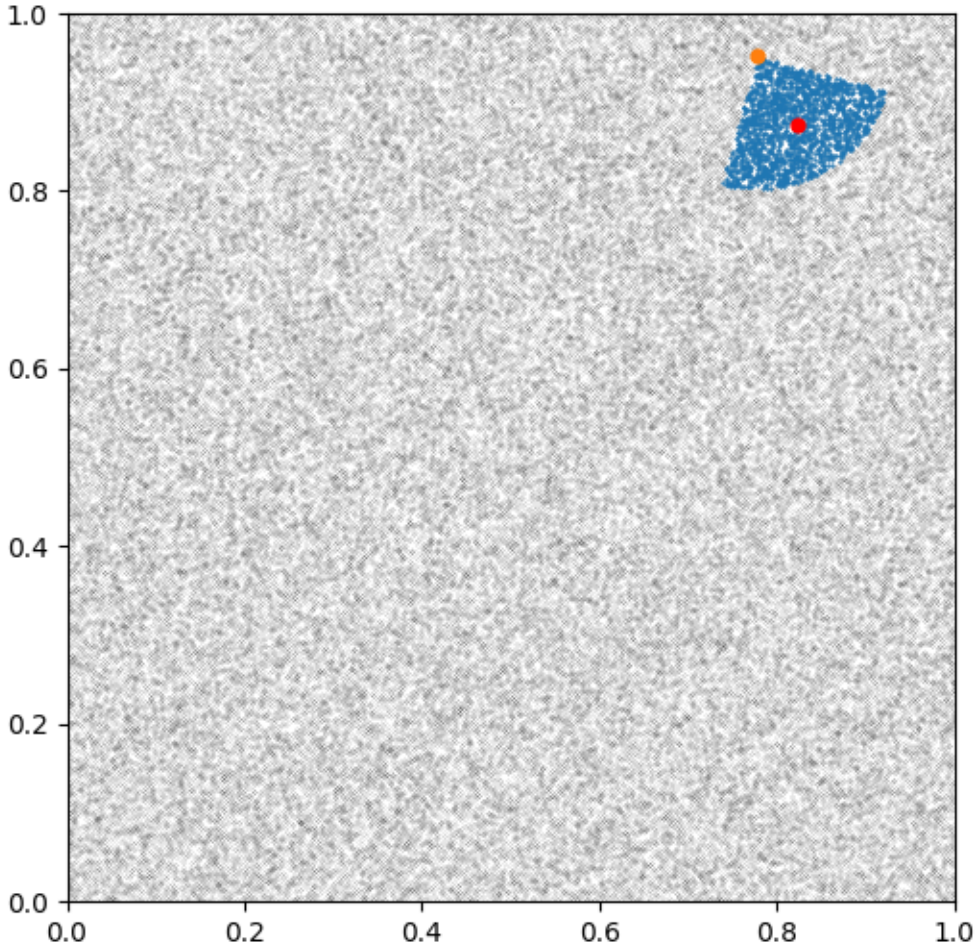
Finally, a fancy display of what we have computed. We plot the full particle system in black, the first particle in orange, its neighbours in blue and the center of mass of the neighbours in red.

```
xall = particles.pos[:,0].cpu().numpy()
yall = particles.pos[:,1].cpu().numpy()

x = particles.pos[neigh0,0].cpu().numpy()
y = particles.pos[neigh0,1].cpu().numpy()

x0 = particles.pos[0,0].item()
y0 = particles.pos[0,1].item()
```

(continues on next page)

(continued from previous page)

```
xc = center_of_mass[0,0].item()
yc = center_of_mass[0,1].item()


fig, ax = plt.subplots(figsize=(6,6))
ax.scatter(xall, yall, s=.003, c='black')
ax.scatter(x, y, s=.3)
ax.scatter(x0, y0, s=24)
ax.scatter(xc, yc, s=24, c='red')
ax.axis([0, L, 0, L])
ax.set_aspect("equal")
```



Nonlinear averages

In some cases, we need to compute a **nonlinear average** of the form

$$J^i = \frac{1}{N}\sum_{j=1}^{N} K(|X^j - X^i|)b(U^i, V^j),$$

where $(U^i)_{1\leq i\leq N}$ and $(V^j)_{1\leq j\leq N}$ are given vectors and b is a given function. When the **binary formula** b can be written as a `LazyTensor`, this can be computed with the method *nonlinear_local_average()*.

For instance, let us compute the local mean square distance:

$$J^i = \frac{\sum_{j=1}^{N} K(|X^j - X^i|)|X^j - X^i|^2}{\sum_{j=1}^{N} K(|X^j - X^i|)}.$$

In this case, we can use the function *sisyphe.kernels.lazy_xy_matrix()* to define a custom binary formula. Given two vectors $X = (X^i)_{1\leq i\leq M}$ and $Y = (Y^j)_{1\leq j\leq N}$, respectively of sizes (M, d) and (N, d), the XY matrix is a (M, N, d) LazyTensor whose $(i, j, :)$ component is the vector $Y^j - X^i$.

```
from sisyphe.kernels import lazy_xy_matrix

def b(x,y):
    K_ij = lazy_xy_matrix(x,y,particles.L)
    return (K_ij ** 2).sum(-1)

x = particles.pos
y = particles.pos
mean_square_dist = N/Nneigh.reshape((N,1)) * particles.nonlinear_local_average(b,x,y)
```

Since the particles are uniformly scattered in the box, the theoretical value is

$$MSD_0 = \frac{\int_0^R \int_0^{\pi/2} r^3 \mathrm{d}r\mathrm{d}\theta}{\int_0^R \int_0^{\pi/2} r\mathrm{d}r\mathrm{d}\theta} = \frac{R^2}{2}$$

```
print("Theoretical value: " + str(R**2/2))
print("Experimental value: " + str(mean_square_dist[0].item()))
```

Out:

```
Theoretical value: 0.01125
Experimental value: 0.01103969756513834
```

Total running time of the script: (2 minutes 49.544 seconds)

4.5 Examples

Some examples for various models.

4.5.1 Bands

The classical Vicsek model in a square periodic domain is known to produce band-like structures in a very dilute regime. These structures also appears in a mean-field regime. To showcase the efficiency of the SiSyPHE library, we simulate a mean-field *Vicsek* model with the target *max_kappa()* and 10^6 particles.

First of all, some standard imports.

```
import os
import sys
import time
import math
import torch
import numpy as np
from matplotlib import pyplot as plt

use_cuda = torch.cuda.is_available()
dtype = torch.cuda.FloatTensor if use_cuda else torch.FloatTensor
```

Set the parameters and create an instance of the Vicsek model.

```
import sisyphe.models as models

N = 1000000
L = 1.
dt = .01

nu = 3
sigma = 1.
kappa = nu/sigma

R = .01
c = .1

pos = L*torch.rand((N,2)).type(dtype)
vel = torch.randn(N,2).type(dtype)
vel = vel/torch.norm(vel,dim=1).reshape((N,1))

simu=models.Vicsek(pos=pos,vel=vel,
              v=c,
              sigma=sigma,nu=nu,
              interaction_radius=R,
              box_size=L,
              boundary_conditions='periodic',
              variant = {"name" : "max_kappa", "parameters" : {"kappa_max" : 10.}},
              options = {},
              numerical_scheme='projection',
              dt=dt,
              block_sparse_reduction=True)
```

Check that we are in a mean field regime…

```
Nneigh = simu.number_of_neighbours()
```

(continues on next page)

(continued from previous page)

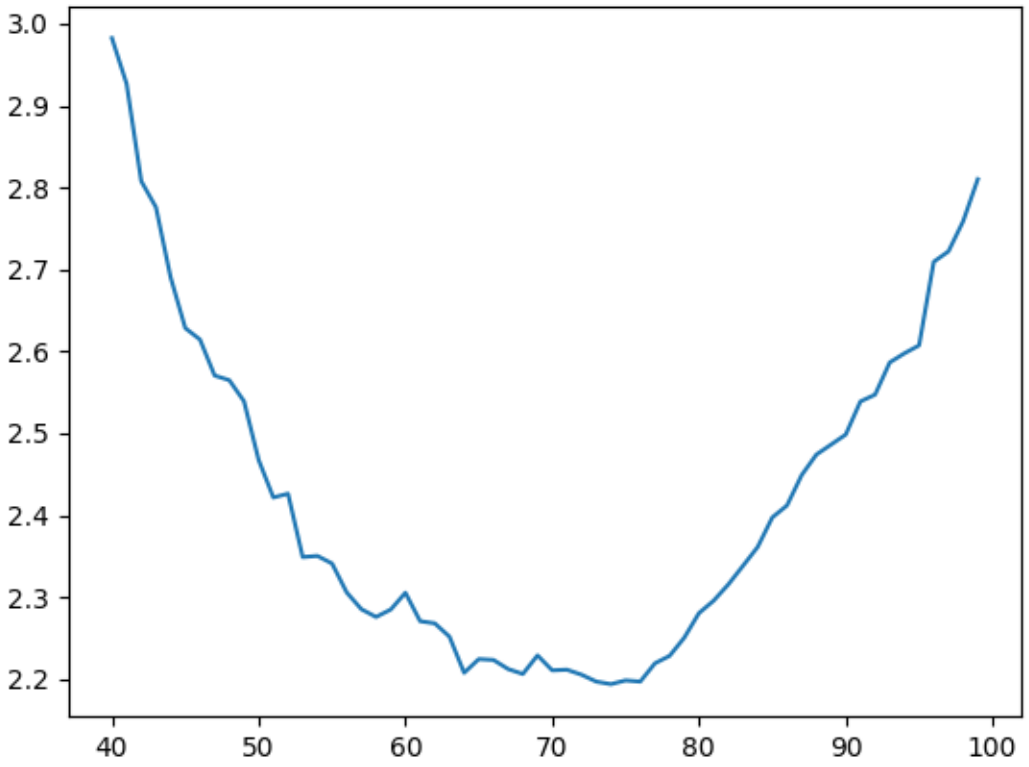
```
print("The most isolated particle has " + str(Nneigh.min().item()) + " neighbours.")
print("The least isolated particle has " + str(Nneigh.max().item()) + " neighbours.")
```

Out:

```
The most isolated particle has 232.0 neighbours.
The least isolated particle has 404.0 neighbours.
```

Set the block sparse parameters to their optimal value.

```
fastest, nb_cells, average_simu_time, simulation_time = simu.best_blocksparse_
↪parameters(40,100)

plt.plot(nb_cells,average_simu_time)
plt.show()
```



Out:

```
Progress:0.0%
Progress:1.67%
Progress:3.33%
Progress:5.0%
Progress:6.67%
```

(continues on next page)

(continued from previous page)

```
Progress:8.33%
Progress:10.0%
Progress:11.67%
Progress:13.33%
Progress:15.0%
Progress:16.67%
Progress:18.33%
Progress:20.0%
Progress:21.67%
Progress:23.33%
Progress:25.0%
Progress:26.67%
Progress:28.33%
Progress:30.0%
Progress:31.67%
Progress:33.33%
Progress:35.0%
Progress:36.67%
Progress:38.33%
Progress:40.0%
Progress:41.67%
Progress:43.33%
Progress:45.0%
Progress:46.67%
Progress:48.33%
Progress:50.0%
Progress:51.67%
Progress:53.33%
Progress:55.0%
Progress:56.67%
Progress:58.33%
Progress:60.0%
Progress:61.67%
Progress:63.33%
Progress:65.0%
Progress:66.67%
Progress:68.33%
Progress:70.0%
Progress:71.67%
Progress:73.33%
Progress:75.0%
Progress:76.67%
Progress:78.33%
Progress:80.0%
Progress:81.67%
Progress:83.33%
Progress:85.0%
Progress:86.67%
Progress:88.33%
Progress:90.0%
Progress:91.67%
Progress:93.33%
```

(continues on next page)

(continued from previous page)

```
Progress:95.0%
Progress:96.67%
Progress:98.33%
```

Create the function which compute the center of mass of the system (on the torus).

```
def center_of_mass(particles):
    cos_pos = torch.cos((2*math.pi / L) * particles.pos)
    sin_pos = torch.sin((2*math.pi / L) * particles.pos)
    average_cos = cos_pos.sum(0)
    average_sin = sin_pos.sum(0)
    center = torch.atan2(average_sin, average_cos)
    center = (L / (2*math.pi)) * torch.remainder(center, 2*math.pi)
    return center
```

Let us save the positions and velocities of 100k particles and the center of mass of the system during 300 units of time.

```
from sisyphe.display import save

frames = [50., 100., 300.]

s = time.time()
data = save(simu,frames,["pos", "vel"],[center_of_mass], Nsaved=100000, save_file=False)
e = time.time()
```

Out:

```
Progress:0%
Progress:1%
Progress:2%
Progress:3%
Progress:4%
Progress:5%
Progress:6%
Progress:7%
Progress:8%
Progress:9%
Progress:10%
Progress:11%
Progress:12%
Progress:13%
Progress:14%
Progress:15%
Progress:16%
Progress:17%
Progress:18%
Progress:19%
Progress:20%
Progress:21%
Progress:22%
Progress:23%
Progress:24%
Progress:25%
```

(continues on next page)

(continued from previous page)

```
Progress:26%
Progress:27%
Progress:28%
Progress:29%
Progress:30%
Progress:31%
Progress:32%
Progress:33%
Progress:34%
Progress:35%
Progress:36%
Progress:37%
Progress:38%
Progress:39%
Progress:40%
Progress:41%
Progress:42%
Progress:43%
Progress:44%
Progress:45%
Progress:46%
Progress:47%
Progress:48%
Progress:49%
Progress:50%
Progress:51%
Progress:52%
Progress:53%
Progress:54%
Progress:55%
Progress:56%
Progress:57%
Progress:58%
Progress:59%
Progress:60%
Progress:61%
Progress:62%
Progress:63%
Progress:64%
Progress:65%
Progress:66%
Progress:67%
Progress:68%
Progress:69%
Progress:70%
Progress:71%
Progress:72%
Progress:73%
Progress:74%
Progress:75%
Progress:76%
Progress:77%
```

(continues on next page)

(continued from previous page)

```
Progress:78%
Progress:79%
Progress:80%
Progress:81%
Progress:82%
Progress:83%
Progress:84%
Progress:85%
Progress:86%
Progress:87%
Progress:88%
Progress:89%
Progress:90%
Progress:91%
Progress:92%
Progress:93%
Progress:94%
Progress:95%
Progress:96%
Progress:97%
Progress:98%
Progress:99%
Progress:100%
```

Print the total simulation time and the average time per iteration.

```
print('Total time: '+str(e-s)+' seconds')
print('Average time per iteration: '+str((e-s)/simu.iteration)+' seconds')
```

Out:

```
Total time: 1619.642599105835 seconds
Average time per iteration: 0.05398628709395804 seconds
```

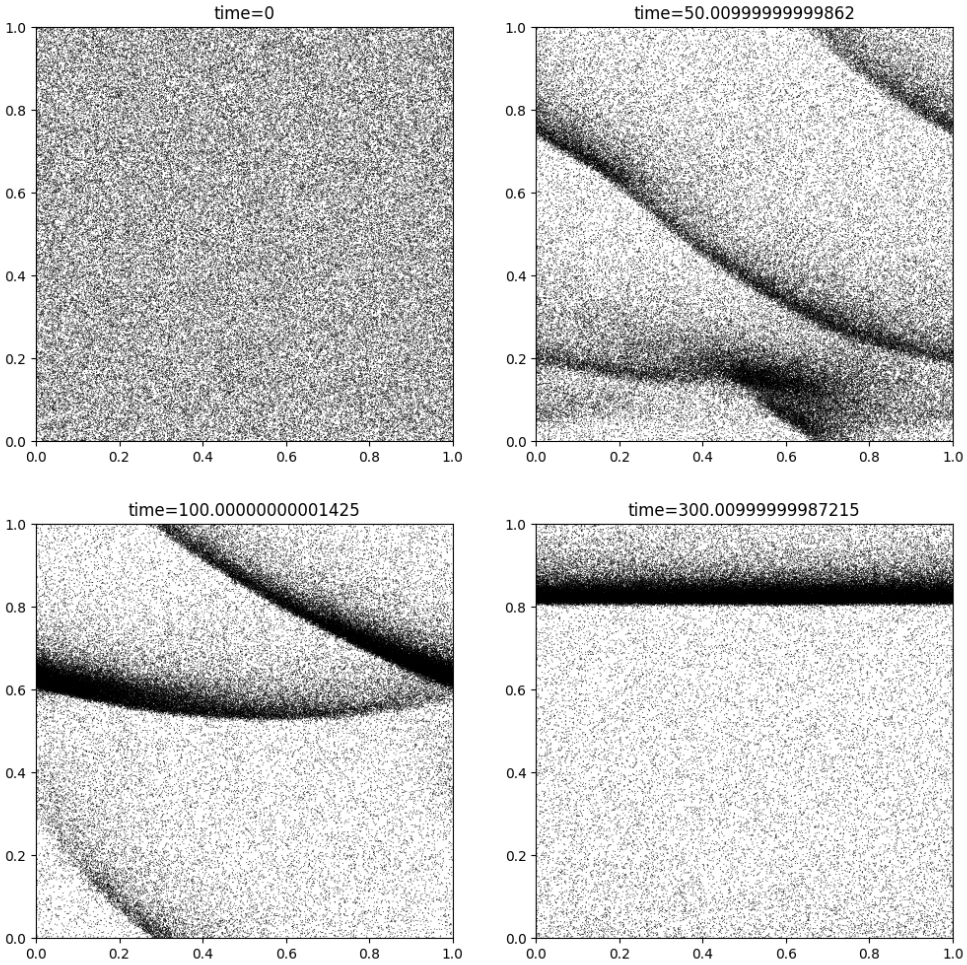
At the end of the simulation, we plot the particles and the evolution of the center of mass.

```
# sphinx_gallery_thumbnail_number = 2
f = plt.figure(0, figsize=(12, 12))
for frame in range(len(data["frames"])):
    x = data["pos"][frame][:,0].cpu()
    y = data["pos"][frame][:,1].cpu()
    u = data["vel"][frame][:,0].cpu()
    v = data["vel"][frame][:,1].cpu()
    ax = f.add_subplot(2,2,frame+1)
    plt.quiver(x,y,u,v)
    ax.set_xlim(xmin=0, xmax=simu.L[0].cpu())
    ax.set_ylim(ymin=0, ymax=simu.L[1].cpu())
    ax.set_title("time="+str(data["frames"][frame]))

center = data["center_of_mass"]

center_x = []
center_y = []
```

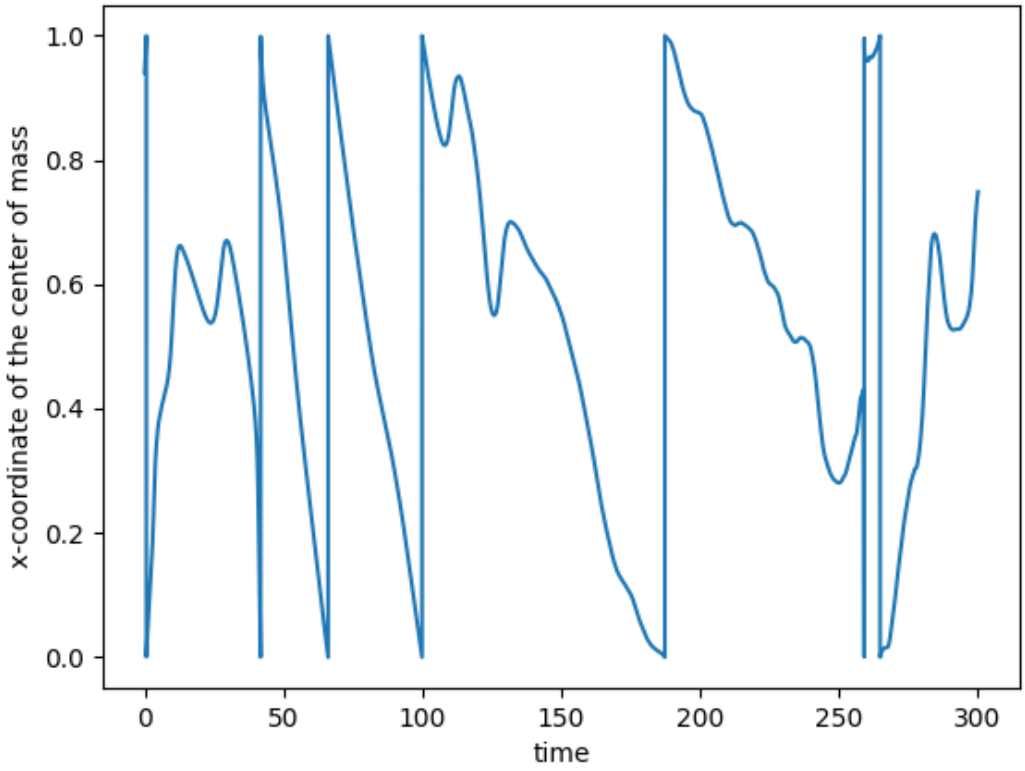
(continues on next page)

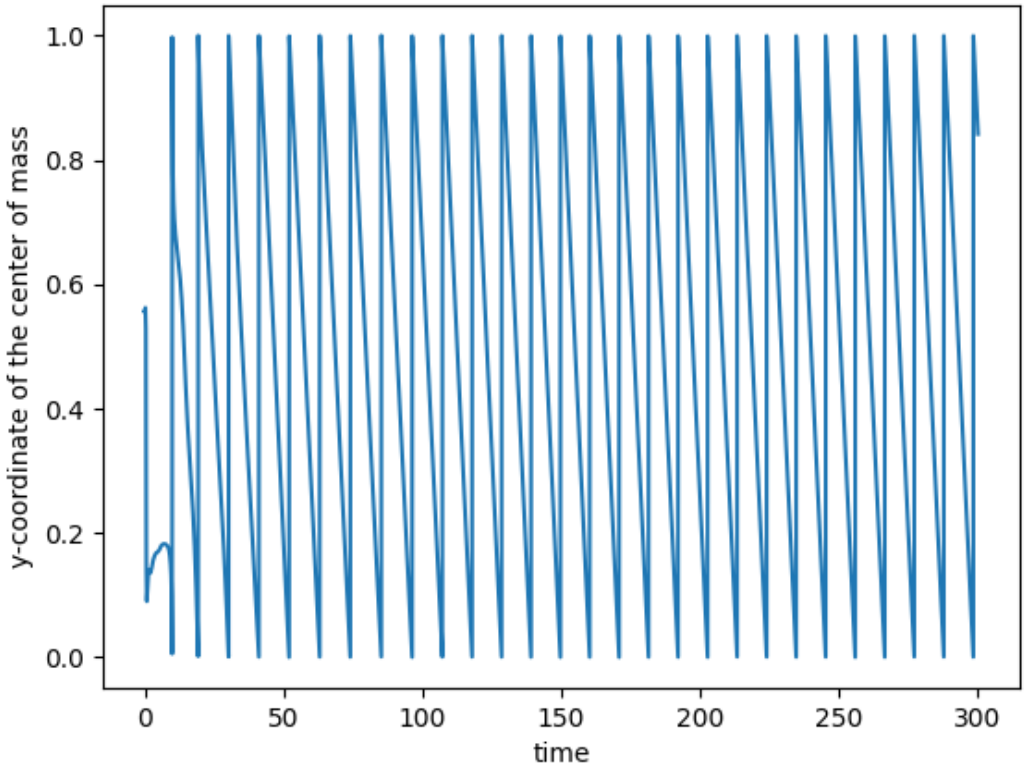
(continued from previous page)

```
for c in center:
    center_x.append(c[0])
    center_y.append(c[1])

f = plt.figure(1)
plt.plot(data["time"],center_x)
plt.ylabel("x-coordinate of the center of mass")
plt.xlabel("time")


f = plt.figure(2)
plt.plot(data["time"],center_y)
plt.ylabel("y-coordinate of the center of mass")
plt.xlabel("time")
plt.show()
```


time=0
time=50.00999999999862
time=100.00000000001425
time=300.00999999987215

-


1.0
0.8
0.6
0.4
0.2
0.0
x-coordinate of the center of mass
0
50
100
150
200
250
300
time



-

We are still in a mean-field regime.

```
Nneigh = simu.number_of_neighbours()

print("The most isolated particle has " + str(Nneigh.min().item()) + " neighbours.")
print("The least isolated particle has " + str(Nneigh.max().item()) + " neighbours.")
```

Out:

```
The most isolated particle has 34.0 neighbours.
The least isolated particle has 6125.0 neighbours.
```

Total running time of the script: (29 minutes 34.952 seconds)

4.5.2 Body-oriented mill

This model is introduced in

P. Degond, A. Diez, M. Na, Bulk topological states in a new collective dynamics model, arXiv:2101.10864, 2021

First of all, some standard imports.

```
import os
import sys
import time
```

(continues on next page)

(continued from previous page)

```
import math
import torch
import numpy as np
from matplotlib import pyplot as plt
import sisyphe.models as models
from sisyphe.display import save

use_cuda = torch.cuda.is_available()
dtype = torch.cuda.FloatTensor if use_cuda else torch.FloatTensor
```

Body-oriented particles with initial perpendicular twist

The system is composed of body-oriented particles which are initially uniformly scattered in a periodic box but their body-orientations are "twisted". The body orientation of a particle at position (x, y, z) is initially:

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos(2\pi z) & -\sin(2\pi z) \\ 0 & \sin(2\pi z) & \cos(2\pi z) \end{pmatrix}.$$

```
from sisyphe.initial import cyclotron_twist_z

N = 1500000
L = 1
R = .025
nu = 40
c = 1
kappa = 10

pos, bo = cyclotron_twist_z(N,L,1,kappa,dtype)

simu = models.BOAsynchronousVicsek(pos=pos,bo=bo,
                  v=c,
                  jump_rate=nu,kappa=kappa,
                  interaction_radius=R,
                  box_size=L,
                  boundary_conditions='periodic',
                  variant = {"name" : "normalised", "parameters" : {}},
                  options = {},
                  sampling_method='vonmises',
                  block_sparse_reduction=True,
                  number_of_cells=15**3)
```

Run the simulation over 5 units of time and save the azimuthal angle of the mean direction of motion defined by:

$$\varphi = \arg(\Omega^1 + i\Omega^2) \in [0, 2\pi],$$

where $\Omega = (\Omega^1, \Omega^2, \Omega^3)$ is the mean direction of motion of the particles with velocities $(\Omega_i)_{1 \leq i \leq N}$:

$$\Omega := \frac{\sum_{i=1}^N \Omega_i}{|\sum_{i=1}^N \Omega_i|}$$

```
frames = [5.]

s = time.time()
data = save(simu,frames,[],["phi"],save_file=False)
e = time.time()
```

Out:

```
Progress:0%
Progress:1%
Progress:2%
Progress:3%
Progress:4%
Progress:5%
Progress:6%
Progress:7%
Progress:8%
Progress:9%
Progress:10%
Progress:11%
Progress:12%
Progress:13%
Progress:14%
Progress:15%
Progress:16%
Progress:17%
Progress:18%
Progress:19%
Progress:20%
Progress:21%
Progress:22%
Progress:23%
Progress:24%
Progress:25%
Progress:26%
Progress:27%
Progress:28%
Progress:29%
Progress:30%
Progress:31%
Progress:32%
Progress:33%
Progress:34%
Progress:35%
Progress:36%
Progress:37%
Progress:38%
Progress:39%
Progress:40%
Progress:41%
Progress:42%
Progress:43%
Progress:44%
```

(continues on next page)

(continued from previous page)

```
Progress:45%
Progress:46%
Progress:47%
Progress:48%
Progress:49%
Progress:50%
Progress:51%
Progress:52%
Progress:53%
Progress:54%
Progress:55%
Progress:56%
Progress:57%
Progress:58%
Progress:59%
Progress:60%
Progress:61%
Progress:62%
Progress:63%
Progress:64%
Progress:65%
Progress:66%
Progress:67%
Progress:68%
Progress:69%
Progress:70%
Progress:71%
Progress:72%
Progress:73%
Progress:74%
Progress:75%
Progress:76%
Progress:77%
Progress:78%
Progress:79%
Progress:80%
Progress:81%
Progress:82%
Progress:83%
Progress:84%
Progress:85%
Progress:86%
Progress:87%
Progress:88%
Progress:89%
Progress:90%
Progress:91%
Progress:92%
Progress:93%
Progress:94%
Progress:95%
Progress:96%
```

(continues on next page)

(continued from previous page)

```
Progress:97%
Progress:98%
Progress:99%
Progress:100%
```

Print the total simulation time and the average time per iteration.

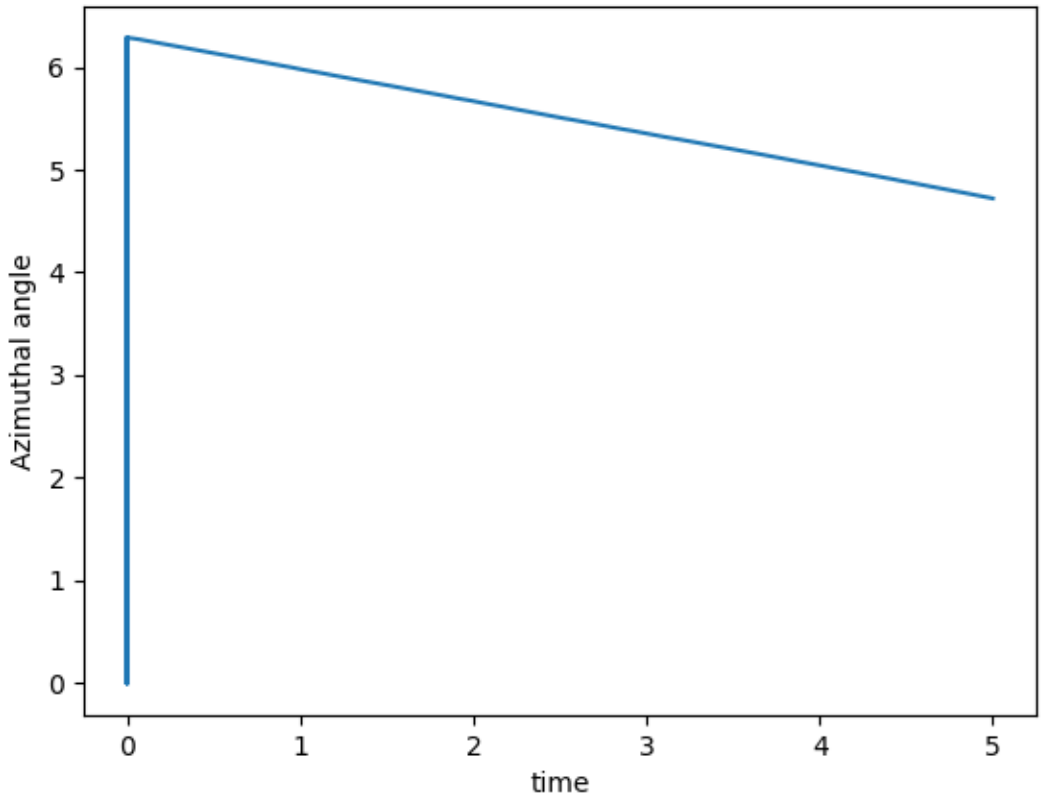
```
print('Total time: '+str(e-s)+' seconds')
print('Average time per iteration: '+str((e-s)/simu.iteration)+' seconds')
```

Out:

```
Total time: 1655.7933239936829 seconds
Average time per iteration: 0.08278966619968414 seconds
```

Plot the azimuthal angle φ.

```
plt.plot(data["time"],data["phi"])
plt.xlabel("time")
plt.ylabel("Azimuthal angle")
```



Out:

```
Text(55.847222222222214, 0.5, 'Azimuthal angle')
```

Total running time of the script: (27 minutes 35.923 seconds)

4.5.3 Mills

Examples of milling behaviours.

First of all, some standard imports.

```
import os
import sys
import time
import math
import torch
import numpy as np
from matplotlib import pyplot as plt
import sisyphe.models as models
from sisyphe.display import display_kinetic_particles

use_cuda = torch.cuda.is_available()
dtype = torch.cuda.FloatTensor if use_cuda else torch.FloatTensor
```

Milling in the D'Orsogna et al. model

Let us create an instance of the attraction-repulsion model introduced in

M. R. D'Orsogna, Y. L. Chuang, A. L. Bertozzi, L. S. Chayes, Self-Propelled Particles with Soft-Core Interactions: Patterns, Stability, and Collapse, *Phys. Rev. Lett.*, Vol. 96, No. 10 (2006).

The particle system satisfies the ODE:

$$\frac{\mathrm{d}X_t^i}{\mathrm{d}t} = V_t^i$$

$$\frac{\mathrm{d}V_t^i}{\mathrm{d}t} = (\alpha - \beta|V_t^i|^2)V_t^i - \frac{m}{N}\nabla_{x^i}\sum_{j\neq i} U(|X_t^i - X_t^j|)$$

where U is the Morse potential

$$U(r) := -C_a \mathrm{e}^{-r/\ell_a} + C_r \mathrm{e}^{-r/\ell_r}$$

```
N = 10000
mass = 1000.
L = 10.

Ca = .5
la = 2.
Cr = 1.
lr = .5

alpha = 1.6
```

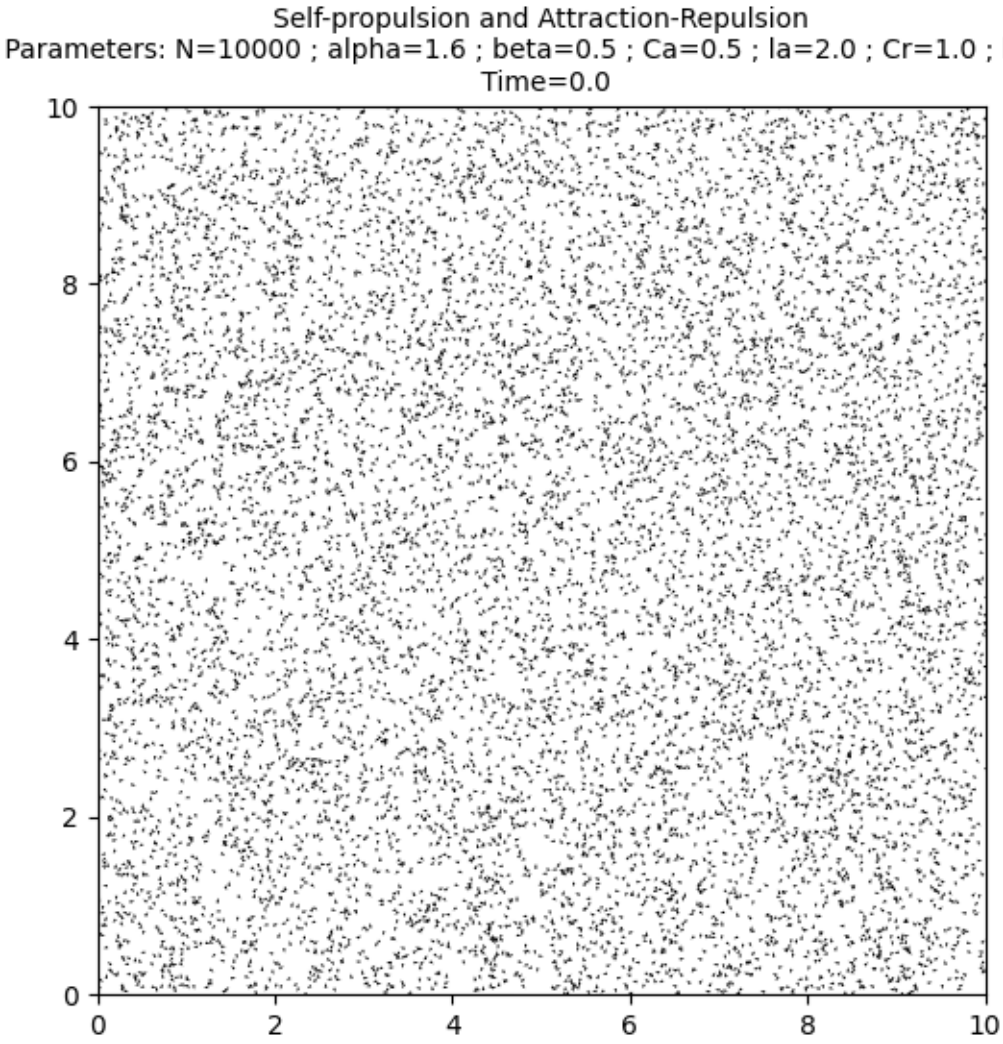
(continues on next page)

(continued from previous page)

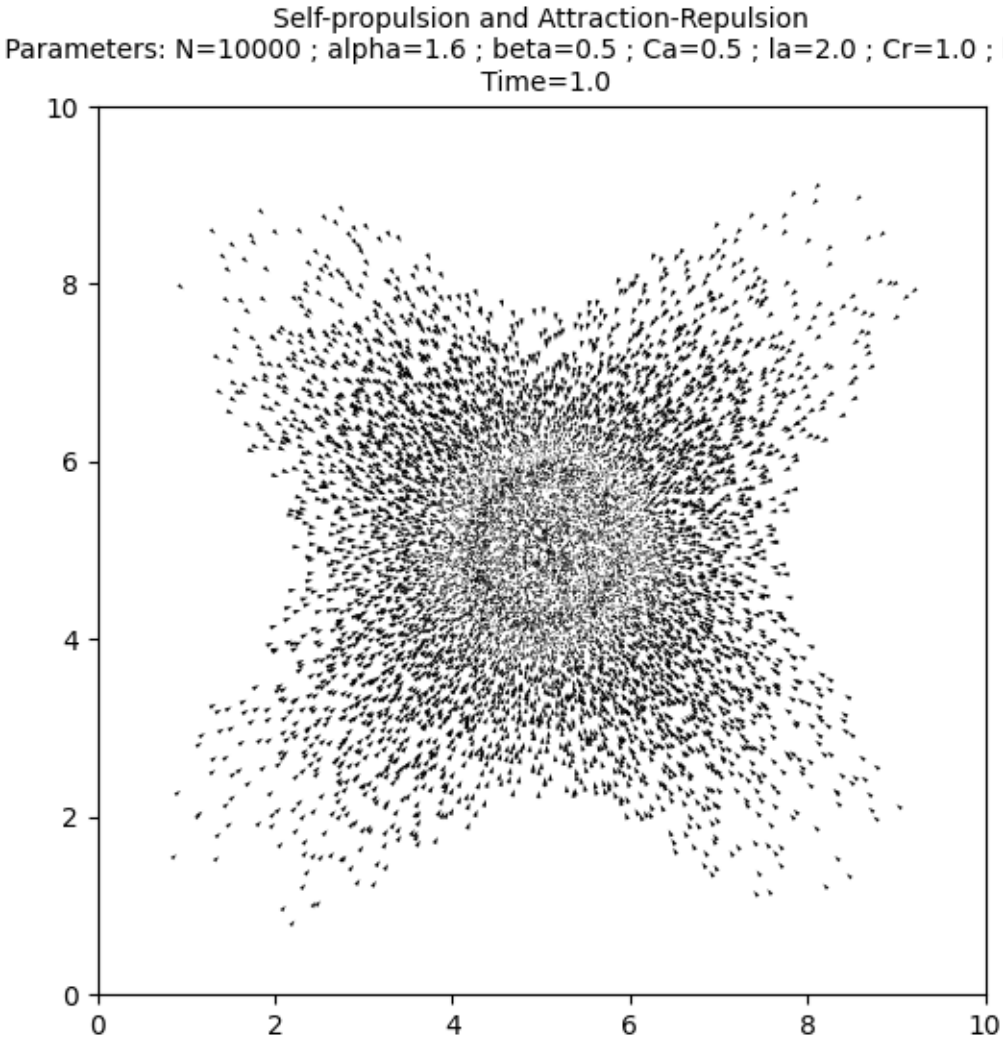
```
beta = .5
v0 = math.sqrt(alpha/beta)

pos = L*torch.rand((N,2)).type(dtype)
vel = torch.randn(N,2).type(dtype)
vel = vel/torch.norm(vel,dim=1).reshape((N,1))
vel = v0*vel

dt = .01

simu = models.AttractionRepulsion(pos=pos,
                  vel=vel,
                  interaction_radius=math.sqrt(mass),
                  box_size=L,
                  propulsion = alpha,
                  friction = beta,
                  Ca = Ca,
                  la = la,
                  Cr = Cr,
                  lr = lr,
                  dt=dt,
                  p=1,
                  isaverage=True)
```

Run the simulation over 100 units of time and plot 10 frames. The ODE system is solved using the Runge-Kutta 4 numerical scheme.

