AM225 Final Project: Exploration of Efficient Simulation for Swarmalators

M. Elaine Cunha, Helen Read, and Jack Scudder

May 12, 2021

Abstract

Swarms have historically been of great interest, both experimentally and mathematically. Here, we present modifications on the model introduced by O'Keefe et al. [4], expanding it to n-dimensions, introducing external forcing, and allowing for a finite interaction distance. Additionally, we introduce a grid structure that allows for hypothetically faster computation in the case of finite interaction distance.

1 Introduction and Motivation

Groups of independent agents that manage to coordinate with one another and move in unison are inherently fascinating–at least, that's what our group decided as we stumbled upon swarmalators in a unique Homework 2 problem of AM225. The idea of swarms that coordinate synchronously invokes images ranging from nature to science fiction. Bees, birds, and fish all move together in time and space and their movements have been studied by biologists and physicists alike [4].[1] Some creative minds have invented fictional scenarios of small artificially intelligent robots wreaking havoc on humanity, such as the nanobots in Michael Crichton's 2002 novel *Prey*[2] or the robotic bees in the *Black Mirror* episode "Hated in the Nation."[3]

Several of these examples and more are presented in a seminal study published in 2017 by O'Keefe et al. In that paper, they coined the term "swarmalators" to represent "hypothetical systems [that] generalize swarms and oscillators" [4]. To the best of our knowledge, O'Keefe's team was the first to do research in this specific area and spawned a whole new branch of study in the years since their first paper. The group gives further description and motivation for their new term by stating that biologists are typically interested in *swarms* - "how animals move, while neglecting the dynamics of their internal states" - while physicists focus on *synchronization* - "oscillators' internal dynamics, not on motion." The models O'Keefe et al. propose form the basis for our project. Further details can be found in Section 2.

To build on our personal motivations for studying swarmalators, we liked that this topic combined differential equations, adaptive integration with dense output, physical and theoretical bases, and

[1] Detailed references can be found in the original paper by O'Keefe et al., particularly references 22-33.

[2] `https://en.wikipedia.org/wiki/Prey_(novel)`

[3] `https://en.wikipedia.org/wiki/Hated_in_the_Nation`

colorful plots. As seen above, it is a relatively new area of study, and lends itself to many exploratory permutations. We set out to accomplish three primary goals. First, we sought to build a code base that implemented models from the three papers briefly described below [4], [3], and [2] . The other papers appeared to use Python or MATLAB as their programming language of choice, so our code base is unique in that it is coded in C++. Second, we wanted to perform a qualitative analysis of our results compared to the reported papers' results to see if there were differences at higher resolution, i.e., using more agents. Third, by applying a custom grid-binning search method we were interested in seeing what kind of speedups we could observe in the finite cutoff method solution.

A brief overview of our method of analysis follows. We extrapolated techniques from O'Keefe et al. [4] to model multi-dimensional swarmalator agents. While our code works in n dimensions, we only plot in 2 and 3 dimensions, as the results are easier to visualize. To extend the swarmalator model further, we added external phase forcing, as described in Lizarraga et al. [3] and a finite interaction radius, based off of work in Lee et al. [2]. Finally, we used a custom grid-binning search method to improve the speed of calculations in the case of finite cutoff interactions between swarmalator agents.

While parts of our code are compatible with analysis in arbitrary dimensions, segments are restricted to 3 dimensions due to implementation strategies. This, as well as more details on how to use our code, is discussed further in Appendix A.

2 Background

2.1 Swarmalators in General

As described in the Introduction, the work by O'Keefe et al. [4] introduced the concept of swarmalators. Their model can be described by the following equations

$$\dot{\mathbf{x}}_i = \mathbf{v}_i + \frac{1}{N}\sum_{j\neq i}^{N}\left(\frac{\mathbf{x}_j - \mathbf{x}_i}{\|\mathbf{x}_j - \mathbf{x}_i\|_2}\cdot(A + J\cos(\theta_j - \theta_i)) - B\frac{\mathbf{x}_j - \mathbf{x}_i}{\|\mathbf{x}_j - \mathbf{x}_i\|_2^2}\right) \tag{1}$$

$$\dot{\theta}_i = \omega_i + \frac{K}{N}\sum_{j\neq i}^{N}\frac{\sin(\theta_j - \theta_i)}{\|\mathbf{x}_j - \mathbf{x}_i\|_2} \tag{2}$$

for $i \in \{1, \ldots, N\}$ where N is the number of swarmalator agents and $\mathbf{x}_i$ is the position vector in $\mathbb{R}^2$. In Section 3.2, we generalize to $\mathbf{x}_i \in \mathbb{R}^n$. For a given i, the values θ_i, ω_i, and $\mathbf{v}_i$ are the ith swarmalator's phase, natural frequency, and self-propulsion velocity respectively. For our experiments, we mimic many of the assumptions and simplifications made in the paper. We set $A = B = 1$ by re-scaling time and space. We set $\mathbf{v}_i = \omega_i = 0$ for all agents, assuming that they all have identical natural frequencies and no initial velocity, which also simplifies plotting as the swarms remain fixed on the same axes.

The parameters J and K are the primary values we modify to observe different output. J represents phase attraction between swarmalators based on their similarity, and K represents phase coupling

strength. There are five primary phases observed depending on various combinations of J and K. The states are as follows: Static synchrony (SS), Static asynchrony (SA), Static phase wave (SPW), Splintered phase wave (SpPW), and Active phase wave (APW). The J and K settings for each are depicted and described in Fig. 1. The first three static phases stop moving as they reach their respective steady states; whereas the splintered phase wave and active phase wave particles continue to move in time.

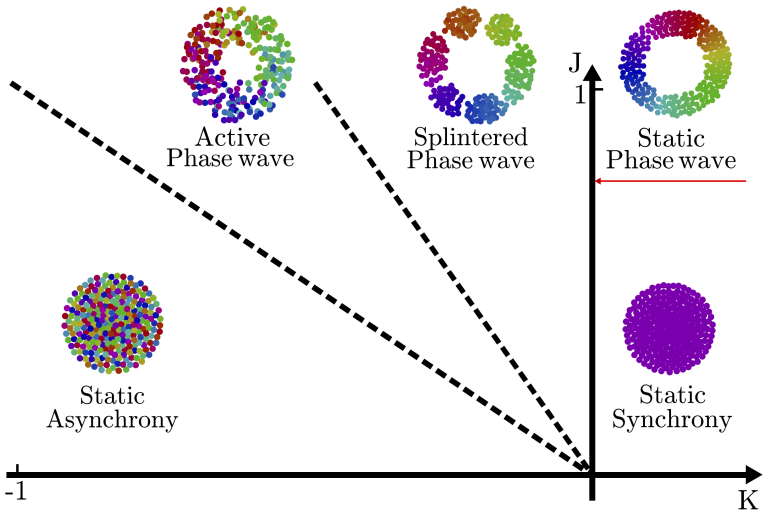

Figure 1: A visual representation of the five states from O'Keefe et al.'s model in [4]. **Static synchrony (SS)** exists for all J when $K > 0$. **Static asynchrony (SA)** exists for $J < |K_c|$ and $K < 0$, where $|K_c| \approx -1.2J$, the bottom-most downward sloping line. **Static phase wave (SPW)** exists where $J > 0$ and $K = 0$, shown by the red arrow beneath the description. **Splintered phase wave (SpPW)** and **Active phase wave (APW)** exist in the remaining portions of the $J > 0$ and $K < 0$ quadrant, but the true dividing line between the two is not as straight as the one depicted, and appears to only be determined through experimentation. This image was taken from [3]. A slightly more descriptive version that shows more accurate lines of division between phases can be found at Figure 1 in [4].

