UNIVERSITY OF LINCOLN

SCHOOL OF MATHEMATICS AND PHYSICS

Modelling the Invincible: Computationally Modelling Self-Propelling Particles

Callum Robinson

ROB15589615

Supervised by Dr Andrei Zvelindovsky

10 May 2019

Mathematics Masters Project

MTH9004M

Abstract

Computational models were created for systems of self-propelling particles using Vicsek's model. This is motivated by a science fiction book called *The Invincible* which contained a system of self-propelling micro-machines. However, many similar systems appear in nature. The observed behaviour of these systems is a sudden shift from random disorder motion to ordered motion. This reason for this behaviour is not fully understood.

This report introduces self-propelling particles and Vicsek's model. The model is then created using C++ to create a variety of models with different parameters. The results show that the particles in these systems do collect together into ordered motion. This process is quickened when the particle density increases. Additionally, when the particle neighbourhoods are restricted, the particles form more compact groups. When the noise amplitude is high these restricted systems exhibit similar behaviour to those observed by flocks of birds.

Table of Contents

Introduction ... 4

Motivations ... 4

Objectives ... 4

Resources and References Needed ... 5

What are Self-Propelling Particles? ... 5

Example Systems ... 6

Method ... 7

Vicsek Model ... 7

Noise ... 8

Code Breakdown ... 8

Headers ... 9

Creating and Initialising Variables ... 11

Plotting the Initial Positions ... 13

Beginning the Time Loop ... 14

Closing the Time Loop and Capturing the Final Positions ... 19

Results ... 22

Format of Results ... 22

10 Particles ... 23

20 Particles ... 25

50 Particles ... 27

200 Particles ... 31

500 Particles ... 33

Limited Neighbourhood ... 35

Multiple Particle types ... 38

Comparisons with Nature/Technology ... 41

Conclusion ... 41

Acknowledgements ... 42

Appendices ... 44

Appendix 1: Program with Standard Interaction Rules ... 44

Appendix 2: Limited Neighbourhood Program ... 49

Appendix 3: Program with Multiple Particles ... 54

Appendix 4: External GnuPlot Library (From Reference [17]) ... 59

Appendix 5: Research Plan ... 60

Appendices 6-52 Links to Video Models 62
Appendix 54: Link to Flock of Birds Video Example 67
Bibliography 68
Links used to help construct the code 68

Introduction

Motivations

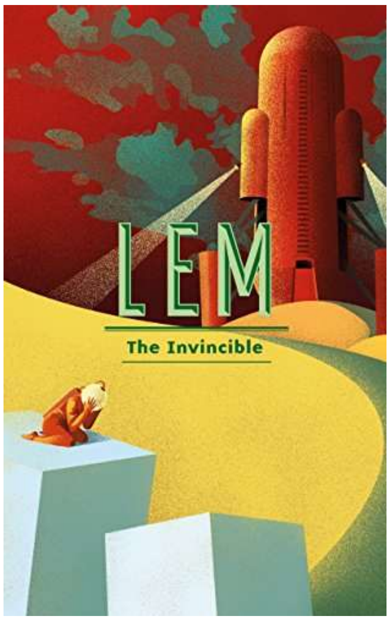

Fig 1.1 Science fiction novel *The Invincible* by Stanisław Lem (ref [5])

The Invincible is a Polish science fiction novel written by Stanisław Lem (fig 1.1). This book contains the story of a powerful spaceship called *The Invincible* which lands on an obscure planet to investigate and determine the fate of another ship. The crew of this ship encounter swarms of miniscule, autonomous, micro machines. These swarms display complex self-organized behaviour (ref plot summary [7]).

The concept of these powerful or even unstoppable self-organised robotic (or biological) swarms have become a common trope in science fiction. However, they closely compare with real-world examples of complex organised systems of self-propelling particles. These systems of self-propelling particles are of great interest and exhibit complex and fascinating behaviours.

Objectives

In this project we will consider what are self-propelling particles and examples of systems of these particles. Vicsek's model will be used to model systems of self-propelling particles where the direction of the velocity for each particle will be calculated for the average direction of the velocity of that particle's neighbourhood. Different models will be created to investigate models with different parameters and with different interaction rules. The code for these programs will be broken down and explained with results that will compare the different models with each other and real-world examples.

Resources and References Needed

Due to the practical nature of this project and the fact that self-propelling particles are not fully understood there are very few physical references required in this project. The book *Collective Behaviour of Self-Propelling Particles with Conservative Kinematic Constraints* by Valeriya I. Ratushna helped to provide the background of self-propelling particles and the equations of the Vicsek model. As mentioned in the motivations the book *The Invincible* by Stanisław Lem motivated this project and provided an artificial (fictional) system of self-propelling particles. Additionally, a few online resources provided example systems of self-propelling particles and basic explanations of self-propelling particles.

To construct the Vicsek models a programming language and compiler needed to be chosen. C++ was chosen as the language and was compiled using Microsoft Visual Studio. This choice of language and compiler was due to their use throughout the Mathematics degree and therefore there was a complete understanding of both. For visual representation of these models, gnuplot was used to show the movement of particles in each modelled system. Multiple online resources assisted in the optimised use of gnuplot in addition to assisting certain calculations. These resources have been separated from the main reference list.

What are Self-Propelling Particles?

Self-propelling particles describe a broad range of systems of autonomous agents (ref [3]). The dynamics of these systems are of great interest for physicists and biologists especially the fascinating phenomenon of the emergence of ordered collective motion (ref [8]). This observed transition from random disordered motion to this ordered motion is not yet fully understood which is why these systems of self-propelling particles are of such interest.

Currently we can distinguish two types of systems of self-propelling particles. The first type of system consists of particles which interact via their background (ref [8]). The driving forces for the systems are due to the gradients of physical or chemical factors. These factors can include temperature, light, chemical concentrations, electric and magnetic fields etc. A direct consequence of these external factors is the absence of conservation of momentum in these systems (ref [8]).

The second type of system consists of particles which interact via kinematic constraints imposed on the velocities of the particles (ref [8]). The models in this project will contain these systems where the particles adjust the direction of their velocities to the directions of the average velocity of that particle's neighbourhood.

Example Systems

Examples of systems of self-propelling particles can be seen throughout nature. The most prominent examples are bird flocks, fish schools, sheep herds and human crowds. If we consider a flock of birds as an example system of self-propelling particles. These flocks are ordered with each bird able to abruptly change direction in unison and the flock may suddenly land with a unanimous group decision. At a smaller scale in nature we can treat cells and bacteria as self-propelling particles. These systems can propel themselves with the presence of chemoattractants (ref [3]).

Janus particles are an artificial example of self-propelling particles. These are special types of nanoparticles whose surfaces have two or more distinct physical properties (ref [1]). These particles have been observed to self-assemble due to the special properties of these particles. This process of self-assembly from a disordered system to an organised structure is an example of ordered motion of self-propelling particles.

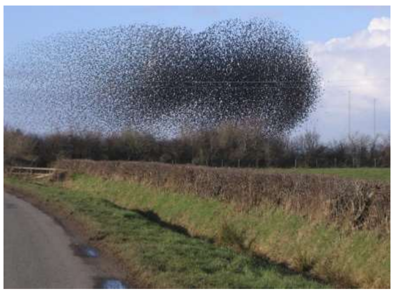
Fig 1.2 Flock of birds (ref [3])

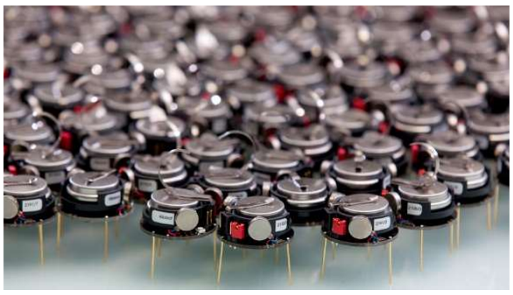
Fig 1.3 Swarm Robots (ref [4])

A final example of systems of self-propelling particles is the concept of swarm robotics. These systems consist of a large number of individually simple physical robots. The science fiction novel *The Invincible* consists of such a system and there exists real-world examples of these swarms of simple physical robots or drones.

Method

Vicsek Model

In the objectives it was mentioned that the Vicsek model will be used to model our systems of self-propelling particles. This aforementioned model is a numerical algorithm introduced by Tamás Vicsek et al. For each time step, this model sets the direction of each particle's velocity to the average direction of velocity of that particle's neighbourhood, plus some noise.