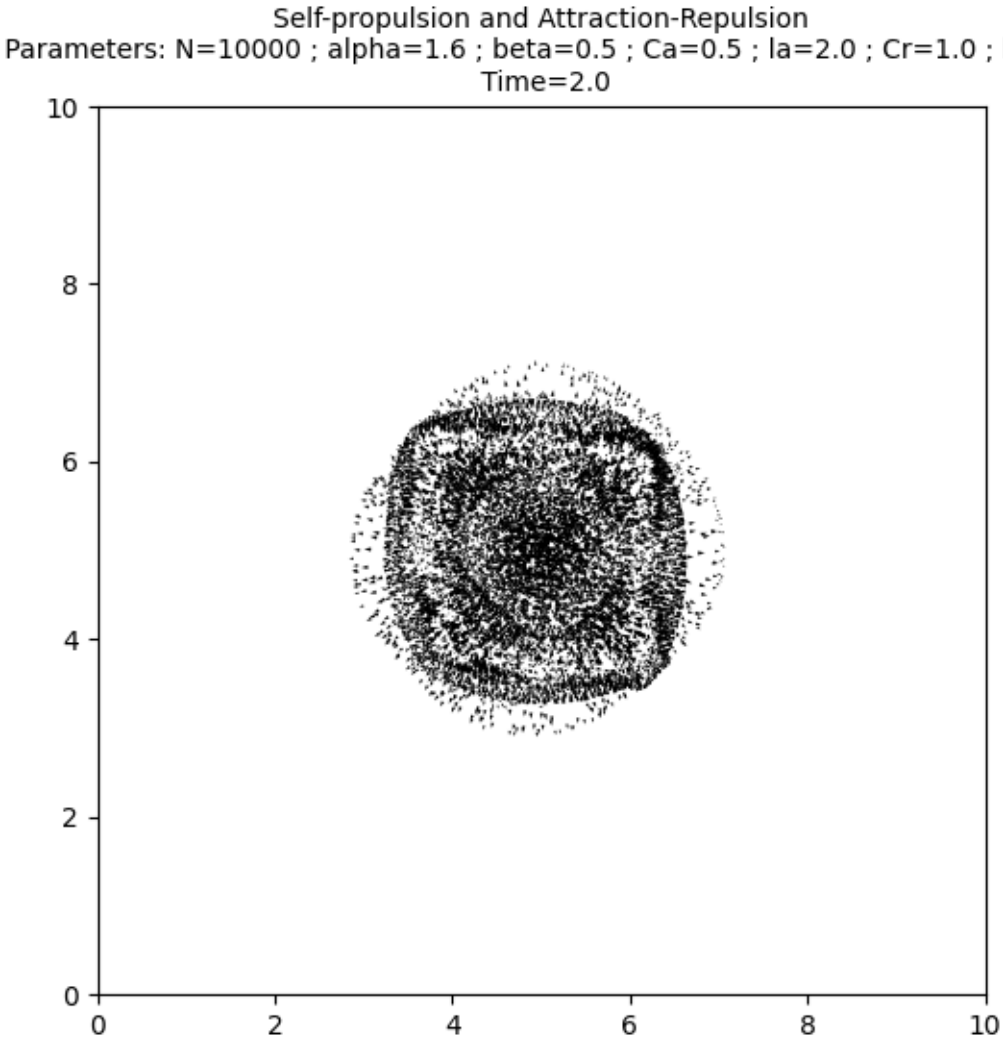
```
frames = [0,1,2,3,4,5,10,40,70,100]

s = time.time()
it, op = display_kinetic_particles(simu,frames)
e = time.time()
```

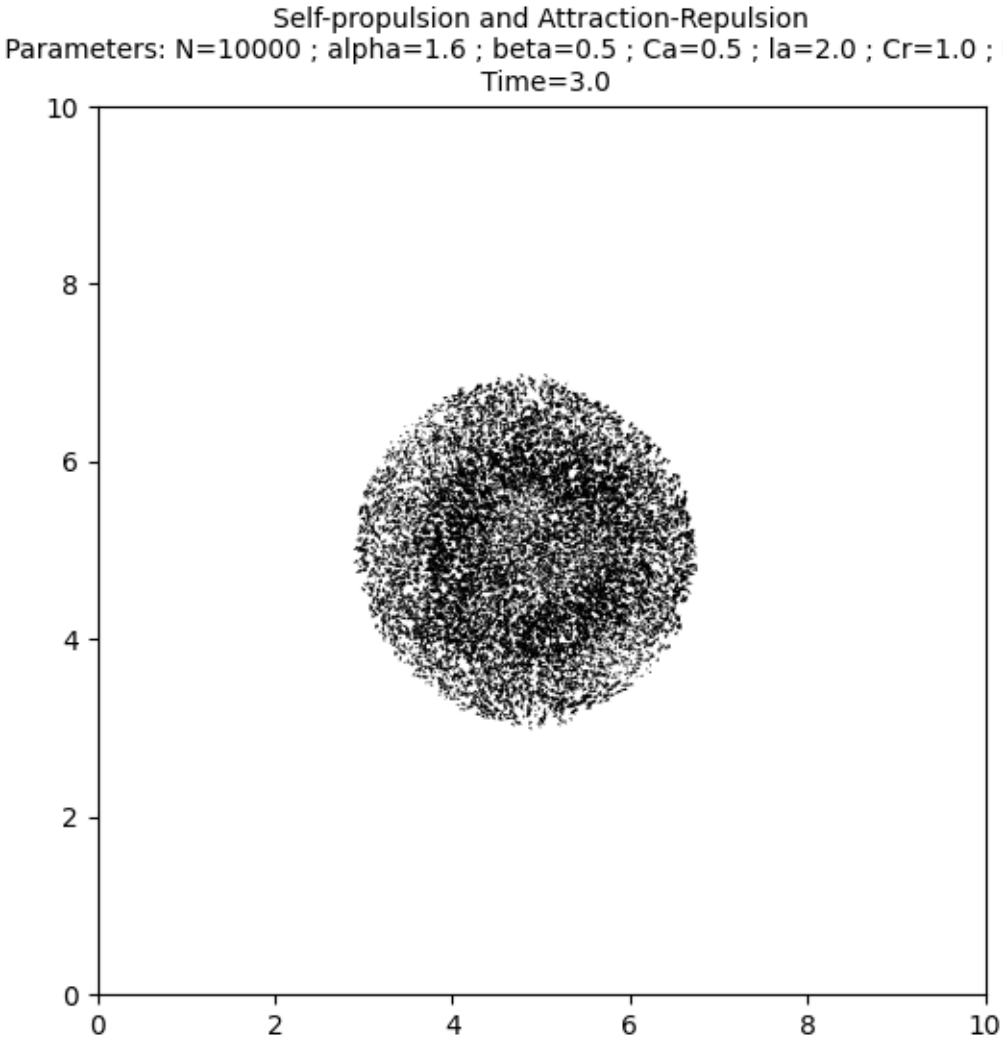

Self-propulsion and Attraction-Repulsion
Parameters: N=10000 ; alpha=1.6 ; beta=0.5 ; Ca=0.5 ; la=2.0 ; Cr=1.0 ; lr=0.5
Time=0.0
10
8
6
4
2
0
0
2
4
6
8
10

-

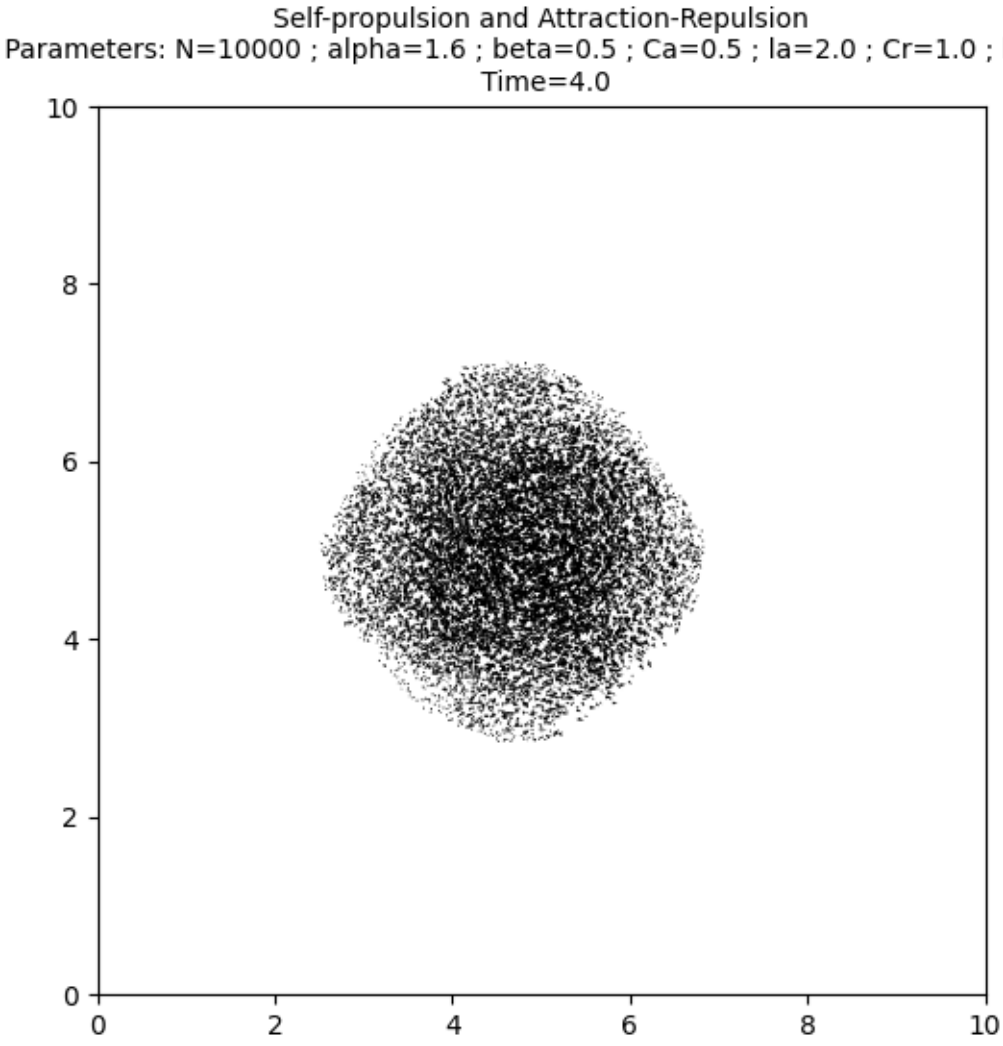

Self-propulsion and Attraction-Repulsion
Parameters: N=10000 ; alpha=1.6 ; beta=0.5 ; Ca=0.5 ; la=2.0 ; Cr=1.0 ; lr=0.5
Time=1.0
10
8
6
4
2
0
0
2
4
6
8
10

•

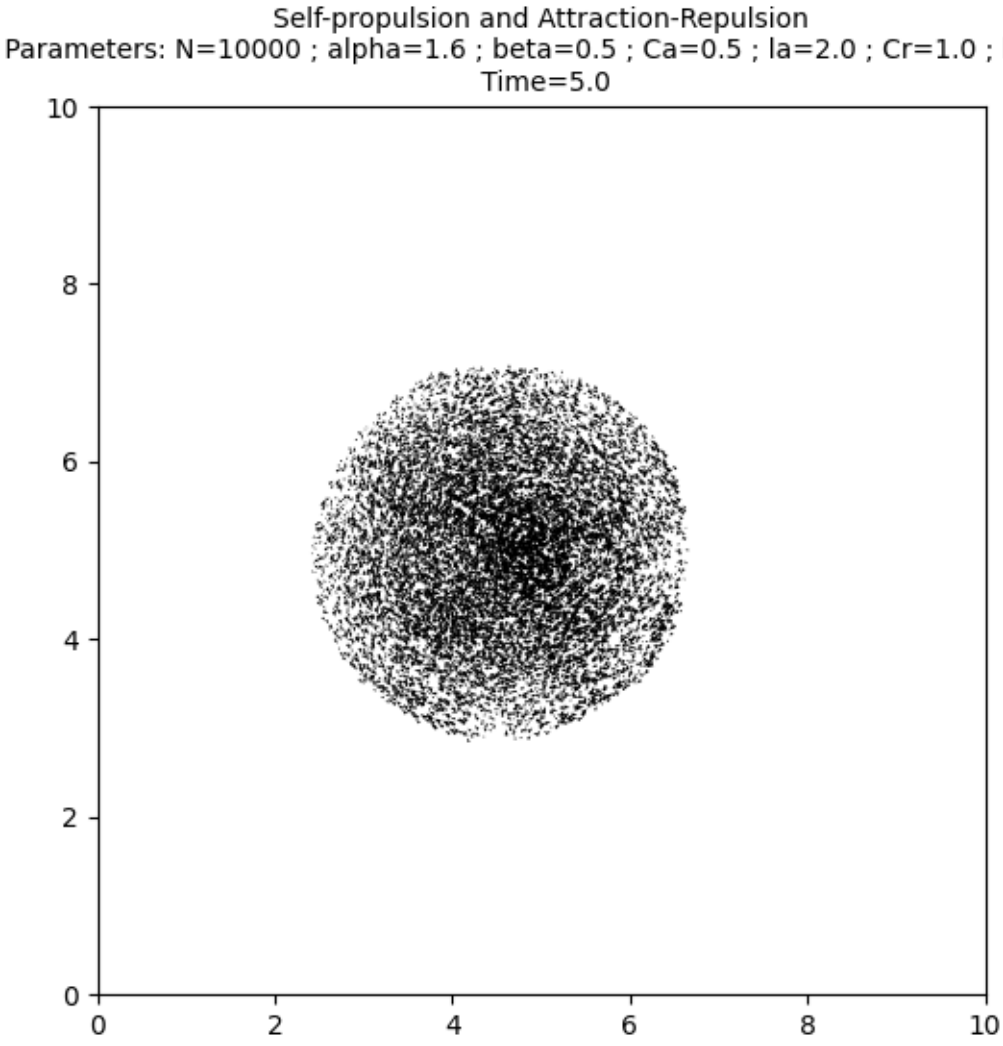

Self-propulsion and Attraction-Repulsion
Parameters: N=10000 ; alpha=1.6 ; beta=0.5 ; Ca=0.5 ; la=2.0 ; Cr=1.0 ; lr=0.5
Time=2.0
10
8
6
4
2
0
0
2
4
6
8
10


Self-propulsion and Attraction-Repulsion
Parameters: N=10000 ; alpha=1.6 ; beta=0.5 ; Ca=0.5 ; la=2.0 ; Cr=1.0 ; lr=0.5
Time=3.0
10
8
6
4
2
0
0
2
4
6
8
10

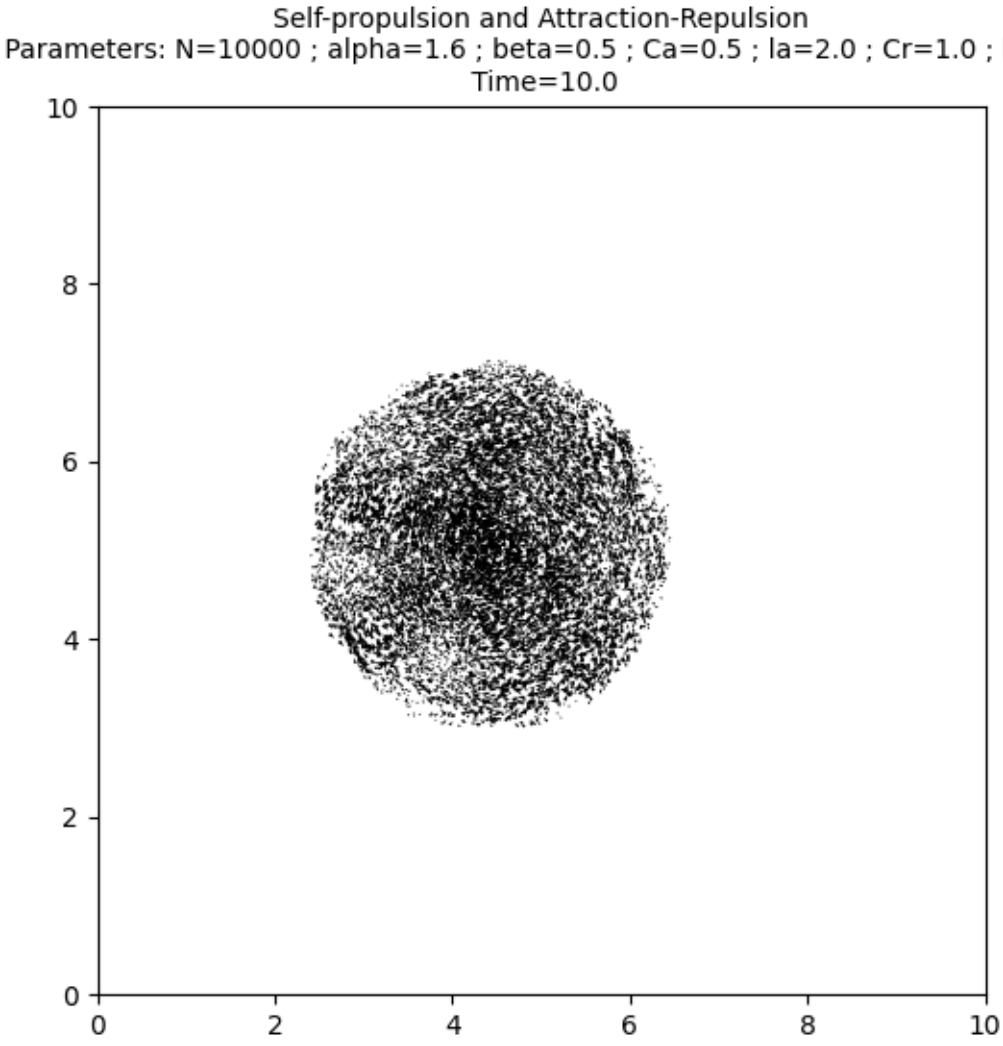
•

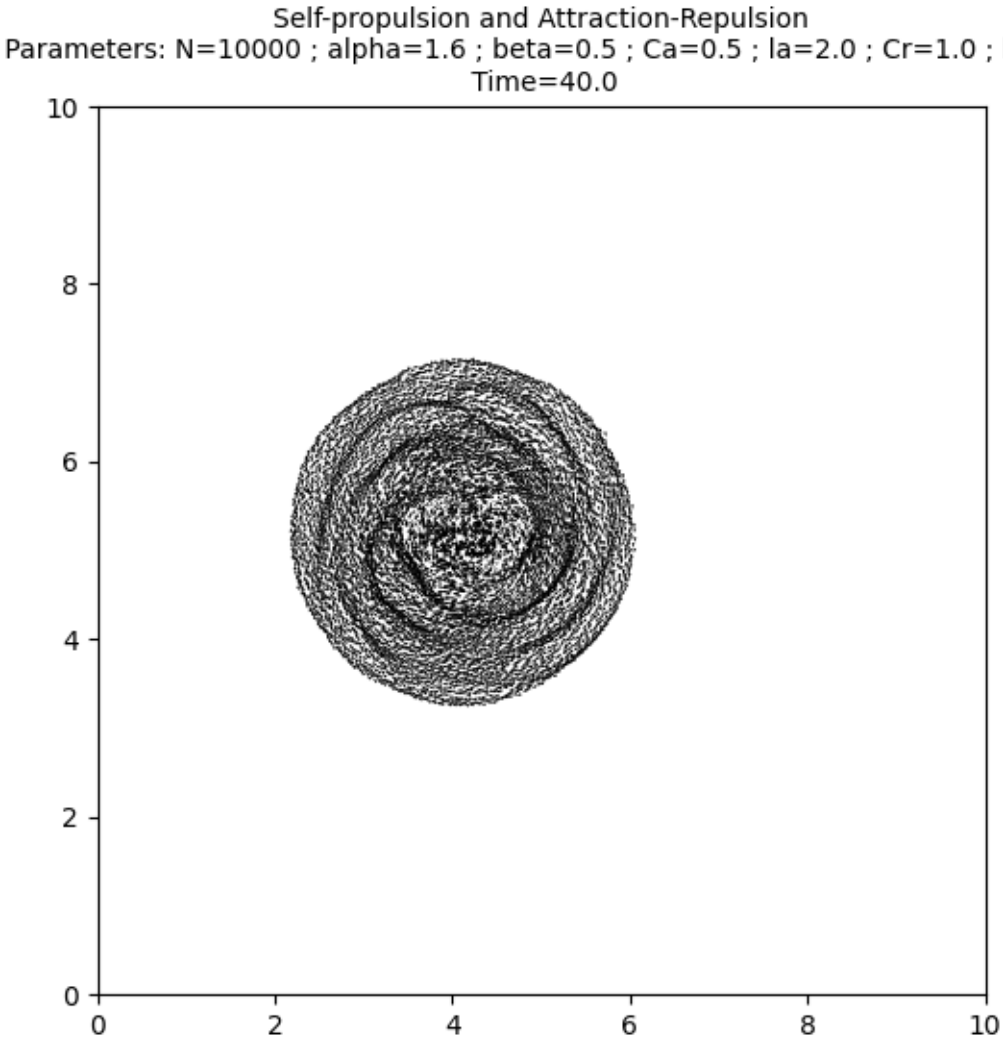

Self-propulsion and Attraction-Repulsion
Parameters: N=10000 ; alpha=1.6 ; beta=0.5 ; Ca=0.5 ; la=2.0 ; Cr=1.0 ; lr=0.5
Time=4.0
10
8
6
4
2
0
0
2
4
6
8
10

•

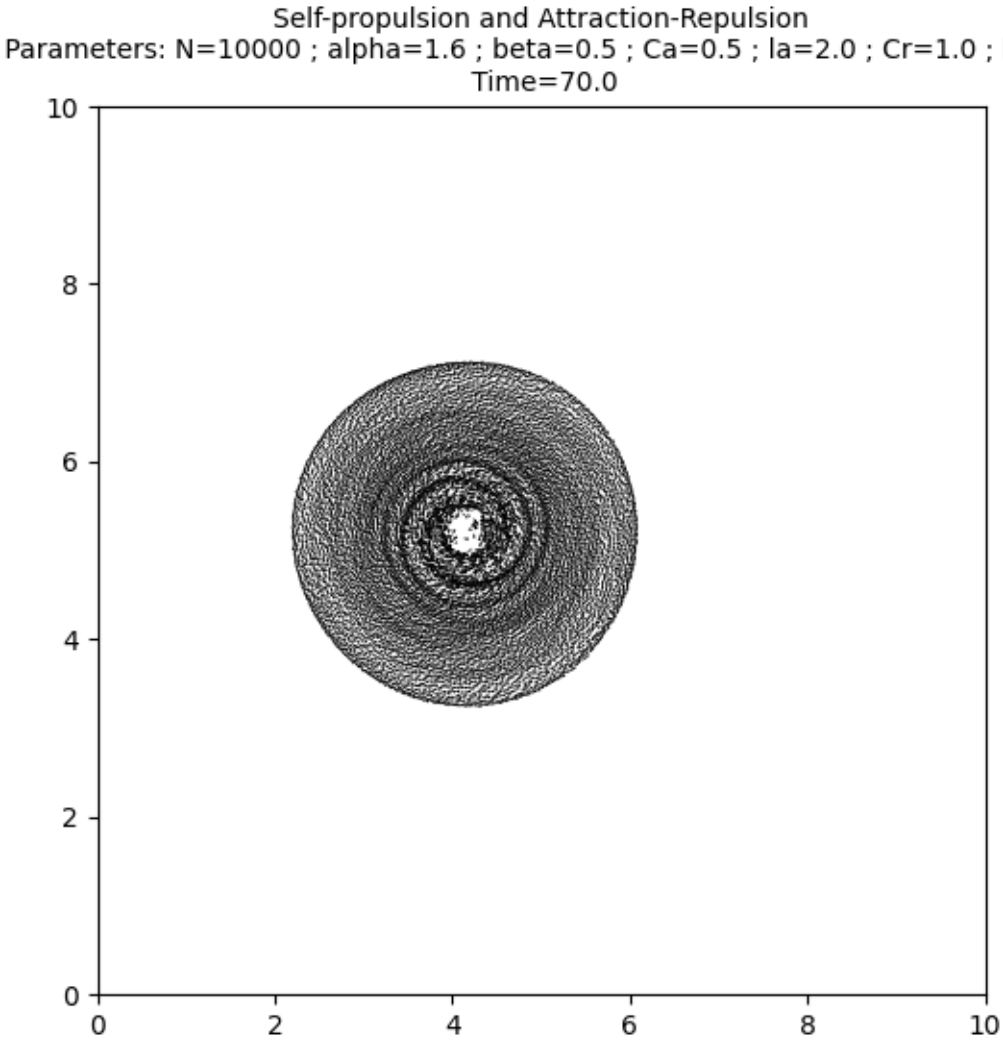

Self-propulsion and Attraction-Repulsion
Parameters: N=10000 ; alpha=1.6 ; beta=0.5 ; Ca=0.5 ; la=2.0 ; Cr=1.0 ; lr=0.5
Time=5.0
10
8
6
4
2
0
0
2
4
6
8
10

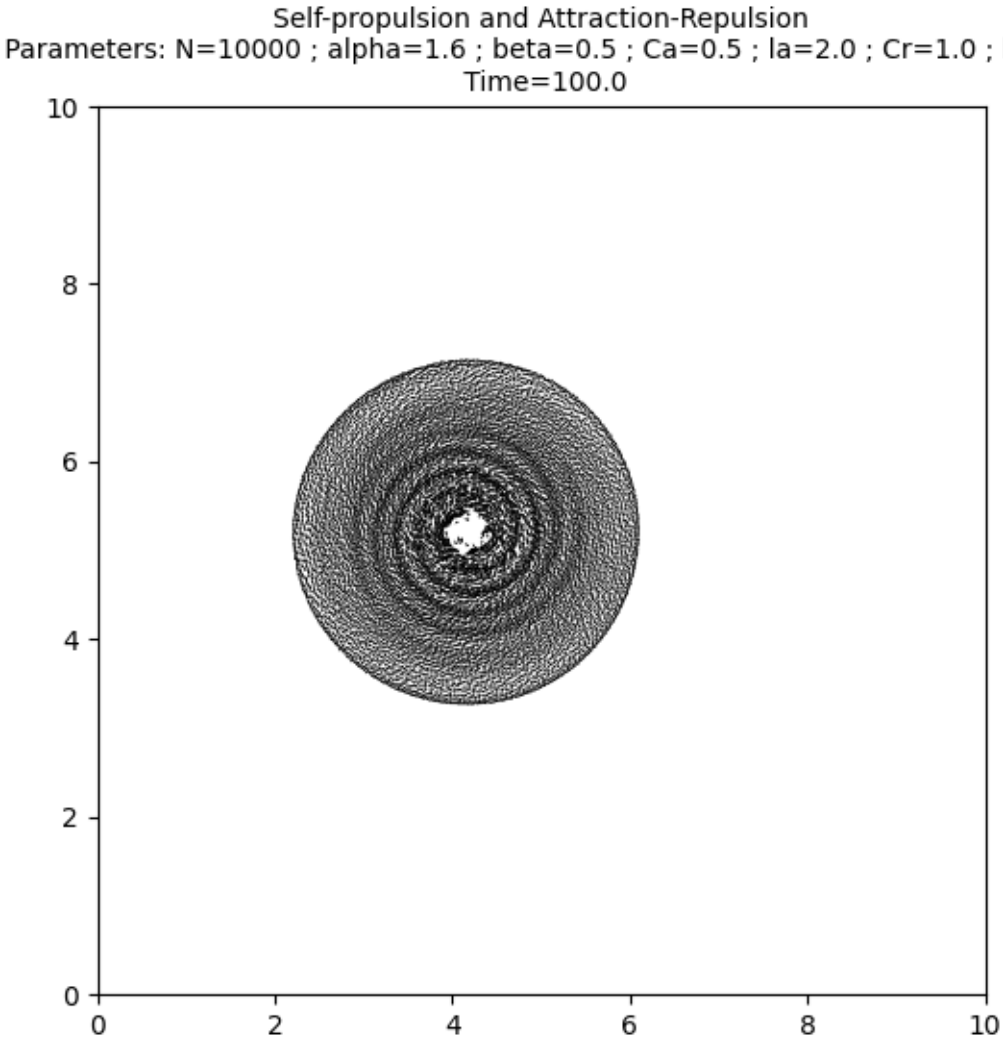
•


Self-propulsion and Attraction-Repulsion
Parameters: N=10000 ; alpha=1.6 ; beta=0.5 ; Ca=0.5 ; la=2.0 ; Cr=1.0 ; lr=0.5
Time=10.0
10
8
6
4
2
0
0
2
4
6
8
10


Self-propulsion and Attraction-Repulsion
Parameters: N=10000 ; alpha=1.6 ; beta=0.5 ; Ca=0.5 ; la=2.0 ; Cr=1.0 ; lr=0.5
Time=40.0
10
8
6
4
2
0
0
2
4
6
8
10

•


Self-propulsion and Attraction-Repulsion
Parameters: N=10000 ; alpha=1.6 ; beta=0.5 ; Ca=0.5 ; la=2.0 ; Cr=1.0 ; lr=0.5
Time=70.0
10
8
6
4
2
0
0
2
4
6
8
10



-

Out:

```
Progress:0%
Progress:1%
Progress:2%
Progress:3%
Progress:4%
Progress:5%
Progress:6%
Progress:7%
Progress:8%
Progress:9%
Progress:10%
Progress:11%
Progress:12%
Progress:13%
Progress:14%
```

(continues on next page)

(continued from previous page)

```
Progress:15%
Progress:16%
Progress:17%
Progress:18%
Progress:19%
Progress:20%
Progress:21%
Progress:22%
Progress:23%
Progress:24%
Progress:25%
Progress:26%
Progress:27%
Progress:28%
Progress:29%
Progress:30%
Progress:31%
Progress:32%
Progress:33%
Progress:34%
Progress:35%
Progress:36%
Progress:37%
Progress:38%
Progress:39%
Progress:40%
Progress:41%
Progress:42%
Progress:43%
Progress:44%
Progress:45%
Progress:46%
Progress:47%
Progress:48%
Progress:49%
Progress:50%
Progress:51%
Progress:52%
Progress:53%
Progress:54%
Progress:55%
Progress:56%
Progress:57%
Progress:58%
Progress:59%
Progress:60%
Progress:61%
Progress:62%
Progress:63%
Progress:64%
Progress:65%
Progress:66%
```

(continues on next page)

(continued from previous page)

```
Progress:67%
Progress:68%
Progress:69%
Progress:70%
Progress:71%
Progress:72%
Progress:73%
Progress:74%
Progress:75%
Progress:76%
Progress:77%
Progress:78%
Progress:79%
Progress:80%
Progress:81%
Progress:82%
Progress:83%
Progress:84%
Progress:85%
Progress:86%
Progress:87%
Progress:88%
Progress:89%
Progress:90%
Progress:91%
Progress:92%
Progress:93%
Progress:94%
Progress:95%
Progress:96%
Progress:97%
Progress:98%
Progress:99%
```

Print the total simulation time and the average time per iteration.

```
print('Total time: '+str(e-s)+' seconds')
print('Average time per iteration: '+str((e-s)/simu.iteration)+' seconds')
```

Out:

```
Total time: 86.00608777999878 seconds
Average time per iteration: 0.008600608777999877 seconds
```

Milling in the Vicsek model

Let us create an instance of the Asynchronuos Vicsek model with a bounded cone of vision and a bounded angular velocity, as introduced in:

A. Costanzo, C. K. Hemelrijk, Spontaneous emergence of milling (vortex state) in a Vicsek-like model, *J. Phys. D: Appl. Phys.*, 51, 134004

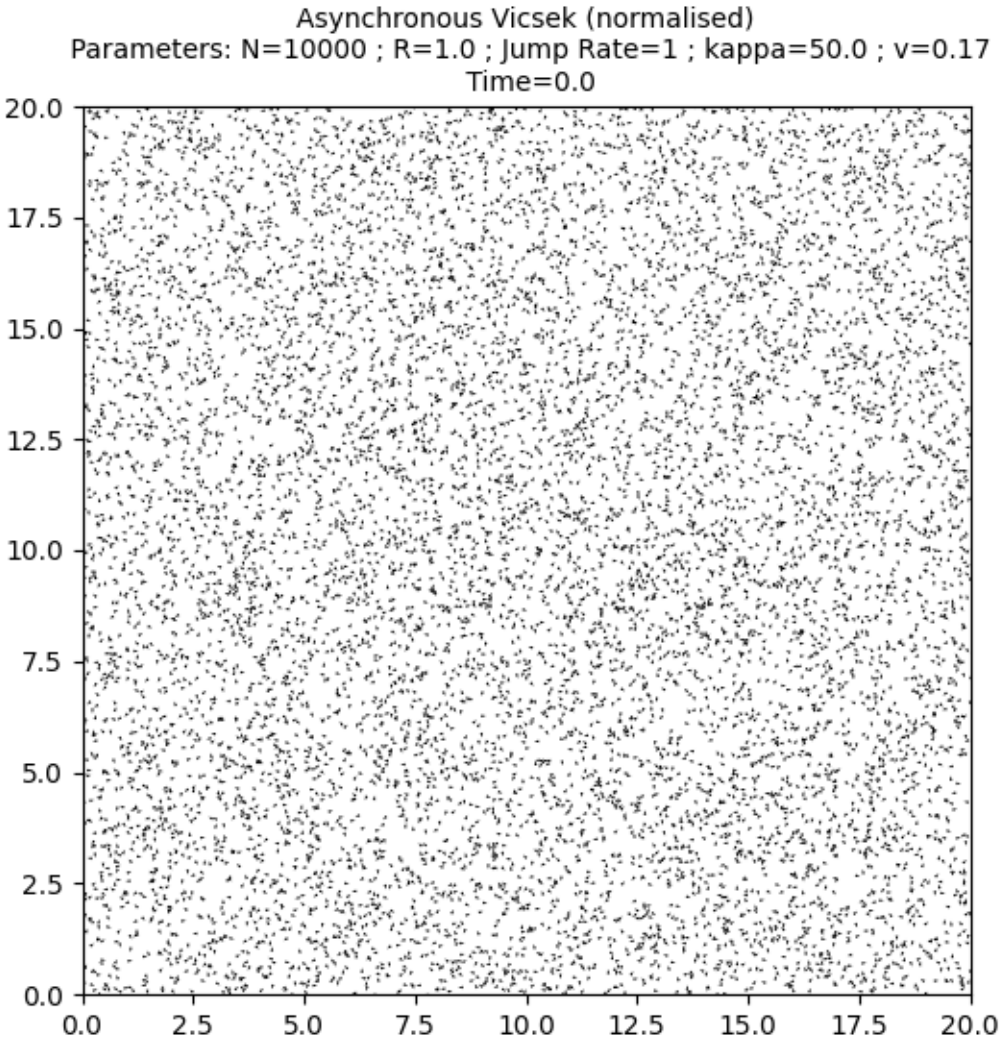
```
N = 10000
R = 1.
L = 20.

nu = 1
sigma = .02
kappa = nu/sigma

c = .175
angvel_max = .175/nu

pos = L*torch.rand((N,2)).type(dtype)
vel = torch.randn(N,2).type(dtype)
vel = vel/torch.norm(vel,dim=1).reshape((N,1))

dt = .01
```

We add an option to the target

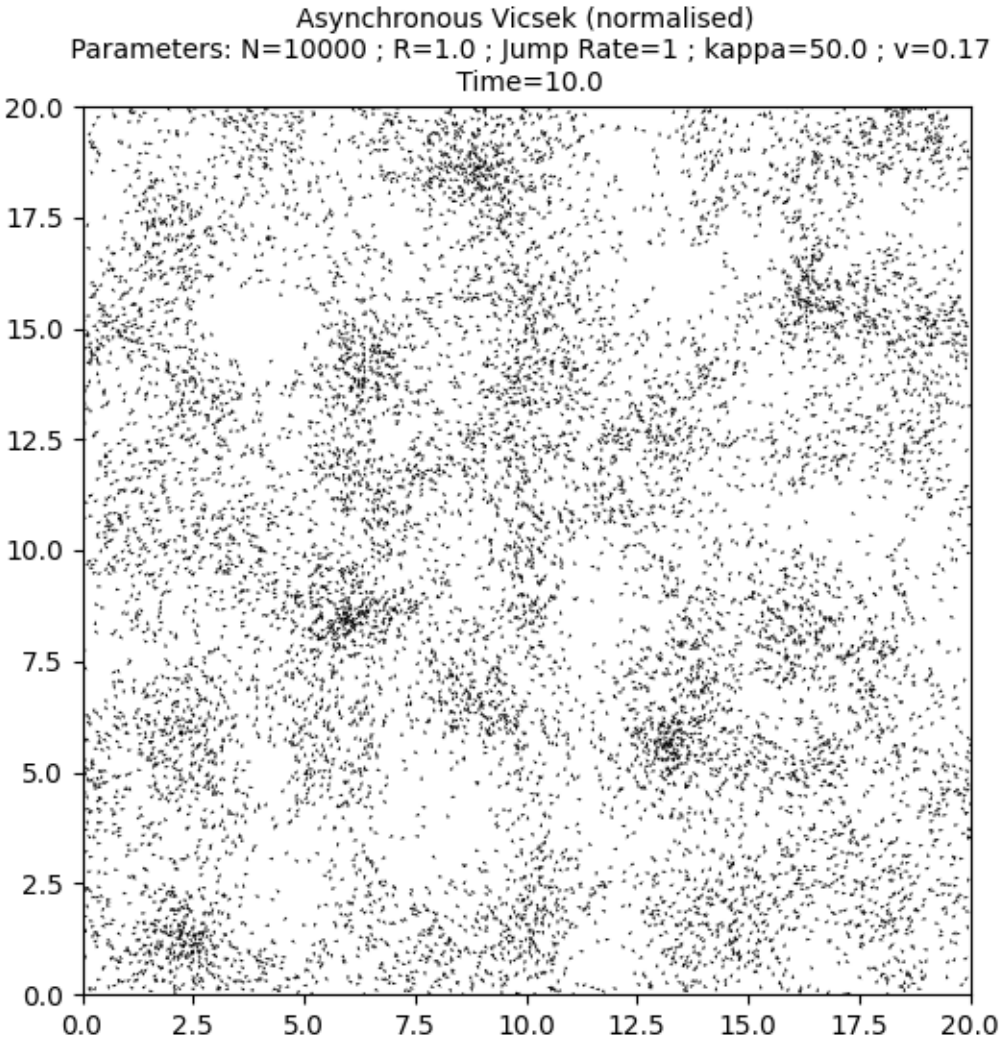
```
target = {"name" : "normalised", "parameters" : {}}
option = {"bounded_angular_velocity" : {"angvel_max" : angvel_max, "dt" : 1./nu}}

simu=models.AsynchronousVicsek(pos=pos,vel=vel,
                  v=c,
                  jump_rate=nu,kappa=kappa,
                  interaction_radius=R,
                  box_size=L,
                  vision_angle=math.pi, axis = None,
                  boundary_conditions='periodic',
                  variant=target,
                  options=option,
                  sampling_method='projected_normal')
```