2.2 Finite Cutoff Interaction Distance

Another study examined swarmalators with a finite cutoff interaction distance, since in reality, an individual agent (like a bee, a fish, etc.) would only be able to receive information from other agents close in physical proximity [2]. We introduce r_c, as the critical radius, such that particles only interact if $\|\mathbf{x}_i - \mathbf{x}_j\|_2 < r_c$. This effectively presents another model parameter, like J and K, and our implementation of this model and results are discussed in Section 3.4. Lee et al. use the following equations for their finite cutoff model in $\mathbb{R}^2$.

$$\dot{\mathbf{r}}_i = \frac{1}{N_i(r_c)} \sum_{j \in \Lambda_i(r_c)}^{N} \left[\Big(1 + J\cos(\theta_j - \theta_i)\Big) - \frac{1}{r_{ij}^2} \right] \Big(\mathbf{r}_j - \mathbf{r}_i\Big) \tag{3}$$

$$\dot{\theta}_i = \omega_i + \frac{K}{N}\sum_{j\neq i}^{N}\frac{\sin(\theta_j - \theta_i)}{r_{ij}} \tag{4}$$

where $r_{ij} = \|\mathbf{x}_i - \mathbf{x}_j\|_2$, and $\mathbf{r}$ are used as the position vectors rather than $\mathbf{x}$. Additionally, they define $\Lambda_i(r_c)$ as the set of agents within r_c of the ith agent (excluding the ith agent itself) and $N_i(r_c) = |\Lambda_i(r_c)| + 1$.

At large r_c, these equations as written do not in fact converge to our original swarmalator equations, defined in Eqs. 1 and 2, as they are missing a factor of $1/r_{ij}$ in the first term. Additionally, they only use the finite cutoff for the spacial terms, while we use it for all terms[4]. Therefore, the equations we use are

$$\dot{\mathbf{x}}_i = \frac{1}{N_i(r_c)}\sum_{j\in\Lambda_i(r_c)}^{N}\left(\frac{\mathbf{x}_j - \mathbf{x}_i}{\|\mathbf{x}_j - \mathbf{x}_i\|_2}\cdot(1 + J\cos(\theta_j - \theta_i)) - \frac{\mathbf{x}_j - \mathbf{x}_i}{\left(\|\mathbf{x}_j - \mathbf{x}_i\|_2\right)^2}\right) \tag{5}$$

$$\dot{\theta}_i = \frac{K}{N}\sum_{j\in\Lambda_j(r_c)}^{N}\frac{\sin(\theta_j - \theta_i)}{\|\mathbf{x}_j - \mathbf{x}_i\|_2}. \tag{6}$$

The implementation of finite cutoff is discussed further in Section 3.4. Additionally, this interaction distance limitation led us to perform code optimizations to experiment with only considering nearby neighbors based on a grid architecture, also discussed in Section 3.4.

2.3 External Phase Forcing

A second swarmalator model we included was introduced by Lizarraga et al. in 2019, which modeled an external force applied to the swarmalators from a particular location. They likened the external perturbation to placing "a pulsing light [...] in the middle of a cloud of fireflies" [3].

The variations to Eqs. 7 and 8 are below for the two-dimensional case.

$$\dot{\mathbf{x}}_i = \frac{1}{N}\sum_{j\neq i}^{N}\left(\frac{\mathbf{x}_j - \mathbf{x}_i}{\|\mathbf{x}_j - \mathbf{x}_i\|_2}\cdot(A + J\cos(\theta_j - \theta_i)) - B\frac{\mathbf{x}_j - \mathbf{x}_i}{\|\mathbf{x}_j - \mathbf{x}_i\|_2^2}\right), \tag{7}$$

$$\dot{\theta}_i = F\cdot\frac{\cos(\Omega t - \theta_i)}{\|\mathbf{x}_0 - \mathbf{x}_i\|_2} + \frac{K}{N}\sum_{j\neq i}^{N}\frac{\sin(\theta_j - \theta_i)}{\|\mathbf{x}_j - \mathbf{x}_i\|_2} \tag{8}$$

The primary changes here are the removal of the $\mathbf{x}_i$ velocity component in Eq. 7 – which simplifies to one frame of reference – and the additional external forcing term in Eq. 8. F is amplitude, Ω is

[4]This made more practical sense to us, since the finite r_c is an approximation of the "observation distance" of an individual agent, and if an agent i cannot "observe" the position of an agent j, why can it observe its internal phase?

the frequency, and $\mathbf{x}_0$ is the location of the external force. Since the stimulus only affects the phase of the particles, not their position or velocity, the particles are able to move through the position of the external forcing function. We thought this would be an interesting addition to explore because the other variations rely on the initialization of the agents themselves – this was one of the options that introduced perturbations introduced from outside the swarmalator system.

3 Methods and Results

As done previously in the homework, we rescale our model such that $A = B = 1$, both for our initial unaltered model, seen in Figure 2 and later alterations. For all simulations, we initialize our points in the unit n-sphere, as discussed in Section 3.1. We generally use $N = 1000$ agents, as done in O'Keeefe et al. [4], but occasionally use less in order to make simulations run faster.

3.1 Initialization

To initialize agents' starting positions and phases, we used the GSL's libray of spherical[5] and uniform[6] distributions. The spherical distributions create uniform distributions along the surface of the n-sphere, so to uniformly distribute inside the n-sphere, we need to multiply each point by some radius r, rescaling it to be in the interior, rather than the boundary of the n-sphere. Since volume of an n-sphere with radius R scales like R^n, we cannot simply take a uniform distribution of $r \in (0, R)$. To get the correct distribution of r, we take the nth root of our uniformly random variable between $(0, R)$.

Then, we re-scale our $\{x_1, \ldots, x_n\}$ values generated from the spherical distributions by this re-scaled r. This allows us to generate a uniform random distribution in the unit n-sphere for arbitrary dimension n, the results of which can be seen in Figure 2.

3.2 Implementation in n Dimensions

While we focus on results in 2 and 3 dimensions here, as they are easy to visualize, we generalized our ODE function to work over an arbitrary number of dimensions n. We assume that the relationship between 2 and 3 dimensions carries over to n dimensions, so our system is

$$\dot{\mathbf{x}}_i = \frac{1}{N} \sum_{i \neq j} \left(\frac{\mathbf{x}_j - \mathbf{x}_i}{\|\mathbf{x}_j - \mathbf{x}_i\|_2} \left(1 + J \cos\left(\theta_j - \theta_i\right)\right) - \frac{\mathbf{x}_j - \mathbf{x}_i}{\left(\|\mathbf{x}_j - \mathbf{x}_i\|_2\right)^n} \right) \tag{9}$$

$$\dot{\theta}_i = \frac{K}{N} \sum_{i \neq j} \frac{\sin\left(\theta_j - \theta_i\right)}{\|\mathbf{x}_j - \mathbf{x}_i\|_2} \tag{10}$$

[5] `https://www.gnu.org/software/gsl/doc/html/randist.html#spherical-vector-distributions`
[6] `https://www.gnu.org/software/gsl/doc/html/rng.html#sampling-from-a-random-number-generator`