For the models in this project we will be considering the original 2-dimensional model consisting of moving particles in a linear box of size L with periodic boundary conditions (ref [8]). The Main equation for this model is as follows:

$$\theta_i(t + \Delta t) = < \theta(t) >_{S(i)} + \ \xi \tag{2.1}$$

Where

$$< \theta >_{S(i)} = Arg\left(\sum_{\substack{j \\ r_j \in S(i)}} v_j \right) \tag{2.2}$$

Breaking this down our direction of velocity of each particle i at time $t + \Delta t$ is the $\theta_i(t + \Delta t)$ term. On the right side of the equation we have the average direction of velocity of the neighbourhood $S(i)$ (for each particle i), plus the noise term ξ.

The neighbourhood is calculated for each particle by calculating the distance between each particle and checking if the distance is within the set interaction radius. The term $< \theta >_{S(i)}$ for the average direction of velocity of the neighbourhood is calculated by taking the argument of the sum of velocities of the neighbourhood. Note that the absolute velocity $|v|$ is constant.

The position of each particle is the updated as follows:

$$x_i(t + \Delta t) = x_i(t) + v_{x_i}(t)\Delta t \tag{2.3}$$

$$y_i(t + \Delta t) = y_i(t) + v_{y_i}(t)\Delta t \tag{2.4}$$

These equations simply calculate the distance travelled across the time step Δt by multiplying the velocity by the time step and adding it to the current position of the particle.

Noise

In these calculations we require a noise term ξ. In real-world systems of self-propelling particles this noise can have several different sources but can be divided into two main groups (ref [2]).

The first type is due to uncertainty in the "communication" between particles. An example of this is the environment that the particles are in. In certain environments the particle may have trouble seeing the direction in which their neighbours are moving. The second is the concept of free-will for the particles. Even though the particle can see the direction in which their neighbours are moving it may "decide" to move a different direction.

For these models the noise ξ in each calculation it is used will be uniformly distributed in the interval $\left[-\frac{\eta}{2}, \frac{\eta}{2}\right]$ where η is the noise amplitude. This noise amplitude is one of our independent variables we will comparing.

Code Breakdown

The computer models of systems of self-propelling particles will be written in C++ using Microsoft Visual Studio. As previously mentioned, this language was chosen due to strong understanding of the language through previous modules in the mathematics degree.

The following pages will contain a breakdown of each of the programs to explain the function on each section of code.

Headers

To start the programs, we need multiple headers:

```
#include "stdafx.h"
//#include <iostream>
#include <fstream>
#include "gnuplot.h"
#include <random>
#include <windows.h>
#include <math.h>
#define N 50 //set the number of particles

//using namespace std;
```

“stdafx.h” is what is known as a precompiled header. This is a standard header for programs in Microsoft Visual Studio and is used to let the compiler know that some of the larger header files have already been compiled and therefore do not be compiled from scratch every time the program is run saving compile time (ref [14]).

<iostream> defines standard input/output objects mainly cout in these programs which is very useful for test outputs when checking certain sections of code while constructing the program. This header is currently commented out as it is not necessary for the final programs.

<fstream> allows us to read and write to external files. This is required in the programs to open a data file and write the x and y coordinates of each particle. This data file can then be used by gnuplot to plot the positions of each particle at each time step.

“gnuplot.h” is custom header which is included as appendix 4. One of the disadvantages of C++ is a lack of visual output which causes a problem to get the outputs needed by these models. To solve this, gnuplot can be used to plot each particle and a custom header is needed to efficiently pass data from Microsoft Visual Studio to gnuplot. This extremely helpful header was posted on reference [17] by Xingguang Liu.

When generating the uniformly distributed random particle positions a random number generator is needed. <random> allows the use of such random number generators.

When communicating between Microsoft Visual Studio and gnuplot there was an issue that arose where gunplot could not keep up with the program when plotting each of the particles. To combat this a delay was added to the main for loop using a sleep function. <windows.h> allows the use of the sleep function.

To use the sine, cosine and arc tan function in C++ the header <math.h> (or <cmath>) is needed. However note that some headers in Microsoft Visual Studio's version of <iostream> indirectly runs a version of <cmath> and therefore the <math.h> is not required but it is better to include especially if there is a possibility of using other compilers which will require the <math.h> header.

When running the programs with different a number of particles (N) there are a few arrays and for loops that need to be updated to be the correct size. This can be done by changing the N values manually each time which can be time consuming or the N value can be defined at the start of the code. Here a #define statement is used for this purpose and is currently set to 50 particles.

The final line [using namespace std] allows the use of cout outputs without the need for [std::] addition. As previously mentioned, this was used for test messages when constructing the codes. This additional line makes these test messages lines easier and "neater". This line is commented out similar to the <iostream> header as it is not required for the final running of the programs.

Beginning Main()

We now move onto the main() code. To start the program needs to be able to open and write to the data file and gnuplot needs to be able to be opened and plot the positions from the data file in the later sections of code.

```
int main()
{
	Gnuplot plot; //Use of external Gnuplot library
	ofstream myfile;
	myfile.open("gnutest.data"); //open data file to export to
```

The line [Gnuplot plot] starts gnuplot and the "gnuplot.h" header and allows commands to be sent to gnuplot using [plot ("insert command")] where the text in quotation marks is sent to gnuplot. Similarly, the next line allows a file to be opened and written to using myfile. This can be seen in the last line which uses this to open a data file named "gnutest.data" to write the x and y positions of each particle.

Creating and Initialising Variables

Any variables or arrays needed throughout the program need to be created and initialised. Each of the following variables and arrays will be commented with their purpose for easier understanding:

```
const long double PI = 3.141592653589793238; //define constant PI
       const double L = 10; //boundary size
       double x[N], y[N], theta[N], xvel[N], yvel[N]; //arrays for each x and
y coordinates, velocities and direction
       double vel = 0.05; //absolute velocity of each particle (constant)
       double r = 1; //interaction radius
       double deltat = 1; //time interval
       double maxt = 1000; //max time period
       double noise = 2; // noise amplitude
       double xvelsum = 0; //sum of velocity x component
       double yvelsum = 0; //sum of velocity y component
       double xvelavg = 0; //average x velocity
       double yvelavg = 0; //average y velocity
       double non = 0; //number of neighbours inside the interaction radius
       double avgtheta = 0; //average theta of neighbourhood
       double difference = 0; //variable for distance between particles
```

The variables above with specific variables are the independent and control variables. In the results the boundary size (L), interaction radius (r) and noise amplitude (noise) will be changed and compared. Additionally, a further program will use two types of particles with different absolute velocities.

Initialising the Particles

When initialising the x and y position and velocity directions a method is needed to generate uniformly random distributed numbers. This is also needed when calculating the noise contribution to the movement of each particle at each time step. This can be achieved with a random number generator (ref [12]):

```
// random number distribution
	random_device rd;
	mt19937 e2(rd()); //Mersenne Twister 19937 generator
	uniform_real_distribution<> dist(0, 1);
```

This generates random numbers uniformly between 0 and 1 using a Mersenne Twister 19937 generator.

Each particle needs to be initialised with an x position, y position, direction of velocity, x velocity and velocity. These are assigned inside a for loop for each particle respectively:

```
for (int i = 0; i < N; i++) //intialise each particle
	{
		x[i] = L*dist(e2); //intial x = L * (random variable between 0 
and 1)
		y[i] = L*dist(e2); //intial y = L * (random variable between 0 
and 1)
		theta[i] = 2 * PI*(dist(e2) - 0.5); //intial theta = 2 * PI * 
(random variable between 0 and 1)
		xvel[i] = 0; //initialise x velocity
		yvel[i] = 0; //initialise y velocity
		myfile << x[i] << "\t" << y[i] << endl;
	}
```

Logically each particle is initially stationary with a randomly generated position and a random direction in which the particle wishes to travel. These random positions and directions use the random number generator at the top of this page. The x and y positions for each particle are then output to the data file that was opened earlier. Note that the \t is an escape sequence for a horizontal tab that separates the two values for each particle in the data file.

Now that the positions of each particle have been written to the data file it now needs to be closed. This is done simply with the following line:

```
myfile.close();
```

Plotting the Initial Positions

These initial positions now need to be plotted in gnuplot. Before this the range of the gnuplot output needs so that it is the size of the boundary L. In this example the boundary size is 10 and therefore the range is set from 0 to 10:

```
plot("set xrange [0:10]"); //sets x range in Gnuplot
plot("set yrange [0:10]"); //sets y range in Gnuplot
```

Note that the text in quotations inside plot() are gnuplot commands and are sent directly to gnuplot.

Now that the range is set the positions need to be plotted form the data file. This simply done by telling gnuplot the data in the "gnutest.data" in the first and second column. This following command is as flows:

```
plot("plot 'gnutest.data' using 1:2\n"); //plot the data file
```

Now that these positions are plotted a question can be raised of how the results of these models can be shown in this report. A video of each model running will be uploaded and linked in the appendices of this report. However, a visual method is needed to directly compare models inside this report in addition to comparing the initial random particle positions with the final positions.