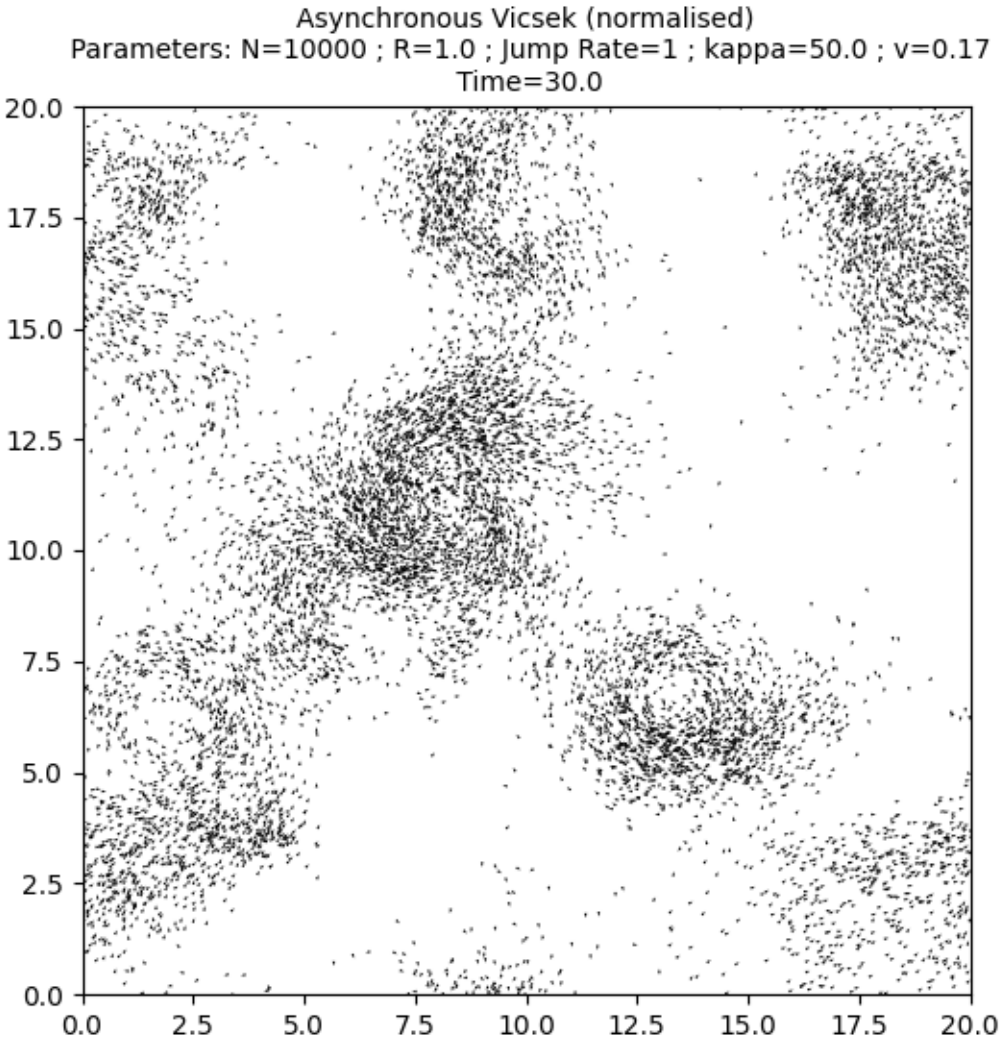
Run the simulation over 200 units of time and plot 10 frames.

```
# sphinx_gallery_thumbnail_number = -1

frames = [0, 10, 30, 50, 75, 100, 125, 150, 175, 200]

s = time.time()
it, op = display_kinetic_particles(simu,frames)
e = time.time()
```

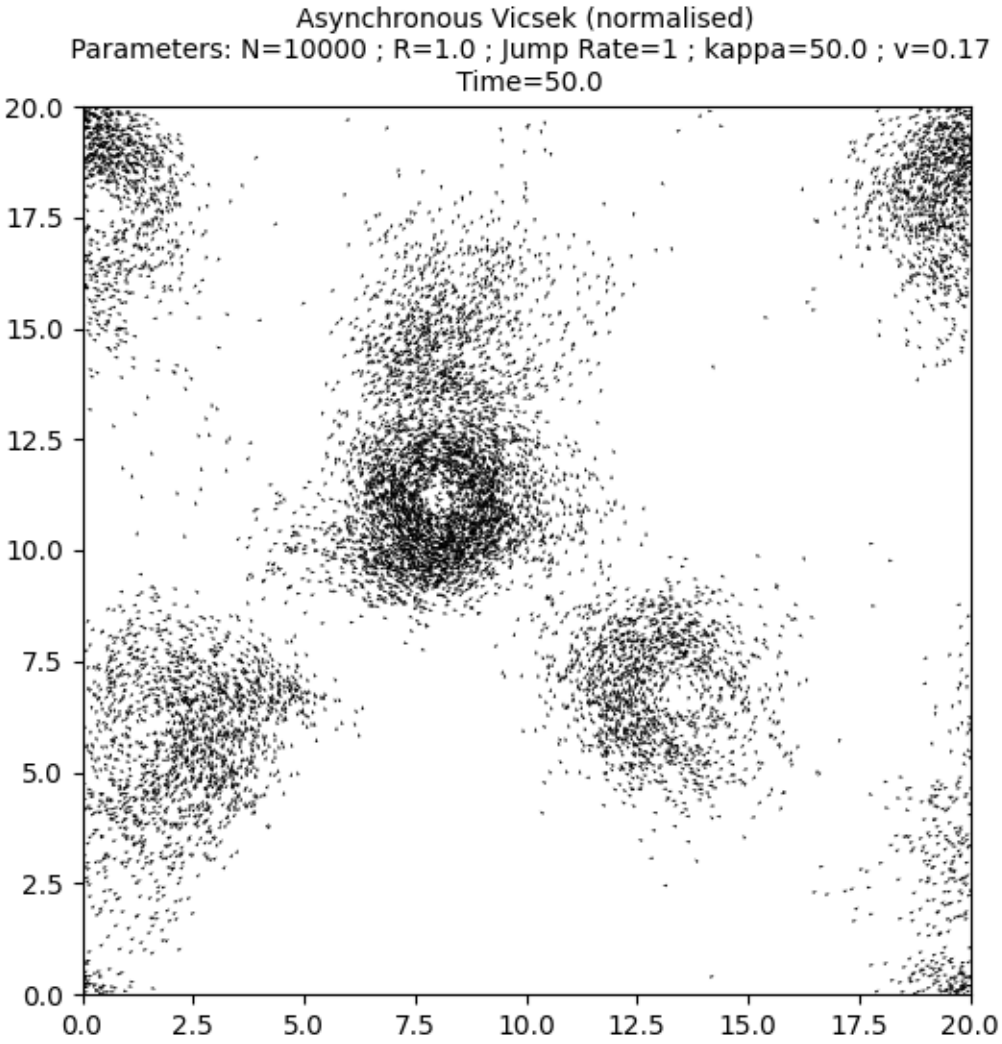

Asynchronous Vicsek (normalised)
Parameters: N=10000 ; R=1.0 ; Jump Rate=1 ; kappa=50.0 ; v=0.17
Time=0.0
20.0
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0

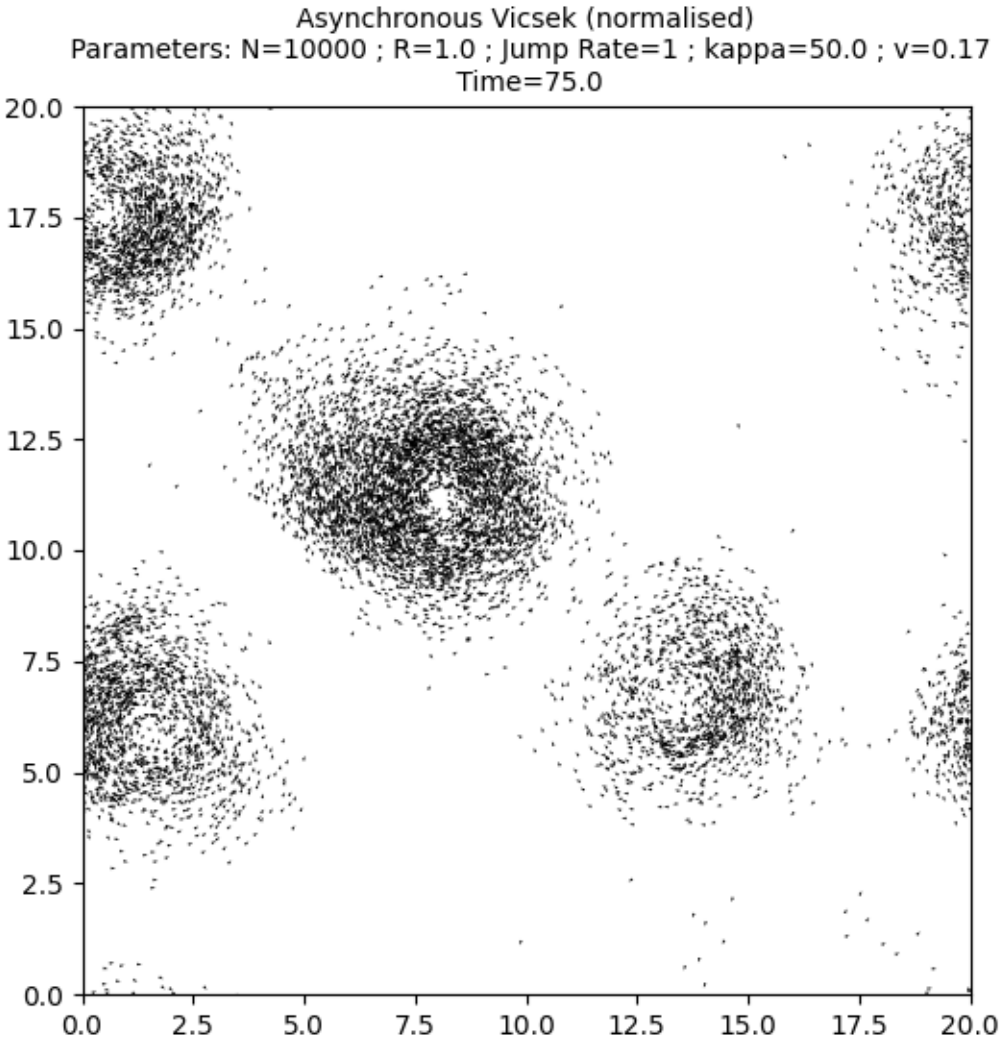
•


Asynchronous Vicsek (normalised)
Parameters: N=10000 ; R=1.0 ; Jump Rate=1 ; kappa=50.0 ; v=0.17
Time=10.0
20.0
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0

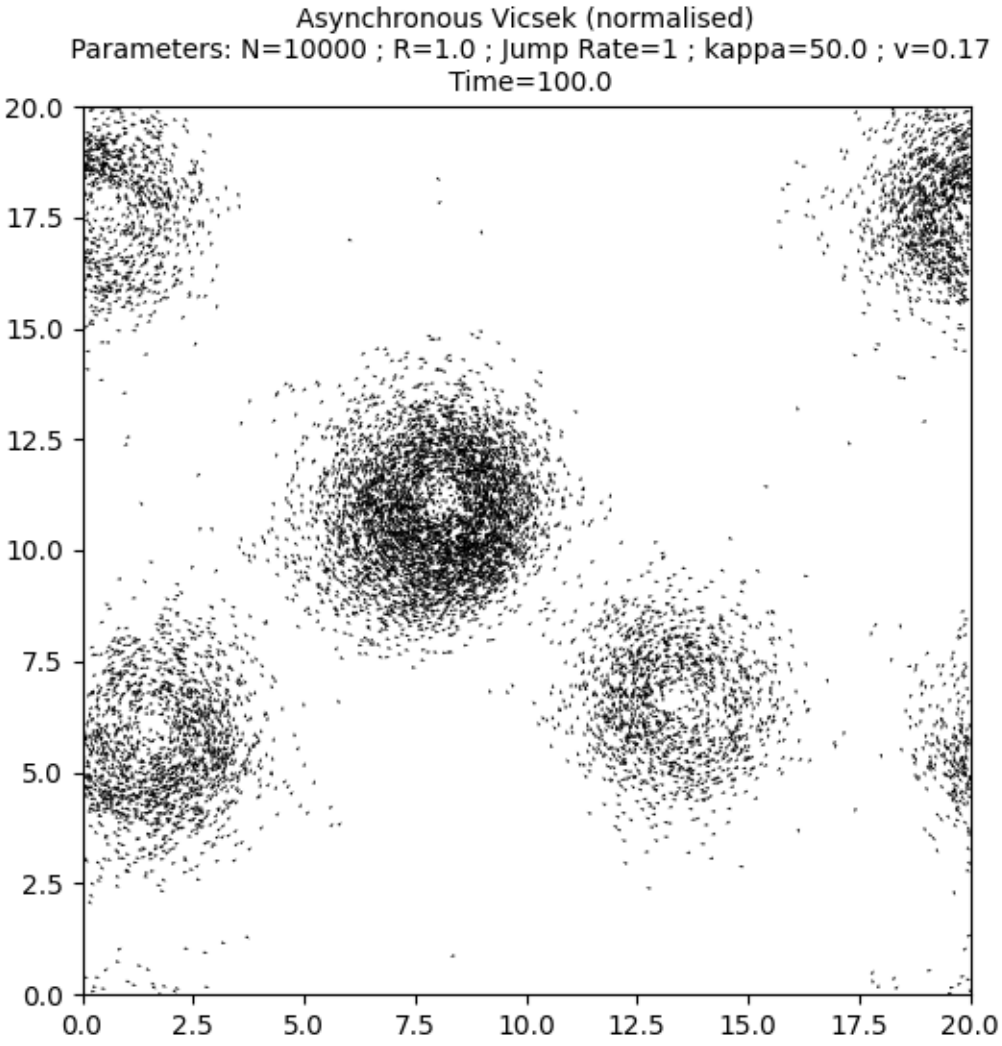

Asynchronous Vicsek (normalised)
Parameters: N=10000 ; R=1.0 ; Jump Rate=1 ; kappa=50.0 ; v=0.17
Time=30.0
20.0
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0

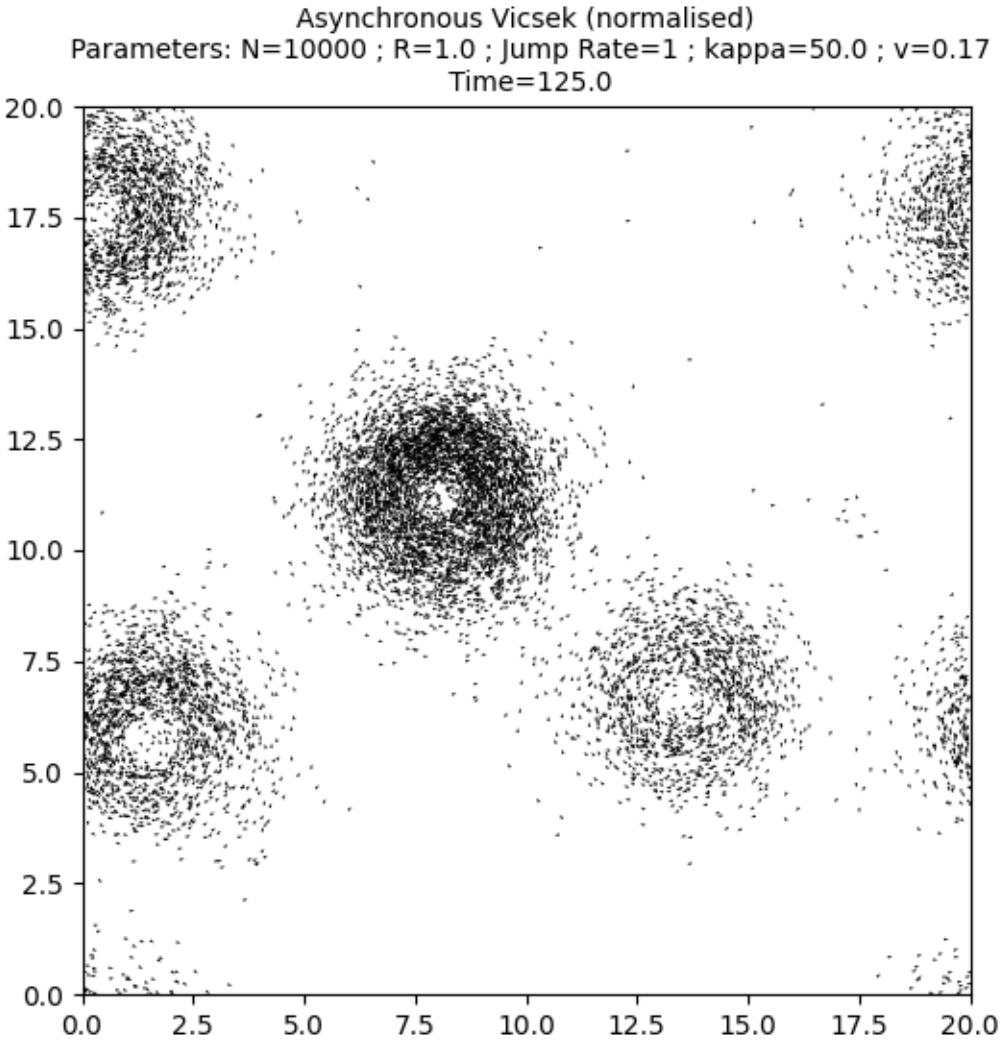
•


Asynchronous Vicsek (normalised)
Parameters: N=10000 ; R=1.0 ; Jump Rate=1 ; kappa=50.0 ; v=0.17
Time=50.0
20.0
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0

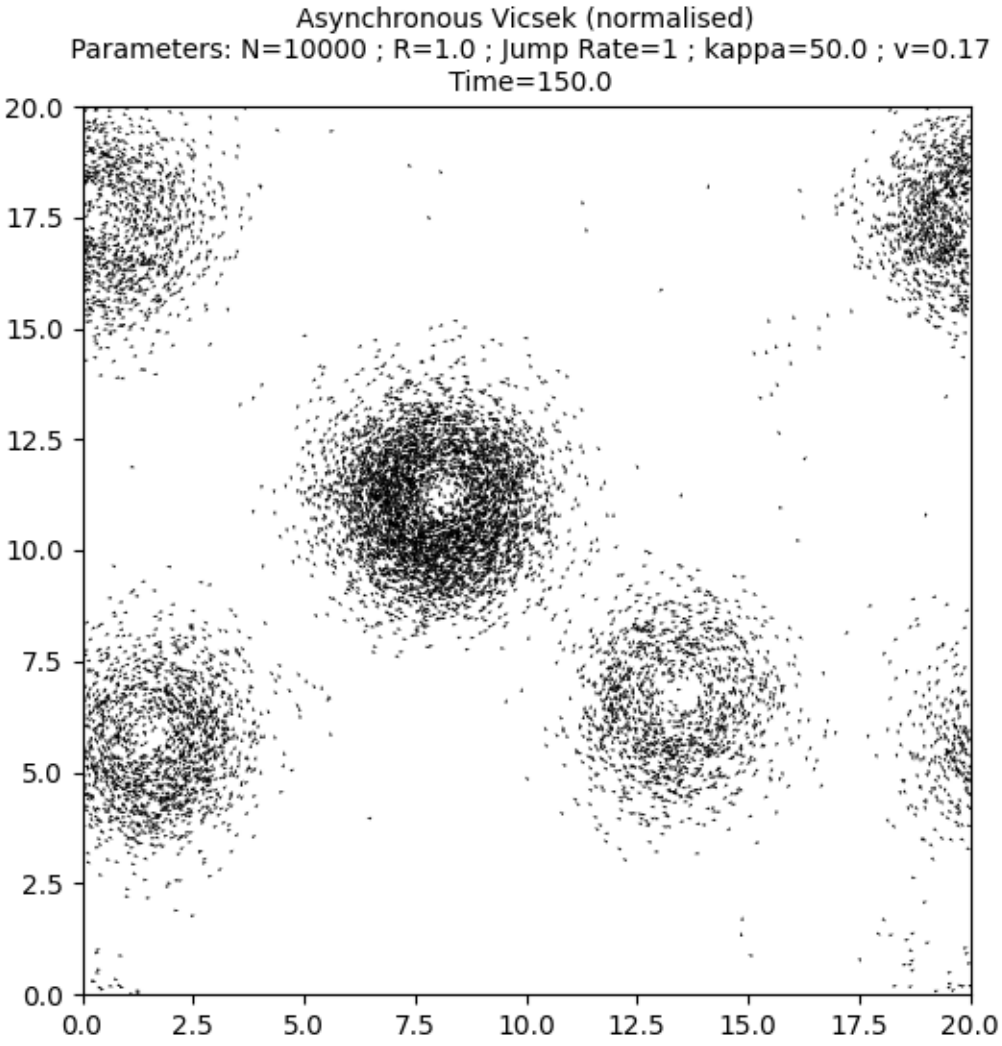

Asynchronous Vicsek (normalised)
Parameters: N=10000 ; R=1.0 ; Jump Rate=1 ; kappa=50.0 ; v=0.17
Time=75.0
20.0
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0

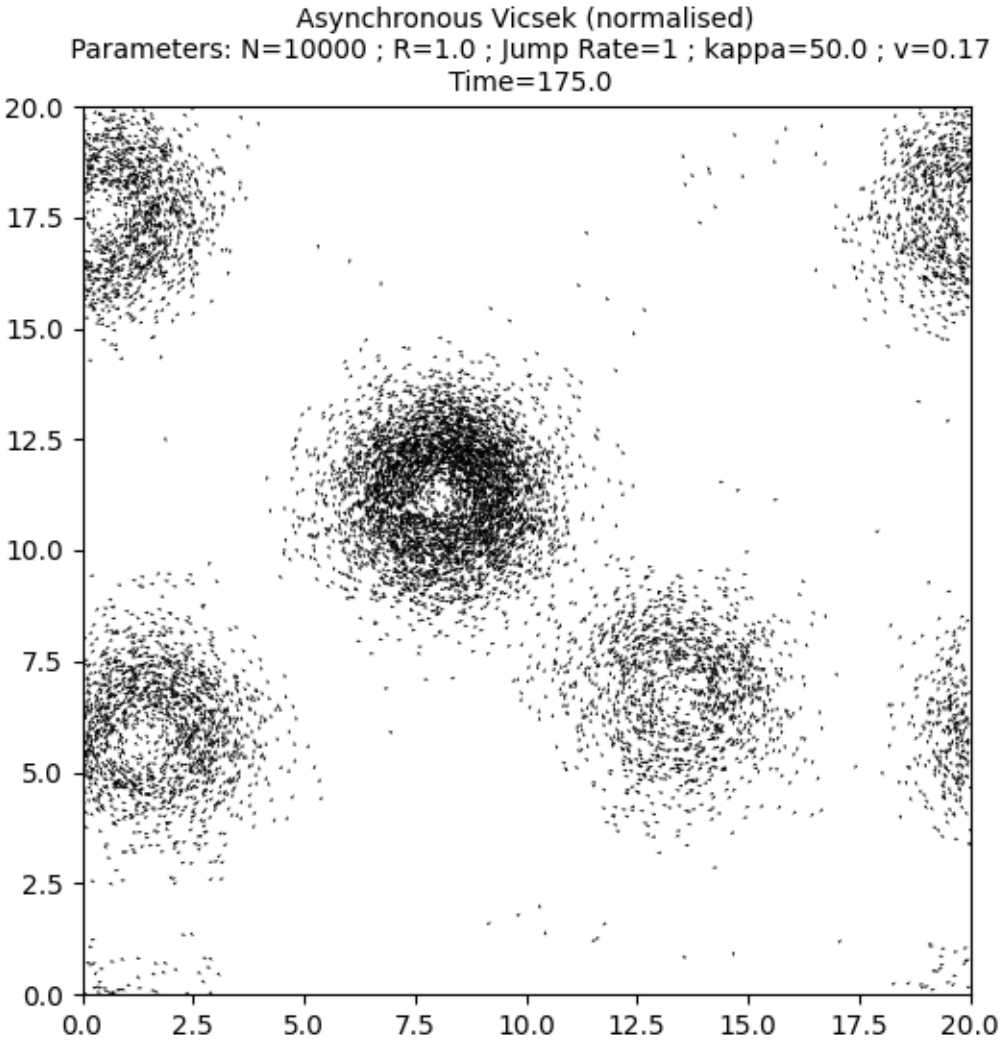
•


Asynchronous Vicsek (normalised)
Parameters: N=10000 ; R=1.0 ; Jump Rate=1 ; kappa=50.0 ; v=0.17
Time=100.0
20.0
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0

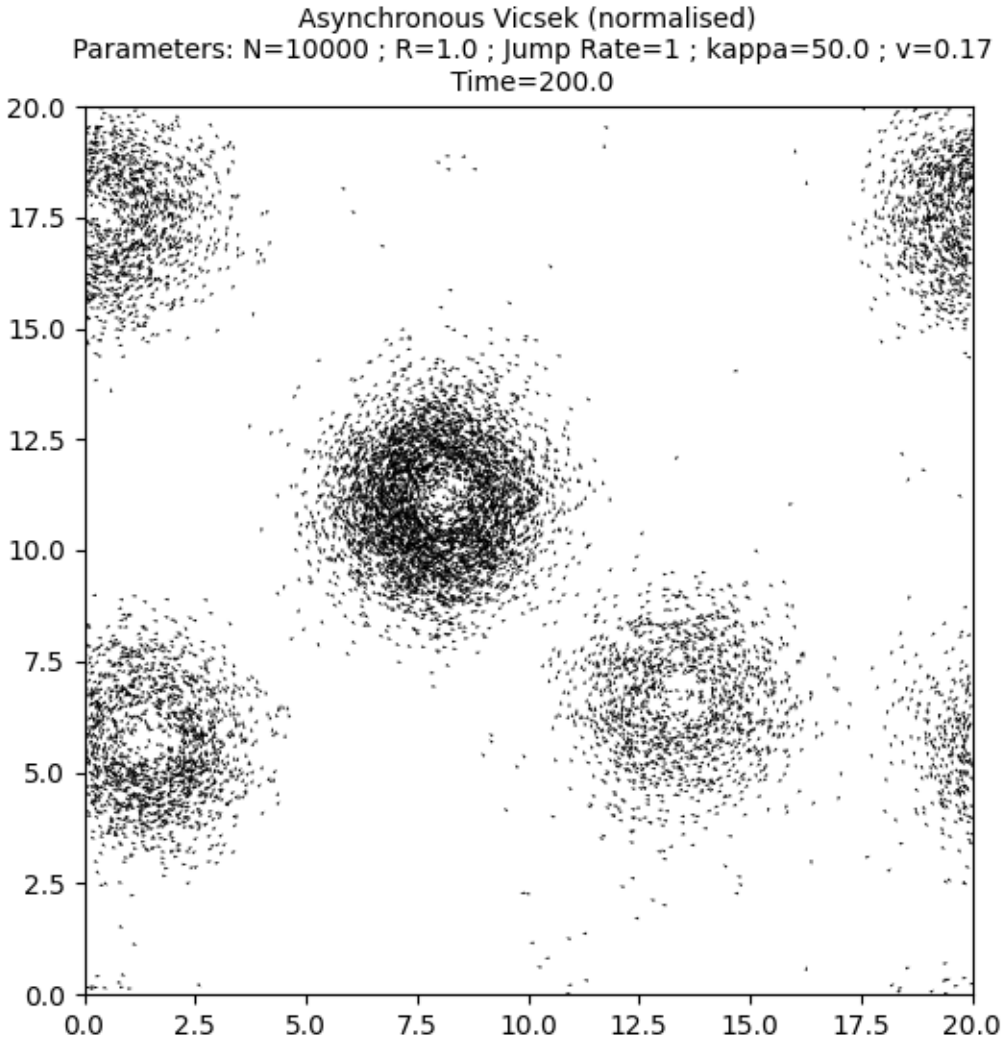

Asynchronous Vicsek (normalised)
Parameters: N=10000 ; R=1.0 ; Jump Rate=1 ; kappa=50.0 ; v=0.17
Time=125.0
20.0
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0

•


Asynchronous Vicsek (normalised)
Parameters: N=10000 ; R=1.0 ; Jump Rate=1 ; kappa=50.0 ; v=0.17
Time=150.0
20.0
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0


Asynchronous Vicsek (normalised)
Parameters: N=10000 ; R=1.0 ; Jump Rate=1 ; kappa=50.0 ; v=0.17
Time=175.0
20.0
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0

•



-

Out:

```
Progress:0%
Progress:1%
Progress:2%
Progress:3%
Progress:4%
Progress:5%
Progress:6%
Progress:7%
Progress:8%
Progress:9%
Progress:10%
Progress:11%
Progress:12%
Progress:13%
Progress:14%
```

(continues on next page)

(continued from previous page)

```
Progress:15%
Progress:16%
Progress:17%
Progress:18%
Progress:19%
Progress:20%
Progress:21%
Progress:22%
Progress:23%
Progress:24%
Progress:25%
Progress:26%
Progress:27%
Progress:28%
Progress:29%
Progress:30%
Progress:31%
Progress:32%
Progress:33%
Progress:34%
Progress:35%
Progress:36%
Progress:37%
Progress:38%
Progress:39%
Progress:40%
Progress:41%
Progress:42%
Progress:43%
Progress:44%
Progress:45%
Progress:46%
Progress:47%
Progress:48%
Progress:49%
Progress:50%
Progress:51%
Progress:52%
Progress:53%
Progress:54%
Progress:55%
Progress:56%
Progress:57%
Progress:58%
Progress:59%
Progress:60%
Progress:61%
Progress:62%
Progress:63%
Progress:64%
Progress:65%
Progress:66%
```

(continues on next page)

(continued from previous page)

```
Progress:67%
Progress:68%
Progress:69%
Progress:70%
Progress:71%
Progress:72%
Progress:73%
Progress:74%
Progress:75%
Progress:76%
Progress:77%
Progress:78%
Progress:79%
Progress:80%
Progress:81%
Progress:82%
Progress:83%
Progress:84%
Progress:85%
Progress:86%
Progress:87%
Progress:88%
Progress:89%
Progress:90%
Progress:91%
Progress:92%
Progress:93%
Progress:94%
Progress:95%
Progress:96%
Progress:97%
Progress:98%
Progress:99%
```

Print the total simulation time and the average time per iteration.

```
print('Total time: '+str(e-s)+' seconds')
print('Average time per iteration: '+str((e-s)/simu.iteration)+' seconds')
```

Out:

```
Total time: 69.85851073265076 seconds
Average time per iteration: 0.003492925536632538 seconds
```

Total running time of the script: (2 minutes 38.151 seconds)

4.5.4 Boundary clusters in a disk

A classical mean-field Vicsek model in a bounded disk domain.

First of all, some standard imports.

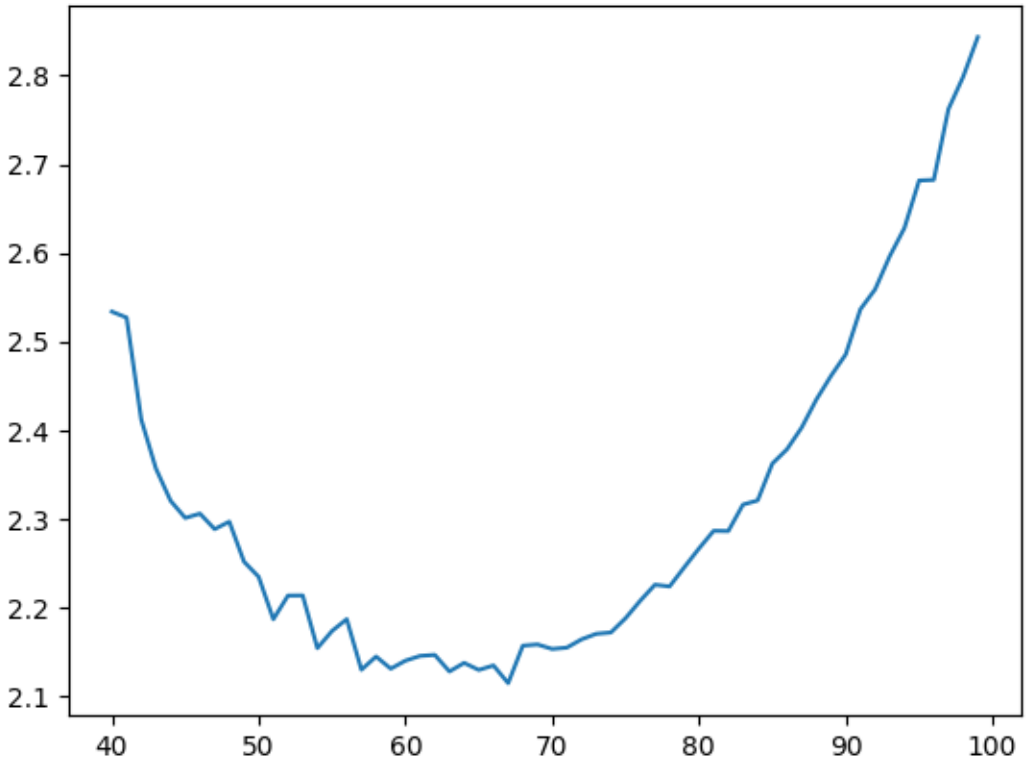
```
import os
import sys
import time
import torch
import numpy as np
from matplotlib import pyplot as plt
from sisyphe.display import display_kinetic_particles


use_cuda = torch.cuda.is_available()
dtype = torch.cuda.FloatTensor if use_cuda else torch.FloatTensor
```

Set the parameters and create an instance of the Vicsek model.

```
import sisyphe.models as models

N = 1000000
L = 10.
dt = .01

nu = 3
sigma = 1.
kappa = nu/sigma

R = .1
c = 1.

center = torch.tensor([L/2,L/2]).type(dtype).reshape((1,2))
radius = L/2
pos = L*torch.rand((N,2)).type(dtype)
out = ((pos-center)**2).sum(1) > radius**2
while out.sum()>0:
    pos[out,:] = L*torch.rand((out.sum(),2)).type(dtype)
    out = ((pos-center)**2).sum(1) > radius**2
vel = torch.randn(N,2).type(dtype)
vel = vel/torch.norm(vel,dim=1).reshape((N,1))

simu=models.Vicsek(pos=pos,vel=vel,
              v=c,
              sigma=sigma,nu=nu,
              interaction_radius=R,
              box_size=L,
              boundary_conditions='spherical',
              variant = {"name" : "max_kappa", "parameters" : {"kappa_max" : 10.}},
              options = {},
              numerical_scheme='projection',
              dt=dt,
              block_sparse_reduction=True)
```

Set the block sparse parameters to their optimal value.

```
fastest, nb_cells, average_simu_time, simulation_time = simu.best_blocksparse_
↪parameters(40,100)

plt.plot(nb_cells,average_simu_time)
plt.show()
```



Out:

```
Progress:0.0%
Progress:1.67%
Progress:3.33%
Progress:5.0%
Progress:6.67%
Progress:8.33%
Progress:10.0%
Progress:11.67%
Progress:13.33%
Progress:15.0%
Progress:16.67%
Progress:18.33%
Progress:20.0%
Progress:21.67%
```

(continues on next page)

(continued from previous page)

```
Progress:23.33%
Progress:25.0%
Progress:26.67%
Progress:28.33%
Progress:30.0%
Progress:31.67%
Progress:33.33%
Progress:35.0%
Progress:36.67%
Progress:38.33%
Progress:40.0%
Progress:41.67%
Progress:43.33%
Progress:45.0%
Progress:46.67%
Progress:48.33%
Progress:50.0%
Progress:51.67%
Progress:53.33%
Progress:55.0%
Progress:56.67%
Progress:58.33%
Progress:60.0%
Progress:61.67%
Progress:63.33%
Progress:65.0%
Progress:66.67%
Progress:68.33%
Progress:70.0%
Progress:71.67%
Progress:73.33%
Progress:75.0%
Progress:76.67%
Progress:78.33%
Progress:80.0%
Progress:81.67%
Progress:83.33%
Progress:85.0%
Progress:86.67%
Progress:88.33%
Progress:90.0%
Progress:91.67%
Progress:93.33%
Progress:95.0%
Progress:96.67%
Progress:98.33%
```

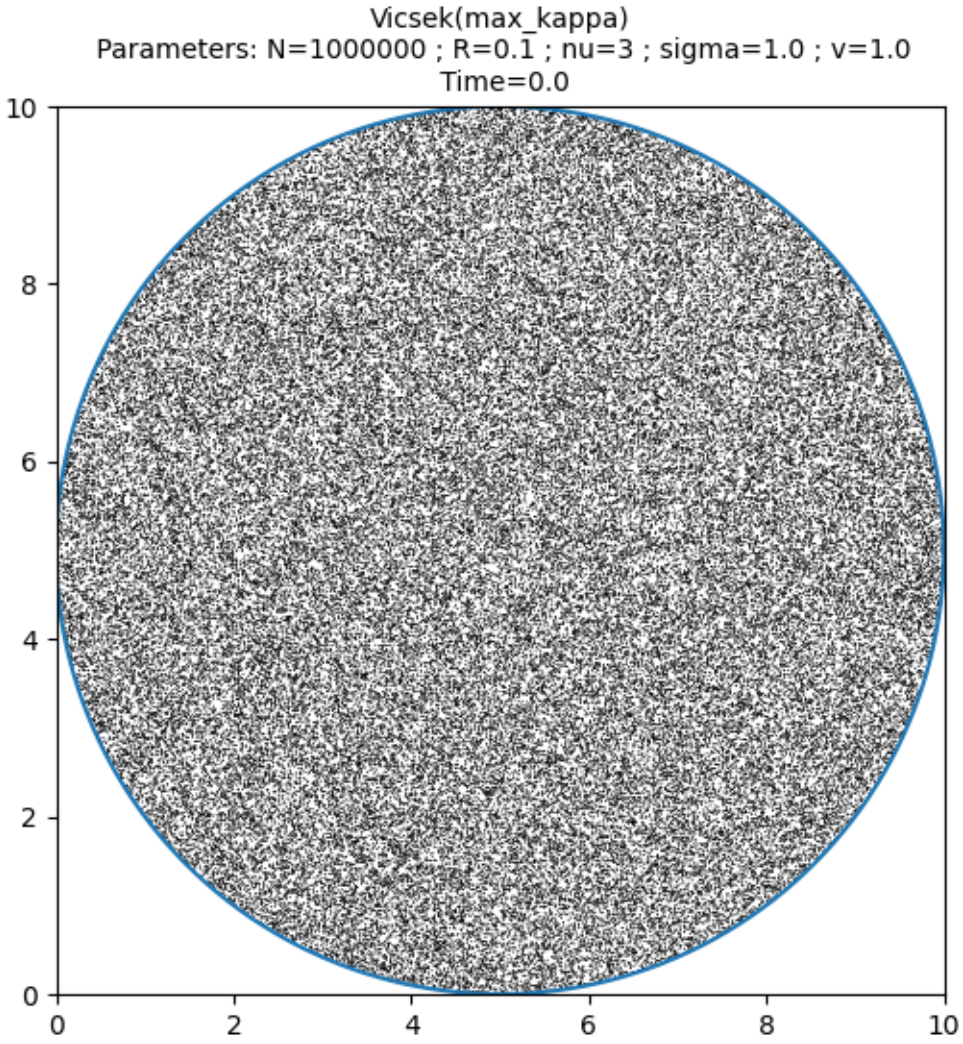
Run the simulation and plot the particles.

```
# sphinx_gallery_thumbnail_number = -1

frames = [0, 10, 40, 70, 100, 150, 200, 250, 300]
```