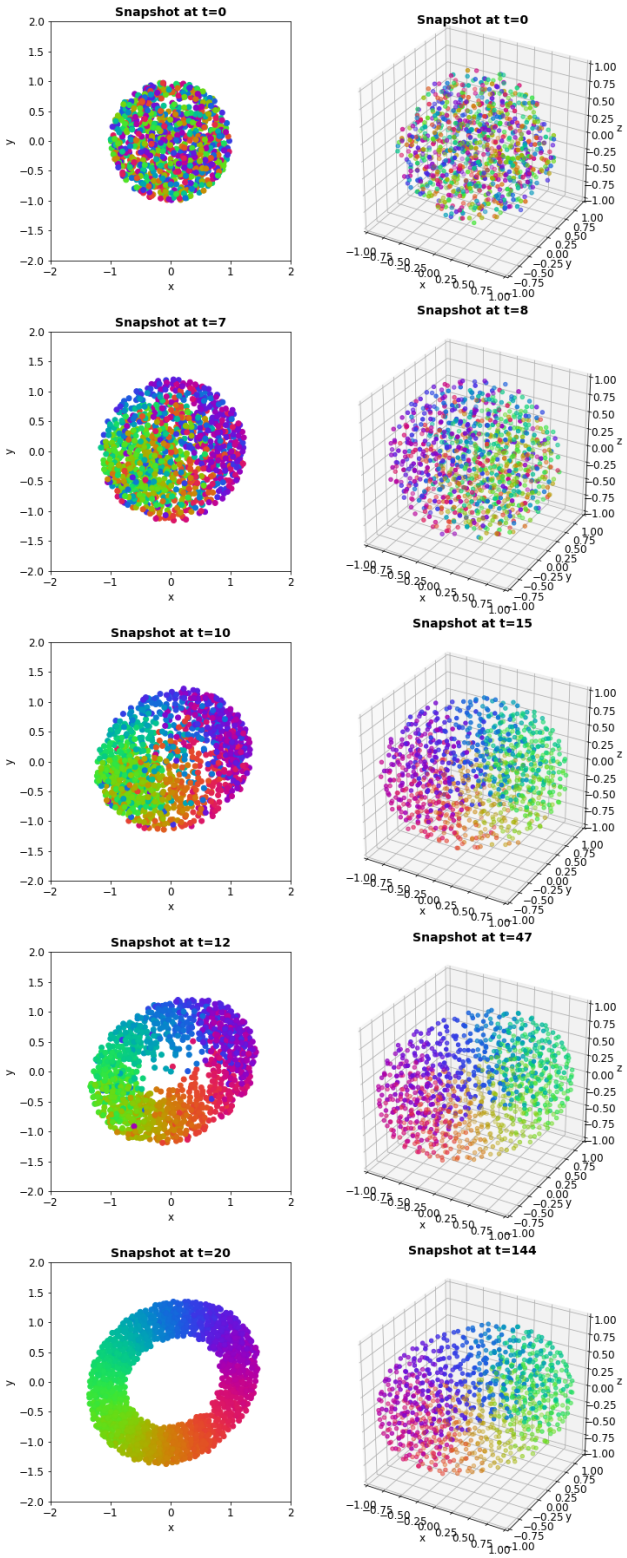

Figure 2: Replicated results in 2D and 3D from [4] for $J = 1$, $K = 0$, and $N = 1000$ which represents the Static Phase Wave (SPW) case.

for a general $\mathbf{x}_i = (x_1, x_2, \ldots, x_n)$. Then, if we define x_k^ℓ as the kth component of the ℓth agent, our state vector q takes the form

$$q = \left(x_1^1, x_2^1, \ldots, x_n^1, \theta^1, x_1^2, \ldots, x_n^2, \theta^2, \ldots, x_1^N, \ldots, x_n^N, \theta^N\right)$$

for N agents in n dimensions. Our vector q has length $N(n+1)$. To implement this, we iterate across each pair of agents. Then, we iterate over the number of dimensions of the system. This gives us three `for` loops, two of length N and one of length n. To more easily index into $\mathbf{x}_i$ and $\mathbf{x}_j$, we use pointer arithmetic. In n dimensions, we have $n+1$ components associated with each agent. Then, given our full state vector, q, indexing into it by $i(n+1)$ gives us x_1^i. In `swarm_ode.cc`, our state vector is called `in`. In our outer `for` loop over the N agents, we define

```
x_i = in + i*(dim + 1)
```

Then, `x_i` points to x_1^i, so we can index into `x_i` directly to obtain the components for $\mathbf{x}_i$. We do the same for $\mathbf{x}_j$ over the next `for` loop.

Besides the length of q, the only other change in our system is the exponent in the repulsion term. After calculating $L = \|\mathbf{x}_j - \mathbf{x}_i\|_2$, by using a private class function, we can calculate L^n by using a `for` loop over n. The results in three dimensions, calculated in this general way, can be seen in Figure 2.

3.3 External Forcing

In order to accommodate the parameters from Eqs. 7 and 8 in our code implementation, we only had to modify the constructor for the `swarm` class, and the update loops of the equations. The constructor now accepts additional inputs of `F, F_freq, F_locx, F_locy, F_locz` which correspond to F, Ω, and the components of $\mathbf{x}$ in 3-dimensions. The parameters are set to 0 by default, allowing the code to perform the other flavors of calculations (unperturbed or finite-cutoff) without including the external forcing effect. Within the update equations, we added a check to confirm whether or not there was a forcing term, F, present. The primary change within the calculation was including the difference between the location of the source, and the agent being evaluated, the denominator of the first term in Eq 8.

Our results in this section align with the paper introducing external forcing [3]. We used twice as many swarmalators in the simulation for a longer time period, $N = 500$ and 250 time steps vs. the paper's $N = 250$ at 110 time steps. For visualization in Figs. 3 and 4, we simulated the Static Synchrony case with the external forcing positioned at the origin of the plot. This state converges to a circle in 2D and a sphere in 3D. We varied the value of F, ranging from $0 - 4$. There didn't seem to be any difference in results from this change (besides our plots being more colorful with less white space). We also performed the tests in 3D, which the original authors did not, to the best of our knowledge. In these cases, the agents near the source within a distance proportional to F all take on the phase of the external phase forcing stimulus. The variation of color down a particular column corresponds to the frequency of the stimulus phase oscillation, fixed at $\Omega = 3\pi/2$