This can be achieved with gnuplot's ability to output the positions to a PNG file. This can therefore be used now to output the initial random particle positions to a PNG file. To output this image gnuplot needs to be set to PNG format and the output the image as follows:

```
plot("set term png");
plot("set output 'outputstart.png' "); //outputs a PNG file with the starting
random distribution
plot("replot");
```

Now that this image is created and saved gnuplot needs to be set back to the initial windows format for further plotting. This is done with a simple command:

```
plot("set term win"); //sets Gnuplot back to the window format
```

Beginning the Time Loop

With the initial particle positions calculated and plotted the main time loop need to be created. This main for loop will contain all of the calculations from this point onwards and represents each time step repeating every time the time increases. This is for loop is as follows:

```
for (double t = 0; t < maxt;) //loop for each timestep up to max time
       {
```

Logically this loop will continue until the time reaches the maximum time limit.

At the beginning of this loop, the data file will need to be opened so that the positions can be written to the file. This is done similarly to when the initial positions were written to the file using the following line:

```
myfile.open("gnutest.data");
```

Each particle needs an updated velocity (x and y components). These updated velocities are calculated using the current direction θ_i with the following equations:

$$vel_{x_i} = Absolute\ Velocity \times \cos(\theta_i) \tag{3.1}$$

$$vel_{y_i} = Absolute\ Velocity\ \times \sin(\theta_i) \tag{3.2}$$

This is done in the program using the following lines:

```
xvel[i] = vel*cos(theta[i]); //update velocity x component
yvel[i] = vel*sin(theta[i]); //update velocity y component
```

Note that the absolute velocity is constant for all particles (except in a model with different particles which will be covered later).

From the velocities the updated x and y positions can be calculated. This is simply calculated using the equations mentioned in the method:

$$x_i(t + \Delta t) = x_i(t) + v_{x_i}(t)\Delta t \tag{3.3}$$

$$y_i(t + \Delta t) = y_i(t) + v_{y_i}(t)\Delta t \qquad (3.4)$$

Using the following lines:

```
x[i] += xvel[i] * deltat; //update x coordinate
y[i] += yvel[i] * deltat; //update y coordinate
```

Unfortunately, a problem is caused when this new particle position lies outside of the boundary. This problem can be tackled multiple ways but in these models, it requires an if statement for each of the four boundaries:

```
//Checks if any particle is outside the boundary
if (x[i] < 0)
{
   x[i] = L + x[i];
}
if (x[i] > L)
{
   x[i] = x[i] - L;
}
if (y[i] < 0)
{
   y[i] = L + y[i];
}
if (y[i] > L)
{
   y[i] = y[i] - L;
}
```

The first statement checks if the x position is negative and if the position is negative then the boundary value L is added to the x position. This means that if any particle moves outside of the left x boundary then it is sent across the right x boundary.

Similarly, the second statement checks if the x position is greater than the boundary value L and if greater the x position is subtracted by the boundary value L to get the new x position. Opposite to the first statement if a particle moves across the right x boundary then it is sent across the left x boundary.

For the bottom and top y boundaries the third and fourth statements are used respectively. The third statement repeats the first replacing x with y for the bottom y boundary. Similarly, the second statement is repeated replacing x with y for the top y boundary. Note that a particle may move across both an x and y boundary is a single time step.

Now that the x and y positions have now been updated for the new time step they need to be written to the data file and then plotted in gnuplot. This is done using the same method when plotting the initial positions:

```
//output and plot these positions
for (int i = 0; i < N; i++)
{
	myfile << x[i] << "\t" << y[i] << endl;
}
myfile.close();
plot("plot 'gnutest.data' using 1:2\n");
```

The lines above write the updated particle positions to the file, close it and then plot the file in gnuplot respectively.

Creating the Neighbourhood

To calculate each particle's next movement using Vicsek's model, a neighbourhood needs to be constructed for each particle. This will require multiple calculations that will be covered in a moment. These calculations will require variables which were created at the start of the code but will be reset to 0. This isn't necessarily required but is considered good practice and helps to determine where errors occur if there is a problem. The variables reset are as follows:

```
for (int i = 0; i < N; i++)
{
	//reset all variables used in loop
	xvelsum = 0;
	yvelsum = 0;
```

```
        non = 0;
        xvelavg = 0;
        yvelavg = 0;
        avgtheta = 0;
```

The above variables are for calculating the velocity of the neighbourhood (x and y components), number of particles in the neighbourhood, average velocity of the neighbourhood (x and y components), and average direction θ of the neighbourhood respectively.

To assemble the neighbourhood, we need to know which particles actually interact. Therefore, the program needs to compare each particle with all the others to see if they lie in the surrounding neighbourhood. This can be done by creating a for loop for every other particle and measuring the distance between the two particles using the simple difference equation:

$$difference = \sqrt{\left(x_j - x_i\right)^2 + \left(y_j - y_i\right)^2} \tag{3.5}$$

With the following lines:

```
for (int j = 0; j < N; j++)
{
        difference = 0;
        difference = sqrt(pow((x[j] - x[i]), 2.0) + pow((y[j] - y[i]), 2.0));
//distance between particles
```

Note that the difference variable is reset to zero in the loop similar to previous variables as it is considered good practice.

To check each particle requires an if statement and an else if statement. If a particle i is the same particle as particle j then it is obviously part of the neighbourhood. Therefore, the velocity (x and y component) are added to the velocity sum of the neighbourhood and we get to the first if statement:

```
if (i == j) //if they are the same particle add the velocities to the sums
{
        xvelsum += xvel[i];
        yvelsum += yvel[i];
        non++; //increase size of the neighbourhood
}
```

Otherwise, if the difference is smaller than the interaction radius r then the particles will interact and therefore this particle becomes part of the neighbourhood. The contents of the else if statement is a repeat of the first if statement and is as follows:

```
else if (difference < r) //if the different particles are in the radius add
the velocities to the sums
{
	xvelsum += xvel[j];
	yvelsum += yvel[j];
	non++; //increase size of the neighbourhood
	//cout << "Particles interact" << endl; //test notification if the
particles are close enough together
}
```

Note that anytime a particle is added to the neighbourhood, the “number of neighbours” variable increases by one (also note that in these models a particle is considered a neighbour to itself).

For the else if statement there is a test line that sends a notification to the windows console when the particles interact. This was used when constructing the code and testing the program with a low number of particles. However, this notification is not necessary and would slow down the program when considering many particles and it is therefore commented out of the code.

Once the neighbourhood is constructed for each particle, the average velocity of the neighbourhood (x and y components) can be calculated using the equations:

$$Avg\ Velocity_x = \frac{Velocity\ sum_x}{Size\ of\ the\ neighbourhood} \tag{3.6}$$

$$Avg\ Velocity_y = \frac{Velocity\ sum_y}{Size\ of\ the\ neighbourhood} \tag{3.7}$$

Using two simple lines as follows:

```
xvelavg = xvelsum / non; //calculate average x velocity
yvelavg = yvelsum / non; //calculate average y velocity
```

These averages can then be used to calculate the average direction of velocity of the neighbourhood. This calculation follows the equation:

$$\theta_{avg} = \tan^{-1}\left(\frac{Avg\ Velocity_y}{Avg\ Velocity_x}\right) \tag{3.8}$$

Where θ_{avg} is the average direction of velocity of the neighbourhood. This uses the arc tan function atan2 and is as follows:

```
avgtheta = atan2(yvelavg, xvelavg); //calculate average theta
```

When constructing the code, a test output was added that output this average value to the console to check the calculation worked as intended:

```
//cout << "average theta = " << avgtheta << endl; //Used to test the calculation
when constructing
```

As with the other test lines included within this code this was used with a small number of particles and is no longer necessary and so is commented out of the code.

The θ_i of each particle i can be calculated using the average direction of velocity of the neighbourhood θ_{avg} using the following equation:

$$noise = noise\ amplitude \times random\ value\ in\ range\ \left[-\frac{1}{2},\frac{1}{2}\right] \tag{3.9}$$

Sources of this noise where covered previously in this report and the above calculation will logically require the uniform random number generator. This is seen in the following:

```
theta[i] = avgtheta + noise*(dist(e2) - 0.5); //update theta
```

Closing the Time Loop and Capturing the Final Positions

Now that all calculations in the time loop are completed, the time loop can be closed. This done simply by increasing the current time by the time step amount Δ_t and closing the bracket of the for loop:

```
t += deltat; //increase timestep
```

However, when testing the program, a problem occurred where gnuplot could not keep up when plotting many particles. This can be seen when the output would stop and start very often and so a delay needed to be introduced to the main time loop. There is a sleep function designed to introduce such a delay:

```
Sleep(12); //add 12 millisecond delay to allow gnuplot to keep up
```

This sleep function adds a 12-millisecond delay and allows gnuplot to run smoothly alongside the C++ program.