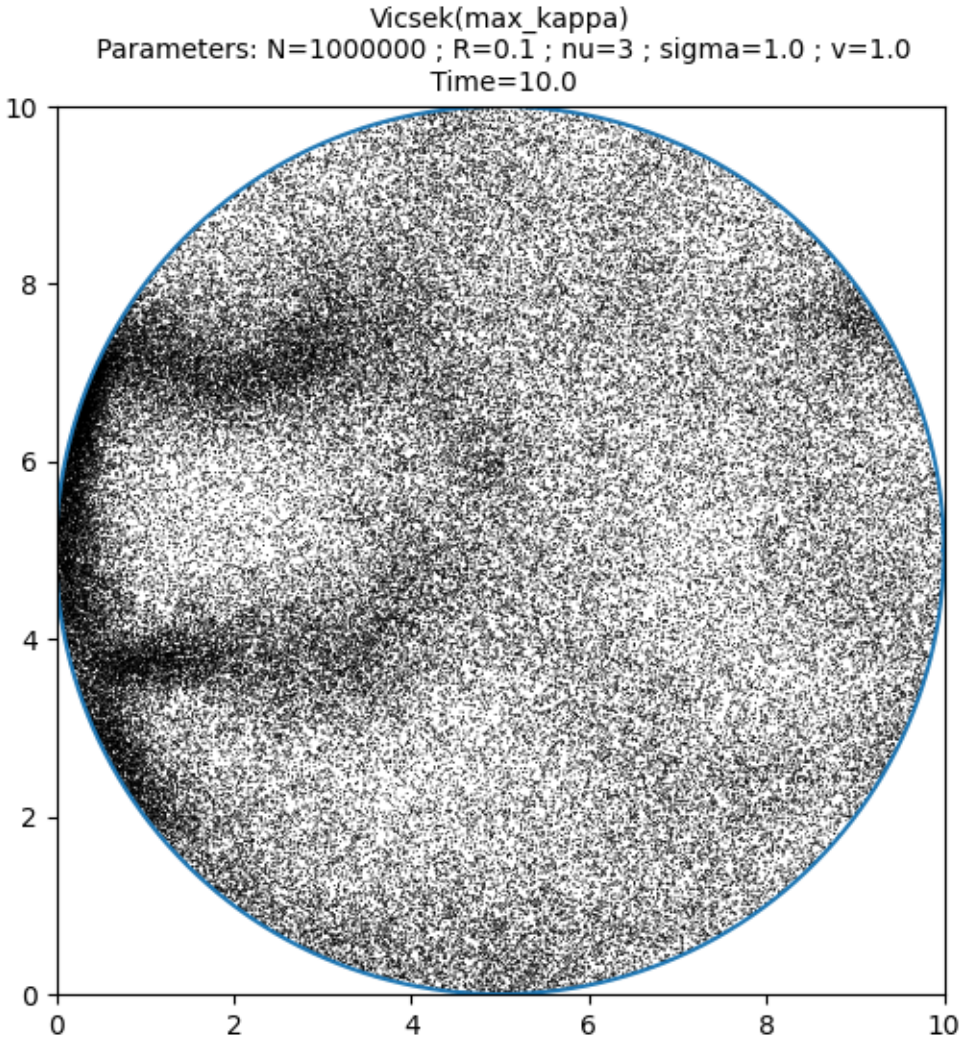
(continues on next page)

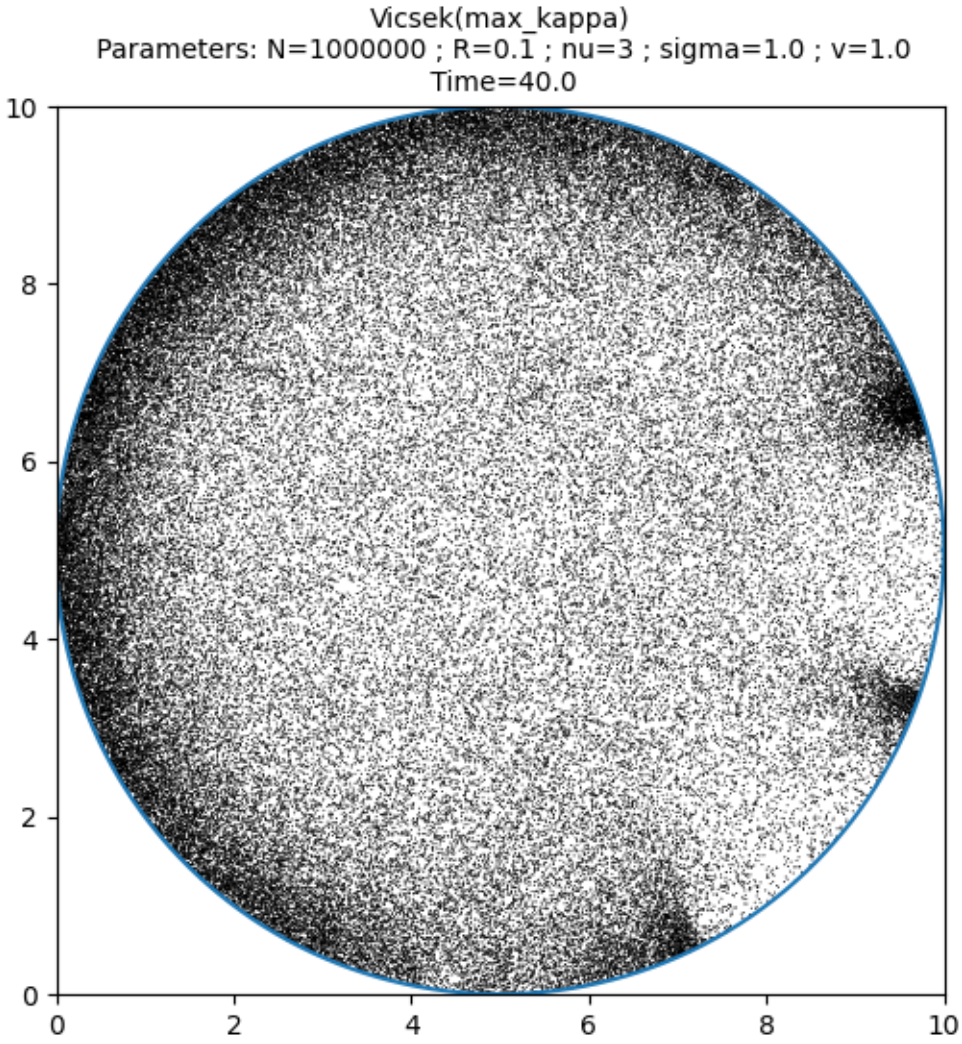
(continued from previous page)

```
s = time.time()
it, op = display_kinetic_particles(simu, frames, N_dispmax=100000)
e = time.time()
```

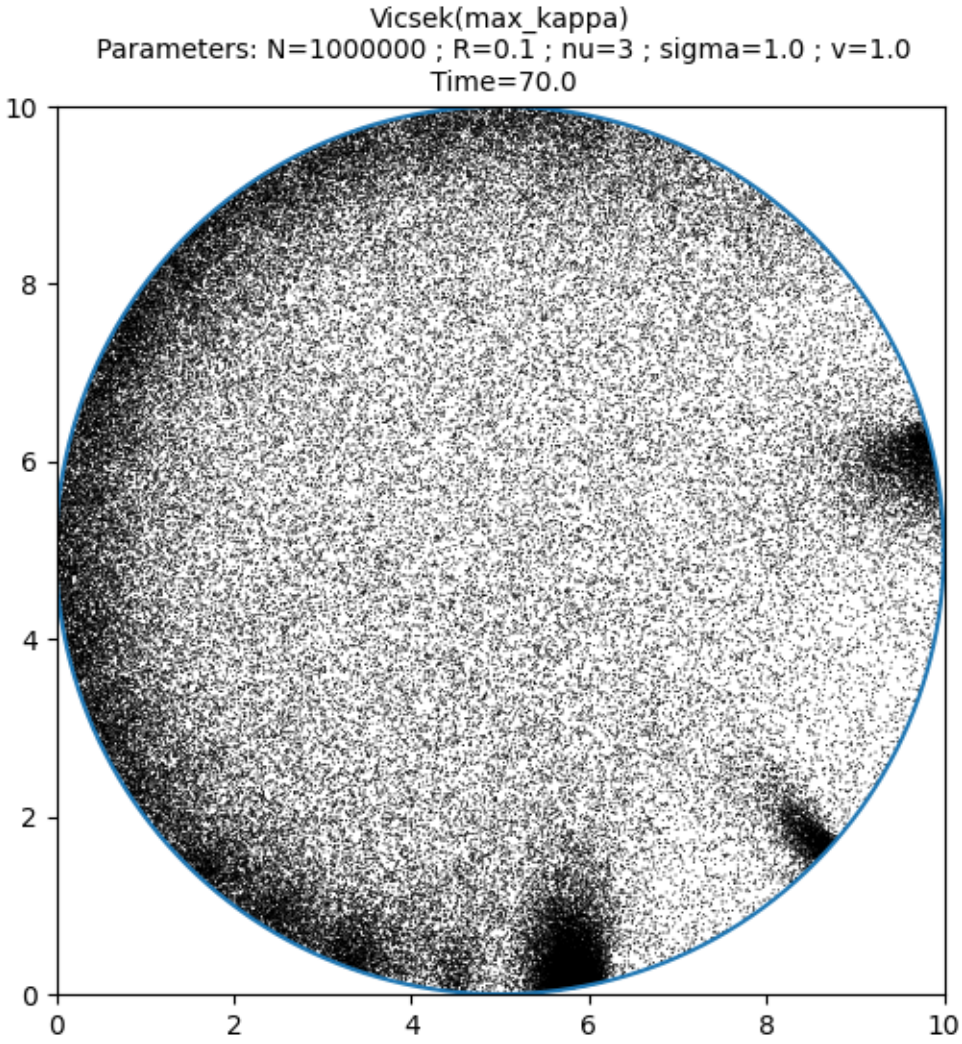


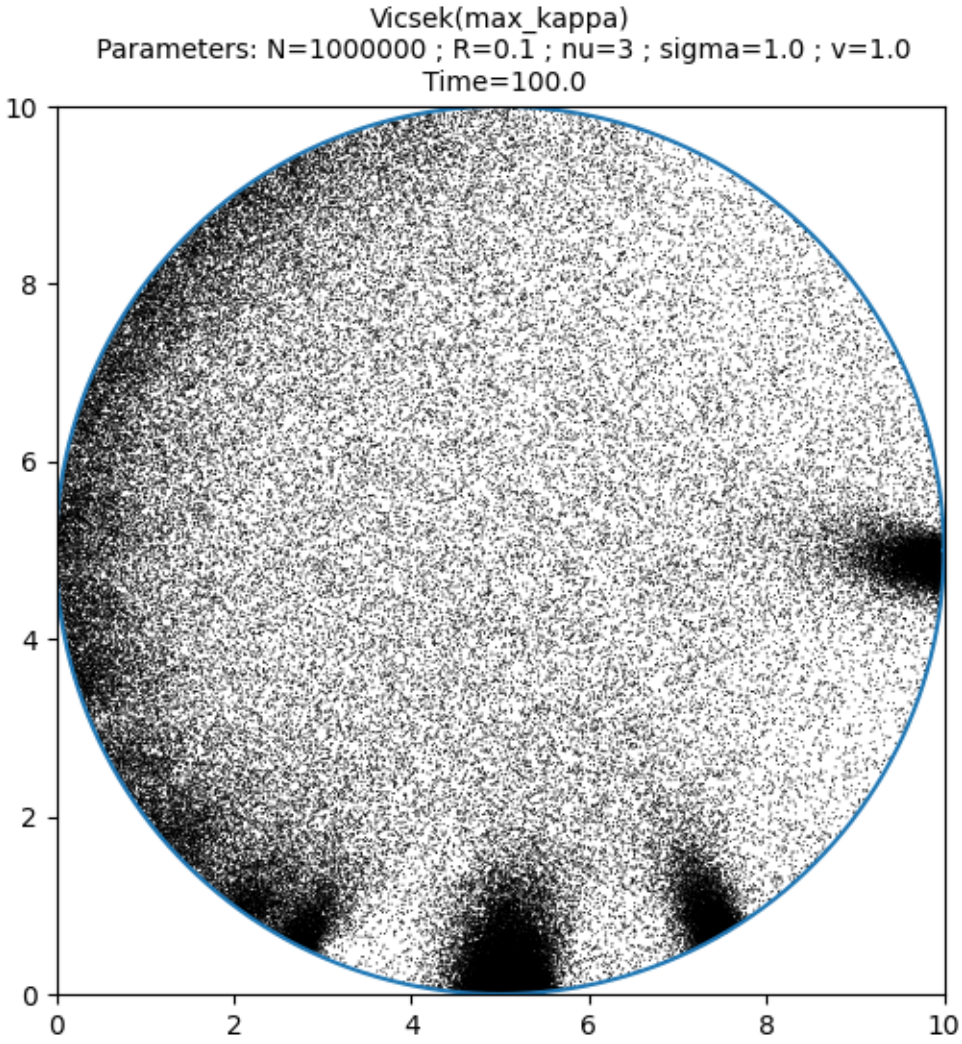
-

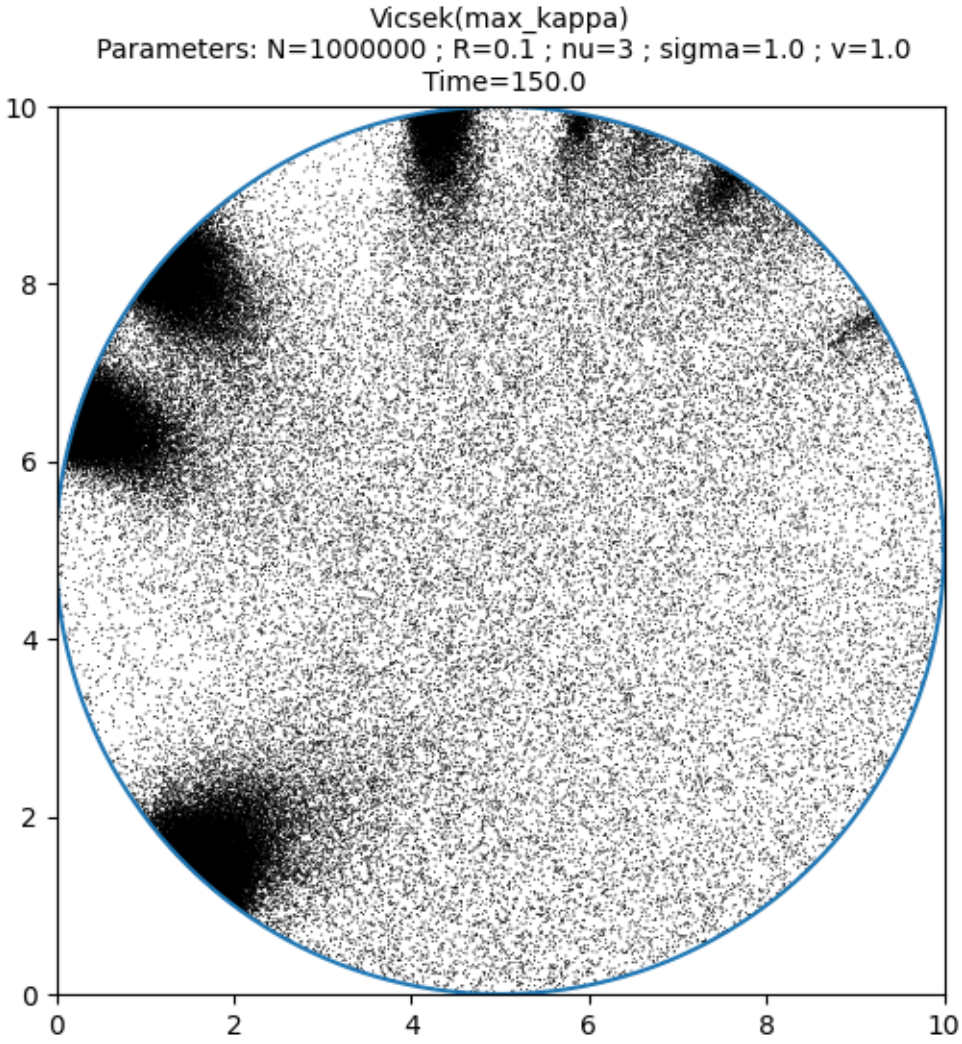

Vicsek(max_kappa)
Parameters: N=1000000 ; R=0.1 ; nu=3 ; sigma=1.0 ; v=1.0
Time=10.0
10
8
6
4
2
0
0
2
4
6
8
10

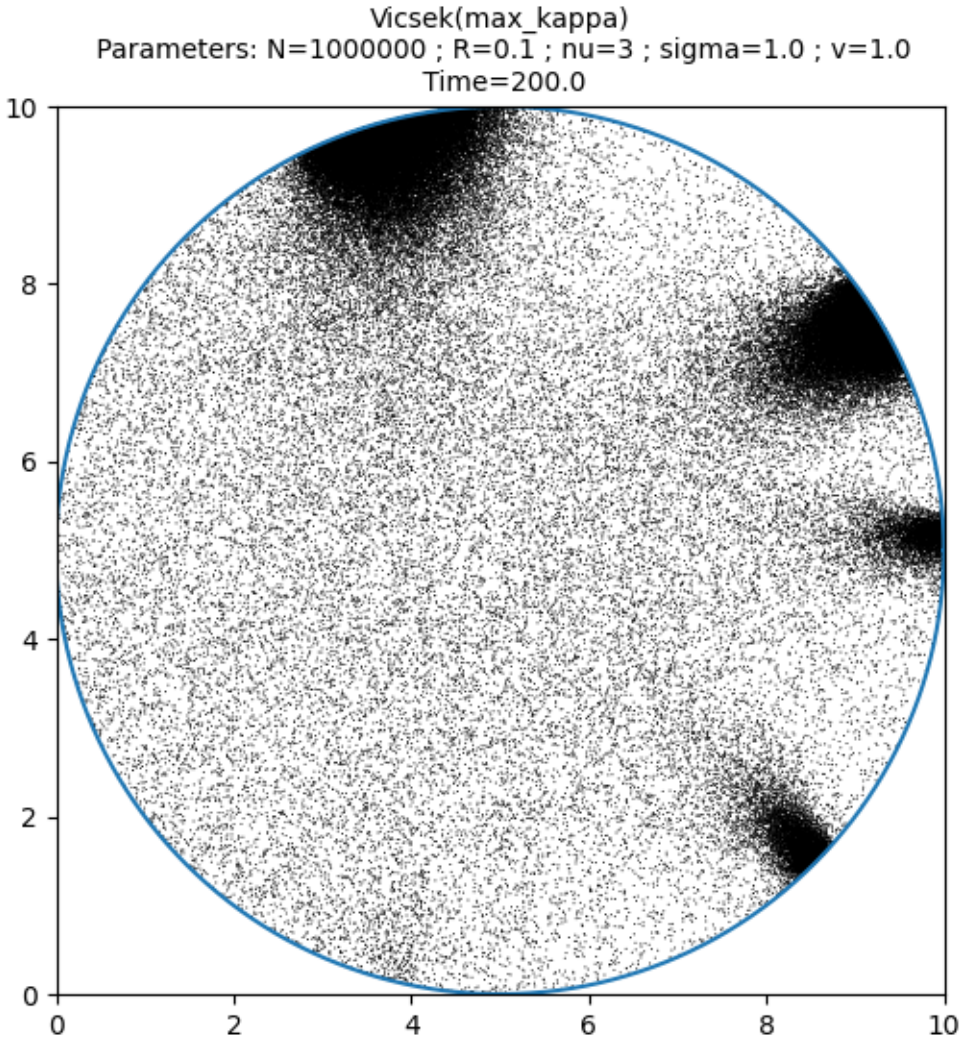

Vicsek(max_kappa)
Parameters: N=1000000 ; R=0.1 ; nu=3 ; sigma=1.0 ; v=1.0
Time=40.0
10
8
6
4
2
0
0
2
4
6
8
10

•

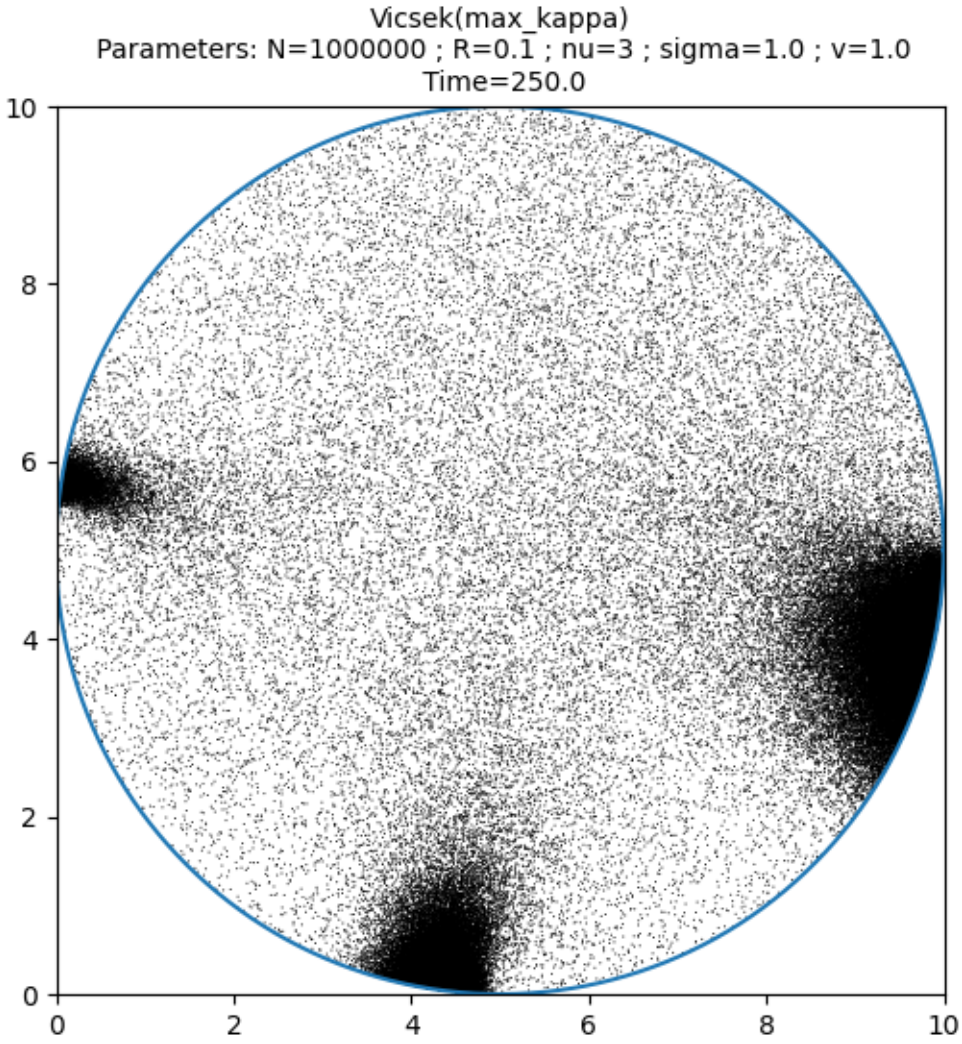

Vicsek(max_kappa)
Parameters: N=1000000 ; R=0.1 ; nu=3 ; sigma=1.0 ; v=1.0
Time=70.0
0
2
4
6
8
10

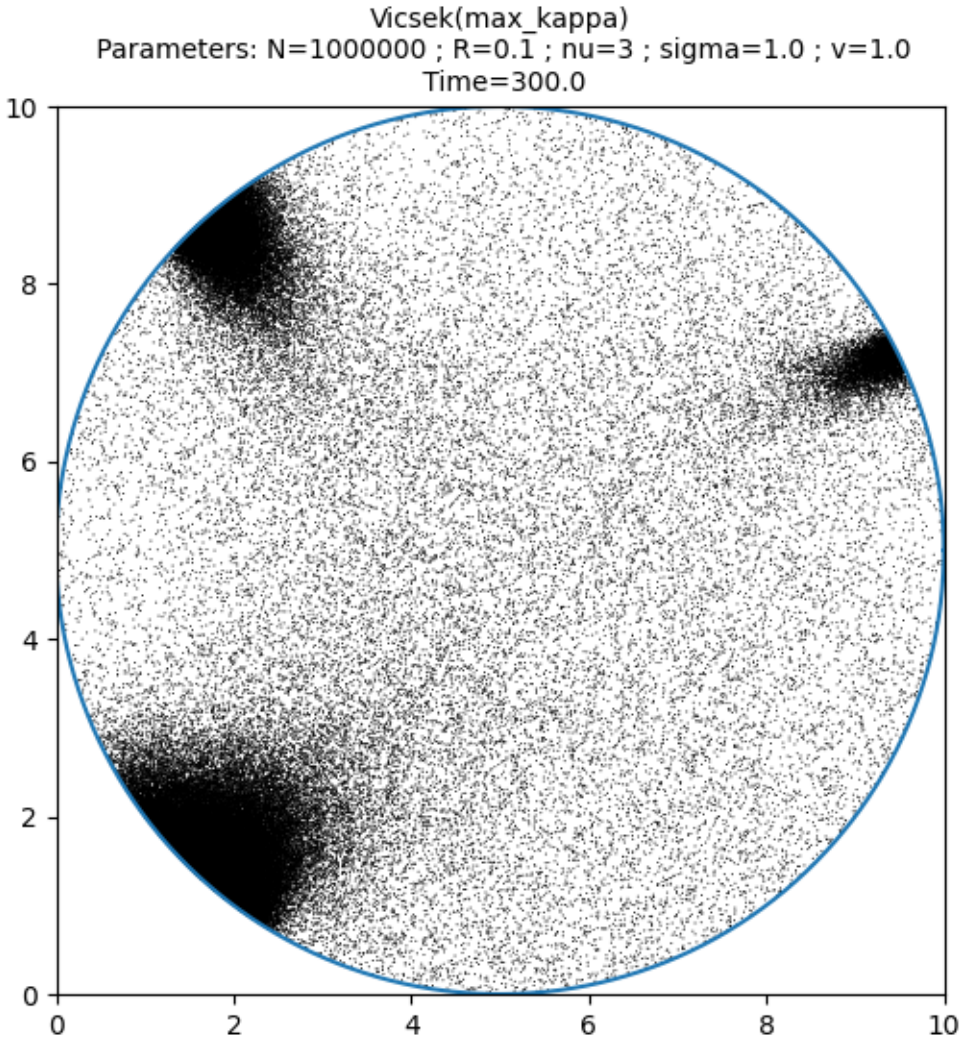

Vicsek(max_kappa)
Parameters: N=1000000 ; R=0.1 ; nu=3 ; sigma=1.0 ; v=1.0
Time=100.0
0
2
4
6
8
10


Vicsek(max_kappa)
Parameters: N=1000000 ; R=0.1 ; nu=3 ; sigma=1.0 ; v=1.0
Time=150.0
0
2
4
6
8
10


Vicsek(max_kappa)
Parameters: N=1000000 ; R=0.1 ; nu=3 ; sigma=1.0 ; v=1.0
Time=200.0
10
8
6
4
2
0
0
2
4
6
8
10

•


Vicsek(max_kappa)
Parameters: N=1000000 ; R=0.1 ; nu=3 ; sigma=1.0 ; v=1.0
Time=250.0
0
2
4
6
8
10



-

Out:

```
Progress:0%
Progress:1%
Progress:2%
Progress:3%
Progress:4%
Progress:5%
Progress:6%
Progress:7%
Progress:8%
Progress:9%
Progress:10%
Progress:11%
Progress:12%
Progress:13%
Progress:14%
```

(continues on next page)

(continued from previous page)

```
Progress:15%
Progress:16%
Progress:17%
Progress:18%
Progress:19%
Progress:20%
Progress:21%
Progress:22%
Progress:23%
Progress:24%
Progress:25%
Progress:26%
Progress:27%
Progress:28%
Progress:29%
Progress:30%
Progress:31%
Progress:32%
Progress:33%
Progress:34%
Progress:35%
Progress:36%
Progress:37%
Progress:38%
Progress:39%
Progress:40%
Progress:41%
Progress:42%
Progress:43%
Progress:44%
Progress:45%
Progress:46%
Progress:47%
Progress:48%
Progress:49%
Progress:50%
Progress:51%
Progress:52%
Progress:53%
Progress:54%
Progress:55%
Progress:56%
Progress:57%
Progress:58%
Progress:59%
Progress:60%
Progress:61%
Progress:62%
Progress:63%
Progress:64%
Progress:65%
Progress:66%
```

(continues on next page)

(continued from previous page)

```
Progress:67%
Progress:68%
Progress:69%
Progress:70%
Progress:71%
Progress:72%
Progress:73%
Progress:74%
Progress:75%
Progress:76%
Progress:77%
Progress:78%
Progress:79%
Progress:80%
Progress:81%
Progress:82%
Progress:83%
Progress:84%
Progress:85%
Progress:86%
Progress:87%
Progress:88%
Progress:89%
Progress:90%
Progress:91%
Progress:92%
Progress:93%
Progress:94%
Progress:95%
Progress:96%
Progress:97%
Progress:98%
Progress:99%
```

Print the total simulation time and the average time per iteration.

```
print('Total time: '+str(e-s)+' seconds')
print('Average time per iteration: '+str((e-s)/simu.iteration)+' seconds')
```

Out:

```
Total time: 1702.3468585014343 seconds
Average time per iteration: 0.05674489528338114 seconds
```

Total running time of the script: (30 minutes 47.639 seconds)

4.5.5 Volume exclusion

This model is introduced in

S. Motsch, D. Peurichard, From short-range repulsion to Hele-Shaw problem in a model of tumor growth, *J. Math. Biology*, Vol. 76, No. 1, 2017.

First of all, some standard imports.

```
import os
import sys
import time
import math
import torch
import numpy as np
from matplotlib import pyplot as plt
import sisyphe.models as models
from sisyphe.display import scatter_particles


use_cuda = torch.cuda.is_available()
dtype = torch.cuda.FloatTensor if use_cuda else torch.FloatTensor
```

Repulsion force

Each particle is a disk with a (fixed) random radius. The particles repel each other when they overlap. The force exerted by a particle located at x_j with radius R_j on a particle located at x_i with radius R_i is

$$F = -\frac{\alpha}{R_i}\nabla_{x_i} U\left(\frac{|x_i - x_j|^2}{(R_i + R_j)^2}\right),$$

where the potential is

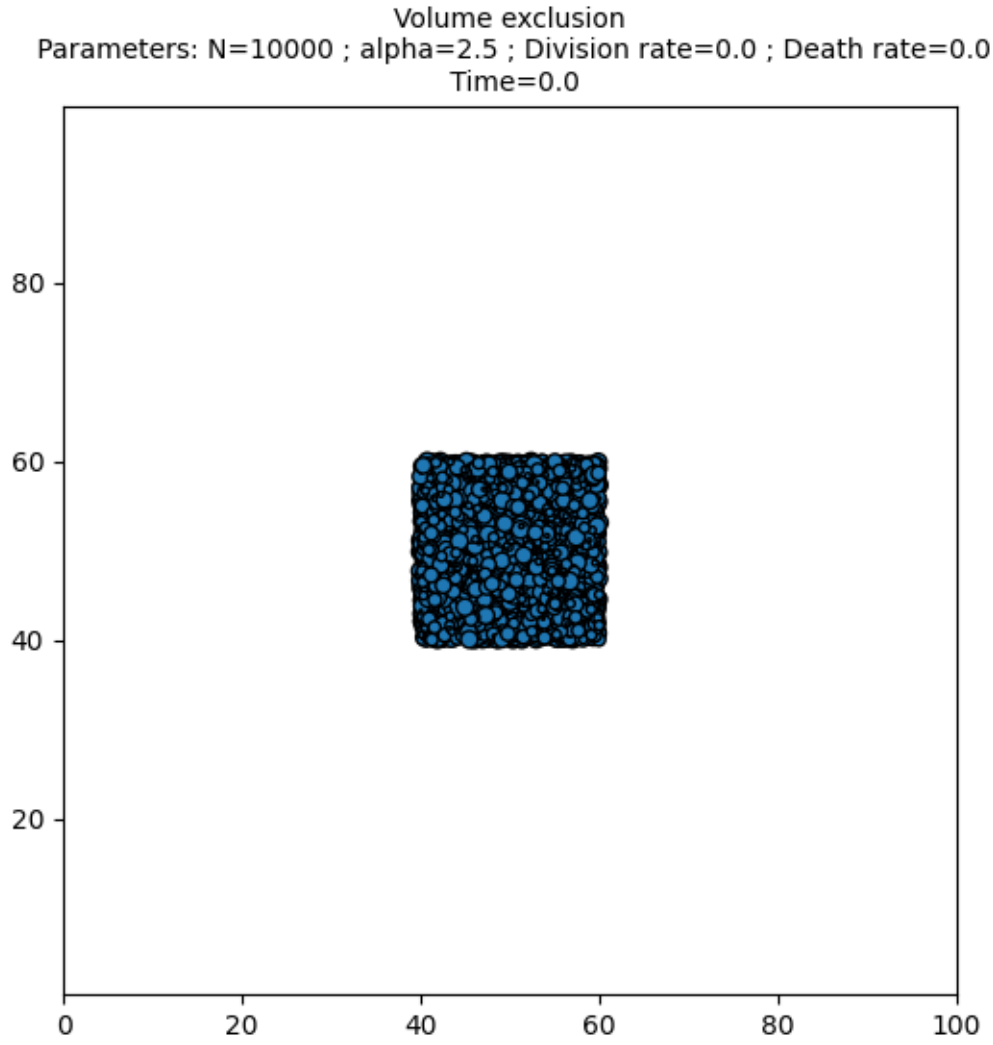
$$U(s) = -\log(s) + s - 1 \text{ for } s < 1 \text{ and } U(s) = 0 \text{ for } s > 1.$$

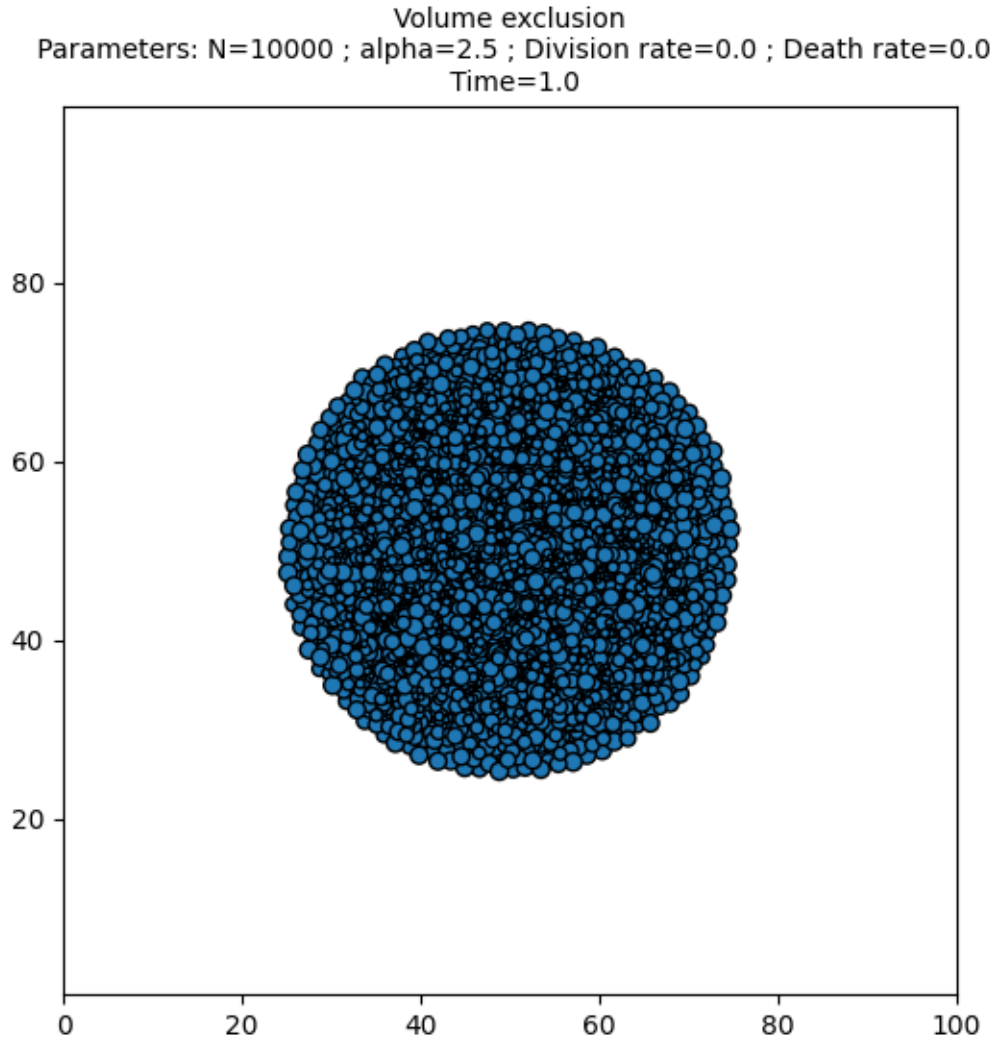
Initially, the particles are clustered in a small region with a strong overlapping.