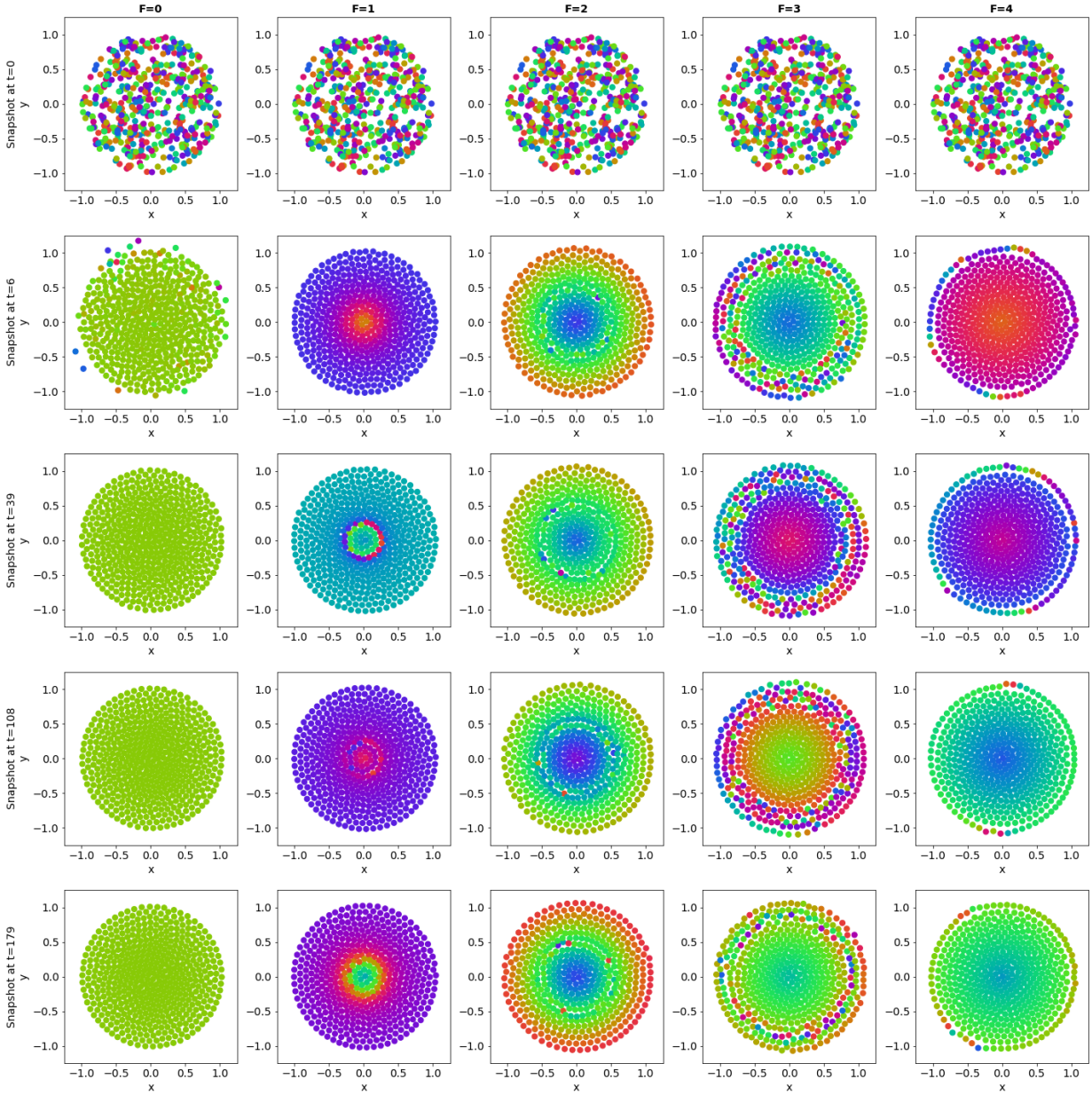

Figure 3: 2D results for an externally forced swarm with parameters $J = 0.1$, $K = 1$, $N = 500$, $\Omega = 3\pi/2$ with the perturbation emanating from the origin. Each column represents a different amplitude of force, F. These settings of J and K represents the Static Synchrony (SS) case.

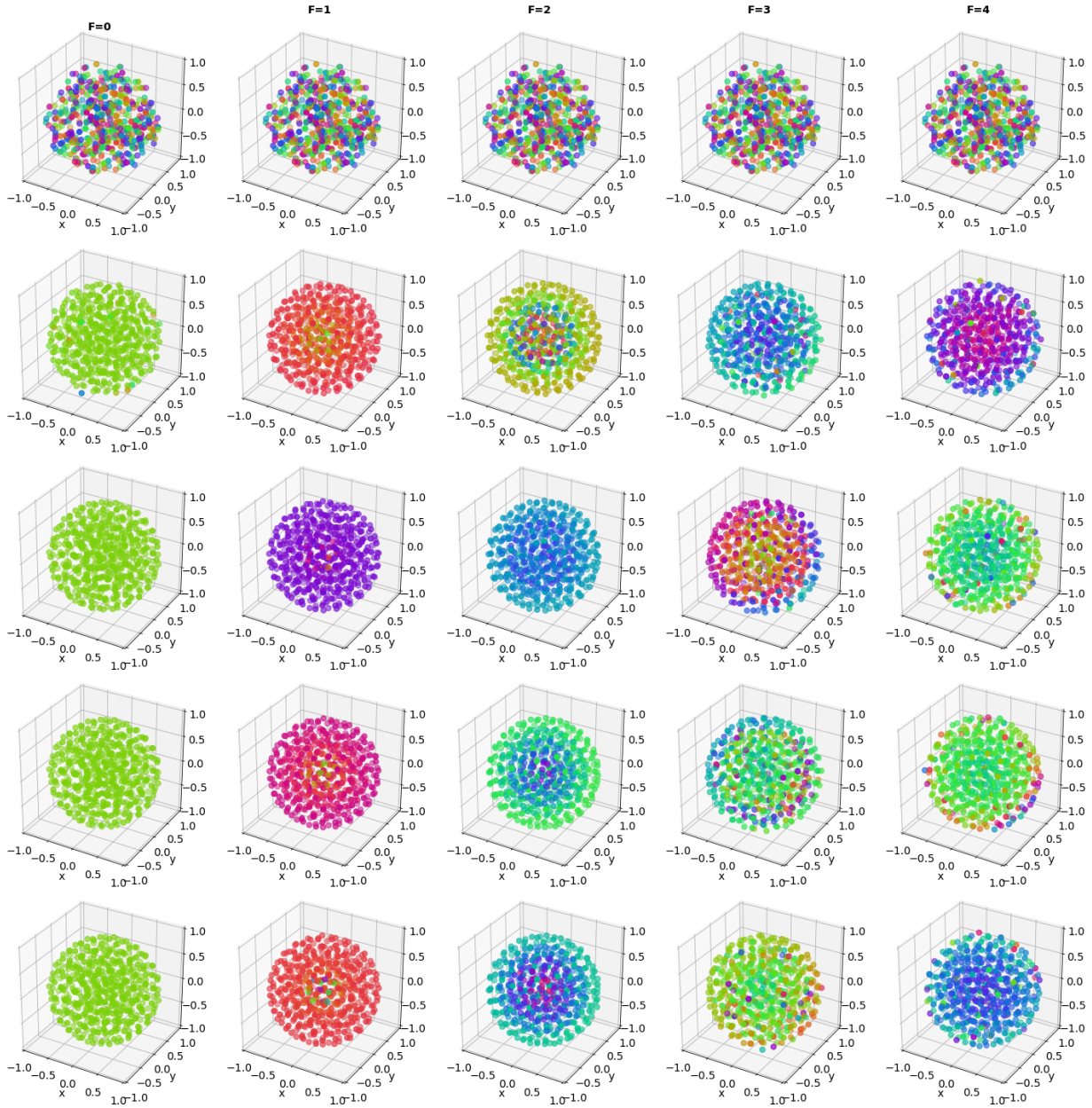

Figure 4: 3D results for externally forced swarm with parameters $J = 0.1$, $K = 1$, $N = 500$, $\Omega = 3\pi/2$ with the perturbation emanating from the origin. Each column represents a different amplitude of force, F. The rows from top to bottom represent snapshots at $t = 0, 9, 55, 119, 184$. These settings of J and K represents the Static Synchrony (SS) case.

in this case. The commentary above applies to both the 2D and 3D case, but is a testament to the fact that our code is flexible enough to model these cases in n dimensions.

3.4 Finite Cutoff

Implementing a finite cutoff radius for agent interactions primarily involved limiting the number of $\mathbf{x}_j$ agents used in evaluating Eq. 3 and 4. At each pair (i, j), we check that $i \neq j$ before proceeding to calculate the interaction between those two agents. To add a finite interaction radius, we add another function, `eval_interaction`, which returns `true` if the particles should interact, i.e., are within a distance r of each other, and `false` otherwise. To preserve the functionality of infinite interaction distance, we encode $r < 0$ to always return true.