When the time loop has reached the maximum time limit and has completed the last loop a PNG image needs to be created so we may see the particles' final positions so that we can compare them later. This is done in a similar way to when the initial positions were captured and is as follows:

```
plot("set term png");
plot("set output 'outputend.png' "); //Output the final result as a PNG file
plot("replot");
```

Gnuplot can be set back to the windows format but this is not required as no further plotting is required.

To exit Main() and complete the code the following is needed:

```
return 0;
}
```

This return 0 is an exit code for main() and the widely accepted convention in C++ to let the program know that main() has worked correctly.

Additions for further models

To expand on these models, two separate additions were made to the code. One considered a system where the size of the neighbourhood was limited. If we consider this in the example of flocks of birds this restriction could be due to limited visibility of its neighbours or the bird may only be able to process the velocity of a certain number of neighbours.

To add this restriction to the model an if statement was added to check if the "number of neighbours" variable equalled three. If this if statement was true, then the for loop comparing each particle would break and no more particles can be considered for the neighbourhood. This very simple to add:

```
if (non == 3)
{
	break; //Particles can interact with a limited number of neighbours
}
```

This uses a break statement which stops the for loop from running anymore steps (for the current particle being considered).

For the second expanded model we consider a system consisting of two (or more) types of particles with different absolute velocities. Logically this will require an additional variable for this second absolute velocity:

```
double vel1 = 0.05; //absolute velocity of particle type one (constant)
double vel2 = 0.10; //absolute velocity of particle type two (constant)
```

A method is then required to decide which type each particle belongs to. For this model this is done by assigning odd particles as one type and even particles as the other. If a velocity is even, then it has the absolute velocity 0.1 otherwise it has the absolute velocity 0.05:

```
for (int i = 0; i < N; i++)
{
	if (i % 2 == 0) //odd and even particles have different velocities
	{
		xvel[i] = vel2*cos(theta[i]); //update velocity x component
		yvel[i] = vel2*sin(theta[i]); //update velocity y component
	}
	else
	{
		xvel[i] = vel1*cos(theta[i]); //update velocity x component
		yvel[i] = vel1*sin(theta[i]); //update velocity y component
	}
```

As seen above this simply uses an if statement to check if the particle is even and uses 0.1 as its absolute velocity. Otherwise the else statement uses the absolute velocity 0.05.

Results

Format of Results

With complete computational models an output is needed to compare results. As seen in the code breakdown, gunplot was used to generate animated visual models of the movement of particles in each system. Clearly this presents a problem of how to represent these models in this physical report. A solution to this is capturing an image of the initial and final particle positions of each model. The method of achieving this was included in the code breakdown.

To illustrate how the results will be formatted, consider the following example:

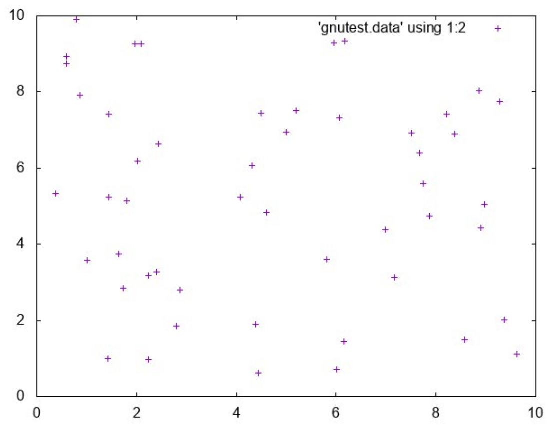

Fig 4.1 Initial Positions of Particles

N = 50

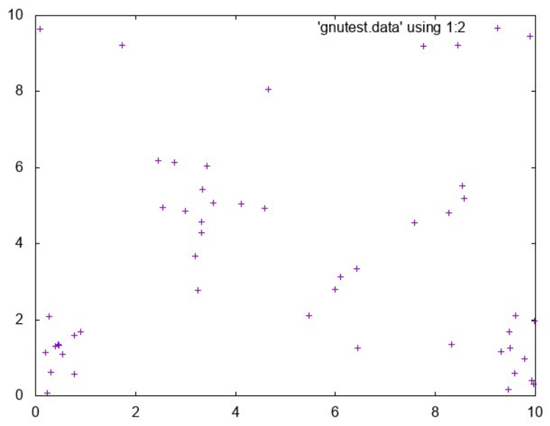

Fig 4.2 Final Positions of Particles

N = 50, L = 10, $\mu = 2$.

Figure 4.1 shows the random initial positions of 50 particles. Next to this, figure 4.2 shows the final positions of a model of 50 particles with a boundary size (L) of 10 and a noise amplitude (μ) of 2. This how each of the models will be represented with the initial particle positions shown and then the finals positions in each model. The results will then be discussed alongside a discussion of the video results that will be linked in the appendices.

For the "main" models two independent variables will be compared. These two variables will be the noise amplitude (μ) which affects the random nature or freewill of each particle and the boundary size (L) which affects the density of particles in the system. Other variables such as time, interaction radius (r) and absolute velocity will be kept constant throughout each model (with exception to further models explained later).

The values of these control variables are as follows:

$$Max\ Time = 1000$$

$$Timestep = 1$$

$$Interaction\ Radius = 1$$

$$Absolute\ Velocity = 0.05$$

10 Particles

For the first set of models a small number of 10 particles was considered. The initial positions are as follows:

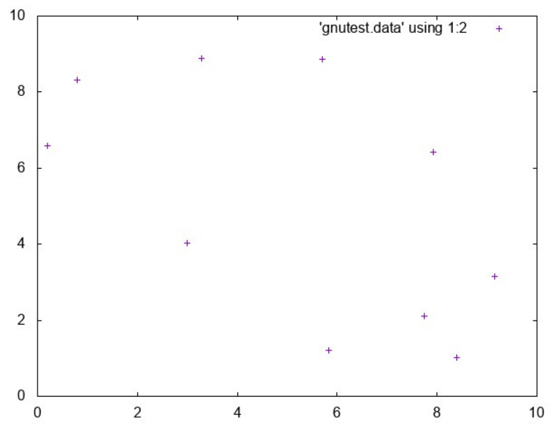

Fig 4.3 Initial Positions of Particles

N = 10

The models compared the effect of high and low noise on systems with different densities of particles.

The final particle positions of each of the models are as follows:

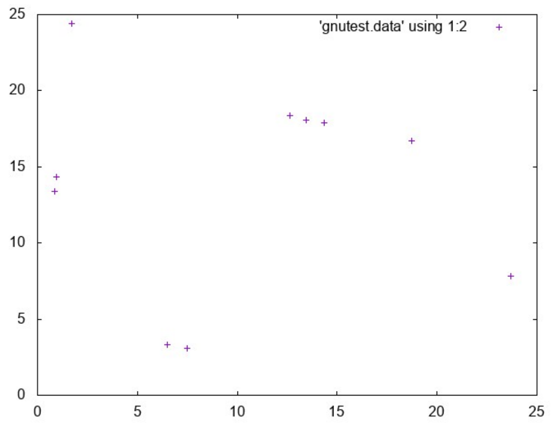

Fig 4.4 Final Positions of Particles

N = 10, L = 25, $\mu = 0.01$.

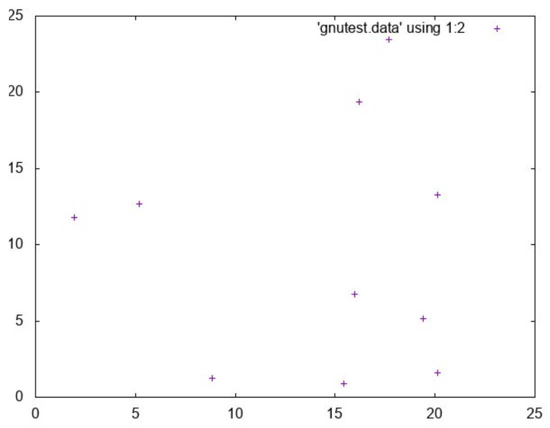

Fig 4.5 Final Positions of Particles

N = 10, L = 25, $\mu = 2$.