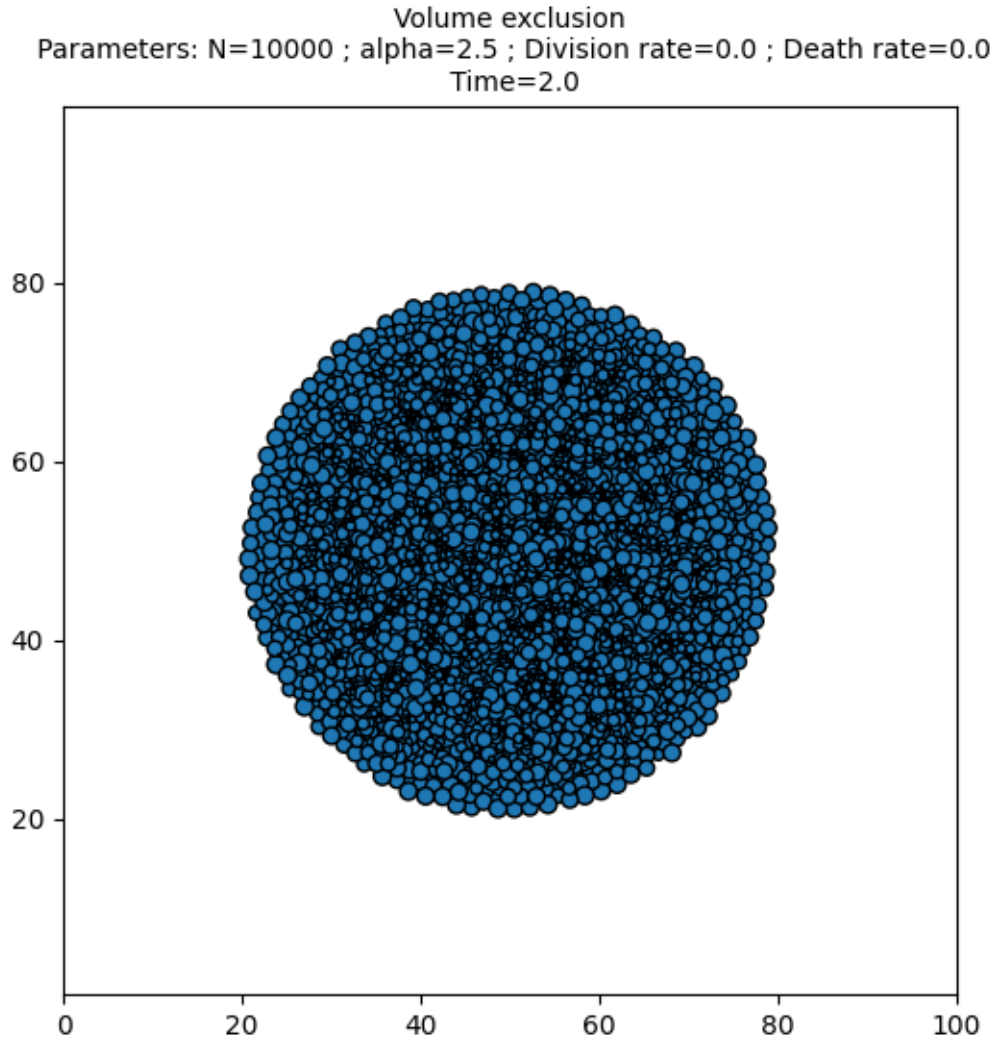
```
N = 10000
rmin = .1
rmax = 1.
R = (rmax-rmin)*torch.rand(N).type(dtype)+rmin
L = 100.
D0 = 20.
pos = (D0*torch.rand((N,2)).type(dtype)-D0/2)+torch.tensor([L/2,L/2]).type(dtype)

dt = .1

simu = models.VolumeExclusion(pos=pos,
                  interaction_radius=R,
                  box_size=L,
                  alpha=2.5,
                  division_rate=0.,
                  death_rate=0.,
                  dt=dt)
```

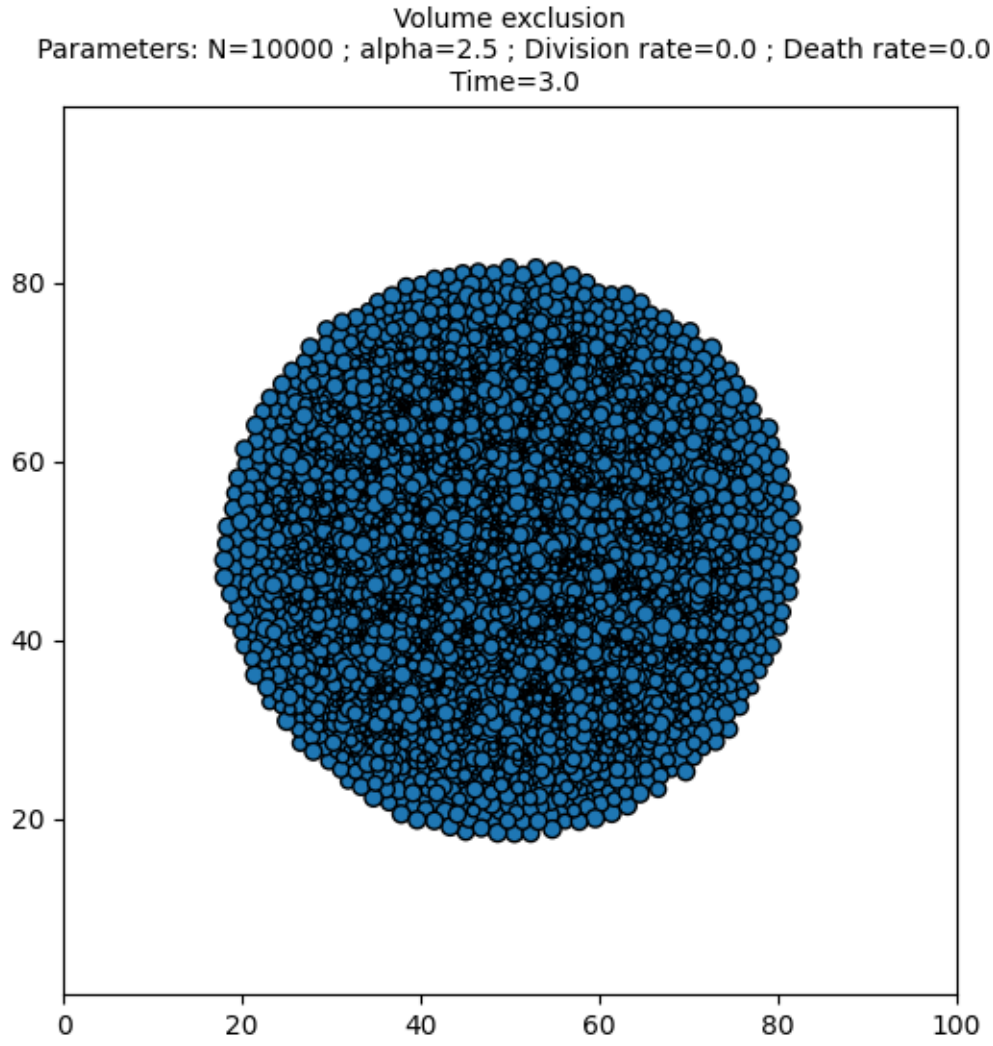
Run the simulation over 200 units of time using an adaptive time-step which ensures that the energy `E` of the system decreases.

```
# sphinx_gallery_thumbnail_number = 13

frames = [0,1,2,3,4,5,10,30,50,75,100,150,200]

s = time.time()
scatter_particles(simu,frames)
e = time.time()
```

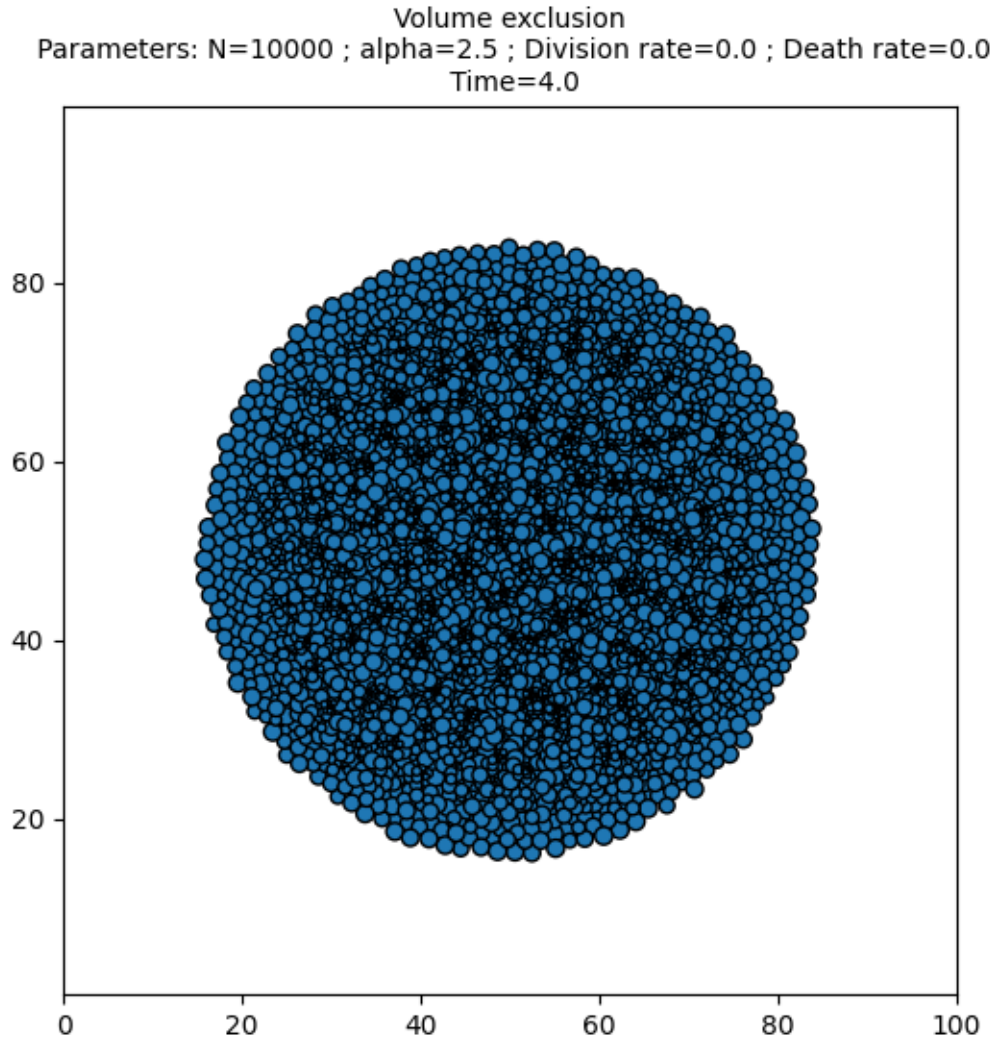



Volume exclusion
Parameters: N=10000 ; alpha=2.5 ; Division rate=0.0 ; Death rate=0.0
Time=1.0
80
60
40
20
0
20
40
60
80
100

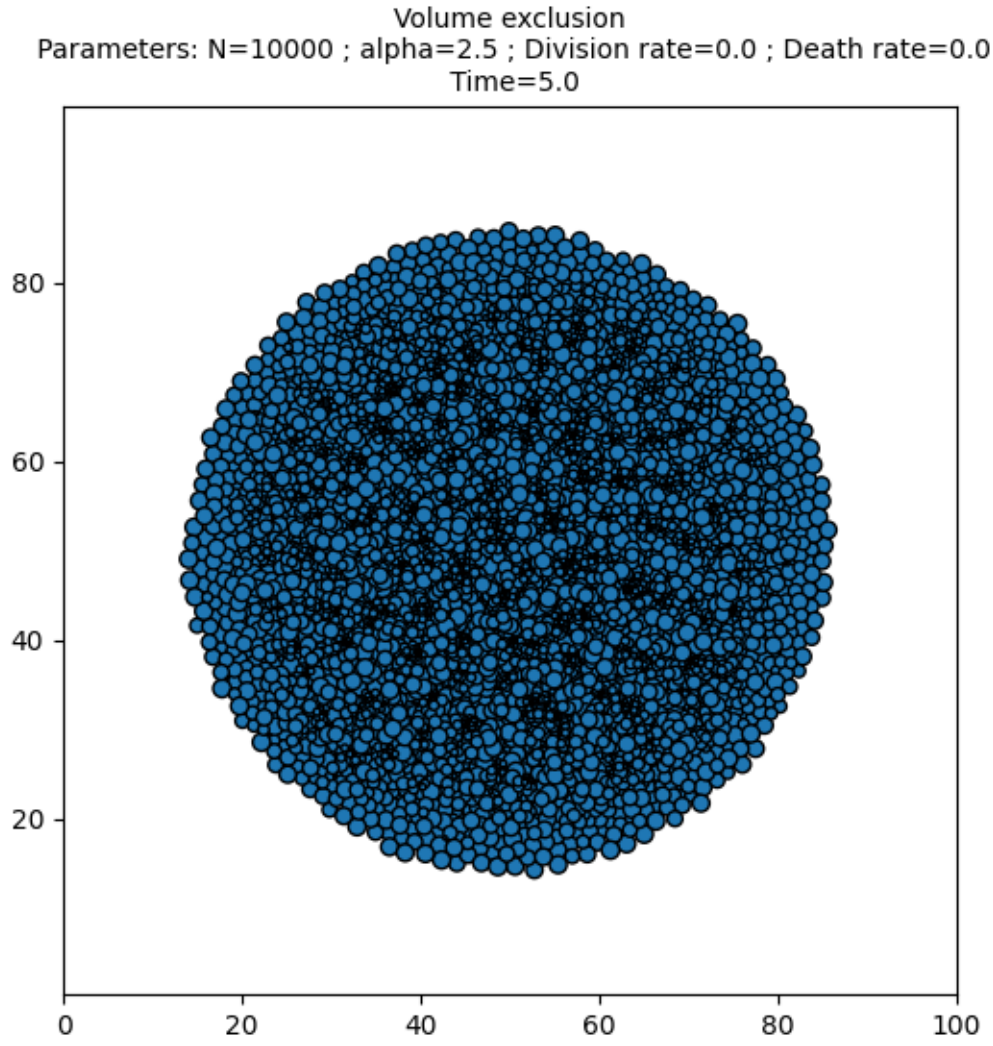

Volume exclusion
Parameters: N=10000 ; alpha=2.5 ; Division rate=0.0 ; Death rate=0.0
Time=2.0
80
60
40
20
0
20
40
60
80
100


Volume exclusion
Parameters: N=10000 ; alpha=2.5 ; Division rate=0.0 ; Death rate=0.0
Time=3.0
80
60
40
20
0
20
40
60
80
100

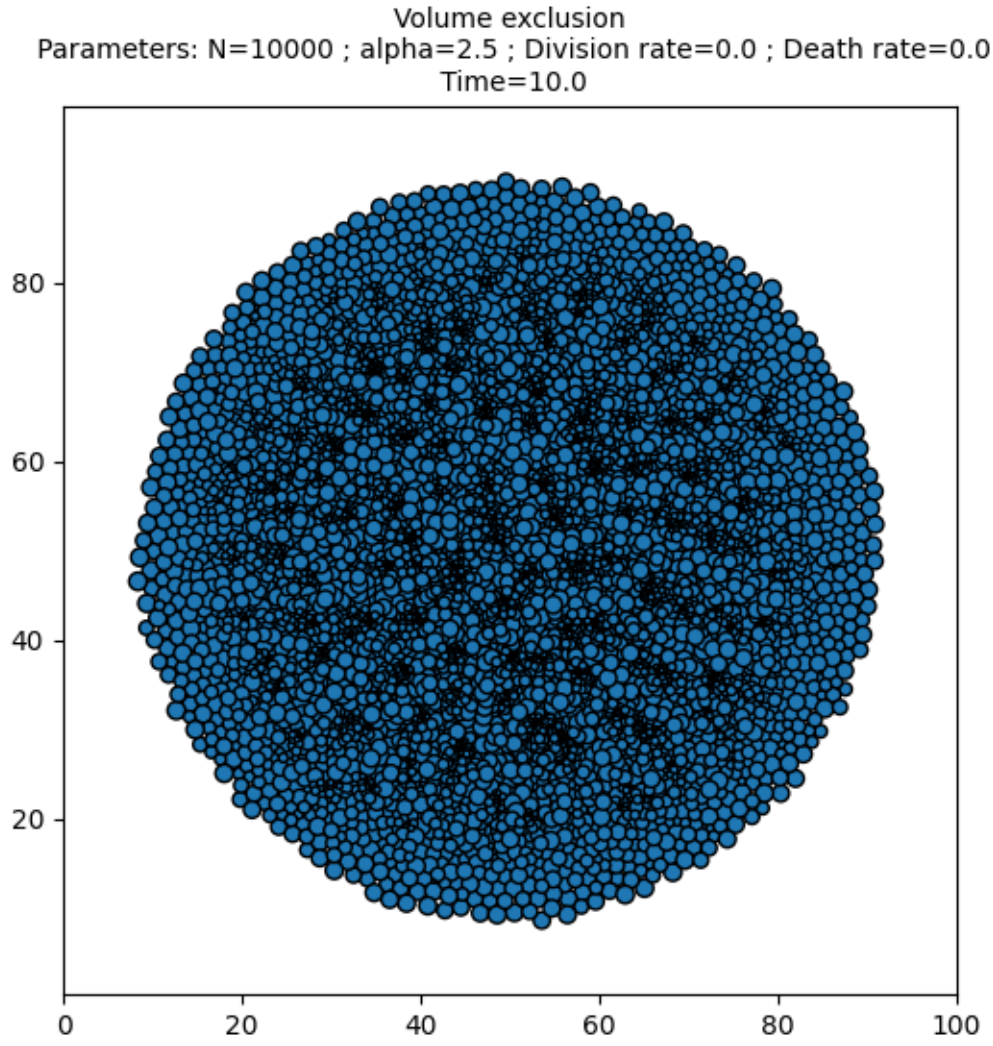
•


Volume exclusion
Parameters: N=10000 ; alpha=2.5 ; Division rate=0.0 ; Death rate=0.0
Time=4.0
80
60
40
20
0
20
40
60
80
100

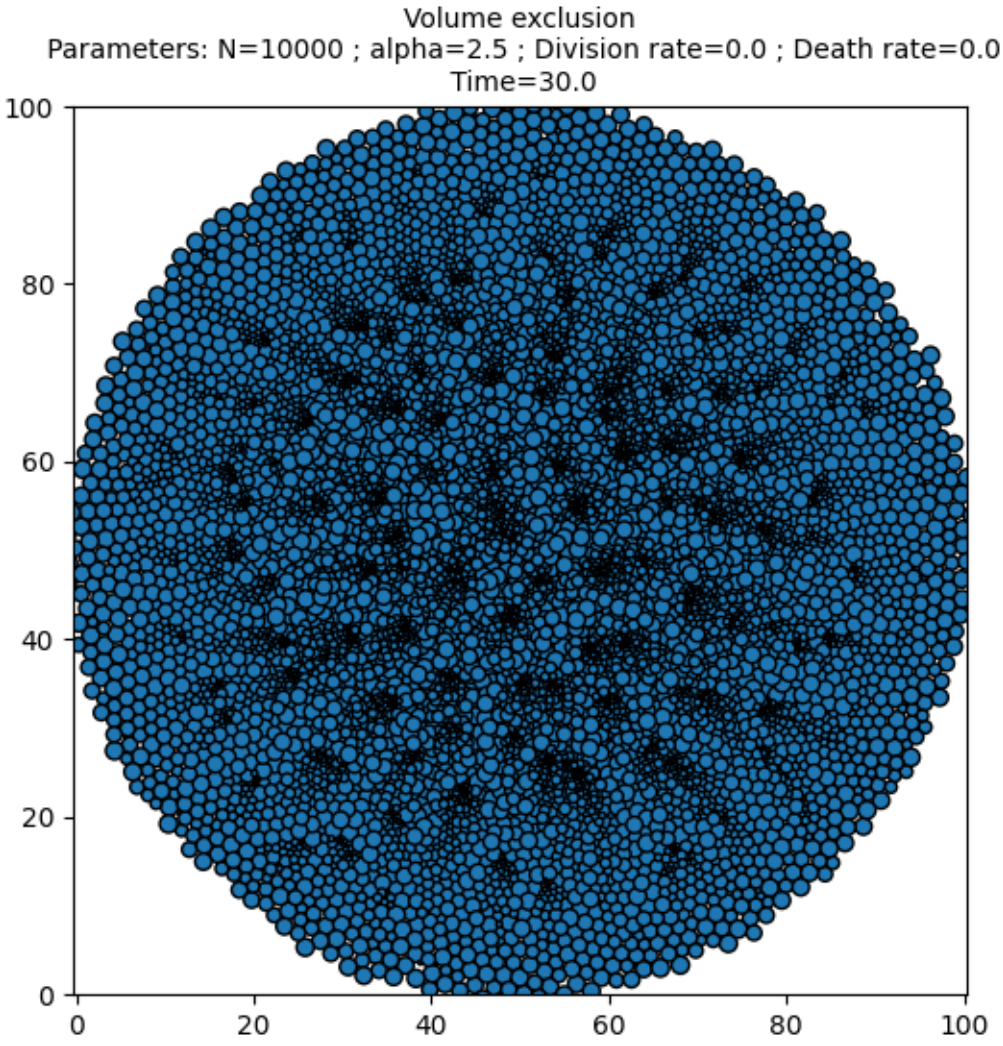
•


Volume exclusion
Parameters: N=10000 ; alpha=2.5 ; Division rate=0.0 ; Death rate=0.0
Time=5.0
80
60
40
20
0
20
40
60
80
100

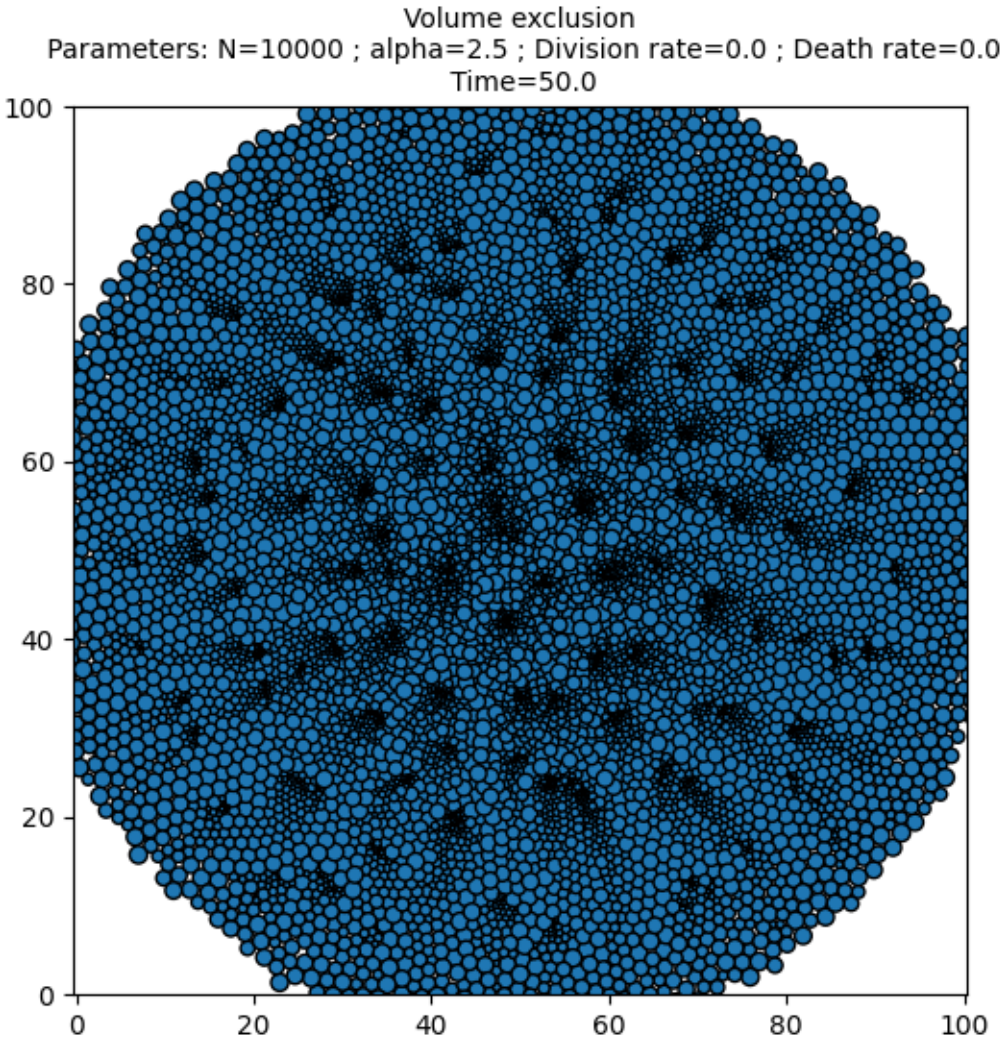
•


Volume exclusion
Parameters: N=10000 ; alpha=2.5 ; Division rate=0.0 ; Death rate=0.0
Time=10.0
80
60
40
20
0
20
40
60
80
100

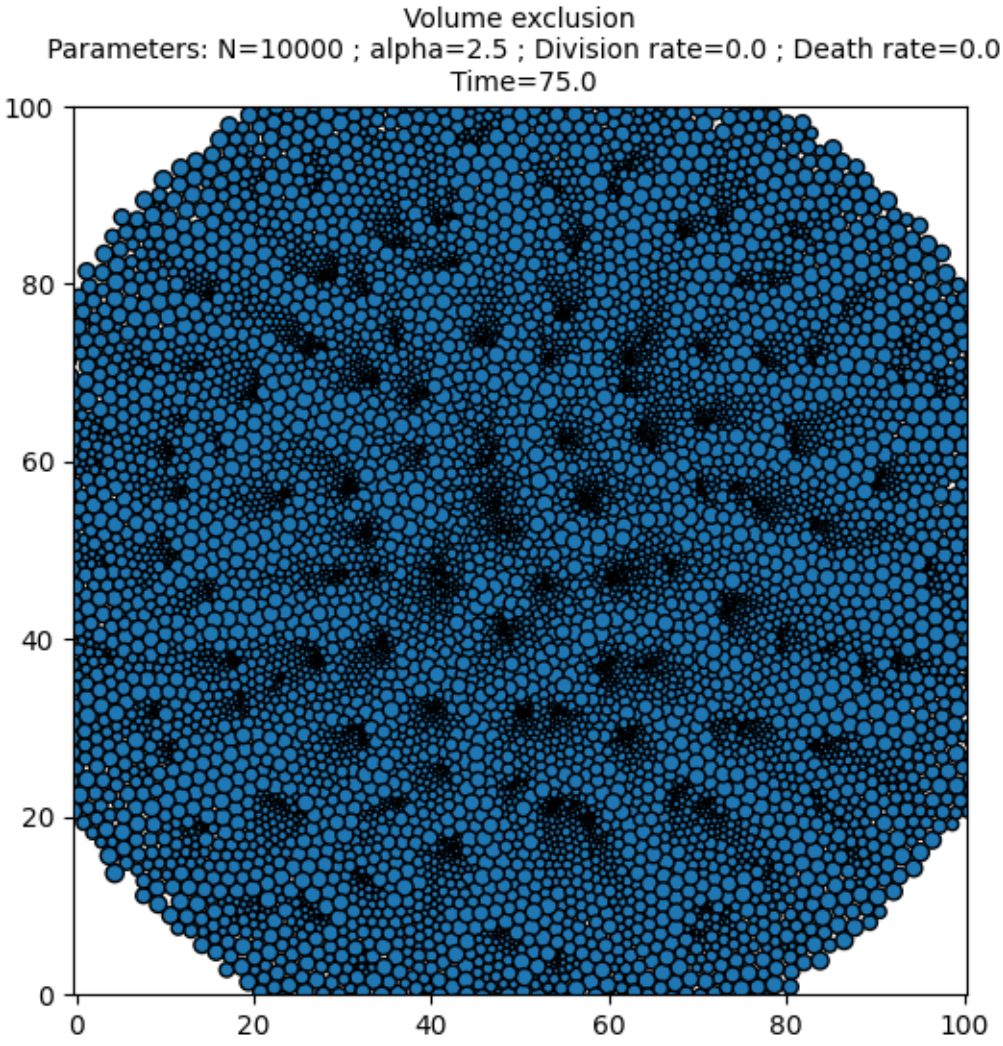
•


Volume exclusion
Parameters: N=10000 ; alpha=2.5 ; Division rate=0.0 ; Death rate=0.0
Time=30.0
100
80
60
40
20
0
0
20
40
60
80
100

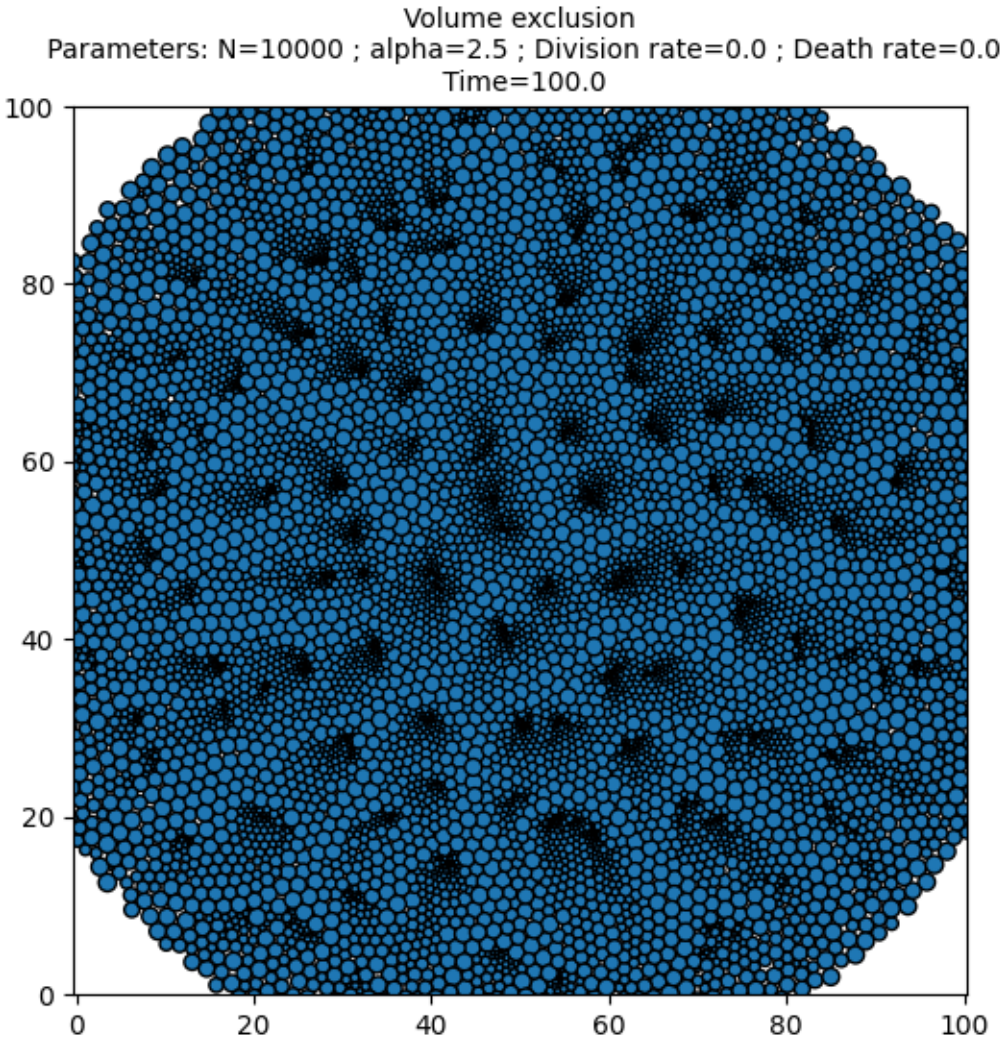
•

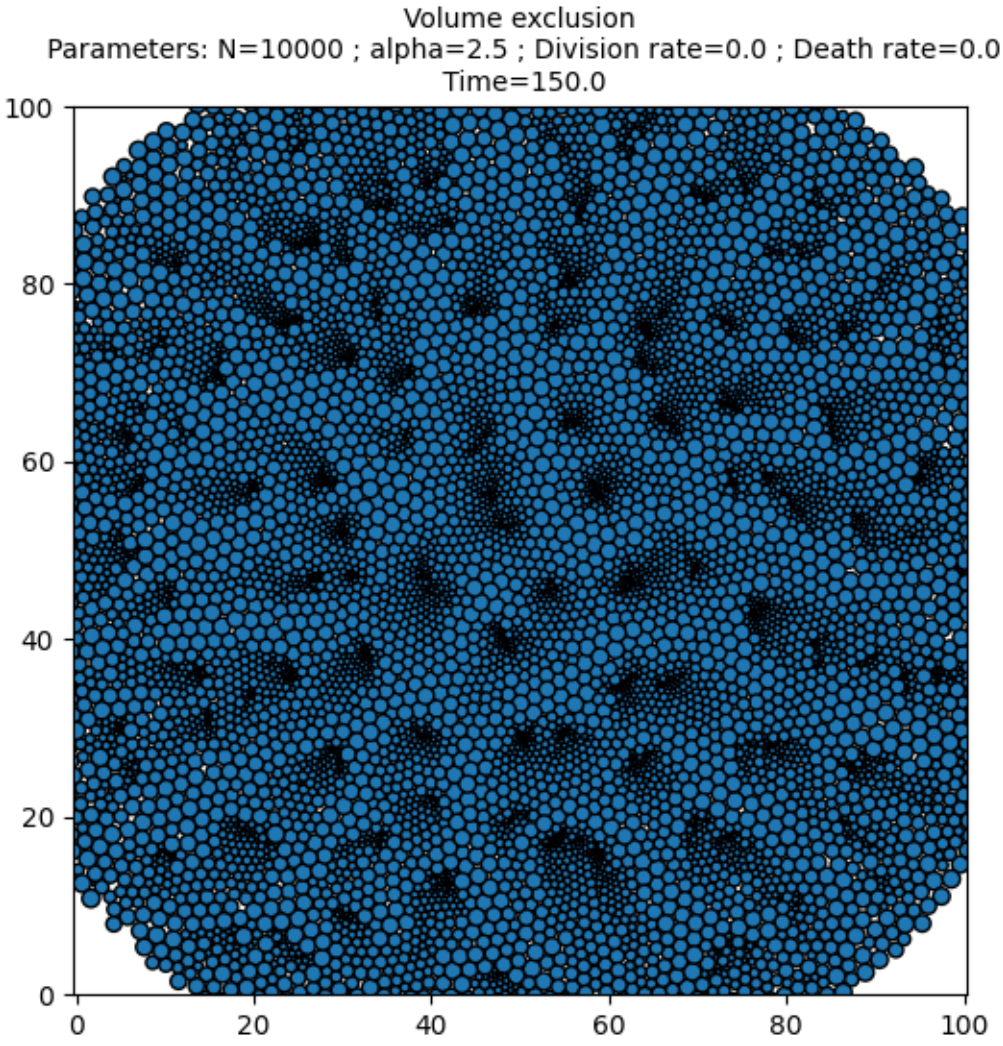

Volume exclusion
Parameters: N=10000 ; alpha=2.5 ; Division rate=0.0 ; Death rate=0.0
Time=50.0
100
80
60
40
20
0
0
20
40
60
80
100

•

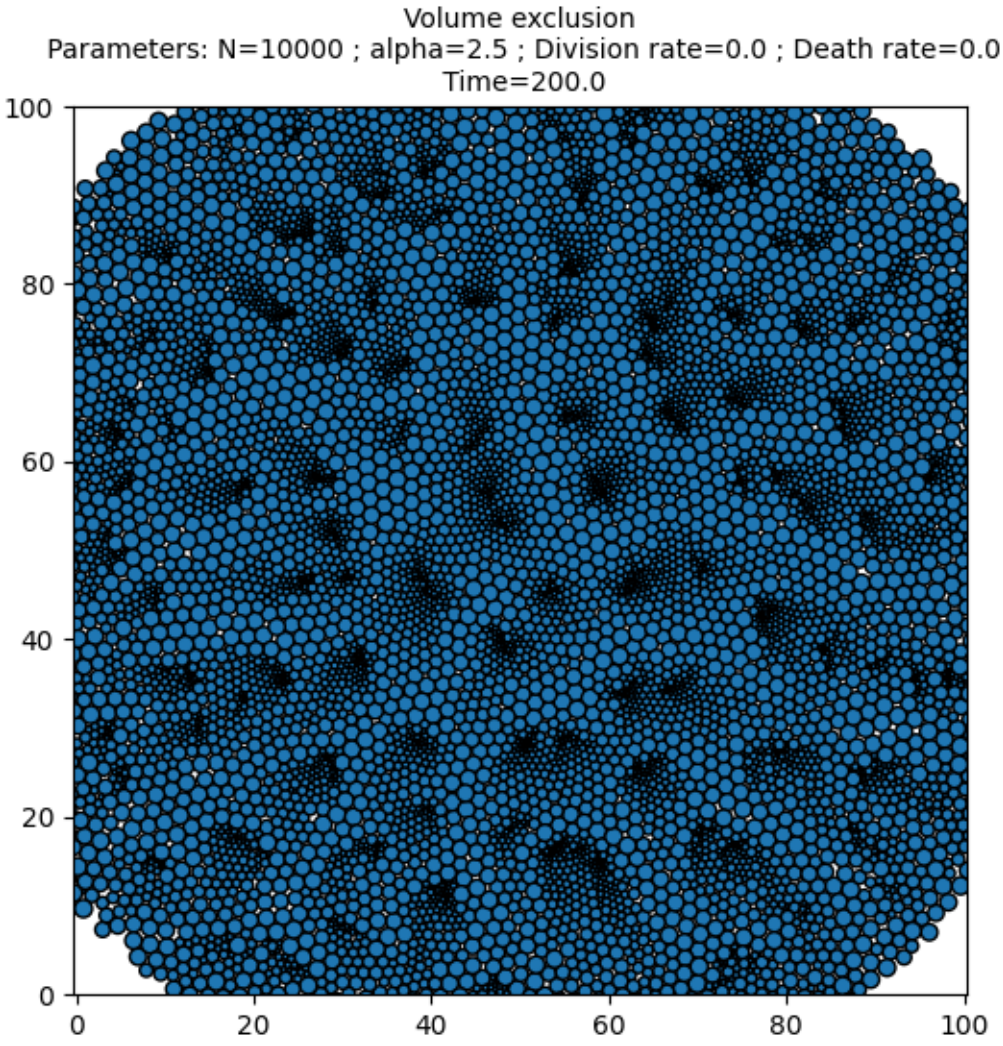

Volume exclusion
Parameters: N=10000 ; alpha=2.5 ; Division rate=0.0 ; Death rate=0.0
Time=75.0
100
80
60
40
20
0
0
20
40
60
80
100

•


Volume exclusion
Parameters: N=10000 ; alpha=2.5 ; Division rate=0.0 ; Death rate=0.0
Time=100.0
100
80
60
40
20
0
0
20
40
60
80
100


Volume exclusion
Parameters: N=10000 ; alpha=2.5 ; Division rate=0.0 ; Death rate=0.0
Time=150.0
100
80
60
40
20
0
0
20
40
60
80
100

-



-

Out:

```
Progress:0%
Progress:1%
Progress:2%
Progress:3%
Progress:4%
Progress:5%
Progress:6%
Progress:7%
Progress:8%
Progress:9%
Progress:10%
Progress:11%
Progress:12%
Progress:13%
Progress:14%
```

(continues on next page)

(continued from previous page)

```
Progress:15%
Progress:16%
Progress:17%
Progress:18%
Progress:19%
Progress:20%
Progress:21%
Progress:22%
Progress:23%
Progress:24%
Progress:25%
Progress:26%
Progress:27%
Progress:28%
Progress:29%
Progress:30%
Progress:31%
Progress:32%
Progress:33%
Progress:34%
Progress:35%
Progress:36%
Progress:37%
Progress:38%
Progress:39%
Progress:40%
Progress:41%
Progress:42%
Progress:43%
Progress:44%
Progress:45%
Progress:46%
Progress:47%
Progress:48%
Progress:49%
Progress:50%
Progress:51%
Progress:52%
Progress:53%
Progress:54%
Progress:55%
Progress:56%
Progress:57%
Progress:58%
Progress:59%
Progress:60%
Progress:61%
Progress:62%
Progress:63%
Progress:64%
Progress:65%
Progress:66%
```

(continues on next page)

(continued from previous page)

```
Progress:67%
Progress:68%
Progress:69%
Progress:70%
Progress:71%
Progress:72%
Progress:73%
Progress:74%
Progress:75%
Progress:76%
Progress:77%
Progress:78%
Progress:79%
Progress:80%
Progress:81%
Progress:82%
Progress:83%
Progress:84%
Progress:85%
Progress:86%
Progress:87%
Progress:88%
Progress:89%
Progress:90%
Progress:91%
Progress:92%
Progress:93%
Progress:94%
Progress:95%
Progress:96%
Progress:97%
Progress:98%
Progress:99%
Progress:100%
```

Print the total simulation time and the average time per iteration.

```
print('Total time: '+str(e-s)+' seconds')
print('Average time per iteration: '+str((e-s)/simu.iteration)+' seconds')
```

Out:

```
Total time: 326.2268440723419 seconds
Average time per iteration: 0.01703801347847401 seconds
```

Note this funny behaviour: the particles are clustered by size!