Since the pair (i, j) should return the same value as (j, i), we can halve the number of distance calculations by using some memory. We use two bool arrays, `interactions`, and `checked_inter`, both of length N^2. These can be thought of representing matrices, called T and C respectively such that

$$T_{ij} = T_{ji} = \begin{cases} 1 & \|\mathbf{x}_j - \mathbf{x}_i\|_2 < r \\ 0 & \text{else} \end{cases}$$

and

$$C_{ij} = C_{ji} = \begin{cases} 1 & \text{We have calculated } \|\mathbf{x}_j - \mathbf{x}_i\|_2 \text{ at this time step} \\ 0 & \text{else} \end{cases}$$

At each distinct pair (i, j), and for a non-negative r[7], we first check C_{ij}. If $C_{ij} = 1$, we have previously checked $\|\mathbf{x}_j - \mathbf{x}_i\|_2$, so we return T_{ij}. If $C_{ij} = 0$, we calculate the distance between the two agents and update $T_{ij} = T_{ji}$ and $C_{ij} = Cji$ accordingly before returning T_{ij}.

While this code works, computing the distance between every pair of points is time consuming, especially in higher dimensions. Therefore, we implement grid-binning, which partitions our points into a finite number of cells. A schematic of this can be seen in Figure 5. Then, for each agent i in grid cell k, we compute which cells possibly contain points within radius r of agent $\mathbf{x}_i$[8].

With the grid-binning search implemented, we run our ODE under two different regimes, static synchrony ($J = 0.1$ and $K = 1$) and static asynchrony ($J = 0.1$ and $K = -1$). In the static synchrony setting, we expect agents of similar phases to move closer to one another because the phase coupling strength K is positive. In the static asynchrony setting, we expect the opposite - agents of similar phases should separate away from one another. In both cases, we used $N = 100$ agents and varied the interaction radius, r, beginning with $r = -1$ (an infinite interaction radius)

[7] We encode $r < 0$ to mean infinite interaction distance, since $r < 0$ is non-physical.

[8] This is done by finding the minimum and maximum x, y, and z values, and finding any grid cells that intersect with the cube these define. Notably, this is like using an L^∞ norm to find valid grid cells rather than the L^2 norm naturally created by the sphere surrounding the point $\mathbf{x}_i$. This is done to make the code simpler, however, since calculating if a grid cell narrowly misses the edge of the sphere is more difficult than our current implementation.

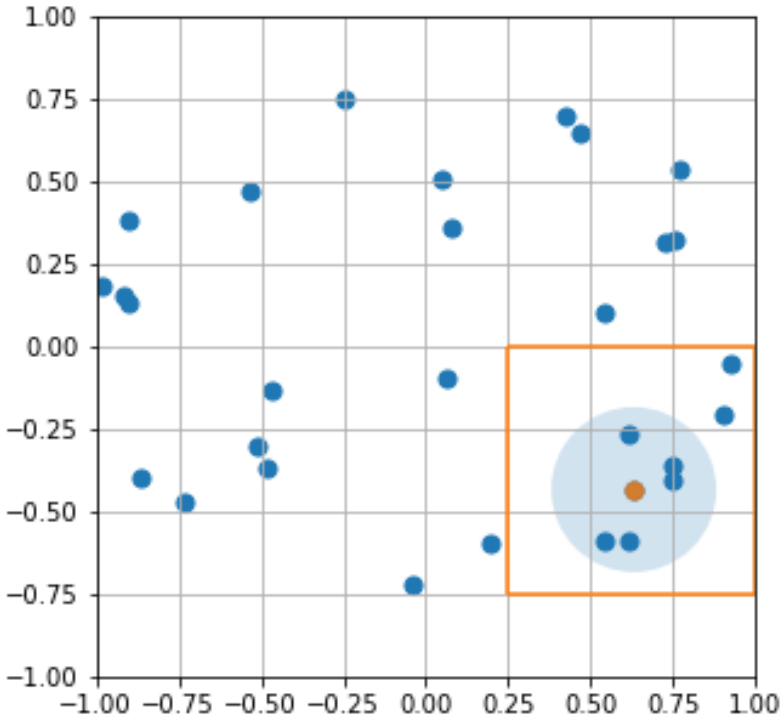

Figure 5: Schematic of how our grid-binning system works. For a given agent i (colored orange), we identify a subgrid of cells that are partially or fully within radius r of agent i. Radius r is illustrated above with a light blue circle and the subgrid for this example is outlined in orange. After this step, we exclusively search within the subgrid for agents that are within radius r of agent i by calculating the distance between each combination of two points. This approach should significantly reduce the total number of distance calculations needed, thus speeding up the code.

and decreasing it to 2, 1, and 0.5. Very long run times were observed for $N > 100$. Results are shown in Figures 6 and 7 for the static synchrony state and in Figures 8 and 9 for the static asynchrony state in 2 and 3 dimensions, respectively.

Comparing Figures 6 and 7 with Figures 8 and 9 reveals the influence of the phase coupling strength parameter K. When K is positive, we see the swarm forming "bubble" of like-colored agents within the specified interaction radius. For an infinite interaction radius, all agents influence each other's phase and position, thus the swarm converges to the same color across all agents. With a smaller interaction radius, the agents group into small sub-swarms of the same color but do not converge on a uniform phase, reflecting that position and phase are only influenced by nearby agents. When K is negative (Figures 8 and 9), the swarm separates into an equally spaced distribution of colors where neighboring agents have different colors. This result illustrates that a negative phase coupling strength physically pushes the position of similarly phased agents away from each other.

These results also demonstrate nuances in the choice of interaction radius r. We see in Figures 6-9 that the swarm size seems to "blow up" at $r = 1$. While we're not entirely sure why this

is occurring, we hypothesize that this has to do with the size of the repelling term. Since the repelling term scales with $1/r_{ij}^2$ (where $r_{ij} = \|\mathbf{x}_j - \mathbf{x}_i\|_2$), whereas the attractive term scales with $1/r_{ij}$, removing terms where $r_{ij} > 1$ as in the case that $r = 1$ would not lessen the repelling force as much as it would the attractive force. However, at smaller radii, like $r = 0.5$, we do begin to remove large repelling terms, making the swarm less likely to "blow up".

4 Conclusion and Future Direction

In this project we created a C++ code base that models swarmalators - oscillators that sync and swarm - a concept introduced by O'Keefe et al. in 2017 [4]. While we accomplished many extensions of the O'Keefe et al. model in our code base, there remain many areas for further analysis. First, we adopted the initial modeling concept from HW2 and expanded the scope of the problem to include 3D modeling, external forcing [3], and finite cutoff for interaction distance [2]. Second, we performed a qualitative analysis of our results and compared to those presented in publications. Finally, we included an alternate "grid-binning" method for measuring distance between swarmalators in the finite-cutoff distance scenario. We are especially proud of the fact that our minimum working example performs in a flexible n-dimensional setting.

We were able to reproduce some of the main results from the papers we analyzed, but didn't observe the runtime improvements we hypothesized we would see. Since this project was heavily reliant on visuals, we had to perform a qualitative comparison between our outputs and those presented in the papers. For the base cases, our plots match. At higher values of N number of agents, our plots did not reveal any new observations, other than the images are more suitable for becoming framed works of art in the vein of HW5.

Given some internal time constraints and code runtimes, we did not perform an extensive analysis of all permutations. For example, the finite-cutoff authors noted a bar pattern appearing for radii between approximately $1.2 - 1.8$ [2] and the external forcing authors noted new phase patterns for the SPW and SpPW at $F = 1$. Furthermore, our grid-binning method didn't perform as well as we had hoped. At small values of N, the original code that analyzed every pairwise comparison between agents was faster, perhaps because the grid-binning method relied on more 'for' loops in its implementation. We assume that the grid-binning method may need high values of N to truly realize its full potential.