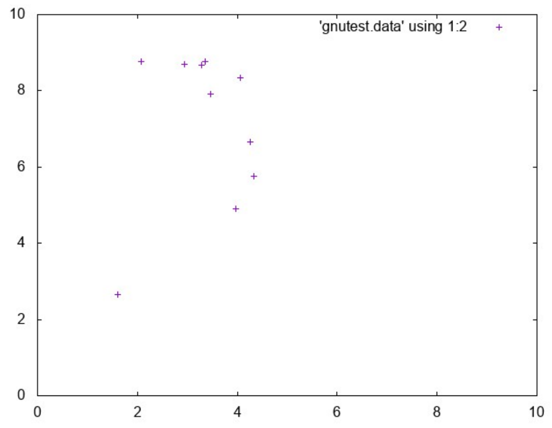

Fig 4.6 Final Positions of Particles

N = 10, L = 10, $\mu = 0.01$.

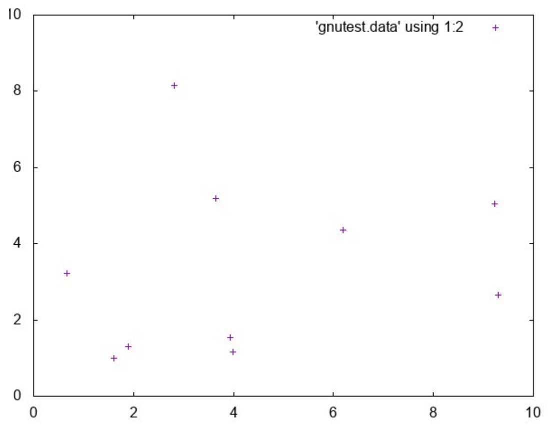

Fig 4.7 Final Positions of Particles

N = 10, L = 10, $\mu = 2$.

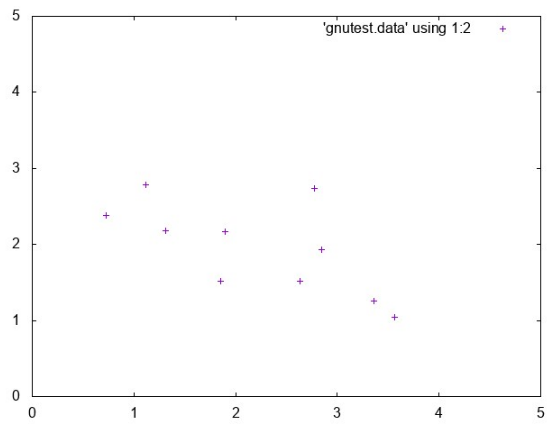

Fig 4.8 Final Positions of Particles

N = 10, L = 5, $\mu = 0.01$.

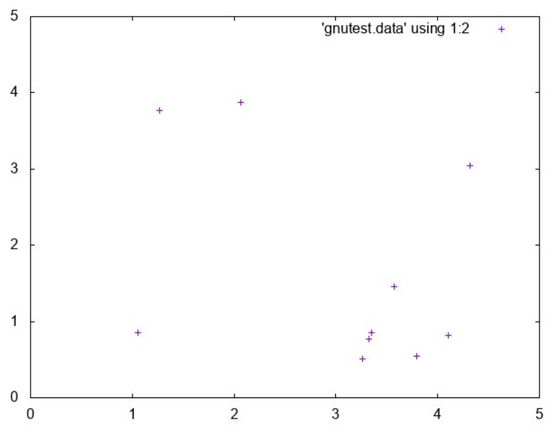

Fig 4.9 Final Positions of Particles

N = 10, L = 5, $\mu = 2$.

The videos of the animated plots will be linked in the appendices as appendix 6, 7, 8, 9, 10 and 11 respectively. With few particles they struggled to interact with each other especially in the systems with higher boundary values. When the noise was minimal, any particle that interacted immediately moved alongside its neighbourhood creating small neighbourhoods moving together which would combine in the systems with smaller boundaries. Eventually these neighbourhoods (or neighbourhood) will move ordered together in the same direction. However, in the systems with high noise these neighbourhoods often separated (or struggled to form at all) causing particles to move around randomly (seen in figures 4.5 and 4.7).

20 Particles

The next set of models increased the number of particles to 20. This is still a seemingly small number in these models but with double the number of particles, interactions should occur more frequently. An example of the starting position of particles is as follows:

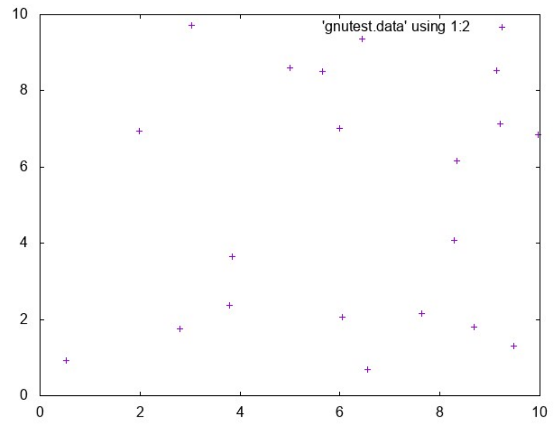

Fig 4.10 Initial Positions of Particles

N = 20

The final positions of the 20 particle models are as follows:

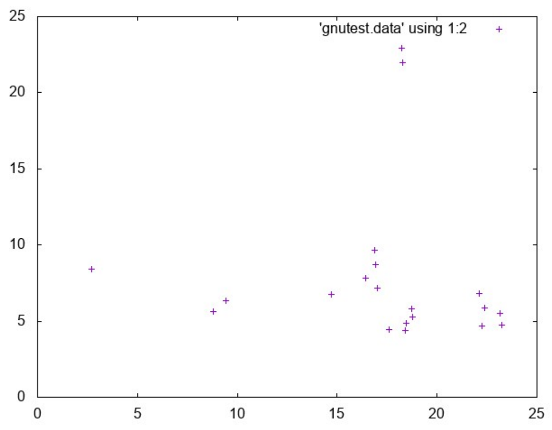

Fig 4.11 Final Positions of Particles

N = 20, L = 25, μ = 0.01.

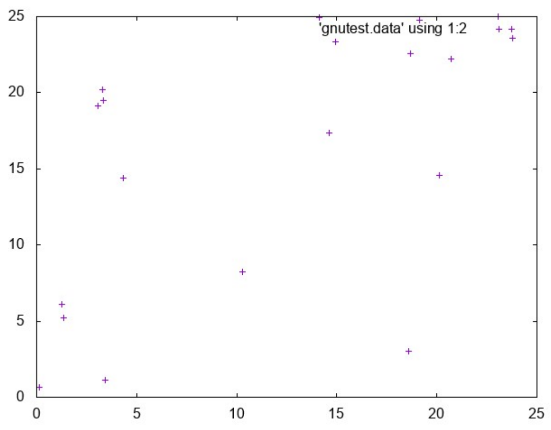

Fig 4.12 Final Positions of Particles

N = 20, L = 25, μ = 2.

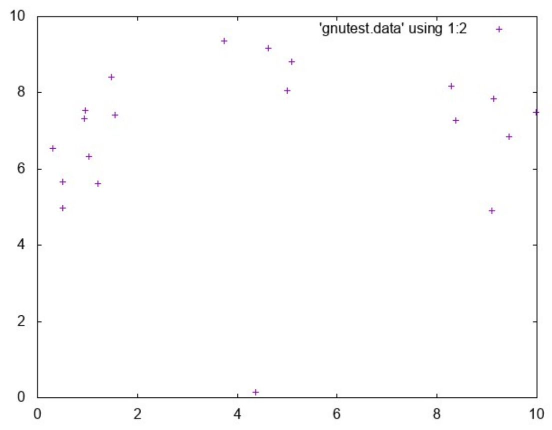

Fig 4.13 Final Positions of Particles

N = 20, L = 10, μ = 0.01.

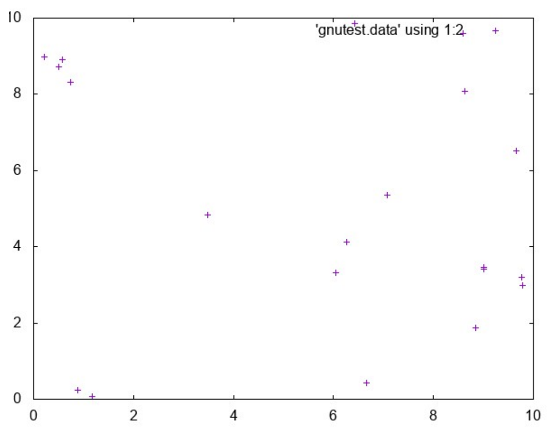

Fig 4.14 Final Positions of Particles

N = 20, L = 10, μ = 2.