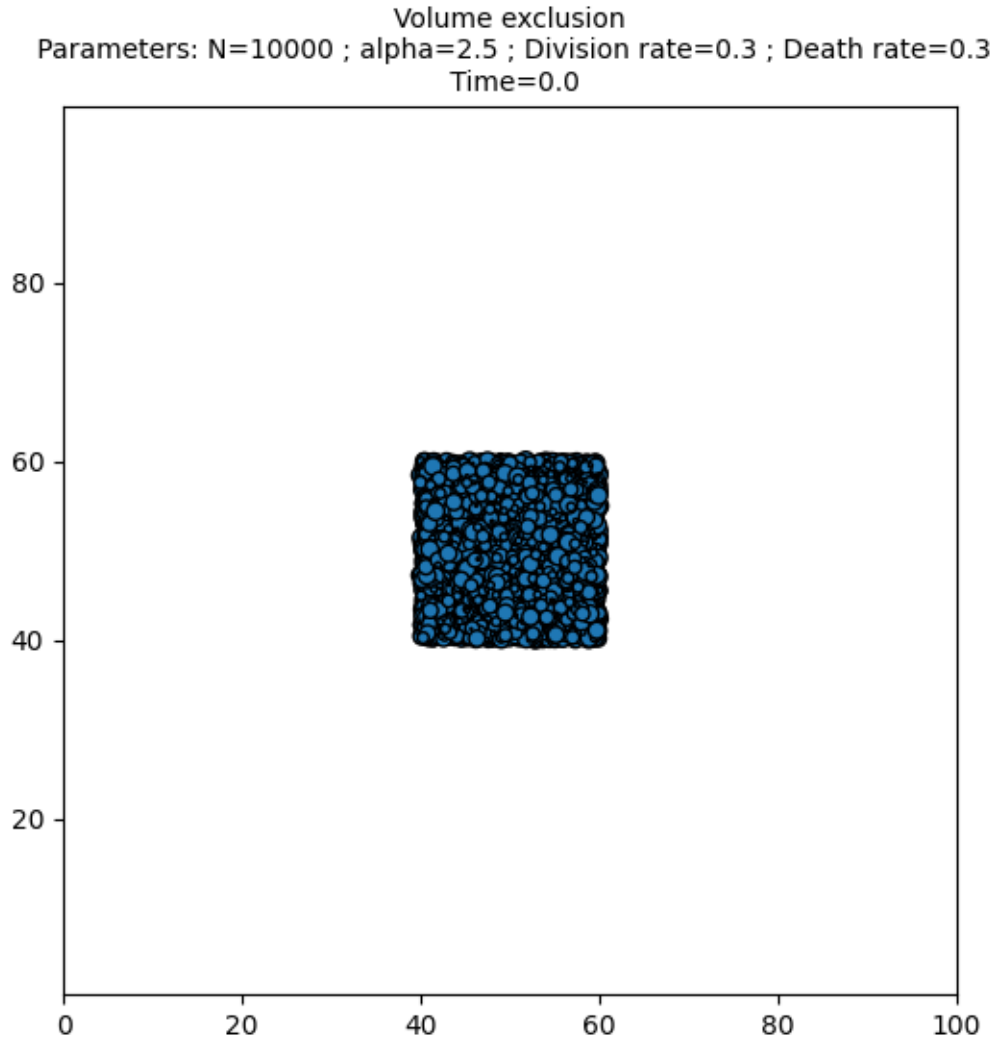
Repulsion force, random births and random deaths

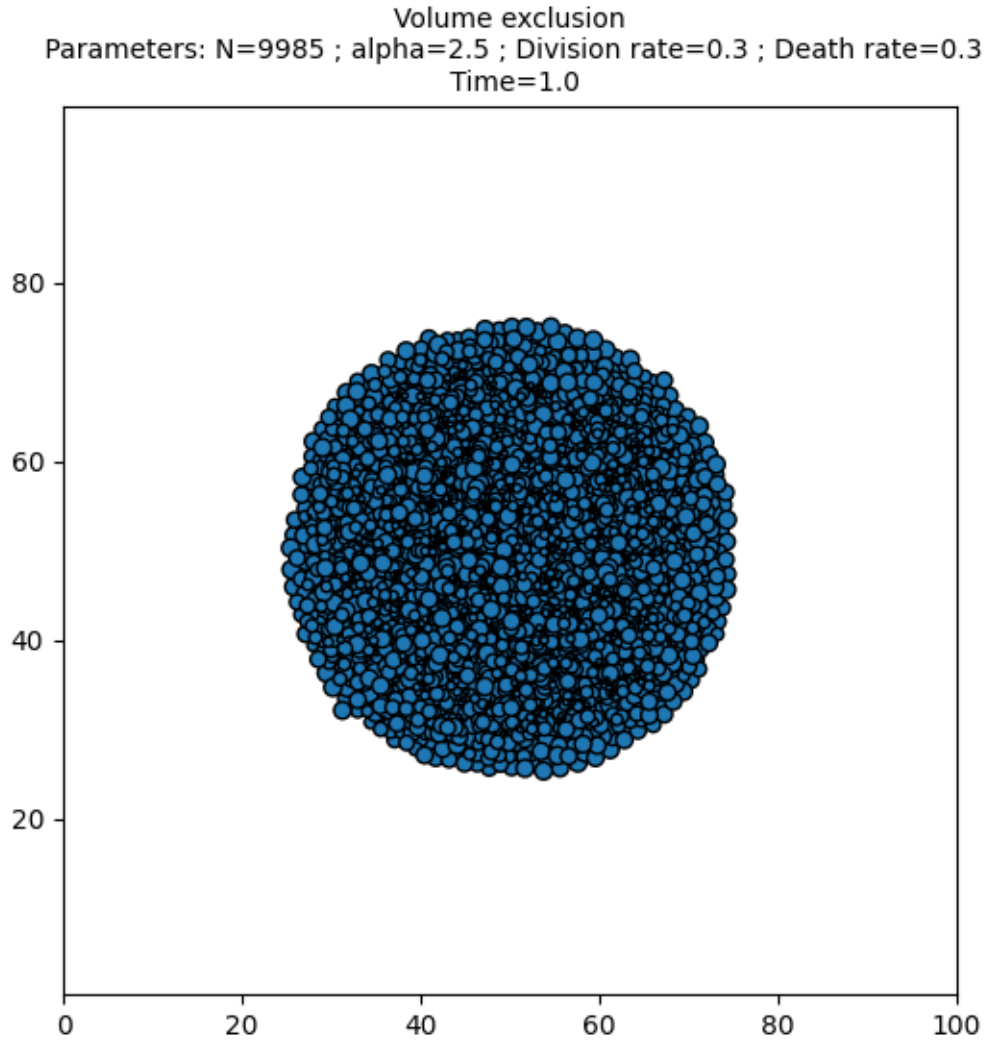
Same system but this time, particles die at a constant rate and give birth to new particles at the same rate. A new particle is added next to its parent and has the same radius.

```
N = 10000
rmin = .1
rmax = 1.
R = (rmax-rmin)*torch.rand(N).type(dtype)+rmin
L = 100.
D0 = 20.
pos = (D0*torch.rand((N,2)).type(dtype)-D0/2)+torch.tensor([L/2,L/2]).type(dtype)

dt = .1

simu = models.VolumeExclusion(pos=pos,
                  interaction_radius=R,
                  box_size=L,
                  alpha=2.5,
                  division_rate=.3,
                  death_rate=.3,
                  dt=dt,
                  Nmax = 20000)
```

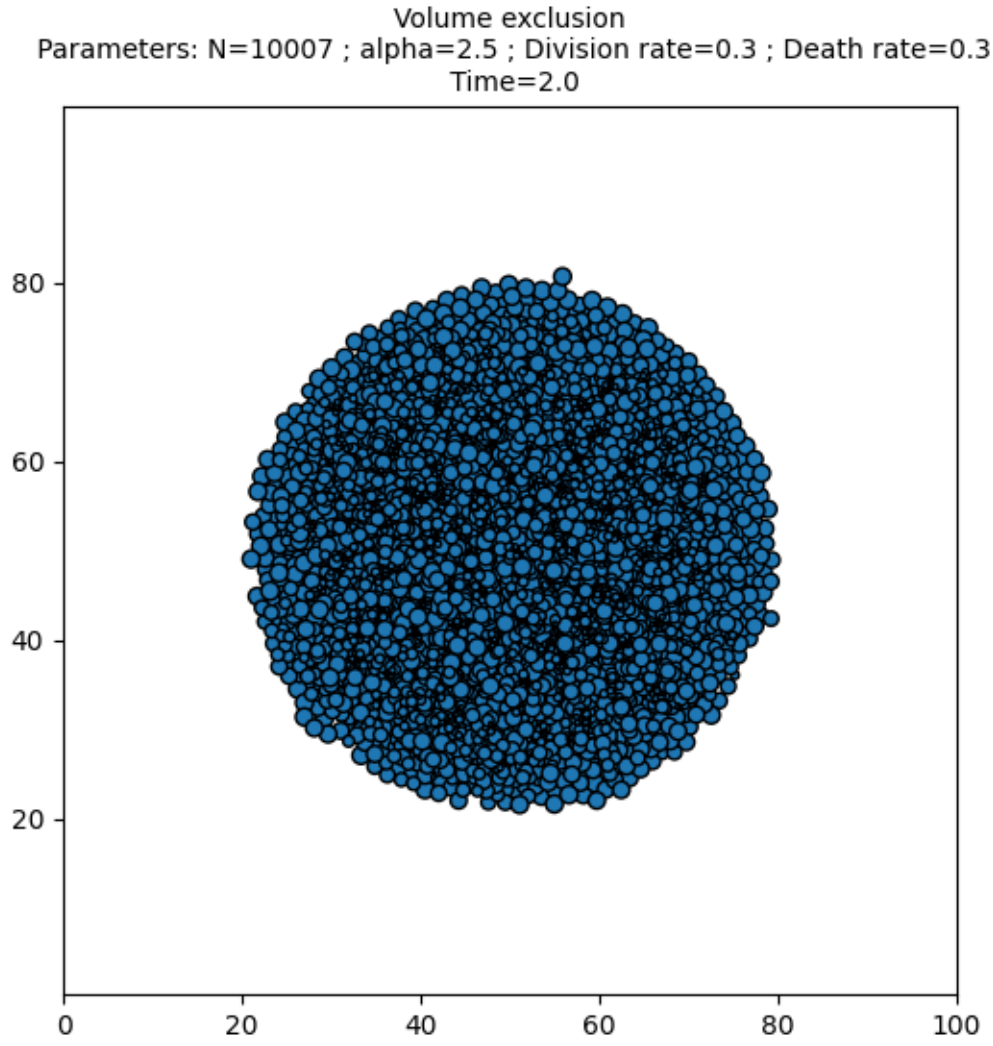
Run the simulation over 200 units of time using an adaptive time-step which ensures that the energy E of the system decreases.

```
frames = [0,1,2,3,4,5,10,30,50,75,100,150,200]

s = time.time()
scatter_particles(simu,frames)
e = time.time()
```

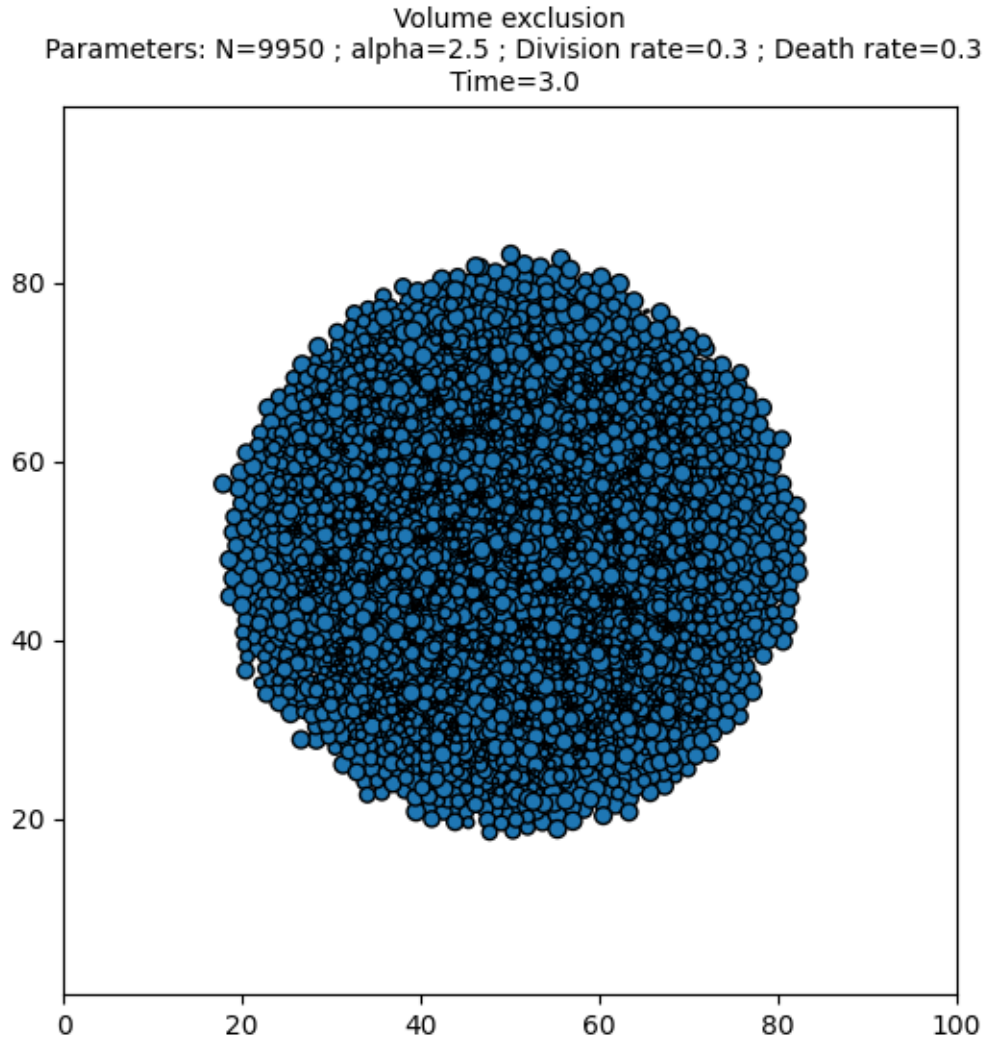

Volume exclusion
Parameters: N=10000 ; alpha=2.5 ; Division rate=0.3 ; Death rate=0.3
Time=0.0
80
60
40
20
0
20
40
60
80
100

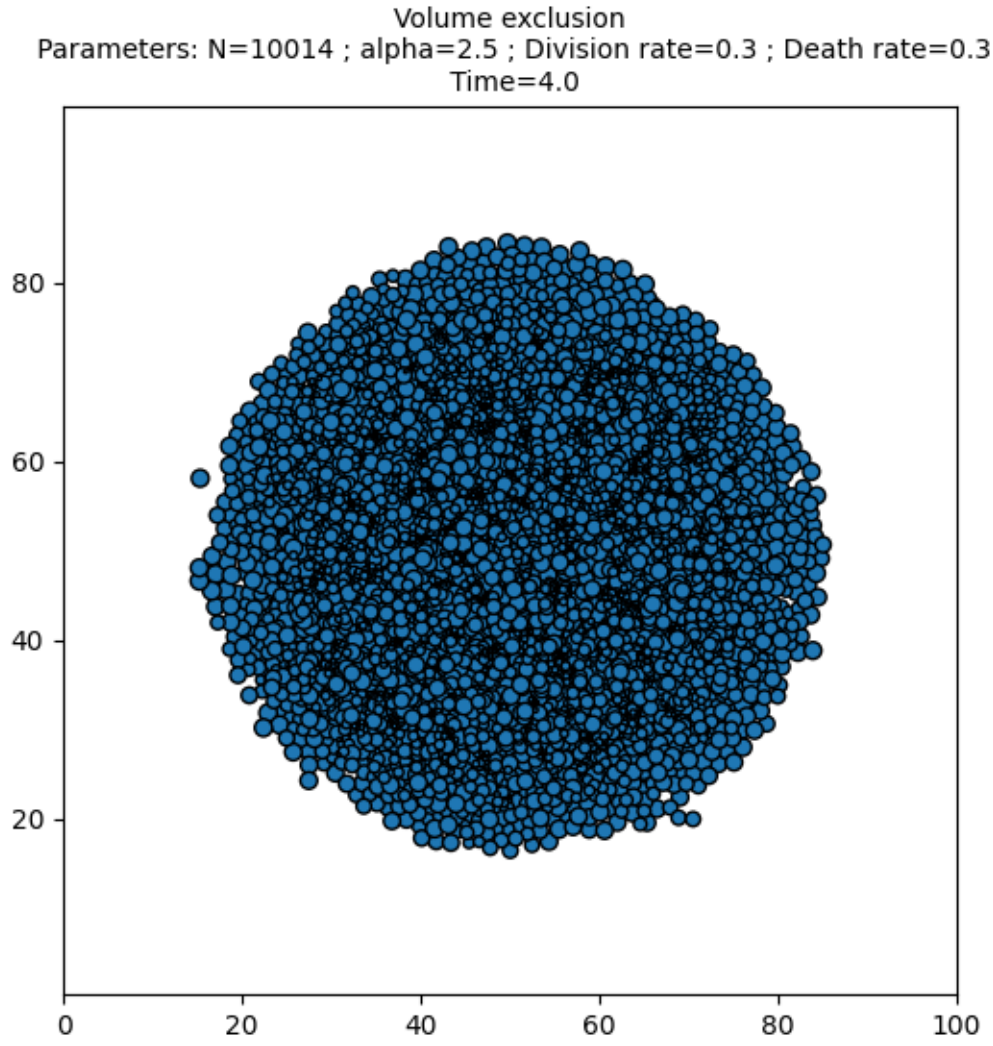

Volume exclusion
Parameters: N=9985 ; alpha=2.5 ; Division rate=0.3 ; Death rate=0.3
Time=1.0
80
60
40
20
0
20
40
60
80
100

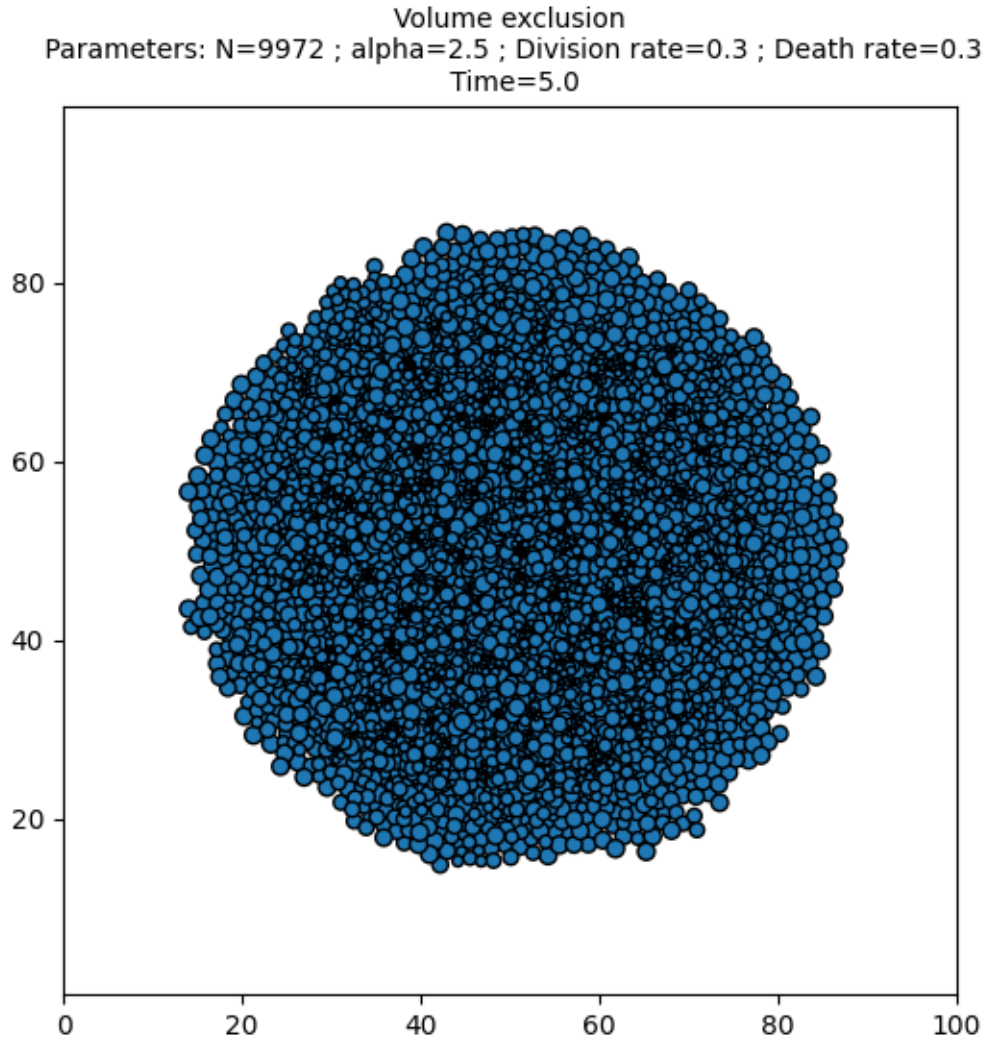
•


Volume exclusion
Parameters: N=10007 ; alpha=2.5 ; Division rate=0.3 ; Death rate=0.3
Time=2.0
80
60
40
20
0
20
40
60
80
100

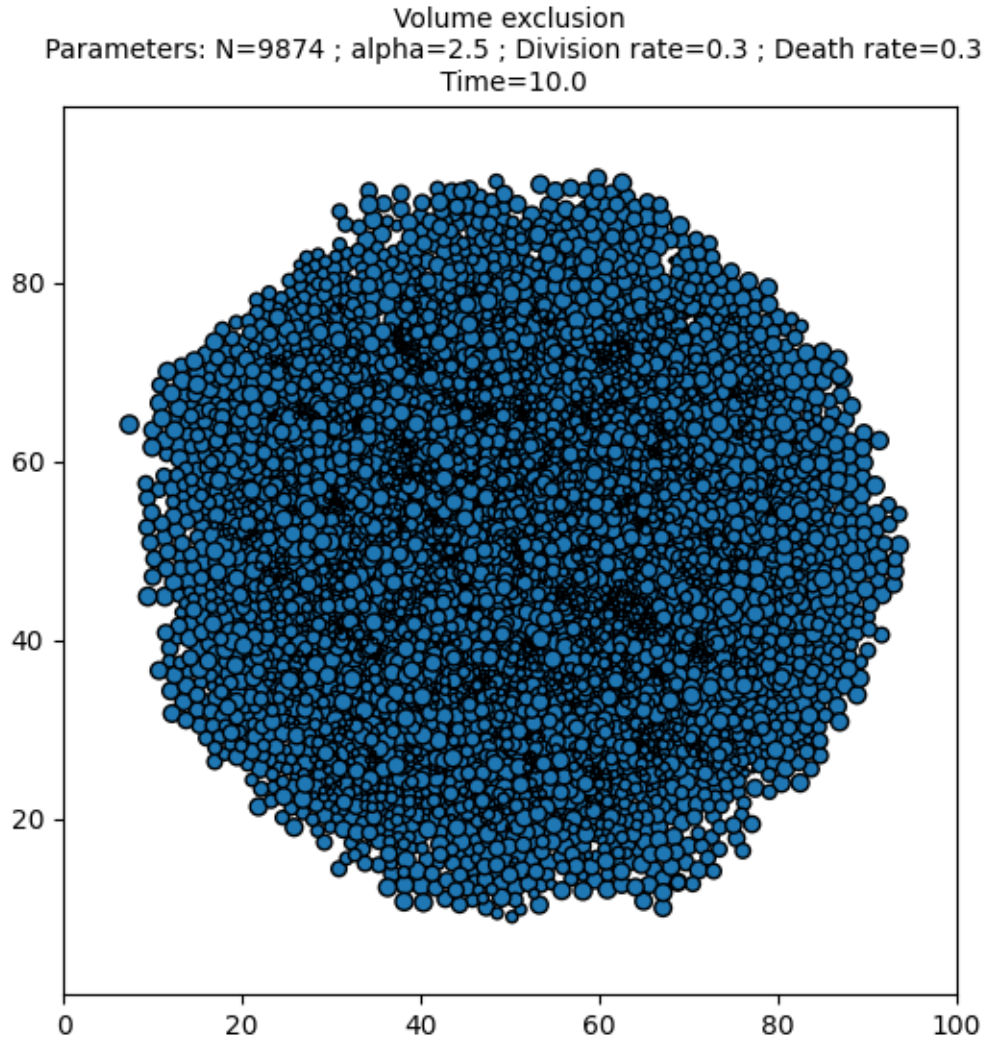
•


Volume exclusion
Parameters: N=9950 ; alpha=2.5 ; Division rate=0.3 ; Death rate=0.3
Time=3.0
80
60
40
20
0
20
40
60
80
100

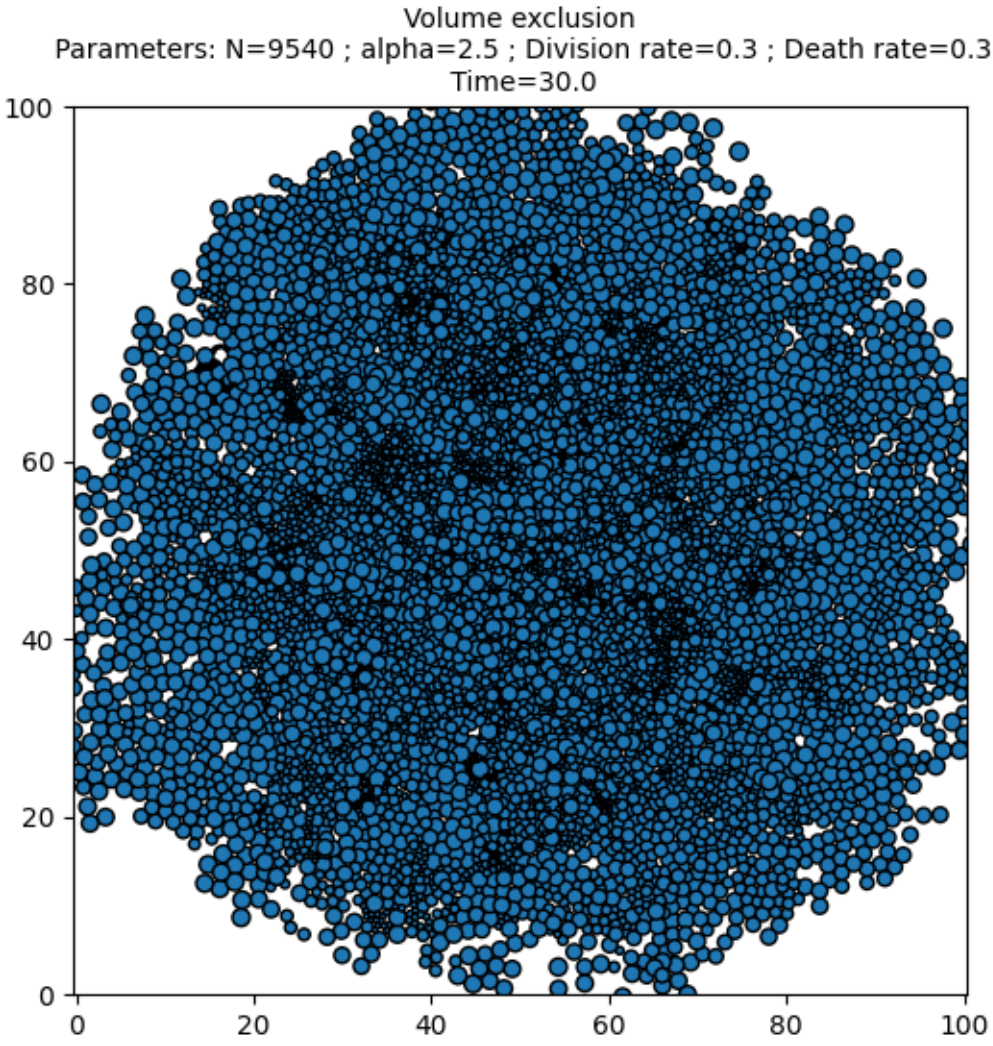

Volume exclusion
Parameters: N=10014 ; alpha=2.5 ; Division rate=0.3 ; Death rate=0.3
Time=4.0
80
60
40
20
0
20
40
60
80
100


Volume exclusion
Parameters: N=9972 ; alpha=2.5 ; Division rate=0.3 ; Death rate=0.3
Time=5.0
80
60
40
20
0
20
40
60
80
100

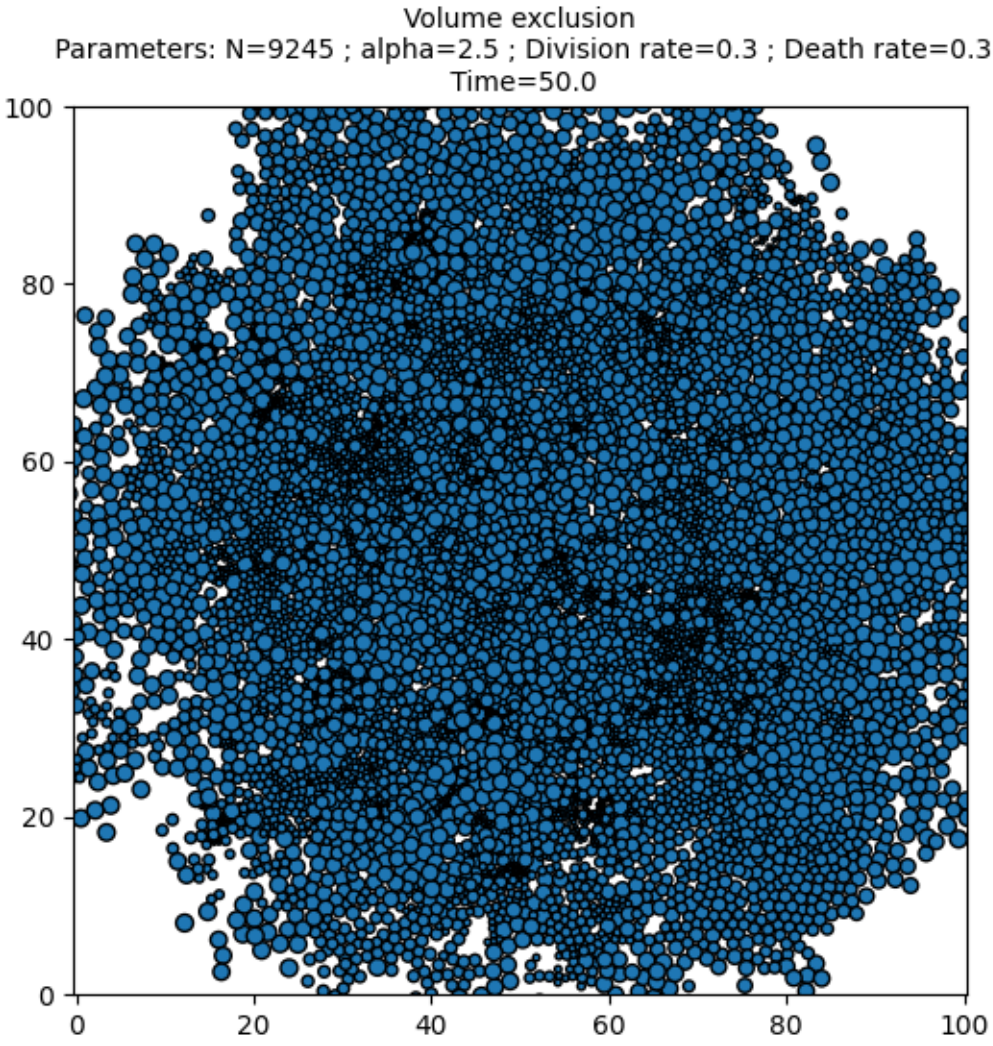
•


Volume exclusion
Parameters: N=9874 ; alpha=2.5 ; Division rate=0.3 ; Death rate=0.3
Time=10.0
80
60
40
20
0
20
40
60
80
100

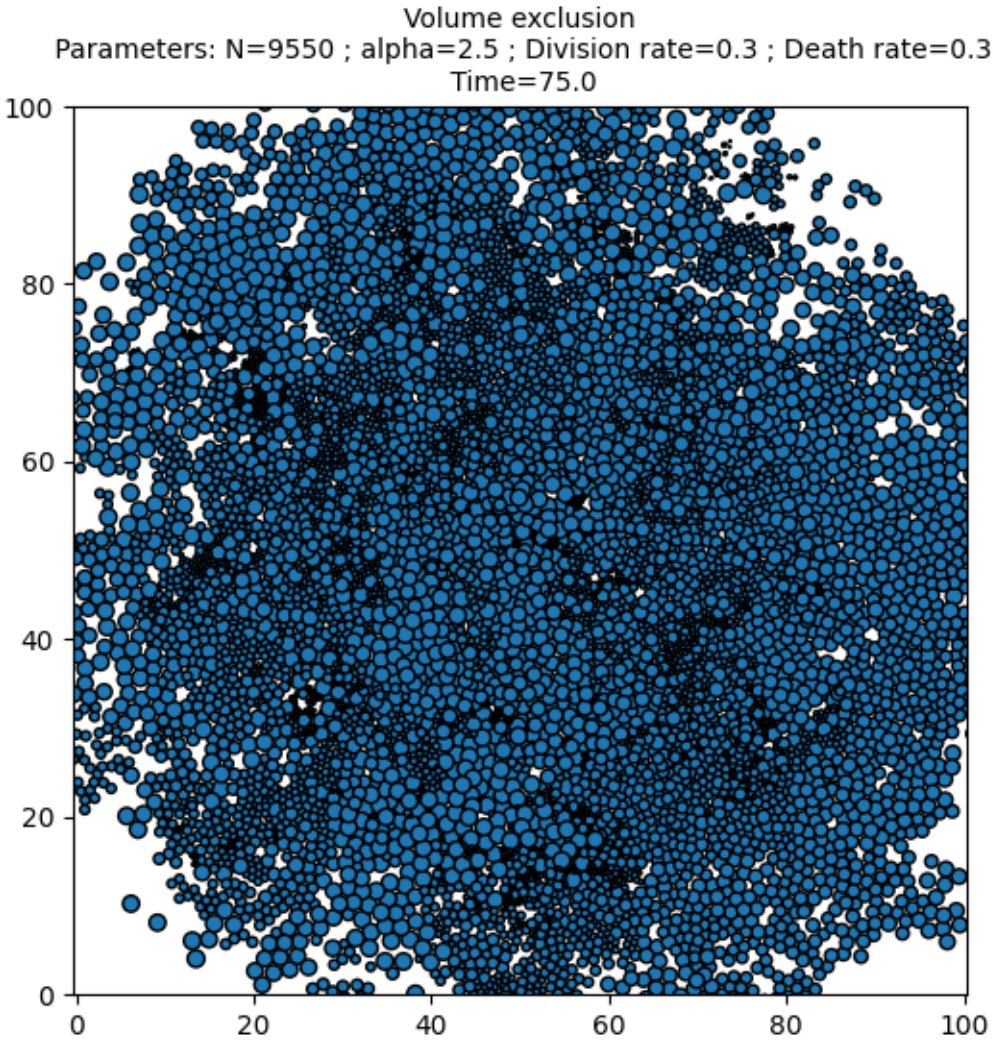
•


Volume exclusion
Parameters: N=9540 ; alpha=2.5 ; Division rate=0.3 ; Death rate=0.3
Time=30.0
100
80
60
40
20
0
0
20
40
60
80
100

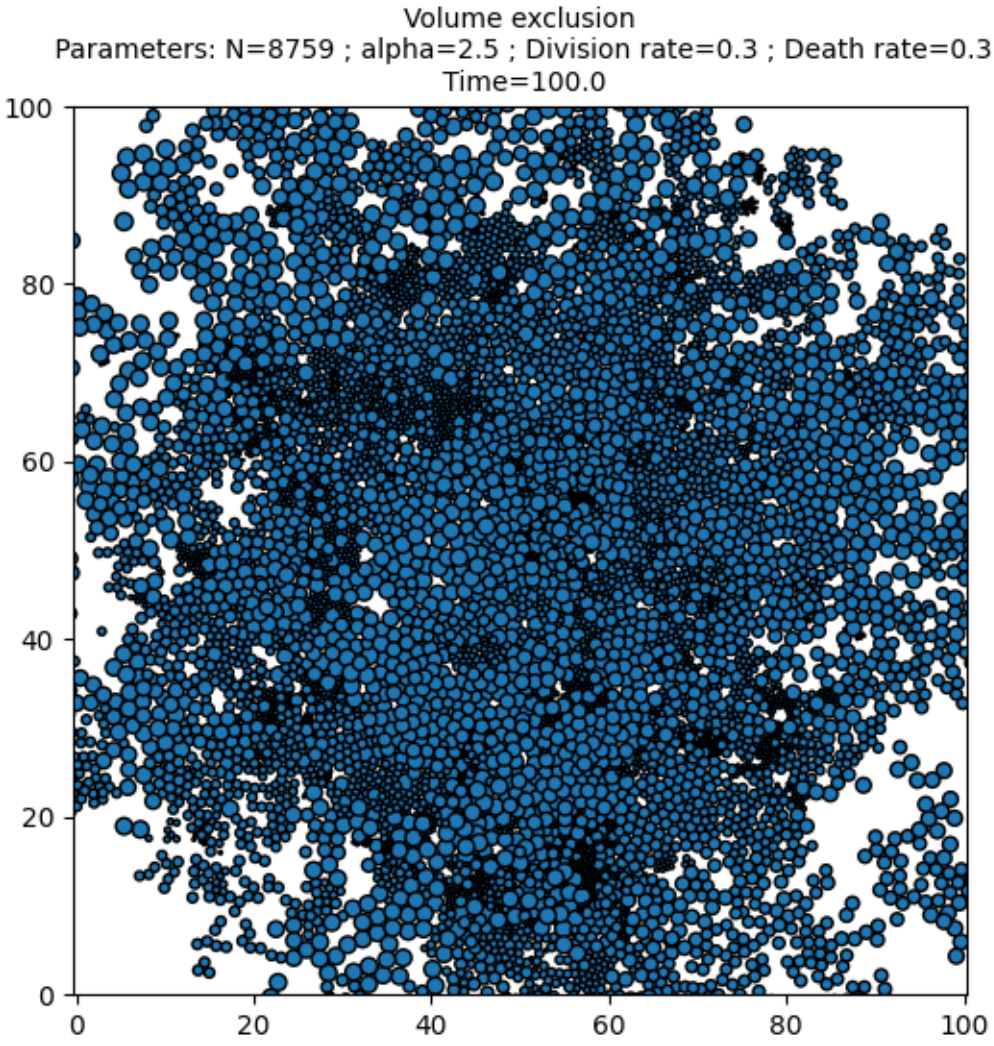
-

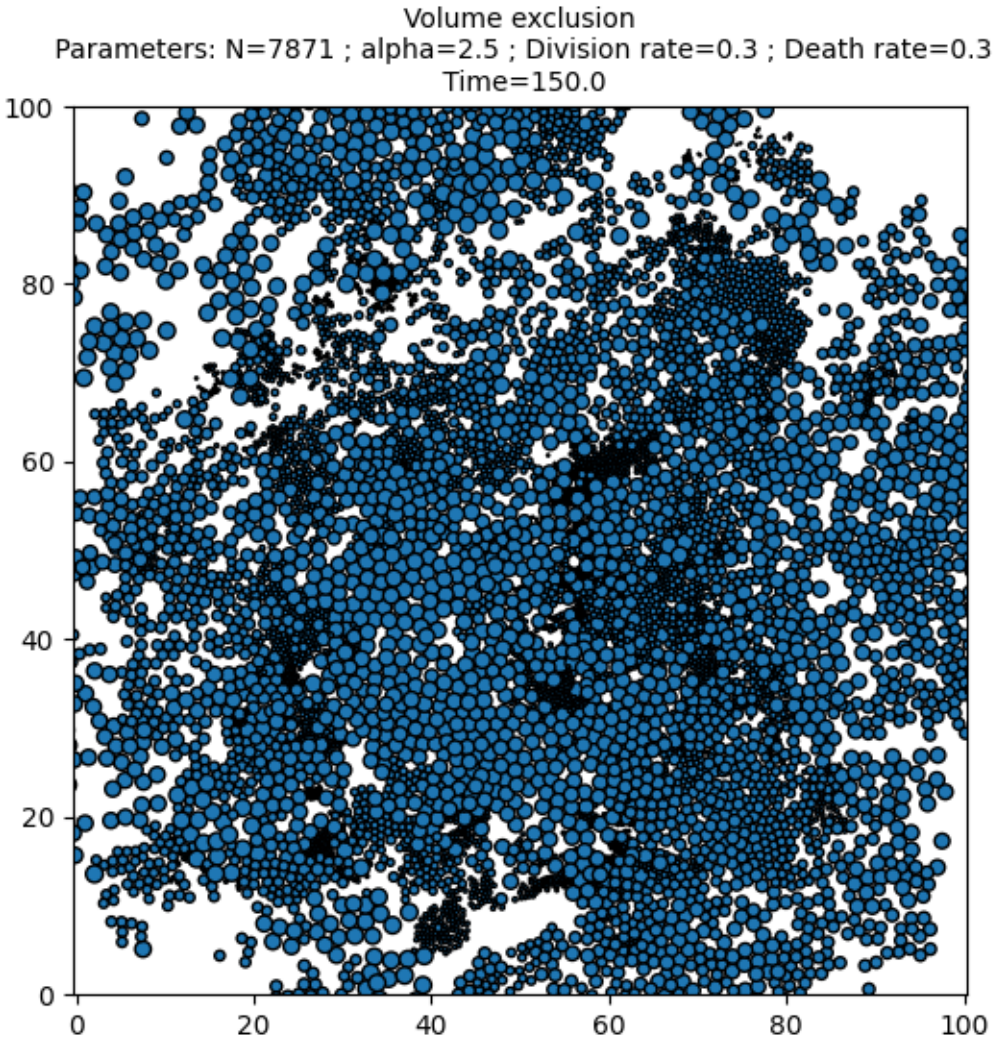

Volume exclusion
Parameters: N=9245 ; alpha=2.5 ; Division rate=0.3 ; Death rate=0.3
Time=50.0
100
80
60
40
20
0
0
20
40
60
80
100

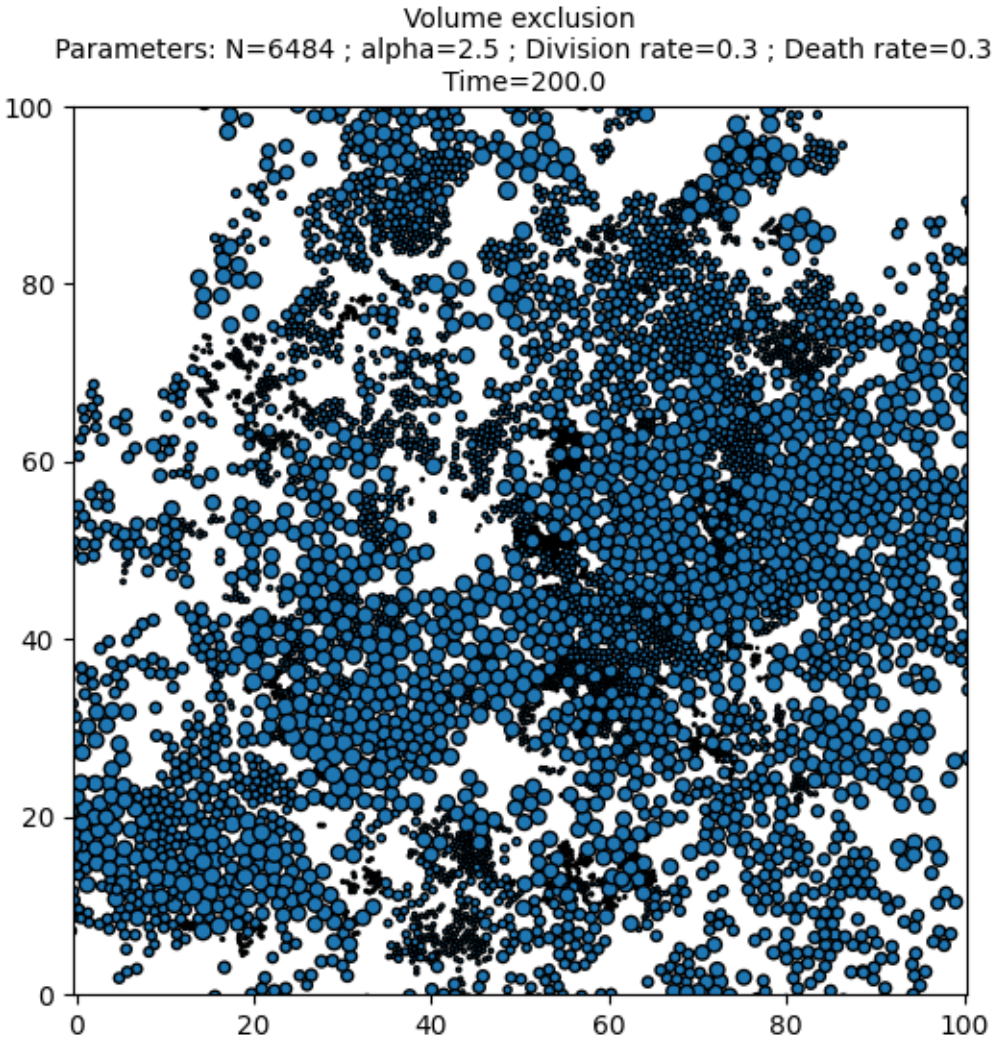
•


Volume exclusion
Parameters: N=9550 ; alpha=2.5 ; Division rate=0.3 ; Death rate=0.3
Time=75.0
100
80
60
40
20
0
0
20
40
60
80
100

-


Volume exclusion
Parameters: N=8759 ; alpha=2.5 ; Division rate=0.3 ; Death rate=0.3
Time=100.0
100
80
60
40
20
0
0
20
40
60
80
100


Volume exclusion
Parameters: N=7871 ; alpha=2.5 ; Division rate=0.3 ; Death rate=0.3
Time=150.0
100
80
60
40
20
0
0
20
40
60
80
100



-

Out:

```
Progress:0%
Progress:1%
Progress:2%
Progress:3%
Progress:4%
Progress:5%
Progress:6%
Progress:7%
Progress:8%
Progress:9%
Progress:10%
Progress:11%
Progress:12%
Progress:13%
Progress:14%
```

(continues on next page)

(continued from previous page)

```
Progress:15%
Progress:16%
Progress:17%
Progress:18%
Progress:19%
Progress:20%
Progress:21%
Progress:22%
Progress:23%
Progress:24%
Progress:25%
Progress:26%
Progress:27%
Progress:28%
Progress:29%
Progress:30%
Progress:31%
Progress:32%
Progress:33%
Progress:34%
Progress:35%
Progress:36%
Progress:37%
Progress:38%
Progress:39%
Progress:40%
Progress:41%
Progress:42%
Progress:43%
Progress:44%
Progress:45%
Progress:46%
Progress:47%
Progress:48%
Progress:49%
Progress:50%
Progress:51%
Progress:52%
Progress:53%
Progress:54%
Progress:55%
Progress:56%
Progress:57%
Progress:58%
Progress:59%
Progress:60%
Progress:61%
Progress:62%
Progress:63%
Progress:64%
Progress:65%
Progress:66%
```

(continues on next page)

(continued from previous page)

```
Progress:67%
Progress:68%
Progress:69%
Progress:70%
Progress:71%
Progress:72%
Progress:73%
Progress:74%
Progress:75%
Progress:76%
Progress:77%
Progress:78%
Progress:79%
Progress:80%
Progress:81%
Progress:82%
Progress:83%
Progress:84%
Progress:85%
Progress:86%
Progress:87%
Progress:88%
Progress:89%
Progress:90%
Progress:91%
Progress:92%
Progress:93%
Progress:94%
Progress:95%
Progress:96%
Progress:97%
Progress:98%
Progress:99%
Progress:100%
```

Print the total simulation time and the average time per iteration.

```
print('Total time: '+str(e-s)+' seconds')
print('Average time per iteration: '+str((e-s)/simu.iteration)+' seconds')
```

Out:

```
Total time: 200.29545760154724 seconds
Average time per iteration: 0.007975450250917705 seconds
```

Total running time of the script: (8 minutes 53.860 seconds)

4.6 Particle systems

Particles(pos, interaction_radius, box_size)	The main class.
KineticParticles(pos, vel, ...[, ...])	Subclass of kinetic particles.
BOParticles(pos, bo, interaction_radius, ...)	3D particle with a body-orientation in $SO(3)$.

class sisyphe.particles.**Particles**(*pos*, *interaction_radius*, *box_size*, *vision_angle=6.283185307179586*, *axis=None*, *boundary_conditions='periodic'*, *block_sparse_reduction=False*, *number_of_cells=1024*)

The main class.