Although we made several code modifications, there are a few areas on which we could improve given an infinite time horizon. First, we would have liked to spend more time on finding ways to speed up the code. We ran into several bugs with each additional layer of complexity, which hampered our ability to optimize speedup. Luckily, the basic implementation and modifications in 2D and 3D work, so a next series of steps would be to (1) add multithreading, (2) improve grid-binning to reduce the number of pairwise comparisons and ideally the number of for-loops present. Second, we would like to reach out to the original authors and see if they would share their code (or at least runtimes) so that we could do a true programming language comparison. We had hoped the sheer speed at which our code ran would convince us of the ultimate superiority of our code, but clearly some limitation in our implementation led us to believe there is a more efficient way, and perhaps a Python or MATLAB version would inspire us to look at a different ordering or use of external libraries. Finally, we would animate our plots to include videos (similar to what the

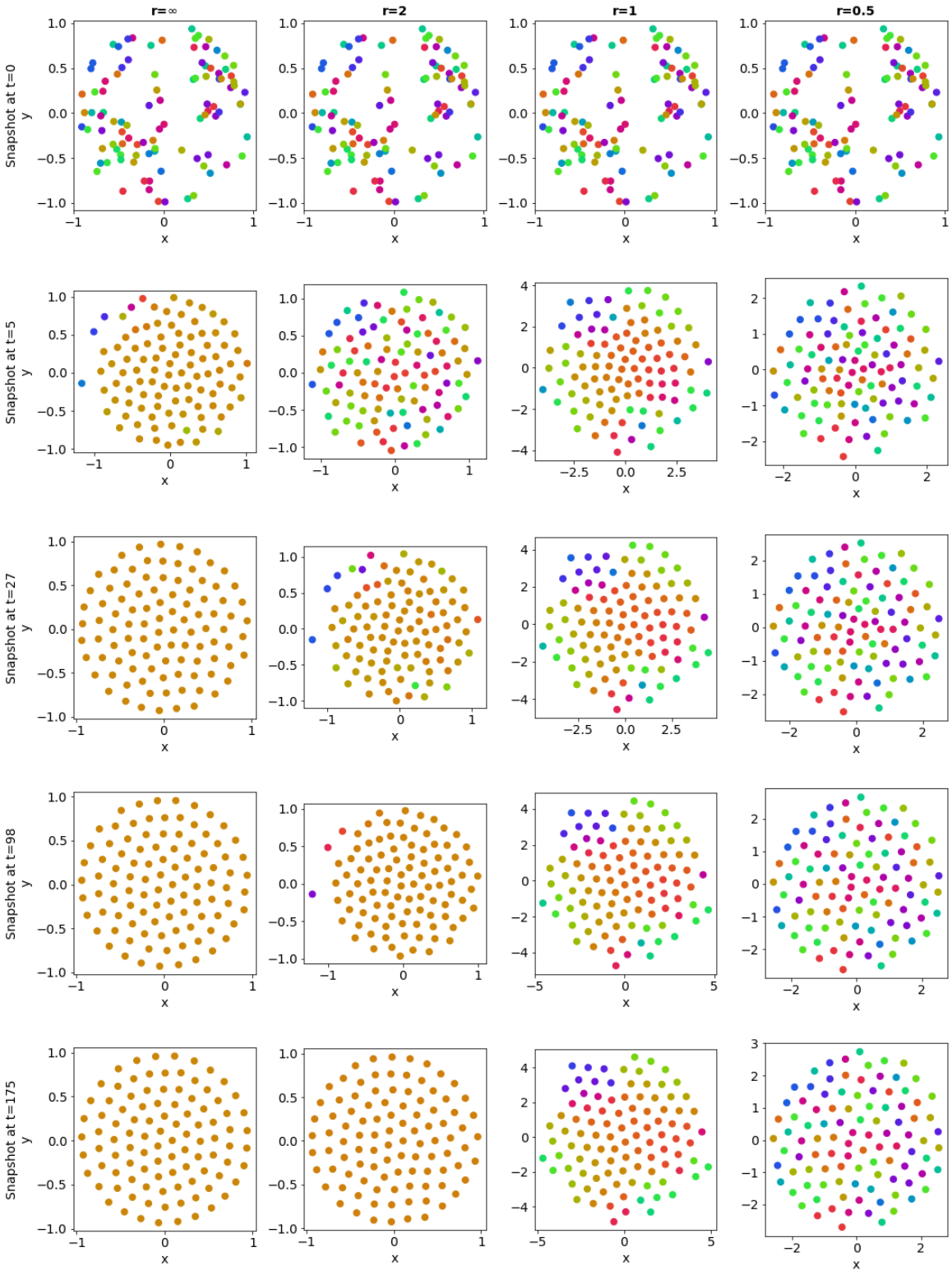

Figure 6: 2D results for finite cutoff with parameters $J = 0.1$, $K = 1$, $N = 100$. We show four cases, (1) infinite interaction distance, (2) $r = 2$, (3) $r = 1$, and (4) $r = 0.5$. While the case of $r = 2$ converges to the infinite distance result, albeit slower, we do not see uniform convergence for smaller r. In the case of $r = 1$, it seems to self organize into an alternative and pointy shape, though this could be due to the small $N = 100$.