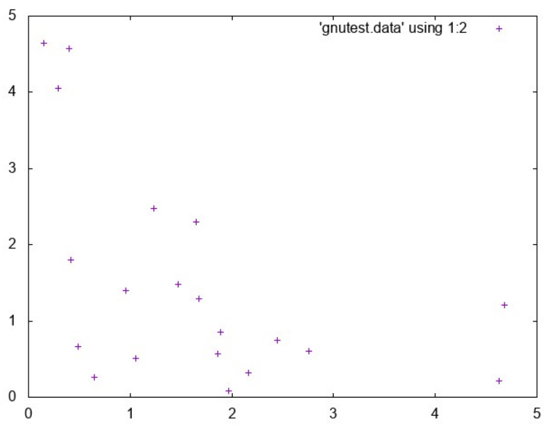

Fig 4.15 Final Positions of Particles

N = 20, L = 5, μ = 0.01.

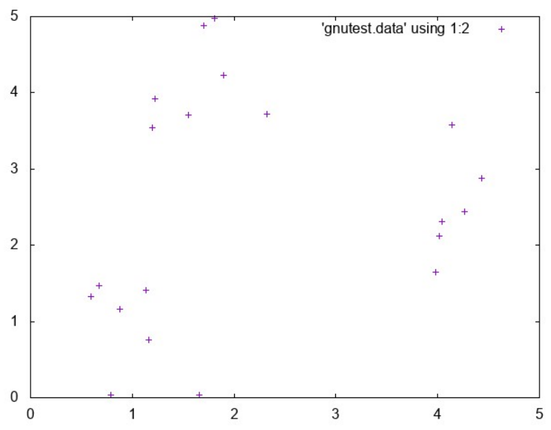

Fig 4.16 Final Positions of Particles

N = 20, L = 5, μ = 2.

The videos of these models have been uploaded and are linked as appendices 12, 13, 14, 15, 16 and 17 respectively. The outputs of these models are very similar to the 10 particle models, but more particles cause neighbourhoods to form quicker and larger. When the boundary was large and the noise was low, the particles formed multiple small neighbourhoods which occasionally combined to form one or multiple neighbourhoods. As the boundary decreased (still with minimal noise) the particles began to move in the same direction in a smaller amount of time. This is easier seen in the video outputs.

With high noise and a large boundary size the particles move seemingly randomly similar to the 10-particle model with these parameters. However small groups occasionally form moving around randomly together before possibly breaking apart. As the boundaries decrease in size, these groups become larger. At the smallest boundary they eventually reach a point where they all move together for a brief moment before separating.

50 Particles

The number of particles is now increased to 50 similar to the example model at the start of the chapter. The starting positions for 50 particles are as follows

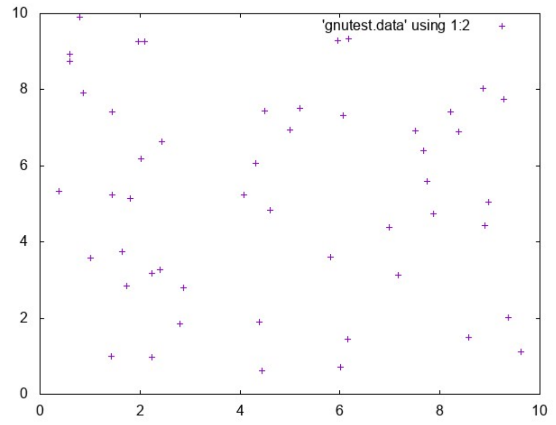

Fig 4.1 from page 21

Initial Positions of Particles

N = 50,

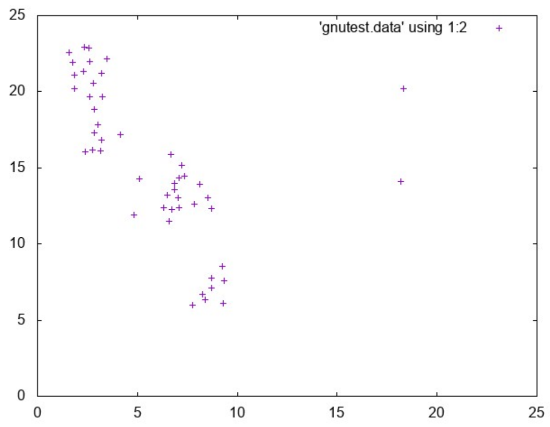

Fig 4.17 Final Positions of Particles

N = 50, L = 25, $\mu = 0.01$.

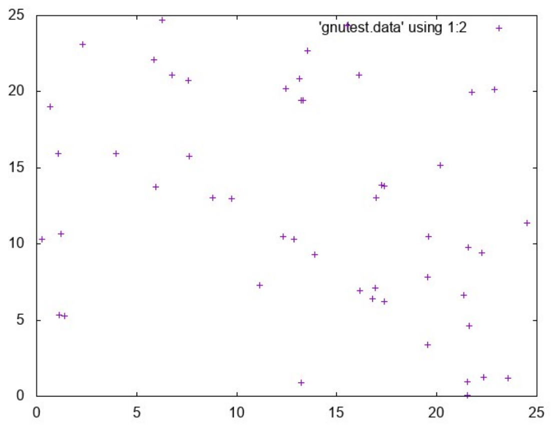

Fig 4.18 Final Positions of Particles

N = 50, L = 25, $\mu = 2$.

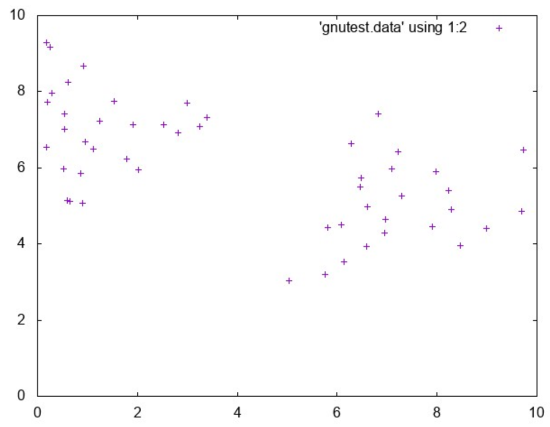

Fig 4.19 Final Positions of Particles

N = 50, L = 10, $\mu = 0.01$.

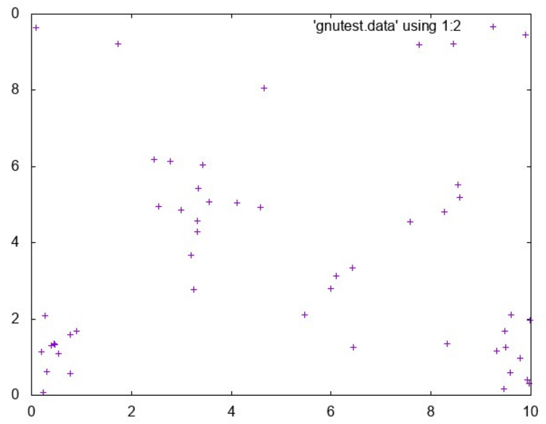

Fig 4.20 Final Positions of Particles

N = 50, L = 10, $\mu = 2$.

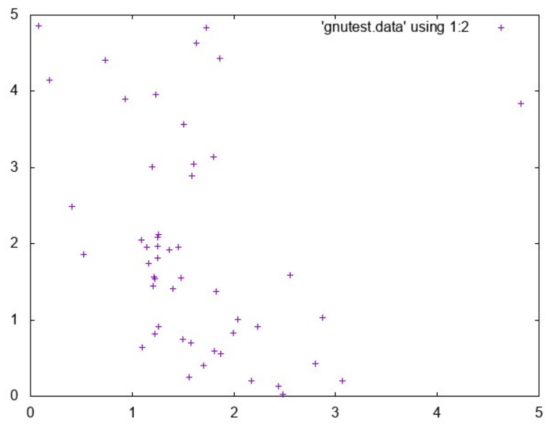

Fig 4.21 Final Positions of Particles

N = 50, L = 5, $\mu = 0.01$.

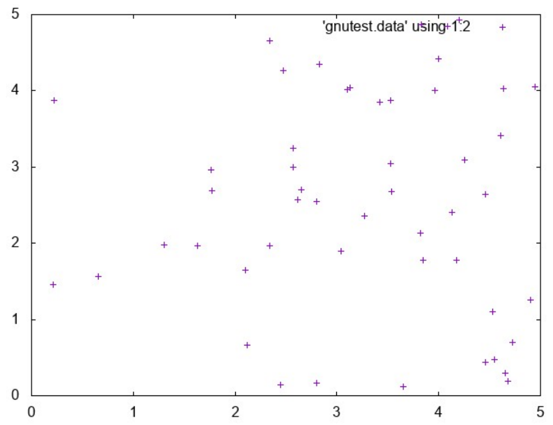

Fig 4.22 Final Positions of Particles

N = 50, L = 5, $\mu = 2$.