N

Number of particles.

Type
int

d

Dimension.

Type
int

pos

Positions of the particles.

Type
(N,d) Tensor

R

Radius of the particles.

Type
float or (N,) Tensor

L

Box size.

Type
(d,) Tensor

angle

Vision angle.

Type
float

axis_is_none

True when an axis is specified.

Type
bool

axis

Axis of the particles.

Type
(N,d) Tensor or None

bc

Boundary conditions. Can be one of the following:

- list of size d containing 0 (periodic) or 1 (wall with reflecting boundary conditions) for each dimension.
- `"open"` : no boundary conditions.
- `"periodic"` : periodic boundary conditions.
- `"spherical"` : reflecting boundary conditions on the sphere of radius `L[0]/2` and center $\mathbf{L}/2$.

Type
list or str

target_method

Dictionary of target methods (see the function *compute_target()*).

Type
dict

target_option_method

Dictionary of options for the targets (see the function *compute_target()*).

Type
dict

blocksparse

Use block sparse reduction or not.

Type
bool

iteration

Iteration.

Type
int

centroids

Centroids.

Type
Tensor

eps

Size of the cells.

Type
Tensor

keep

`keep[i,j]` is True when the cells `i` and `j` are contiguous.

Type
BoolTensor

compute_blocksparse_parameters(*number_of_cells*)

Compute the centroids, the size of the cells and the keep matrix.

Parameters
number_of_cells (*int*) – The maximal number of cells. It will be rounded to the nearest lower power d.

Returns

(nb_cells, d) Tensor, (nb_cells,) Tensor, (nb_cells, nb_cells) BoolTensor:

The centroids the size of the cells and the keep matrix.

`best_blocksparse_parameters`(*min_nb_cells*, *max_nb_cells*, *step=1*, *nb_calls=100*, *loop=1*, *set_best=True*)

Returns the optimal block sparse parameters.

This function measures the time to execute **nb_calls** calls to the `__next__()` for different values of the block sparse parameters given by the number of cells to the power $1/d$.

Parameters

- **min_nb_cells** (*int*) – Minimal number of cells to the power $1/d$.
- **max_nb_cells** (*int*) – Maximal number of cells to the power $1/d$.
- **step** (*int, optional*) – Step between two numbers of cells. Default is 1.
- **nb_calls** (*int, optional*) – Number of calls to the function `__next__()` at each test. Default is 100.
- **loop** (*int, optional*) – Number of loops. Default is 1.
- **set_best** (*bool, optional*) – Set the block sparse parameters to the optimal ones. Default is True.

Returns

- **fastest** ***(int)*****: The number of cells which gave the** fastest result.
- **nb_cells** *(list)*: The number of cells tested.
- **average_simu_time** ***(list)*****: The average simulation** times.
- **simulation_time** ***(numpy array)*****: The total simulation** times.

Return type
tuple

`add_target_method`(*name*, *method*)

Add a method to the dictionary *target_method*.

Parameters

- **name** (*str*) – The name of the method.
- **method** (*func*) – A method.

`add_target_option_method`(*name*, *method*)

Add an option to the dictionary *target_option_method*

Parameters

- **name** (*str*) – The name of the option.
- **method** (*func*) – An option.

compute_target(*which_target*, *which_options*, ***kwargs*)

Compute a target and apply some options.

Parameters

- **which_target** (*dict*) – Dictionary of the form:

```
{"name"       : "method_name",
 "parameters" : {"param1" : param1,
                 "param2" : param2,
                  ...
                }
}
```

The sub-dictionary "parameters" (eventually empty) contains the keyword arguments to be passed to the function found in the dictionary *target_method* with the keyword "method_name".

- **which_options** (*dict*) – Dictionary of the form:

```
{"name_of_option1" : parameters_of_option1,
 "name_of_option2" : parameters_of_option2,
...
}
```

The dictionary `**parameters_of_option1**` contains the keyword arguments to be passed to the method found in the dictionary *target_option_method* with the keyword "name_of_option1".

- ****kwargs** – Keywords arguments to be passed to the functions which compute the target and the options.

Returns

Compute a target using **which_target** and then apply the options in **which_options** successively.

Return type

Tensor

linear_local_average(**to_average*, *who=None*, *with_who=None*, *isaverage=True*, *kernel=<function lazy_interaction_kernel>*)

Fast computation of a linear local average.

A linear local average is a matrix vector product between an interaction kernel (LazyTensor of size (M,N)) and a Tensor of size (N,dimU).

Parameters

- ***to_average** (*Tensor*) – The quantities to average. The first dimension must be equal to *N*.
- **who** (*(N,) BoolTensor or None, optional*) – The rows of the interaction kernel is pos[who,:] (or *pos* if **who** is None). Default is None.
- **with_who** (*(N,) BoolTensor or None, optional*) – The columns of the interaction kernel is pos[with_who,:] (or *pos* if **with_who** is None). Default is None.
- **isaverage** (*bool, optional*) – Divide the result by *N* if **isaverage** is True. Default is True.

- **kernel** (*func, optional*) – The function which returns the interaction kernel as a LazyTensor. Default is `lazy_interaction_kernel()`.

Returns
The linear local averages.

Return type
tuple of Tensors

nonlinear_local_average(*binary_formula*, *arg1*, *arg2*, *who=None*, *with_who=None*, *isaverage=True*, *kernel=<function lazy_interaction_kernel>*)

Fast computation of a nonlinear local average.

A nonlinear local average is a reduction of the form

$$\sum_j K_{ij} U_{ij}$$

where K_{ij} is an interaction kernel (LazyTensor of size (M,N)) and U_{ij} is a LazyTensor of size (M,N,dim) computed using a binary formula.

Parameters

- **binary_formula** (*func*) – Takes two arguments **arg1** and **arg2** and returns a LazyTensor of size (M,N,dim).
- **arg1** (*(M,D1) Tensor*) – First argument of the binary formula.
- **arg2** (*(N,D2) Tensor*) – Second argument of the binary formula.
- **who** (*(N,) BoolTensor or None, optional*) – The rows of the interaction kernel is `pos[who,:]` (or *pos* if **who** is None). Default is None.
- **with_who** (*(N,) BoolTensor or None, optional*) – The columns of the interaction kernel is `pos[with_who,:]` (or *pos* if **with_who** is None). Default is None.
- **isaverage** (*bool, optional*) – Divide the result by *N* if isaverage is True. Default is True.
- **kernel** (*func, optional*) – The function which returns the interaction kernel as a LazyTensor. Default is `lazy_interaction_kernel()`.

Returns
The nonlinear local average.

Return type
(M,dim) Tensor

number_of_neighbours(*who=None*, *with_who=None*)

Compute the number of neighbours.

The number of neighbours of `x[i,:]` is the number of ones in the row corresponding to `x[i,:]` in the interaction kernel.

Parameters

- **who** (*(N,) BoolTensor or None, optional*) – The rows of the interaction kernel is `pos[who,:]` (or *pos* if **who** is None). Default is None.
- **with_who** (*(N,) BoolTensor or None, optional*) – The columns of the interaction kernel is `pos[with_who,:]` (or *pos* if **with_who** is None). Default is None.

Returns
The number of neighbours.

Return type
(M,) Tensor

morse_target(*Ca*, *la*, *Cr*, *lr*, *p=2*, *mass=1.0*, *who=None*, *with_who=None*, *local=False*, *isaverage=True*, *kernel=<function lazy_interaction_kernel>*, ***kwargs*)

Compute the force exerted on each particle due to the Morse potential.

Parameters

- **Ca** (*float*) – Attraction coefficient.
- **la** (*float*) – Attraction length.
- **Cr** (*float*) – Repulsion coefficient.
- **lr** (*float*) – Repulsion length.
- **p** (*int, optional*) – Exponent. Default is 2.
- **mass** (*(N,) Tensor or float, optional*) – Mass of the particles. Default is 1.
- **who** (*(N,) BoolTensor or None, optional*) – The rows of the interaction kernel is `pos[who,:]` (or *pos* if **who** is None). Default is None.
- **with_who** (*(N,) BoolTensor or None, optional*) – The columns of the interaction kernel is `pos[with_who,:]` (or *pos* if **with_who** is None). Default is None.
- **local** (*bool, optional*) – If **local** is true then the sum only considers the y particles within the interaction kernel. Default is False.
- **isaverage** (*bool, optional*) – If isaverage is True then divide the total force by N where N is the number of True in :Tensor"*with_who*. Default is True.
- **kernel** (*func, optional*) – The interaction kernel to use. Default is `lazy_interaction_kernel()`
- ****kwargs** – Arbitrary keyword arguments.

Returns
The force exerted on each particle located at `pos[who,:]`.

Return type
Tensor

quadratic_potential_target(*who=None*, *with_who=None*, *kernel=<function lazy_interaction_kernel>*, *isaverage=True*, ***kwargs*)

Compute the force exerted on each particle due to the quadratic potential.

Parameters

- **who** (*(N,) BoolTensor or None, optional*) – The rows of the interaction kernel is `pos[who,:]` (or *pos* if **who** is None). Default is None.
- **with_who** (*(N,) BoolTensor or None, optional*) – The columns of the interaction kernel is `pos[with_who,:]` (or *pos* if **with_who** is None). Default is None.
- **isaverage** (*bool, optional*) – If isaverage is True then divide the total force by N where N is the number of True in :Tensor"*with_who*. Default is True.
- **kernel** (*func, optional*) – The interaction kernel to use. Default is `lazy_interaction_kernel()`
- ****kwargs** – Arbitrary keyword arguments.

Returns
The force exerted on each particle located at `pos[who,:]`.

Return type
Tensor

`overlapping_repulsion_target`(*who=None*, *with_who=None*, ***kwargs*)

Compute the repulsion force exerted on each particle due to the overlapping.

The force exerted on a particle located at x_i derives from a logarithmic potential and is given by:

$$F = \sum_j K(x_i - x_j) \left(\frac{(R_i + R_j)^2}{|x_i - x_j|^2} - 1 \right) (x_i - x_j)$$

where x_j is the position of the other particles; R_i and R_j are the radii of the particles at x_i and x_j respectively and K is the overlapping kernel.

Note: in the present case, it is simpler to write explicitly the reduction formula but the same function could be implemented using a binary formula in *nonlinear_local_average()* with the kernel given by `lazy_overlapping_kernel()`.

Parameters

- **who** (*(N,) BoolTensor or None, optional*) – The rows of the interaction kernel is `pos[who,:]` (or *pos* if **who** is None). Default is None.
- **with_who** (*(N,) BoolTensor or None, optional*) – The columns of the interaction kernel is `pos[with_who,:]` (or *pos* if **with_who** is None). Default is None.
- ****kwargs** – Arbitrary keyword arguments.

Returns
The force exerted on each particle located at `pos[who,:]`.

Return type
Tensor

`class sisyphe.particles.KineticParticles`(*pos*, *vel*, *interaction_radius*, *box_size*, *vision_angle=6.283185307179586*, *axis=None*, *boundary_conditions='periodic'*, *block_sparse_reduction=False*, *number_of_cells=1024*)

Subclass of kinetic particles.

`vel`

Velocities.

Type
(N,d) Tensor

`property axis`

If a None axis is given, then the default axis is the normalised velocity.

Returns
(N,d) Tensor

`property order_parameter`

Compute the norm of the average velocity.

Returns
The order parameter.

Return type
float

`check_boundary()`

Update the positions and velocities of the particles which are outside the boundary.

`normalised(`*kappa*, *who=None*, *with_who=None*, ***kwargs*`)`

The normalised average velocity.

Given a system of particles with positions $(x_i)_i$ and velocities $(v_i)_i$, the normalised average velocity around particle (x_i, v_i) is

$$\Omega_i = \frac{\sum_j K(x_i - x_j)v_j}{\left|\sum_j K(x_i - x_j)v_j\right|},$$

where K is the interaction kernel.

Parameters

- **kappa** (*(1,) Tensor or (M,) Tensor*) – The concentration parameter. If the size of kappa is bigger than 1 then its size must be equal to the number of True values in **who**.
- **who** (*(N,) BoolTensor or None, optional*) – The rows of the interaction kernel is `pos[who,:]` (or `pos` if **who** is None). Default is None.
- **with_who** (*(N,) BoolTensor or None, optional*) – The columns of the interaction kernel is `pos[with_who,:]` (or `pos` if **with_who** is None). Default is None.
- ****kwargs** – Arbitrary keyword arguments.

Returns
The normalised averaged velocity multiplied by the concentration parameter.

Return type
Tensor

`motsch_tadmor(`*kappa*, *who=None*, *with_who=None*, ***kwargs*`)`

The average velocity of the neighbours only.

Given a system of particles with positions $(x_i)_i$ and velocities $(v_i)_i$, the average velocity around particle (x_i, v_i) with the Motsch-Tadmor normalisation is:

$$\Omega_i = \frac{\sum_j K(x_i - x_j)v_j}{\sum_j K(x_i - x_j)},$$

where K is the interaction kernel.

Parameters

- **kappa** (*(1,) Tensor or (M,) Tensor*) – The concentration parameter. If the size of kappa is bigger than 1 then its size must be equal to the number of True values in **who**.
- **who** (*(N,) BoolTensor or None, optional*) – The rows of the interaction kernel is `pos[who,:]` (or `pos` if **who** is None). Default is None.
- **with_who** (*(N,) BoolTensor or None, optional*) – The columns of the interaction kernel is `pos[with_who,:]` (or `pos` if **with_who** is None). Default is None.
- ****kwargs** – Arbitrary keyword arguments.

Returns
The average velocity of the neighbours multiplied by the concentration parameter.

Return type
Tensor

mean_field(*kappa*, *who=None*, *with_who=None*, ***kwargs*)

Non-normalised average velocity.

Given a system of particles with positions $(x_i)_i$ and velocities $(v_i)_i$, the mean-field average velocity around particle (x_i, v_i) (without normalisation) is:

$$J_i = \frac{1}{N} \sum_j K(x_i - x_j) v_j,$$

where K is the interaction kernel.

Parameters

- **kappa** (*(1,) Tensor or (M,) Tensor*) – The concentration parameter. If the size of kappa is bigger than 1 then its size must be equal to the number of True values in **who**.
- **who** (*(N,) BoolTensor or None, optional*) – The rows of the interaction kernel is `pos[who,:]` (or `pos` if **who** is None). Default is None.
- **with_who** (*(N,) BoolTensor or None, optional*) – The columns of the interaction kernel is `pos[with_who,:]` (or `pos` if **with_who** is None). Default is None.
- ****kwargs** – Arbitrary keyword arguments.

Returns
The mean-field average velocity multiplied by a scaling parameter and by the concentration parameter. The scaling parameter is equal to the volume of the box divided by the volume of the ball of radius `R` in dimension `d`.

Return type
Tensor

max_kappa(*kappa*, *kappa_max*, *who=None*, *with_who=None*, ***kwargs*)

The mean-field target with a norm threshold.

Given a system of particles with positions $(x_i)_i$ and velocities $(v_i)_i$, the mean-field average velocity around particle (x_i, v_i) with a cutoff is:

$$\Omega_i = \frac{1}{\frac{1}{\kappa} + \frac{|J_i|}{\kappa_{\max}}} J_i$$

where

$$J_i = \frac{1}{N} \sum_j K(x_i - x_j) v_j,$$

and K is the interaction kernel multiplied by the scaling parameter. The scaling parameter is equal to the volume of the box divided by the volume of the ball of radius `R` in dimension `d`. The norm of Ω_i is thus bounded by $\kappa_{\max}$.

Parameters

- **kappa** (*(1,) Tensor or (M,) Tensor*) – The concentration parameter. If the size of kappa is bigger than 1 then its size must be equal to the number of True values in **who**.
- **who** (*(N,) BoolTensor or None, optional*) – The rows of the interaction kernel is `pos[who,:]` (or `pos` if **who** is None). Default is None.

- **with_who** (*(N,) BoolTensor or None, optional*) – The columns of the interaction kernel is `pos[with_who,:]` (or `pos` if **with_who** is None). Default is None.
- ****kwargs** – Arbitrary keyword arguments.

Returns
The cutoff average velocity.

Return type
Tensor

nematic(*kappa*, *who=None*, *with_who=None*, ***kwargs*)

Normalised average of the nematic velocities.

Given a system of particles with positions $(x_i)_i$ and velocities $(v_i)_i$, the nematic average velocity around particle (x_i, v_i) is:

$$\Omega_i = (v_i \cdot \overline{\Omega}_i)\overline{\Omega}_i,$$

where $\overline{\Omega}_i$ is any unit eigenvector associated to the maximal eigenvalue of the average Q-tensor:

$$Q_i = \sum_j K(x_i - x_j)\left(v_j \otimes v_j - \frac{1}{d} I_d\right),$$

where K is the interaction kernel.

Parameters

- **kappa** (*(1,) Tensor or (M,) Tensor*) – The concentration parameter. If the size of kappa is bigger than 1 then its size must be equal to the number of True values in **who**.
- **who** (*(N,) BoolTensor or None, optional*) – The rows of the interaction kernel is `pos[who,:]` (or `pos` if **who** is None). Default is None.
- **with_who** (*(N,) BoolTensor or None, optional*) – The columns of the interaction kernel is `pos[with_who,:]` (or `pos` if **with_who** is None). Default is None.
- ****kwargs** – Arbitrary keyword arguments.

Returns
The nematic average velocity multiplied by the concentration parameter.

Return type
Tensor

bounded_angular_velocity(*target*, *angvel_max*, *dt*, *who=None*, ***kwargs*)

Bounded angle between the velocity and the target.

The maximum angle between the velocity and the new target cannot exceed the value `angvel_max * dt`. The output tensor **new_target** is of the form:

```
new_target = a * vel + b * target
```

where $a, b > 0$ are such that the cosine between **new_target** and *vel* is equal to:

```
max_cos := max(cos(angvel_max * dt), cos_target)
```

where **cos_target** is the cosine between **target** and *vel*. The solution is:

```
b = (1 - max_cos ** 2) / (1 - cos_target ** 2)
a = |target| / |vel| * (max_cos - b * cos_target)
```

Parameters

- **target** (*(M,d) Tensor*) – Target.
- **angvel_max** (*float*) – Maximum angular velocity.
- **dt** (*float*) – Interval of time.
- **who** (*(N,) BoolTensor, optional*) – The velocities of the particles associated to the target are `vel[who,:]` (or *vel* if **who** is None). Default is None.
- ****kwargs** – Arbitrary keywords arguments.

Returns
The new target.

Return type
(M,d) Tensor

pseudo_nematic(*target*, *who=None*, ***kwargs*)

Choose the closest to the velocity between the target and its opposite.

Parameters

- **target** (*(M,d) Tensor*) – Target.
- **who** (*(N,) BoolTensor, optional*) – The velocities of the particles associated to the target are `vel[who,:]` (or *vel* if **who** is None). Default is None.
- ****kwargs** – Arbitrary keywords arguments.

Returns
Take the opposite of **target** if the angle with the velocity is larger than $\pi/2$.

Return type
(M,d) Tensor

class sisyphe.particles.**BOParticles**(*pos*, *bo*, *interaction_radius*, *box_size*, *vision_angle=6.283185307179586*, *axis=None*, *boundary_conditions='periodic'*, *block_sparse_reduction=False*, *number_of_cells=1024*)

3D particle with a body-orientation in $SO(3)$.

A body-orientation is a rotation matrix is $SO(3)$ or equivalently a unit quaternion. If $q = (x, y, z, t)$ is a unit quaternion, the associated rotation matrix is $A = [e_1, e_2, e_3]$ where the column vectors are defined by:

$$e_1 = (x^2 + y^2 - z^2 + t^2, 2(xt + yz), 2(yt - xz))^T$$

$$e_2 = (2(yz - xt), x^2 - y^2 + z^2 - t^2, 2(xy + zt))^T$$

$$e_3 = (2(xz + yt), 2(zt - xy), x^2 - y^2 - z^2 + t^2)^T$$

Conversely, if A is the rotation of angle θ around the axis $u = (u_1, u_2, u_3)^T$ then the associated unit quaternion is:

$$q = \cos(\theta/2) + \sin(\theta/2)(iu_1 + ju_2 + ku_3).$$

or its opposite.

bo

Body-orientation stored as a unit quaternion.

Type
(N,4) Tensor

property vel

The velocity is the first column vector of the body-orientation matrix.

Returns
The velocity.

Return type
(N,3) Tensor

property axis

If a None axis is given, then the default axis is the (normalised) velocity.

Return type
(N,3) Tensor

property orth_basis

An othogonal basis of the orthogonal plane of the velocity.

Returns

- **e2** *((N,3) Tensor)*: Second column vector of the body-orientation matrix.
- **e3** *((N,3) Tensor)*: Third column vector of the body-orientation matrix.

Return type
tuple

property qtensor

Compute the normalised uniaxial Q-tensors of the particles.

If q is a unit quaternion seen as a colum vector of dimension 4, the normalised uniaxial Q-tensor is defined by:

$$Q = q \otimes q - \frac{1}{4} I_4$$

where $\otimes$ is the tensor product and I_4 the identity matrix of size 4.

Returns
Q-tensors of the N particles.

Return type
(N,4,4) Tensor

property omega_basis

Local frame associated to the direction of motion.

The unit vector Ω is the direction of motion (it is equal to *vel*) and $[\Omega, e_\theta, e_\varphi]$ is the local orthomormal frame associated to the spherical cooordinates.

Returns

- **p** *((N,3) Tensor)*: e_φ.
- **q** *((N,3) Tensor)*: $-e_\theta$.

Return type
tuple

property u_in_omega_basis

Coordinates of u in the frame $[\Omega, p, q]$ given by *omega_basis* where u is the second column vector of the body-orientation matrix.

normalised(*who=None*, *with_who=None*, ***kwargs*)

Projection of the average body-orientation on $SO(3)$.

Given a system of particles with positions $(x_i)_i$ and body-orientation matrices $(A_i)_i$, the mean-field average body-orientation around the particle (x_i, A_i) is is the arithmetic average:

$$J_i = \sum_j K(x_i - x_j)A_j$$

where K is the interaction kernel. The projection on $SO(3)$ is

$$P_{SO(3)}(J_i) = argmax_{A \in SO(3)}(A \cdot J_i)$$

for the dot product

$$A \cdot B = \frac{1}{2}Tr(A^T B).$$

The unit quaternion associated to $P_{SO(3)}(J_i)$ is any eigenvector associated to the maximal eigenvalue of the average Q-tensor:

$$\overline{Q}_i = \sum_j K(x_i - x_j)Q_j$$

where Q_j is the Q-tensor of particle (x_j, A_j) given by *qtensor*.

Parameters

- **who** (*(N,) BoolTensor or None, optional*) – The rows of the interaction kernel is `pos[who,:]` (or `pos` if **who** is None). Default is None.
- **with_who** (*(N,) BoolTensor or None, optional*) – The columns of the interaction kernel is `pos[with_who,:]` (or `pos` if **with_who** is None). Default is None.
- ****kwargs** – Arbitrary keyword arguments.

Returns

The unit quaternion associated to the projection of the mean-field average body-orientation on $SO(3)$.

Return type

(M,4) Tensor

property theta

Polar angle of the direction of motion

property phi

Azimuthal angle of the direction of motion

property global_order

Returns

$$\frac{1}{N(N-1)}\sum_{i,j}(q_i \cdot q_j)^2.$$

build_reflection_quat(*who*, *axis*)

Returns the quaternion associated to the rotation of angle 2α and axis n where α is the angle between `vel[who,:]` and the plane orthogonal to **axis** and n is the element of this plane obtained by a rotation of $\pi/2$ around **axis** of the normalised projection of `vel[who,:]`.

Parameters

- **who** (*(N,) BoolTensor*) –
- **axis** (*int*) – The axis number either 0 (x axis), 1 (y axis) or 2 (z axis).

Returns
Rotation (unit quaternion).

Return type
(M,4) Tensor

check_boundary()

Update the positions and body-orientations of the particles which are outside the boundary.

4.7 Models

AsynchronousVicsek(pos, vel, v, jump_rate, ...)	Vicsek model with random jumps.
Vicsek(pos, vel, v, sigma, nu, ...[, ...])	Vicsek model with gradual alignment and diffusion.
SynchronousVicsek(pos, vel, v, dt, kappa, ...)	Vicsek model with synchronous random jumps.
BOAsynchronousVicsek(pos, bo, v, jump_rate, ...)	Asynchronous Vicsek model for body-oriented particles.
VolumeExclusion(pos, interaction_radius, ...)	Spherical particles interacting via short-range repulsion.
AttractionRepulsion(pos, vel, ...[, p, ...])	Self-propelled particles with attraction and repulsion forces.

class sisyphe.models.**AsynchronousVicsek**(*pos*, *vel*, *v*, *jump_rate*, *kappa*, *interaction_radius*, *box_size*, *vision_angle=6.283185307179586*, *axis=None*, *boundary_conditions='periodic'*, *variant=None*, *options=None*, *sampling_method='vonmises'*, *block_sparse_reduction=False*, *number_of_cells=1024*)

Vicsek model with random jumps.

The asynchronous Vicsek Model is described in [DM2016].

The velocity of each particle is a unit vector in the sphere $\mathbb{S}^{d-1}$. The velocity of each particle is updated at random times by computing first a *target* (Tensor) in $\mathbb{R}^d$ using the parameters contained in *target* and then a new velocity in $\mathbb{S}^{d-1}$ is sampled using a method specified by *sampling_method*. Between two jumps, a particle moves in straight line at a constant speed.

name

`"Asynchronous Vicsek"` and the value of the key `"name"` in the dictionary *target*.

Type
str

v

The speed of the particles.

Type
float

dt

Discretisation time-step. It is equal to 0.01 divided by *jumprate*.

Type
float

jumprate

Each particle has an independent Poisson clock with parameter *jumprate*.

Type
float

kappa

Concentration parameter.

Type
(1,) Tensor

target

Target dictionary. Contains the target name and its parameters. To be passed to the method `compute_target()` in the argument `which_target`.

Type
dict

options

Options dictionary. Each key is an option's name and its value contains the parameters of the option. To be passed to the method `compute_target()` in the argument `which_options`.

Type
dict

sampling_method

The new velocity is sampled using one of the following methods.

- `"vonmises"` : Von Mises distribution with center given by the normalised *target* and the concentration parameter given by the norm of *target*.
- `"uniform"` : Uniform distribution in the ball around the normalised *target* with deviation given by the inverse of the norm of *target*. Dimension 2 only.
- `"projected_normal"` : Sample first a gaussian random variable in $\mathbb{R}^d$ with center the normalised *target* and standard deviation the inverse of the norm of *target*. Then project the result on the sphere $\mathbb{S}^{d-1}$.
- `"vectorial_noise"` : Same as `"projected_normal"` where the normally distributed random variable is replaced by a uniformly distributed random variable on the sphere, divided by the norm of *target*.

Type
str

parameters

The parameters of the model.

Type
str

update()

Update the positions and velocities.

Update the positions using a first order explicit Euler method during the time step *dt*. The particles which update their velocities in the time interval *dt* are given by the boolean mask of size `N` **who_jumps**. The probability for a particle to jump is equal to `exp(-jumprate * dt)`.

Returns
Positions and velocities.

Return type
dict

class sisyphe.models.**Vicsek**(*pos*, *vel*, *v*, *sigma*, *nu*, *interaction_radius*, *box_size*, *vision_angle=6.283185307179586*, *axis=None*, *boundary_conditions='periodic'*, *variant=None*, *options=None*, *numerical_scheme='projection'*, *dt=0.01*, *block_sparse_reduction=False*, *number_of_cells=1600*)

Vicsek model with gradual alignment and diffusion.

The Vicsek model is described in [DM2008].

The velocity of each particle is a unit vector in the sphere $\mathbb{S}^{d-1}$. The velocity of each particle is continuously updated by a combination of two effects.

- A deterministic drift tends to relax the velocity towards a *target* computed using *target*. The intensity of the drift may depend on the norm of the target.
- A random diffusion.

The speed of the particles is constant.

name

`"Vicsek"` and the value of the key `"name"` in the dictionary *target*.

Type
str

v

The speed of the particles.

Type
float

dt

Discretisation time-step.

Type
float

sigma

The diffusion coefficient.

Type
float

nu

The drift coefficient

Type
float

kappa

Concentration parameter equal to the ratio of *nu* over *sigma*.

Type
(1,) Tensor

target

Target dictionary. Contains the target name and its parameters. To be passed to the method `compute_target()` in the argument `which_target`.

Type
dict

options

Options dictionary. Each key is an option's name and its value contains the parameters of the option. To be passed to the method `compute_target()` in the argument `which_options`.

Type
dict

scheme

The numerical scheme used to discretise the SDE. Can be one of the following:

- `"projection"` : Projected Euler-Maruyama scheme on the manifold $\mathbb{S}^{d-1}$.
- `"expEM"` : In dimension 2, Euler-Maruyama scheme in the Lie algebra and exponentiation.

Type
str

parameters

The parameters of the model.

Type
str

update()

Update the positions and velocities.

Update the positions using a first order explicit Euler method during the time step *dt*. Update the velocities using the numerical scheme *scheme* and the method *one_step_velocity_scheme()*.

Returns
positions and velocities.

Return type
dict

one_step_velocity_scheme(*targets*)

One time step of the numerical scheme for the velocity.

Parameters
targets (*(N,d) Tensor*) –

Returns
(N,d) Tensor

class sisyphe.models.**SynchronousVicsek**(*pos*, *vel*, *v*, *dt*, *kappa*, *interaction_radius*, *box_size*, *vision_angle=6.283185307179586*, *axis=None*, *boundary_conditions='periodic'*, *variant=None*, *options=None*, *sampling_method='uniform'*, *block_sparse_reduction=False*, *number_of_cells=1024*)

Vicsek model with synchronous random jumps.

This is the original Vicsek model described in [CGGR2008].

The velocity of each particle is a unit vector in the sphere $\mathbb{S}^{d-1}$. The velocity of each particle is updated at each iteration by computing first a *target* (Tensor) in $\mathbb{R}^d$ using the parameters contained in the dictionary *target* and then a new velocity in $\mathbb{S}^{d-1}$ is sampled using a method specified by *sampling_method*. The particles move in straight line at a constant speed.

name

"Synchronous Vicsek" and the value of the key "name" in the dictionary *target*.

Type
str

v

The speed of the particles.

Type
float

dt

Discretisation time-step.

Type
float

kappa

Concentration parameter.

Type
(1,) Tensor

target

Target dictionary. Contains the target name and its parameters. To be passed to the method compute_target() in the argument which_target.

Type
dict

options

Options dictionary. Each key is an option's name and its value contains the parameters of the option. To be passed to the method compute_target() in the argument which_options.

Type
dict

sampling_method

The new velocity is sampled using one of the following methods.

- "vonmises" : Von Mises distribution with center given by the normalised *target* and the concentration parameter given by the norm of *target*.
- "uniform" : Uniform distribution in the ball around the normalised *target* with deviation given by the inverse of the norm of *target*. Dimension 2 only.
- "projected_normal" : Sample first a gaussian random variable in $\mathbb{R}^d$ with center the normalised *target* and standard deviation the inverse of the norm of *target*. Then project the result on the sphere $\mathbb{S}^{d-1}$.

- "vectorial_noise" : Same as "projected_normal" where the normally distributed random variable is replaced by a uniformly distributed random variable on the sphere, divided by the norm of *target*.

Type
str

parameters

The parameters of the model.

Type
str

update()

Update the positions and velocities.

Update the positions using a first order explicit Euler method during the time step *dt*. Update the velocities by sampling new velocities using the method given by *sampling_method*.

Returns
Positions and velocities.

Return type
dict

class sisyphe.models.**CuckerSmale**(*pos*, *vel*, *sigma*, *nu*, *interaction_radius*, *box_size*, *alpha*, *beta*, *vision_angle=6.283185307179586*, *axis=None*, *boundary_conditions='periodic'*, *variant=None*, *options=None*, *dt=0.01*, *block_sparse_reduction=False*, *number_of_cells=1600*)

class sisyphe.models.**BOAsynchronousVicsek**(*pos*, *bo*, *v*, *jump_rate*, *kappa*, *interaction_radius*, *box_size*, *vision_angle=6.283185307179586*, *axis=None*, *boundary_conditions='periodic'*, *variant=None*, *options=None*, *sampling_method='vonmises'*, *block_sparse_reduction=False*, *number_of_cells=1600*)

Asynchronous Vicsek model for body-oriented particles.

The asynchronous Vicsek Model is described in [DFM2017].

The body-orientation of each particle is updated at random times by computing first a *target* (Tensor) in $SO(3)$ using the parameters contained in the dictionary *target* and then a new body-orientation in $SO(3)$ is sampled using a method specified by *sampling_method*. Between two jumps, a particle moves in straight line at a constant speed.

name

"Body-Orientation Asynchronous Vicsek" and the value of the key "name" in the dictionary *target*.

Type
str

v

The speed of the particles.

Type
float

dt

Discretisation time-step. It is equal to 0.01 divided by *jumprate*.

Type
float

jumprate

Each particle has an independent Poisson clock with parameter *jumprate*.

Type
float

kappa

Concentration parameter.

Type
(1,) Tensor

target

Target dictionary. Contains the target name and its parameters. To be passed to the method `compute_target()` in the argument `which_target`.

Type
dict

options

Options dictionary. Each key is an option's name and its value contains the parameters of the option. To be passed to the method `compute_target()` in the argument `which_options`.

Type
dict

sampling_method

The new velocity is sampled using one of the following methods.

- `"vonmises"` : Von Mises distribution with center given by the normalised *target* and the concentration parameter given by the norm of *target*.
- `"uniform"` : Uniform distribution in the ball around the normalised *target* with deviation given by the inverse of the norm of *target*. Dimension 2 only.

Type
str

parameters

The parameters of the model.

Type
str

update()

Update the positions and body-orientations.

Update the positions using a first order explicit Euler method during the time step *dt*. The particles which update their body-orientations in the time interval *dt* are given by the boolean mask of size `N` **who_jumps**. The probability for a particle to jump is equal to `exp(-jumprate * dt)`.

Returns
Positions and body-orientations.

Return type
dict

class sisyphe.models.**VolumeExclusion**(*pos*, *interaction_radius*, *box_size*, *alpha*, *division_rate*, *death_rate*, *dt*, *Nmax=20000*, *boundary_conditions='open'*, *block_sparse_reduction=False*, *number_of_cells=1024*)

Spherical particles interacting via short-range repulsion.

The model is described in [MP2017].