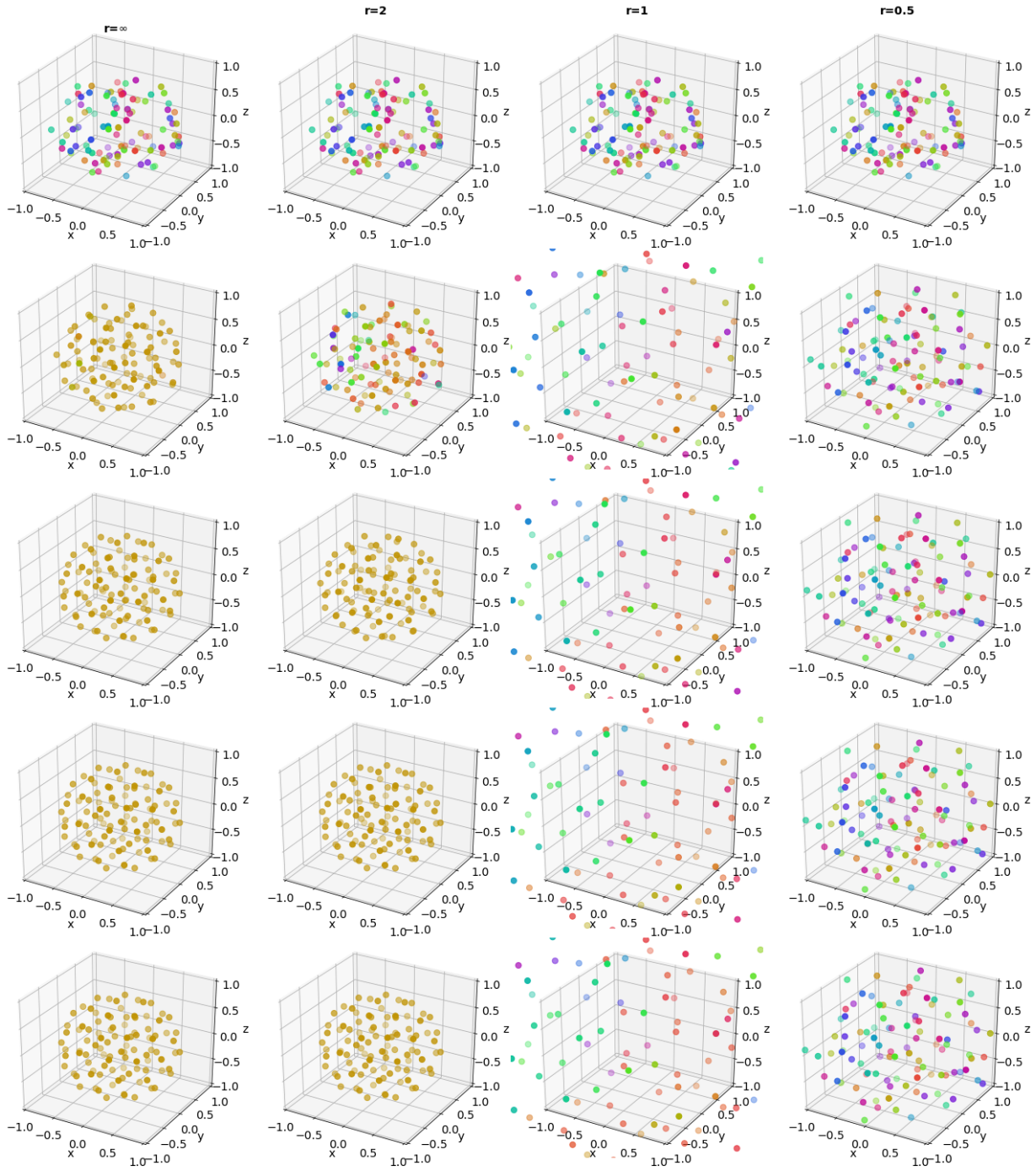

Figure 7: 3D results for finite cutoff with parameters $J = 0.1$, $K = 1$, $N = 100$. We show four cases, (1) infinite interaction distance, (2) $r = 2$, (3) $r = 1$, and (4) $r = 0.5$. While the case of $r = 2$ converges to the infinite distance result, albeit slower, we do not see uniform convergence for smaller r. In the case of $r = 1$, the swarmalators disassociate in a wide pattern and for $r = 0.5$ they disassociate in a smaller pattern. This could be due to the attractive forces not being strong enough to retain a spherical shape for convergence.

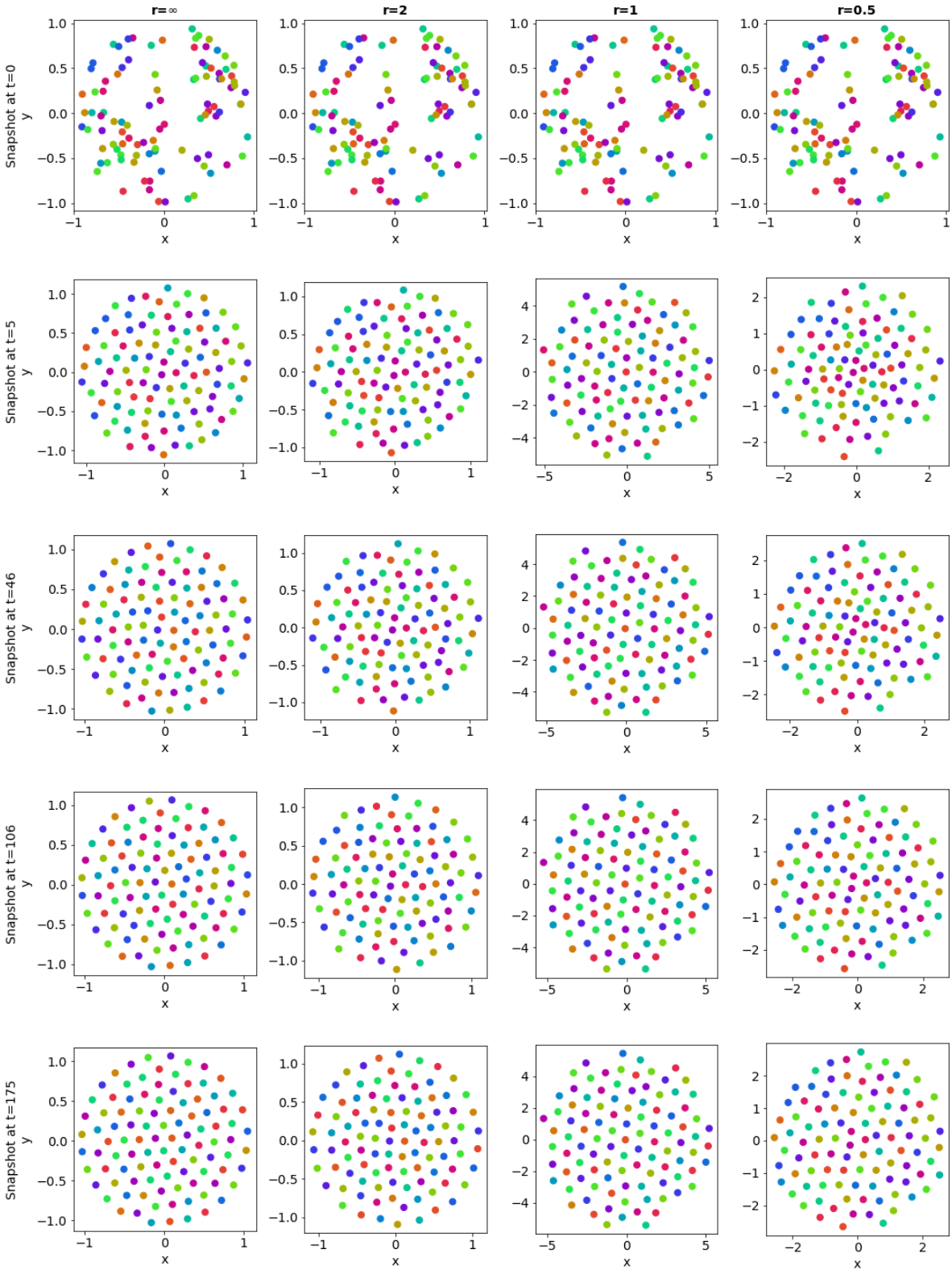

Figure 8: 2D results for finite cutoff with parameters $J = 0.1$, $K = -1$, $N = 100$. We show four cases, (1) infinite interaction distance, (2) $r = 2$, (3) $r = 1$, and (4) $r = 0.5$. While the case of $r = 2$ converges to the infinite distance result, albeit slower, we do not see full convergence for smaller r. In the case of $r < 1$, the swarmalators expand and don't take on a perfect circular shape.