The videos for these models will be included as appendices 18, 19, 20, 21, 22 and 23 respectively. When the noise was low and the boundary was large, the particles again formed small "communities". However, this time they formed many more which began to combine and towards the time limit most had become ordered. When the boundary decreased the particles became ordered almost immediately.

With a high noise and a large boundary many small groups formed moving around randomly. As the boundary decreased some of these groups combined to form larger groups moving around randomly. At the smallest boundary most particles eventually moved together in a large group with rapidly changing direction. Occasionally a small group would break off to be added quick back to the main group causing a drastic change in the main group's direction.

100 Particles

Doubling the particles to 100 particles for the next set of models. An example of the starting positions is as follows:

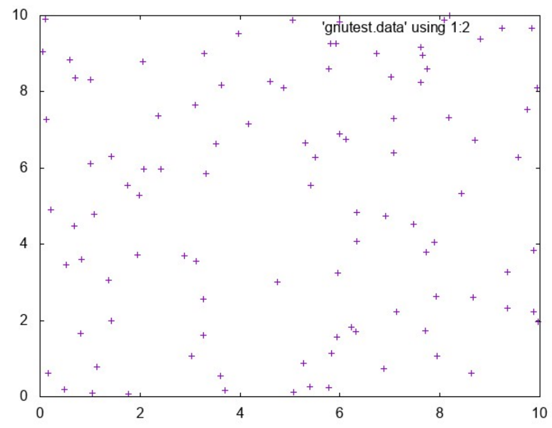

Fig 4.23 Initial Positions of Particles

N = 100

The 100 particle models are shown on the next page:

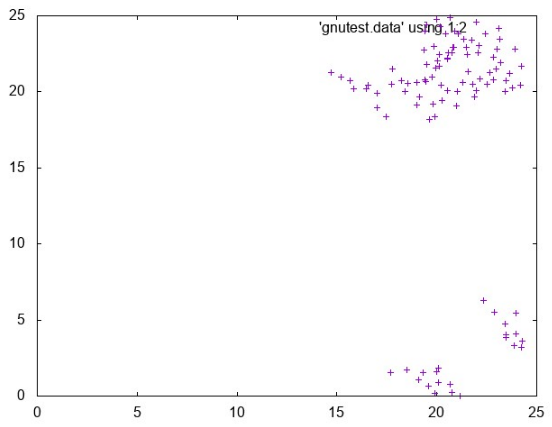

Fig 4.24 Final Positions of Particles

N = 100, L = 25, $\mu = 0.01$.

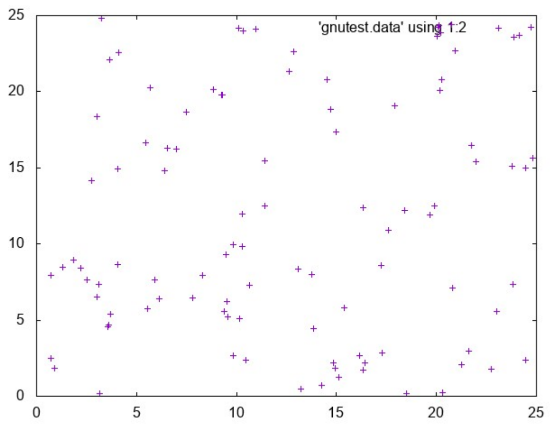

Fig 4.25 Final Positions of Particles

N = 100, L = 25, $\mu = 2$.

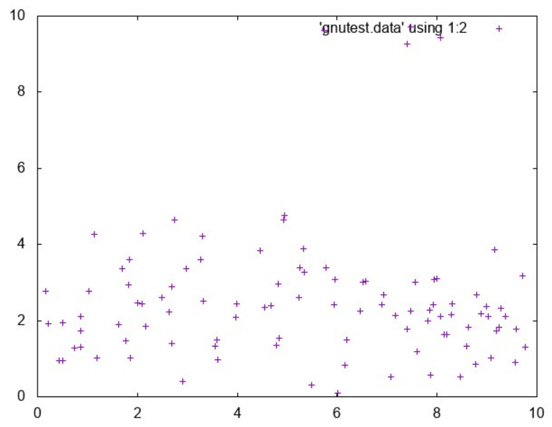

Fig 4.26 Final Positions of Particles

N = 100, L = 10, $\mu = 0.01$.

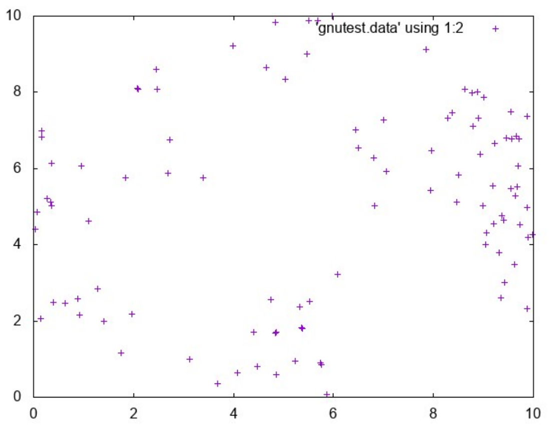

Fig 4.27 Final Positions of Particles

N = 100, L = 10, $\mu = 2$.

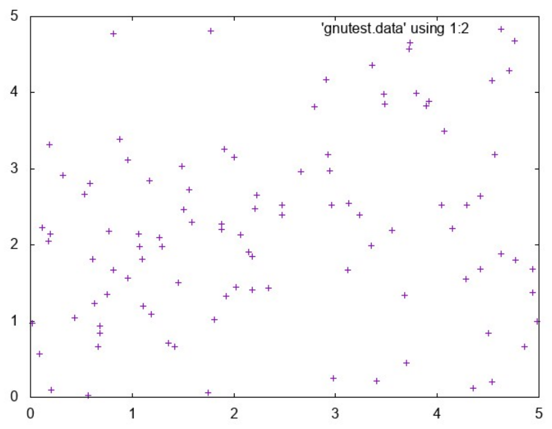

Fig 4.28 Final Positions of Particles

N = 100, L = 5, $\mu = 0.01$.

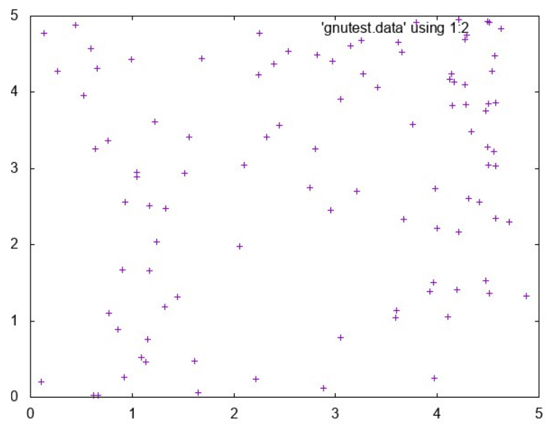

Fig 4.29 Final Positions of Particles

N = 100, L = 5, $\mu = 2$.

The 100 particle models are uploaded and linked in appendices 24, 25, 26, 27, 28 and 29 respectively. In the model with minimal noise and a large boundary, the particle formed the expected larger groups. However due to the larger number of particles in the groups they interacted more and began to "stick" when they came into contact, this can be seen in the large group in figure 4.24. When the boundary size was decreased the particles began to very quickly order themselves into one large stream of particles. This can be seen as a long stream in 4.26.

For a high noise amplitude and a large boundary, the particles still moved around randomly as expected although many small clusters did form temporarily. Similar to the 50-particle model the clusters became larger when the boundary decreased in size. These larger groups attempted to combine a couple of times but were immediately torn apart into multiple clusters. At the minimum boundary size, the particles all moved in the same rapidly changing direction. However, with more particles the direction change was not as severe as in the 50-particle model.

200 Particles

The next set of models will contain systems of 200 particles. An example of the initial random positions of the particles is as follows:

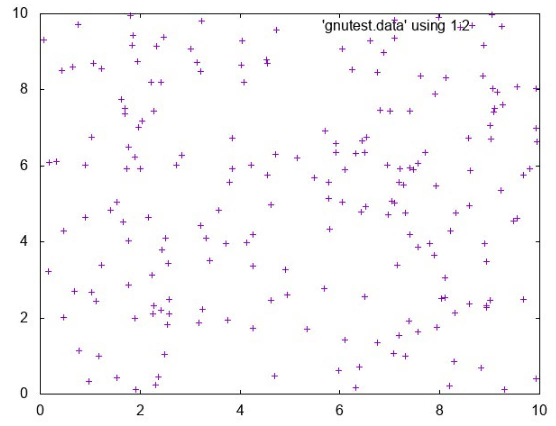

Fig 4.30 Initial Positions of Particles

N = 200

The captured final positions of these models are on the following page:

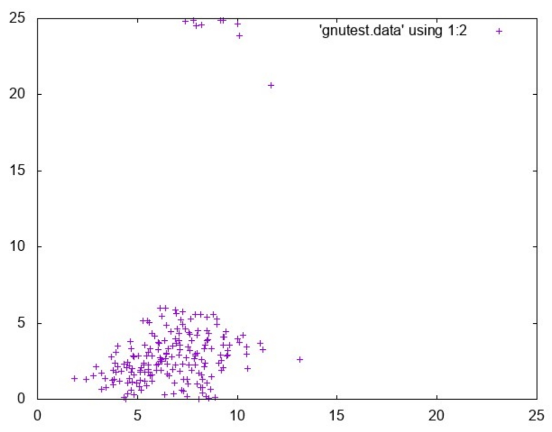

Fig 4.31 Final Positions of Particles

N = 200, L = 25, μ = 0.01.