Each particle has a fixed radius and the particles repulse each other when they overlap. In addition the particles die and can divide at a constant rate. Currently implemented in an open domain only.

alpha

Drift coefficient.

Type
float

mu

Division rate.

Type
float

nu

Death rate.

Type
float

dt0

Default time step.

Type
float

dt

Current discretisation time step (adaptive).

Type
float

Nmax

Maximum number of particles.

Type
int

age

Age of the particles.

Type
(self.N,) Tensor

name

"Volume exclusion".

Type
str

property E

The energy of the particle system.

The force exerted by a particle located at x_j with radius R_j on a particle located at x_i with radius R_i is

$$F = -\frac{\alpha}{R_i}\nabla_{x_i} U\left(\frac{|x_i - x_j|^2}{(R_i + R_j)^2}\right),$$

where the potential is

$$U(s) = -\log(s) + s - 1 \text{ for } s < 1 \text{ and } U(s) = 0 \text{ for } s > 1.$$

The energy of the couple is

$$E_{ij} = U\left(\frac{|x_i - x_j|^2}{(R_i + R_j)^2}\right)(R_i + R_j)^2.$$

The total energy of the system is

$$E = -\frac{1}{2}\sum_{i \neq j} E_{ij}.$$

Returns
The total energy of the system.

Return type
(1,) Tensor

add_particle(*who*)

Some particle divide.

When a particle at x_i with radius R_i divides, a new particle is added at position $x_i + 2R_i u$ where u is uniformly sampled on the unit sphere. The radius of the new particle is R_i.

Parameters
who (*(N,) BoolTensor*) – Boolean mask giving the particles which divide.

remove_particle(*who*)

The particles in the boolean mask **who** die.

update()

Update the positions.

First compute the force. The Energy satisfies

$$\frac{\mathrm{d}}{\mathrm{d}t}E = -\frac{1}{2}\sum_{i \neq j}\frac{\alpha}{R_i}\left|\frac{\mathrm{d}x_i}{\mathrm{d}t}\right|^2 \leq 0.$$

Update the position using an adaptive time step:

1. Compute the energy of the system *E*.
2. Set `dt = dt0`.
3. Compute the new positions with time step *dt*.
4. Compute the energy of the new system.
5. If the energy of the new system is higher than *E* then set `dt = dt/2` and return to 3.

Once the positions are updated, compute the set of particles which divide during the time interval *dt*. Each particle has a probability `exp(-dt * mu)` to divide. Divisions are not allowed if there are more than *Nmax* particles. Finally compute the set of particles which die during the interval *dt*. Each particle has a probability `exp(-dt * nu)` to die. Death is not allowed if there are less that 2 particles.

Returns
Positions.

Return type
dict

class sisyphe.models.**AttractionRepulsion**(*pos*, *vel*, *interaction_radius*, *box_size*, *propulsion*, *friction*, *Ca*, *la*, *Cr*, *lr*, *dt*, *p=2*, *isaverage=False*, *vision_angle=6.283185307179586*, *axis=None*, *boundary_conditions='open'*, *block_sparse_reduction=False*, *number_of_cells=1000*)

Self-propelled particles with attraction and repulsion forces.

The model is studied in [DCBC2006].

The attraction-repulsion force is derived from the Morse potential. The self-propulsion force tends to relax the speed of the particles towards the ratio of the parameters *alpha* and *beta*.

alpha

Propulsion parameter.

Type
float

beta

Friction parameter.

Type
float

Ca

Attraction coefficient.

Type
float

la

Attraction length.

Type
float

Cr

Repulsion coefficient.

Type
float

lr

Repulsion length.

Type
float

p

Exponent of the Morse potential.

Type
int

dt

Time step.

Type
float

isaverage

Use the mean-field scaling or not.

Type
bool

mass

The mass is proportional to the volume.

Type
(N,) Tensor

name

'Self-propulsion and Attraction-Repulsion'.

Type
str

parameters

Parameters.

Type
str

compute_force()

Return the sum of the self-propulsion and attraction-repulsion forces.

update()

Update the positions and velocities. RK4 numerical scheme.

Returns
Positions and velocities.

Return type
dict

class sisyphe.models.**HardSpheres**(*pos*, *vel*, *radius*, *box_size*, *boundary_conditions*, *dt0*, *fig=None*)

4.8 Kernels

lazy_xy_matrix(x, y, L[, boundary_conditions])	XY matrix as LazyTensors.
lazy_interaction_kernel(x, y, Rx, L, ...[, ...])	Interaction kernel as LazyTensor.
lazy_overlapping_kernel(x, y, Rx, Ry, L, ...)	Overlapping kernel as LazyTensor.
lazy_morse(x, y, Ca, la, Cr, lr[, p, mx, my])	LazyTensor of the forces derived from the Morse potential.
lazy_quadratic(x, y, R, L[, boundary_conditions])	LazyTensor of the forces derived from of a quadratic potential.

sisyphe.kernels.**squared_distances_tensor**(*x*, *y*, *L*, *boundary_conditions='periodic'*)

Squared distances matrix as torch Tensors.

Parameters

- **x** (*(M,d) Tensor*) – Rows.
- **y** (*(N,d) Tensor*) – Columns.
- **L** (*(d,) Tensor*) – Box size.
- **boundary_conditions** (*list or str, optional*) – Boundary conditions. Can be one of the following:
  - list of size d containing 0 (periodic) or 1 (wall with reflecting boundary conditions) for each dimension.
  - "open" : no boundary conditions.
  - "periodic" : periodic boundary conditions.
  - "spherical" : reflecting boundary conditions on the sphere of radius L[0]/2 and center L/2.

  Default is "periodic".

Returns

The tensor whose (i, j) coordinate is the squared distance between x[i,:] and y[j,:].

Return type

(M,N) Tensor

sisyphe.kernels.**lazy_xy_matrix**(*x*, *y*, *L*, *boundary_conditions='periodic'*)

XY matrix as LazyTensors.

Parameters

- **x** (*(M,d) Tensor*) – Rows.
- **y** (*(N,d)*) – Columns.
- **L** (*(d,) Tensor*) – Box size.
- **boundary_conditions** (*list or str, optional*) – Boundary conditions. Can be one of the following:
  - list of size d containing 0 (periodic) or 1 (wall with reflecting boundary conditions) for each dimension.
  - "open" : no boundary conditions.
  - "periodic" : periodic boundary conditions.

- "spherical" : reflecting boundary conditions on the sphere of radius L[0]/2 and center **L**/2.

Default is "periodic".

Returns
The LazyTensor A such that for each coordinate (i, j), A[i,j,:] is the LazyTensor LazyTensor(y[j,:]-x[i,:]).

Return type
(M,N,d) LazyTensor

sisyphe.kernels.**squared_distances**(*x*, *y*, *L*, *boundary_conditions='periodic'*)

Squared distances LazyTensor.

Parameters

- **x** (*(M,d) Tensor*) – Rows.
- **y** (*(N,d)*) – Columns.
- **L** (*(d,) Tensor*) – Box size.
- **boundary_conditions** (*list or str, optional*) – Boundary conditions. Can be one of the following:
  - list of size d containing 0 (periodic) or 1 (wall with reflecting boundary conditions) for each dimension.
  - "open" : no boundary conditions.
  - "periodic" : periodic boundary conditions.
  - "spherical" : reflecting boundary conditions on the sphere of radius L[0]/2 and center **L**/2.

  Default is "periodic".

Returns
The LazyTensor whose (i, j) coordinate is the squared distance between x[i,:] and y[j,:].

Return type
(M,N) LazyTensor

sisyphe.kernels.**sqdist_angles**(*x*, *y*, *x_orientation*, *L*, *boundary_conditions='periodic'*)

Squared distances and angles LazyTensors.

Parameters

- **x** (*(M,d) Tensor*) – Rows.
- **y** (*(N,d)*) – Columns.
- **x_orientation** (*(M,d) Tensor*) – Orientation of x.
- **L** (*(d,) Tensor*) – Box size.
- **boundary_conditions** (*list or str, optional*) – Boundary conditions. Can be one of the following:
  - list of size d containing 0 (periodic) or 1 (wall with reflecting boundary conditions) for each dimension.
  - "open" : no boundary conditions.
  - "periodic" : periodic boundary conditions.

- `"spherical"` : reflecting boundary conditions on the sphere of radius `L[0]/2` and center **L**/2.

Default is `"periodic"`.

Returns

The LazyTensor whose (i, j) coordinate is the squared distance between `x[i,:]` and `y[j,:]` and the LazyTensor whose (i, j) coordinate is the cosine of the angle between `y[j,:]-x[i,:]` and `x_orientation[i,:]`.

Return type

(M,N) LazyTensor, (M,N) LazyTensor

sisyphe.kernels.**lazy_interaction_kernel**(*x*, *y*, *Rx*, *L*, *boundary_conditions*, *vision_angle=6.283185307179586*, *axis=None*, ***kwargs*)

Interaction kernel as LazyTensor.

Parameters

- **x** (*(M,d) Tensor*) – Row.
- **y** (*(N,d) Tensor*) – Columns.
- **Rx** (*(N,) Tensor or float*) – Interaction radius of x.
- **L** (*(d,) Tensor*) – Box size.
- **boundary_conditions** (*list or str, optional*) – Boundary conditions. Can be one of the following:
  - list of size d containing 0 (periodic) or 1 (wall with reflecting boundary conditions) for each dimension.
  - `"open"` : no boundary conditions.
  - `"periodic"` : periodic boundary conditions.
  - `"spherical"` : reflecting boundary conditions on the sphere of radius `L[0]/2` and center **L**/2.

  Default is `"periodic"`.
- **vision_angle** (*float, optional*) – Angle of vision of x. Default is 2π.
- **axis** (*(M,d) Tensor, optional*) – Orientation of **x**. Default is `None`. Must be specified if **vision_angle** is not 2π.
- ****kwargs** – Arbitrary keywords arguments.

Returns

The LazyTensor whose (i, j) coordinate is 1 if `y[j,:]` is in the cone of vision of `x[i,:]` and 0 otherwise.

Return type

(M,N) LazyTensor

sisyphe.kernels.**lazy_overlapping_kernel**(*x*, *y*, *Rx*, *Ry*, *L*, *boundary_conditions*, ***kwargs*)

Overlapping kernel as LazyTensor.

Parameters

- **x** (*(M,d) Tensor*) – Rows.
- **y** (*(N,d) Tensor*) – Columns.
- **Rx** (*(M,) Tensor or float*) – Radius of x.

- **Ry** (*(N,) Tenosr or float*) – Radius of y.
- **L** (*(d,) Tensor*) – Box size.
- **boundary_conditions** (*list or str, optional*) – Boundary conditions. Can be one of the following:
  - list of size d containing 0 (periodic) or 1 (wall with reflecting boundary conditions) for each dimension.
  - "open" : no boundary conditions.
  - "periodic" : periodic boundary conditions.
  - "spherical" : reflecting boundary conditions on the sphere of radius L[0]/2 and center **L**/2.

  Default is "periodic".
- ****kwargs** – Arbitrary keywords arguments.

Returns
The LazyTensor whose (i, j) coordinate is 1 if x[i,:] and y[j,:] are at a distance smaller than Rx[i]+Ry[j] (or smaller than Rx+Ry if all the **x** and **y** particles have the same radius).

Return type
(M,N) LazyTensor

Raises
NotImplementerError – if **Rx** and **Ry** do not have the same type, or when **Rx** and **Ry** are float but **Rx** is not equal to **Ry**.

sisyphe.kernels.**lazy_morse**(*x*, *y*, *Ca*, *la*, *Cr*, *lr*, *p=2*, *mx=1.0*, *my=1.0*, ***kwargs*)

LazyTensor of the forces derived from the Morse potential.

The Morse potential is defined by

$$U(z) = -C_a \exp(-|z|^p/\ell_a^p) + C_r \exp(-|z|^p/\ell_r^p)$$

The force exerted by a particle located in y_j on a particle located on x_i is

$$F = -m_x m_y \nabla U(x_i - y_j)$$

where m_x and m_y are the masses of the particles in x_i and y_j respectively.

Parameters

- **x** (*(M,d) Tensor*) – Rows.
- **y** (*(N,d) Tensor*) – Columns.
- **Ca** (*float*) – Attraction coefficient.
- **la** (*float*) – Attraction length.
- **Cr** (*float*) – Repulsion coefficient.
- **lr** (*float*) – Repulsion length.
- **p** (*int, optional*) – Exponent. Default is 2.
- **mx** (*(M,) Tensor or float, optional*) – Mass of **x**. Default is 1.
- **my** (*(N,) Tensor or float, optional*) – Mass of **y**. Default is 1.
- ****kwargs** – Arbitrary keywords arguments.

Returns
The LazyTensor whose $(i, j, :)$ coordinate is the force exerted by `y[j,:]` on `x[i,:]` derived from the Morse potential.

Return type
(M,N,d) LazyTensor

`sisyphe.kernels.`**`lazy_quadratic`**(*x*, *y*, *R*, *L*, *boundary_conditions='periodic'*, ***kwargs*)

LazyTensor of the forces derived from of a quadratic potential.

The quadratic potential is defined by:

$$U(z) = \frac{1}{2R}|z|^2 - |z|$$

where R is a fixed parameter. The force exerted on a particle located on y_j on a particle located on x_i is

$$F = \left(\frac{1}{R} - \frac{1}{|y_j - x_i|}\right)(y_j - x_i).$$

Parameters

- **`x`** (*`(M,d) Tensor`*) – Rows.
- **`y`** (*`(N,d) Tensor`*) – Columns.
- **`R`** (*`(M,) Tensor or float`*) – Radius of the **x** particles.
- **`L`** (*`(d,) Tensor`*) – Box size.
- **`boundary_conditions`** (*`list or str, optional`*) – Boundary conditions. Can be one of the following:
  - list of size d containing 0 (periodic) or 1 (wall with reflecting boundary conditions) for each dimension.
  - `"open"` : no boundary conditions.
  - `"periodic"` : periodic boundary conditions.
  - `"spherical"` : reflecting boundary conditions on the sphere of radius `L[0]/2` and center **L**/2.

  Default is `"periodic"`.
- **`**kwargs`** – Arbitrary keywords arguments.

Returns
The LazyTensor whose $(i, j, :)$ coordinate is the force exerted by `y[j,:]` on `x[i,:]` derived from the quadratic potential.

Return type
(M,N,d) LazyTensor

4.9 Sampling

uniform_sphere_rand(N, d, device[, dtype])	Uniform sample on the unit sphere $\mathbb{S}^{d-1}$.
uniform_angle_rand(mu, eta)	Uniform sample in an angle interval in dimension 2.
vonmises_rand(mu, kappa)	Von Mises sample on the sphere $\mathbb{S}^{d-1}$ using the Ulrich-Wood algorithm [W1994].
uniformball_unitquat_rand(q, scales)	Sample rotations at random in balls centered around q, with radii given by the scales array.
vonmises_quat_rand(q, kappa)	Von Mises random variables on the space of quaternions with center **q** and concentration parameter **kappa**.

sisyphe.sampling.**uniform_sphere_rand**(*N*, *d*, *device*, *dtype=torch.float32*)

Uniform sample on the unit sphere $\mathbb{S}^{d-1}$.

Parameters

- **N** (*int*) – Number of samples.
- **d** (*int*) – Dimension
- **device** (*torch.device*) – Device on which the samples are created
- **dtype** (*torch.dtype, optional*) – Default is torch.float32

Returns
Sample.

Return type
(N,D) Tensor

sisyphe.sampling.**uniform_angle_rand**(*mu*, *eta*)

Uniform sample in an angle interval in dimension 2.

Add a uniform angle in $[-\pi\eta, \pi\eta]$ to the angle of **mu**.

Parameters

- **mu** (*(N,2) Tensor*) – Center of the distribution for each of the N samples.
- **eta** (*(N,2) Tensor or float*) – Deviation for each of the N samples.

Returns
Sample.

Return type
(N,2) Tensor

sisyphe.sampling.**sample_W**(*kappa*, *N*, *d*)

Sample the first coordinate of a von Mises distribution with center at $(1, 0, ...0)^T$ and concentration **kappa**.

Parameters

- **kappa** (*(N,) Tensor*) – Concentration parameter. Also accept (1,) Tensor if all the samples have the same concentration parameter.
- **N** (*int*) – Number of samples. Must be equal to the size of kappa when the size of kappa is >1.
- **d** – Dimension.

Returns
Random sample in $\mathbb{R}$.

Return type
(N,) Tensor

sisyphe.sampling.**vonmises_rand**(*mu*, *kappa*)

Von Mises sample on the sphere $\mathbb{S}^{d-1}$ using the Ulrich-Wood algorithm [W1994].

Parameters

- **mu** (*(N,d) Tensor*) – Centers of the N samples.
- **kappa** – ((N,) Tensor or (1,) Tensor): Concentration parameter.

Returns
Sample with center **mu** and concentration parameter **kappa**.

Return type
(N,d) Tensor

sisyphe.sampling.**sample_angles**(*N*, *scales*)

Sample angles by rejection sampling.

sisyphe.sampling.**uniformball_unitquat_rand**(*q*, *scales*)

Sample rotations at random in balls centered around q, with radii given by the scales array.

sisyphe.sampling.**uniform_unitquat_rand**(*N*, *device*, *dtype=torch.float32*)

Uniform sample in the space of unit quaternions.

Parameters

- **N** (*int*) – Number of samples.
- **device** (*torch.device*) – Device on which the samples are created.
- **dtype** (*torch.dtype, optional*) – Default is torch.float32.

Returns
Samples.

Return type
(N,4) Tensor

sisyphe.sampling.**sample_q0**(*kappa*, *N*)

Sample **N** random variables distributed according to the von Mises distribution on the space of quaternions with center $(1, 0, 0, 0)$ and concentration parameter **kappa**.

Use the BACG method described in [KGM2018].

Parameters

- **kappa** (*(N,) Tensor or (1,) Tensor*) – Concentration parameter.
- **N** (*int*) – Number of samples. Must be equal to the size of kappa when the size of kappa is >1.

Returns
Sample.

Return type
(N,) Tensor

sisyphe.sampling.**vonmises_quat_rand**(*q*, *kappa*)

Von Mises random variables on the space of quaternions with center **q** and concentration parameter **kappa**.

Multiply **q** with a sample from the von Mises distribution on the space of quaternions with center $(1, 0, 0, 0)$ and concentration parameter **kappa**. See *sample_q0()*.

Parameters

- **q** (*(N,4) Tensor*) – Center of the distribution.
- **kappa** (*(N,) Tensor or (1,) Tensor*) – Concentration parameter.

Returns
Sample.

Return type
(N,4) Tensor

4.10 Display

save(simu, frames, attributes, ...[, ...])	Basic saving function.
display_kinetic_particles(simu, time[, ...])	Basic quiver plot in dimension 2 and scatter plot in dimension 3.
scatter_particles(simu, time[, show, save, path])	Scatter plot with the radii of the particles.

sisyphe.display.**save**(*simu*, *frames*, *attributes*, *attributes_alltimes*, *Nsaved=None*, *save_file=True*, *file_name='data'*)

Basic saving function.

Parameters

- **simu** (*Particles*) – A model.
- **frames** (*list*) – List of times (float) to save.
- **attributes** (*list*) – List of strings containing the names of the attributes to save at the times in the list **frames**.
- **attributes_alltimes** (*list*) – List of strings containing the names of the attributes which will be saved at each iteration until the last time in frame.
- **Nsaved** (*int, optional*) – Default is None which means that all the particles will be saved.
- **save_file** – (bool, optional): Default is True which means that a pickle file will be generated and saved in the current directory.
- **file_name** (*str, optional*) – Default is "data".

Returns
A dictionary which contains the entry keywords "time", "frames" and all the elements of the lists **attributes** and **attributes_alltimes**. The dictionary is saved in the current directory.

Return type
dict

sisyphe.display.**display_kinetic_particles**(*simu*, *time*, *N_dispmax=None*, *order=False*, *color=False*, *show=True*, *figsize=(6, 6)*, *save=False*, *path='simu'*)

Basic quiver plot in dimension 2 and scatter plot in dimension 3.

Parameters

- **simu** (Particles) – A model.
- **time** (*list*) – List of times to plot.
- **N_dispmax** (*int, optional*) – Default is None.
- **order** (*bool, optional*) – Compute the order parameter or not. Default is False.
- **color** (*bool, optional*) – The color of the particle depends on the velocity angle (dimension 2). Default is False.
- **show** (*bool, optional*) – Show the plot. Default is True.
- **figsize** (*tuple, optional*) – Figure size, default is (6,6).
- **save** (*bool, optional*) – Save the plots. Default is False.
- **path** (*str, optional*) – The plots will be saved in the directory `'./path/frames'`.

Returns
Times and order parameter

Return type
list, list

`sisyphe.display.`**`scatter_particles`**(*simu*, *time*, *show=True*, *save=False*, *path='simu'*)

Scatter plot with the radii of the particles.

Parameters

- **simu** (Particles) – A model.
- **time** (*list*) – List of times to plot.
- **show** (*bool, optional*) – Show the plot. Default is True.
- **save** (*bool, optional*) – Save the plots. Default is False.
- **path** (*str, optional*) – The plots will be saved in the directory `'./path/frames'`.

4.11 Toolbox

`sisyphe.toolbox.`**`volume_ball`**(*d*)

Volume of the ball of radius 1 in dimension d

`sisyphe.toolbox.`**`uniform_grid_separation`**(*x*, *r*, *L*)

Return the label of the cell to which each row of **x** belongs.

The box $[0, L[0]] \times ... \times [0, L[d-1]]$ is divided in cells of size r. The cells are numbered as follows in dimension 2:

(K[1]-1)*K[0]	(K[1]-1)*K[0]+1	...	K[0]*K[1]-1
...	...	...	...
K[0]	K[0]+1	...	2*K[0]-1
0	1	...	K[0]-1

where `K[i]` is the number of cells along the dimension `i`.

Parameters

- **x** (*(N,d) Tensor*) – Positions.
- **r** (*(d,) Tensor*) – Size of the cells at each dimension.
- **L** (*(d,) Tensor*) – Box size.

Returns
The Tensor of labels.

Return type
(N,) IntTensor

Raises
NotImplementedError – if the dimension is $d = 1$ or $d > 3$.

sisyphe.toolbox.**uniform_grid_centroids**(*r*, *L*, *d*)

Return the coordinates of the centroids.

The box $[0, L[0]] \times ... \times [0, L[d-1]]$ is divided in cells of size r. The cells are numbered as follows in dimension 2:

(K[1]-1)*K[0]	(K[1]-1)*K[0]+1	...	K[0]*K[1]-1
...	...	...	...
K[0]	K[0]+1	...	2*K[0]-1
0	1	...	K[0]-1

where K[i] is the number of cells along the dimension i.

Parameters

- **r** (*(d,) Tensor*) – Size of the cells at each dimension.
- **L** (*(d,) Tensor*) – Box size.
- **d** (*int*) – Dimension.

Returns
The Tensor of centroids: the i component is the center of the cell numbered i. The first dimension N_cell is the total number of cells (in the example above, N_cell = K[0]*K[1]).

Return type
(N_cell,d) Tensor

Raises
NotImplementedError – if $d = 1$ or $d > 3$.

sisyphe.toolbox.**block_sparse_reduction_parameters**(*x*, *y*, *centroids*, *eps*, *L*, *keep*, *where_dummies=False*)

Compute the block sparse reduction parameters.

Classical block sparse reduction as explained in the documentation of the KeOps library. The main difference is that the centroids and labels are computed using the *uniform_grid_centroids()* and the *uniform_grid_separation()* respectively. With this method a cell may be empty. Dummy particles are thus added to **x** and **y** at the positions of the centroids.

Parameters

- **x** (*(M,d) Tensor*) – Rows.
- **y** (*(N,d) Tensor*) – Columns

- **centroids** (*(nb_centr,d) Tensor*) – Centroids.
- **eps** (*(d,) Tensor*) – Size of the cells.
- **L** (*(d,) Tensor*) – Size of the box.
- **keep** (*(nb_centro,nb_centro) BoolTensor*) – `keep[i,j]` is True when `centroids[i,:]` and `centroids[j,:]` are contiguous.
- **where_dummies** (*bool, optional*) – Default is False.

Returns

The tuple of block sparse parameters:

- **x_sorted** *((M+nb_centro,d) Tensor)*: The sorted **x_plus** where **x_plus** is the concatenation of **x** and the centroids.
- **y_sorted** *((N+nb_centro,d) Tensor)*: The sorted **y_plus** where **y_plus** is the concatenation of **y** and the centroids.
- **nb_centro** *(int)*: Number of centroids.
- **labels_x** *((M+nb_centro,) IntTensor)*: The labels of **x_plus**.
- **labels_y** *((n+nb_centro,) IntTensor)*: The labels of **y_plus**.
- **permutation_x** *((M+nb_centro,) IntTensor)*: The permutation of the **x_plus** such that `x_plus[permutation_x,:]` is equal to **x_sorted**.
- **ranges_ij**: Result of `from_matrix()`.
- **map_dummies** *((M+nb_centro,N+nb_centro) LazyTensor)*: The (i, j) coordinate is 0 if either `x_sorted[i,:]` or `y_sorted[j,:]` is a centroid and 1 otherwise. **map_dummies** is None when **where_dummies** is False.

Return type
tuple

sisyphe.toolbox.**maximal_eigenvector_dim2**(*J*)

Eigenvectors of maximum eigenvalue for symmetric matrices in dimension 2.

Parameters
J (*(N,2,2) Tensor*) – Batch of N 2D symmetric matrices.

Returns
Unit eigenvector associated to the maximal eigenvalue of each matrix.

Return type
(N,2) Tensor

sisyphe.toolbox.**quat_mult**(*q_1*, *q_2*)

Multiplication in the space of quaternions.

sisyphe.toolbox.**angles_directions_to_quat**(*angles*, *directions*)

Represents a list of rotation angles and axes as quaternions.

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[1] G. Albi, N. Bellomo, L. Fermo, S.-Y. Ha, J. Kim, L. Pareschi, D. Poyato, and J. Soler. Vehicular traffic, crowds, and swarms: From kinetic theory and multiscale methods to applications and research perspectives. *Math. Models Methods Appl. Sci.*, 29(10):1901–2005, 2019. URL: https://www.worldscientific.com/doi/abs/10.1142/S0218202519500374, doi:10.1142/S0218202519500374.

[2] L. Andreis, P. Dai Pra, and M. Fischer. McKean–Vlasov limit for interacting systems with simultaneous jumps. *Stoch. Anal. Appl.*, 36(6):960–995, 2018. doi:10.1080/07362994.2018.1486202.

[3] O. Cappé, A. Guillin, J. M. Marin, and C. P. Robert. Population Monte Carlo. *J. Comput. Graph. Statist.*, 13(4):907–929, 2004. URL: http://www.tandfonline.com/doi/abs/10.1198/106186004X12803, doi:10.1198/106186004X12803.

[4] J. A. Carrillo, M. Fornasier, G. Toscani, and F. Vecil. Particle, kinetic, and hydrodynamic models of swarming. In G. Naldi, L. Pareschi, and G. Toscani, editors, *Mathematical Modeling of Collective Behavior in Socio-Economic and Life Sciences*, pages 297–336. Birkhäuser Boston, 2010. URL: http://link.springer.com/10.1007/978-0-8176-4946-3_12, doi:10.1007/978-0-8176-4946-3_12.

[5] B. Charlier, J. Feydy, J. A. Glaunès, F.-D. Collin, and G. Durif. Kernel Operations on the GPU, with Autodiff, without Memory Overflows. *J. Mach. Learn. Res.*, 22(74):1–6, 2021.

[6] H. Chaté, F. Ginelli, G. Grégoire, and F. Raynaud. Collective motion of self-propelled particles interacting without cohesion. *Phys. Rev. E*, 77(4):046113, 2008. URL: https://link.aps.org/doi/10.1103/PhysRevE.77.046113, doi:10.1103/PhysRevE.77.046113.

[7] L. Chizat and F. Bach. On the Global Convergence of Gradient Descent for Over-parameterized Models using Optimal Transport. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31 (NeurIPS 2018)*, 3040–3050. Montreal, Canada, 2018. Curran Associates, Inc.

[8] G. Clarté, A. Diez, and J. Feydy. Collective Proposal Distributions for Nonlinear MCMC samplers: Mean-Field Theory and Fast Implementation. *arXiv preprint: arXiv:1909.08988*, 2021. URL: http://arxiv.org/abs/1909.08988.

[9] A. Creppy, O. Praud, X. Druart, P. L. Kohnke, and F. Plouraboué. Turbulence of swarming sperm. *Phys. Rev. E*, 92(3):032722, 2015. URL: https://link.aps.org/doi/10.1103/PhysRevE.92.032722, doi:10.1103/PhysRevE.92.032722.

[10] Dan Crisan and Arnaud Doucet. A survey of convergence results on particle filtering methods for practitioners. *IEEE Trans. Signal Process.*, 50(3):736–746, 2002. URL: http://ieeexplore.ieee.org/document/984773/ (visited on 2021-05-12), doi:10.1109/78.984773.

[11] F. Cucker and S. Smale. On the mathematics of emergence. *Jpn. J. Math.*, 2(1):197–227, 2007. URL: http://link.springer.com/10.1007/s11537-007-0647-x, doi:10.1007/s11537-007-0647-x.

[12] V. De Bortoli, A. Durmus, X. Fontaine, and U. Simsekli. Quantitative Propagation of Chaos for SGD in Wide Neural Networks. *arXiv preprint: arXiv:2007.06352*, 2020. URL: http://arxiv.org/abs/2007.06352.

[13] P. Degond. Mathematical models of collective dynamics and self-organization. In *Proceedings of the International Congress of Mathematicians ICM 2018*, volume 4, 3943–3964. Rio de Janeiro, Brazil, August 2018. doi:10.1142/9789813272880_0206.

[14] P. Degond, A. Diez, and M. Na. Bulk topological states in a new collective dynamics model. *arXiv preprint: arXiv:2101.10864*, 2021. URL: http://arxiv.org/abs/2101.10864.

[15] P. Degond, A. Frouvelle, S. Merino-Aceituno, and A. Trescases. Alignment of Self-propelled Rigid Bodies: From Particle Systems to Macroscopic Equations. In G. Giacomin, S. Olla, E. Saada, H. Spohn, and G. Stoltz, editors, *Stochastic Dynamics Out of Equilibrium, Institut Henri Poincaré, Paris, France, 2017*, number 282 in Springer Proceedings in Mathematics & Statistics, pages 28–66. Springer, Cham, 2019. URL: http://link.springer.com/10.1007/978-3-030-15096-9_2, doi:10.1007/978-3-030-15096-9_2.

[16] P. Degond and S. Motsch. Continuum limit of self-driven particles with orientation interaction. *Math. Models Methods Appl. Sci.*, 18(Suppl.):1193–1215, 2008. URL: https://www.worldscientific.com/doi/abs/10.1142/S0218202508003005, doi:10.1142/S0218202508003005.

[17] P. Del Moral. *Feynman-Kac Formulae, Genealogical and Interacting Particle Systems with Applications*. Probability and Its Applications. Springer-Verlag New York, 2004.

[18] P. Del Moral. *Mean field simulation for Monte Carlo integration*. Number 126 in Monographs on Statistics and Applied Probability. CRC Press, Taylor & Francis Group, 2013. ISBN 9781466504059.

[19] G. Dimarco and S. Motsch. Self-alignment driven by jump processes: Macroscopic limit and numerical investigation. *Math. Models Methods Appl. Sci.*, 26(07):1385–1410, 2016. URL: https://www.worldscientific.com/doi/abs/10.1142/S0218202516500330, doi:10.1142/S0218202516500330.

[20] Arnaud Doucet, Nando Freitas, and Neil Gordon, editors. *Sequential Monte Carlo Methods in Practice*. Information Science and Statistics. Springer-Verlag New York, 2001. ISBN 978-0-387-95146-1. URL: http://link.springer.com/10.1007/978-1-4757-3437-9 (visited on 2021-05-13), doi:10.1007/978-1-4757-3437-9.

[21] N. Fournier and E. Löcherbach. On a toy model of interacting neurons. *Ann. Inst. Henri Poincaré Probab. Stat.*, 2016. URL: https://projecteuclid.org/journals/annales-de-linstitut-henri-poincare-probabilites-et-statistiques/volume-52/issue-4/On-a-toy-model-of-interacting-neurons/10.1214/15-AIHP701.full, doi:10.1214/15-AIHP701.

[22] S. Grassi and L. Pareschi. From particle swarm optimization to consensus based optimization: stochastic modeling and mean-field limit. *Math. Models Methods Appl. Sci.*, 31(8):1625–1657, 2020. URL: http://arxiv.org/abs/2012.05613, doi:10.1142/s0218202521500342.

[23] L. Greengard and V. Rokhlin. A fast algorithm for particle simulations. *J. Comput. Phys.*, 73(2):325–348, 1987. URL: https://linkinghub.elsevier.com/retrieve/pii/0021999187901409, doi:10.1016/0021-9991(87)90140-9.

[24] S.-Y. Ha, K. Lee, and D. Levy. Emergence of time-asymptotic flocking in a stochastic Cucker-Smale system. *Commun. Math. Sci.*, 7(2):453–469, 2009. doi:10.4310/cms.2009.v7.n2.a9.

[25] M. Hauray and S. Mischler. On Kac's chaos and related problems. *J. Funct. Anal.*, 266:6055–6157, 2014. doi:10.1016/j.jfa.2014.02.030.

[26] S. Jin, L. Li, and J.-G. Liu. Random batch methods (RBM) for interacting particle systems. *arXiv preprint: arXiv:1812.10575*, 2019. URL: http://arxiv.org/abs/1812.10575, doi:10.1016/j.jcp.2019.108877.

[27] Mark Kac. Foundations of kinetic theory. In *Proceedings of the Third Berkeley Symposium on Mathematical Statistics and Probability*, volume 3, 171–197. University of California Press Berkeley and Los Angeles, California, 1956.

[28] Nikolas Kantas, Arnaud Doucet, Sumeetpal Sidhu Singh, and Jan M. Maciejowski. An Overview of Sequential Monte Carlo Methods for Parameter Estimation in General State-Space Models. *IFAC Proceedings Volumes*, 42(10):774–785, 2009. URL: https://linkinghub.elsevier.com/retrieve/pii/S1474667016387432 (visited on 2021-05-12), doi:10.3182/20090706-3-FR-2004.00129.

[29] J. Kennedy and R. Eberhart. Particle swarm optimization. In *Proceedings of ICNN'95 - International Conference on Neural Networks*, volume 4, 1942–1948. Perth, WA, Australia, 1995. IEEE. URL: http://ieeexplore.ieee.org/document/488968/, doi:10.1109/ICNN.1995.488968.

[30] H. P. McKean. Propagation of chaos for a class of non-linear parabolic equations. In A. K. Aziz, editor, *Lecture Series in Differential Equations, Volume 2*, number 19 in Van Nostrand Mathematical Studies, pages 177–194. Van Nostrand Reinhold Company, 1969.

[31] S. Mei, A. Montanari, and P.-M. Nguyen. A mean field view of the landscape of two-layer neural networks. *Proc. Natl. Acad. Sci. USA*, 115(33):E7665–E7671, 2018. URL: http://www.pnas.org/lookup/doi/10.1073/pnas.1806579115, doi:10.1073/pnas.1806579115.

[32] S. Méléard. Asymptotic Behaviour of some interacting particle systems; McKean-Vlasov and Boltzmann models. In D. Talay and L. Tubaro, editors, *Probabilistic Models for Nonlinear Partial Differential Equations*, number 1627 in Lecture Notes in Mathematics. Springer-Verlag Berlin Heidelberg, 1996. doi:10.1007/bfb0093177.

[33] R. Pinnau, C. Totzeck, O. Tse, and S. Martin. A consensus-based model for global optimization and its mean-field limit. *Math. Models Methods Appl. Sci.*, 27(01):183–204, 2017. URL: https://www.worldscientific.com/doi/abs/10.1142/S0218202517400061, doi:10.1142/S0218202517400061.

[34] G. M. Rotskoff and E. Vanden-Eijnden. Trainability and Accuracy of Neural Networks: An Interacting Particle System Approach. *arXiv preprint: arXiv:1805.00915*, 2019. URL: http://arxiv.org/abs/1805.00915.

[35] J. Sirignano and K. Spiliopoulos. Mean Field Analysis of Neural Networks: A Law of Large Numbers. *SIAM J. Appl. Math.*, 80(2):725–752, 2020. URL: https://epubs.siam.org/doi/10.1137/18M1192184, doi:10.1137/18M1192184.

[36] A.-S. Sznitman. Topics in propagation of chaos. In *Éc. Été Probab. St.-Flour XIX—1989*, pages 165–251. Springer, 1991. doi:10.1007/bfb0085169.

[37] J. Toner and Y. Tu. Flocks, herds, and schools: A quantitative theory of flocking. *Phys. Rev. E*, 58(4):4828–4858, 1998. URL: https://link.aps.org/doi/10.1103/PhysRevE.58.4828, doi:10.1103/PhysRevE.58.4828.

[38] C. Totzeck. Trends in Consensus-based optimization. *arXiv preprint: arXiv:2104.01383*, 2021. URL: http://arxiv.org/abs/2104.01383.

[39] C. Totzeck, R. Pinnau, S. Blauth, and S. Schotthöfer. A Numerical Comparison of Consensus-Based Global Optimization to other Particle-based Global Optimization Schemes. *PAMM. Proc. Appl. Math. Mech.*, 2018. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/pamm.201800291, doi:10.1002/pamm.201800291.

[40] T. Vicsek, A. Czirók, E. Ben-Jacob, I. Cohen, and O. Shochet. Novel Type of Phase Transition in a System of Self-Driven Particles. *Phys. Rev. Lett.*, 75(6):1226–1229, 1995. URL: https://link.aps.org/doi/10.1103/PhysRevLett.75.1226, doi:10.1103/PhysRevLett.75.1226.

[41] T. Vicsek and A. Zafeiris. Collective motion. *Phys. Rep.*, 517(3-4):71–140, 2012. URL: https://linkinghub.elsevier.com/retrieve/pii/S0370157312000968, doi:10.1016/j.physrep.2012.03.004.

[42] H. H. Wensink, J. Dunkel, S. Heidenreich, K. Drescher, R. E. Goldstein, H. Lowen, and J. M. Yeomans. Meso-scale turbulence in living fluids. *Proc. Natl. Acad. Sci. USA*, 109(36):14308–14313, 2012. URL: http://www.pnas.org/cgi/doi/10.1073/pnas.1202032109, doi:10.1073/pnas.1202032109.

[DM2016] G. Di Marco, S. Motsch, Self-alignment driven by jump processes: Macroscopic limit and numerical investigation, *Math. Models Methods Appl. Sci.* Vol. 26, No. 7 (2016).

[DM2008] P. Degond, S. Motsch, Continuum limit of self-driven particles with orientation interaction, *Math. Models Methods Appl. Sci.* Vol. 18, Suppl. (2008).

[CGGR2008] H. Chaté, F. Ginelli, G. Grégoire, F. Raynaud, Collective motion of self-propelled particles interacting without cohesion, *Phys. Rev. E.*, Vol. 77, No. 4 (2008).

[DFM2017] P. Degond, A. Frouvelle, S. Merino-Aceituno, A new flocking model through body attitude coordination, *Math. Models Methods Appl. Sci.* Vol. 27, No. 6 (2017).

[MP2017] S. Motsch, D. Peurichard, From short-range repulsion to Hele-Shaw problem in a model of tumor growth, *J. Math. Biology*, Vol. 76, No. 1, 2017.

[DCBC2006] M. R. D'Orsogna, Y. L. Chuang, A. L. Bertozzi, L. S. Chayes, Self-Propelled Particles with Soft-Core Interactions: Patterns, Stability, and Collapse, *Phys. Rev. Lett.*, Vol. 96, No. 10 (2006).

[W1994] A. Wood, Simulation of the von Mises Fisher distribution, *Comm. Statist. Simulation Comput.*, Vol. 23, No. 1 (1994).

[KGM2018] J. Kent, A. Ganeiber & K. Mardia, A New Unified Approach for the Simulation of a Wide Class of Directional Distributions, *J. Comput. Graph. Statist.*, Vol. 27, No. 2 (2018).

PYTHON MODULE INDEX

S

INDEX

J

K

L

M

N

O

P