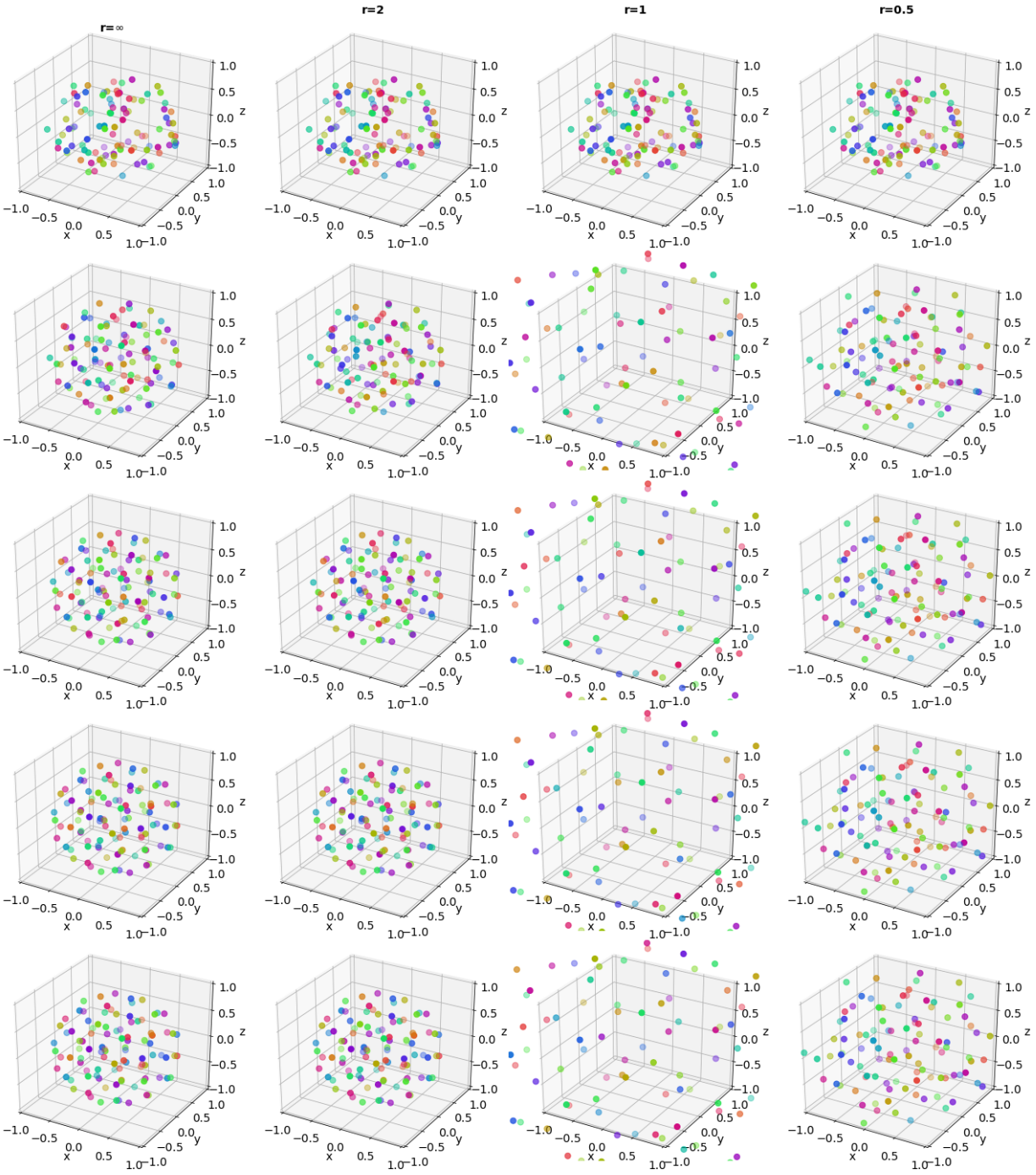

Figure 9: 3D results for finite cutoff with parameters $J = 0.1$, $K = -1$, $N = 100$. We show four cases, (1) infinite interaction distance, (2) $r = 2$, (3) $r = 1$, and (4) $r = 0.5$. While the case of $r = 2$ converges to the infinite distance result, albeit slower, we do not see full convergence for smaller r. In the case of $r < 1$, the swarmalators expand and don't take on a perfect spherical shape, and the radius of the outermost particles to the center is wider.

papers did) to allow for more qualitative analysis with each time step.

Acknowledgements

Thank you to the AM225 teaching team! First, thanks to Nick for giving us the ideas and feedback to form this final project. Second, we appreciate Eder's office hours and mostly his educational and humorous Piazza posts. Finally, a huge shout out to Professor Rycroft! Thank you for investing so much time and effort into helping us better understand these complex topics. We admire your stamina and patience during office hour sessions lasting upwards of two hours at 10 pm on a Tuesday – all while drawing educational works of art with only your mouse. Thank you for making us better mathematicians and programmers. We all look forward to seeing you in person next year!

References

[1] Ernst Hairer, Syvert P. Nørsett, and Gerhard Wanner. Solving Ordinary Differential Equations I: Nonstiff Problems. Springer Series in Computational Mathematics, Springer Ser.Comp.Mathem. Hairer,E.:Solving Ordinary Diff. Springer-Verlag, Berlin Heidelberg, 2 edition, 1993.

[2] Hyun Keun Lee, Kangmo Yeo, and Hyunsuk Hong. Collective steady-state patterns of swarmalators with finite-cutoff interaction distance. Chaos: An Interdisciplinary Journal of Nonlinear Science, 31(3):033134, 2021. _eprint: https://doi.org/10.1063/5.0038591.

[3] Joao U. F. Lizarraga and Marcus A. M. de Aguiar. Synchronization and spatial patterns in forced swarmalators. Chaos: An Interdisciplinary Journal of Nonlinear Science, 30(5):053112, 2020. _eprint: https://doi.org/10.1063/1.5141343.

[4] Kevin P. O'Keeffe, Hyunsuk Hong, and Steven H. Strogatz. Oscillators that sync and swarm. Nature Communications, 8(1):1504, November 2017.

Appendix

A Code

A link to our GitHub repository is below:

`https://github.com/jscuds/am225_swarmalators`

Our ODE is defined in two files, `swarm_ode.cc` and its associated header file. The `swarm_ode` class has two `ff()` functions, one (`ff()`) which does external forcing and grid-binning, but is restricted to 3 dimensions, and one (`ff2()`) which lacks those features, but works in arbitrary dimension n. Both can implement finite interaction distance.

As was done in example code for AM 225, we build our ODE solver as a subclass from the ODE itself and our solver class, `fsal_rk4`. Each ODE solver sets a dimension n, since this determines the number of degrees of freedom to use. Therefore, we build `swarm_sol` as the 2D solver and `swarm_sol_3d` as the 3D solver.

Additional details are in the README file in our git repo, but broadly, there are three files, `run_basic_swarm.cc`, `run_finite_swarm.cc`, and `run_forced_swarm.cc` that are used to generate our results. These are the functions that should be built into executables and run to obtain the data used for our figures.

The figures themselves are generated in a Jupyter notebook, `Figures.ipynb`.

B Runge-Kutta Adaptive Integration and Dense Output

Here, we discuss the solver we used and how we output at specific times for our solutions. Although a detailed explanation of Runge-Kutta methods is beyond the scope of this paper, we want to give a short summary about how our code solves the various swarmalator equations. Runge-Kutta (RK) methods are a body of iterative methods used to solve differential equations. They encompass varying orders of accuracy, including Euler's method (first order accurate) and DOP853 (eighth order) among many others [1]. For this project, we used a First Same as Last (FSAL) adaptive integration scheme which took a five-step RK method and implemented guidelines described in Hairer et al.'s textbook [1] and AM225 Unit 1 slides.[9] The goal behind adaptive integration methods are to take efficient, flexible step sizes while integrating instead of a fixed step, allowing us to obtain the same order of accuracy as RK4 with less function calls. Since our ODE calculates pairwise interactions on hundreds of agents, minimizing function calls is an invaluable asset.

Since by definition, the adaptive integration technique used doesn't have a fixed step size, if we want the integration results at a particular snapshot in time, we need to modify the code. Dense output is a method that assists in solving this problem. Again, the whole description is beyond the scope of this paper, but Hairer et al. dedicate all of their chapter II.6 to the idea [1], and the AM225 notes[9] have concise descriptions and examples. It is sufficient to say that dense output creates a

[9] `https://courses.seas.harvard.edu/courses/am225/slides/am225_unit1.pdf`

polynomial interpolation between two computed integration steps, and can therefore return the value of a function at a particular snapshot in time, or a series of specific snapshots in time.

While we are obviously unable to compare runtimes, previous work [4, 3] used built-in ODE solvers in the Python library Scipy or MATLAB. By using C++ directly, we suspect that our implementation runs faster, allowing us to more easily use larger N without the simulation time becoming unreasonable.