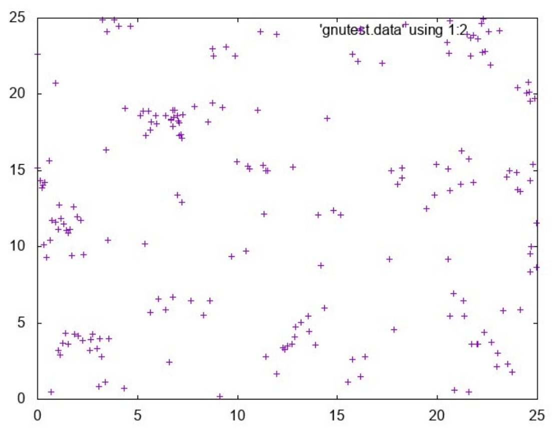

Fig 4.32 Final Positions of Particles

N = 200, L = 25, μ = 2.

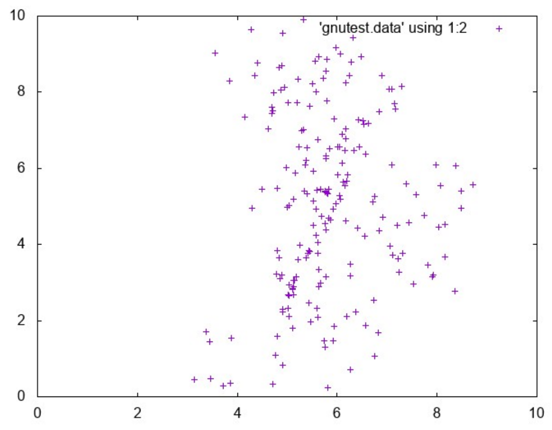

Fig 4.33 Final Positions of Particles

N = 200, L = 10, μ = 0.01.

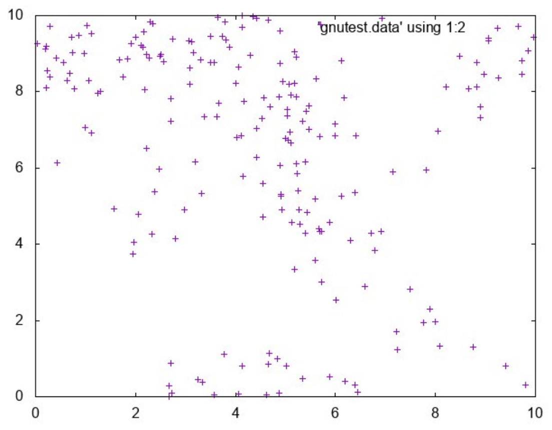

Fig 4.34 Final Positions of Particles

N = 200, L = 10, μ = 2.

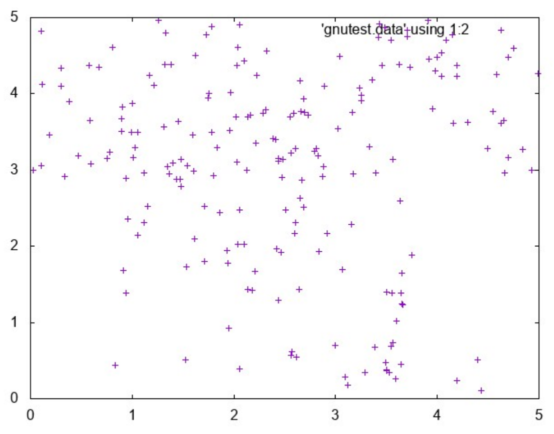

Fig 4.35 Final Positions of Particles

N = 200, L = 5, μ = 0.01.

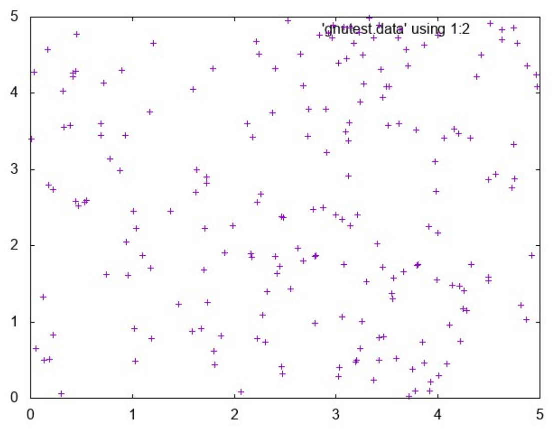

Fig 4.36 Final Positions of Particles

N = 200, L = 5, μ = 2.

The videos of these models are linked in appendices 30, 31, 32, 33, 34 and 35 respectively. For low noise and a large boundary, the particles formed many large groups as expected. These groups eventually formed one large group as seen in figure 4.31. Similar to previous models with the same parameters, this process of collecting together took less time as the boundary size decreased.

Similar to the 100-particle model with high noise and a large boundary the particles formed many groups of various sizes moving around randomly. As the boundary size decreased the particles began to move in the same direction. Occasionally however, the group would divide and then reconnect. At the smallest boundary size all particles flowed together but with each particle "shaking" independently. The direction of this "flow" changed frequently but with very small changes in direction.

500 Particles

For the final set of models with these interaction rules we have models with 500 particles. An example of the starting positions of the particles in these models is as follows:

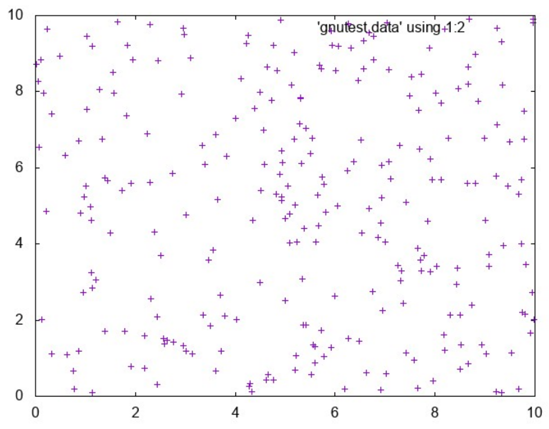

Fig 4.37 Initial Positions of Particles

N = 500

The final positions of the particles in these models are on the following page:

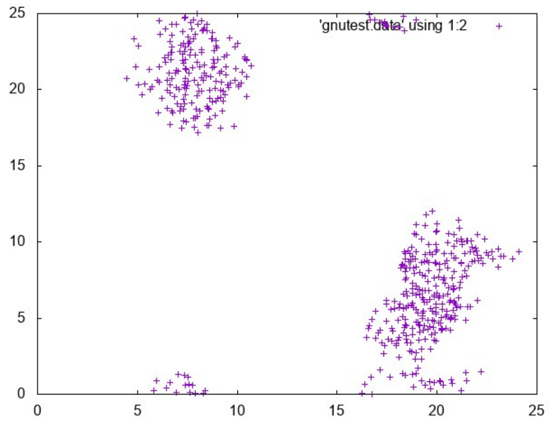

Fig 4.38 Final Positions of Particles

N = 500, L = 25, $\mu = 0.01$.

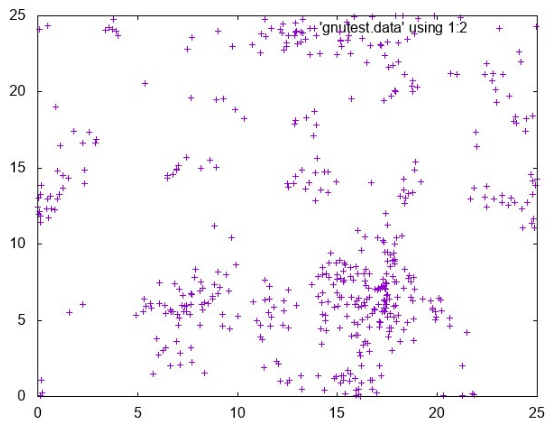

Fig 4.39 Final Positions of Particles

N = 500, L = 25, $\mu = 2$.

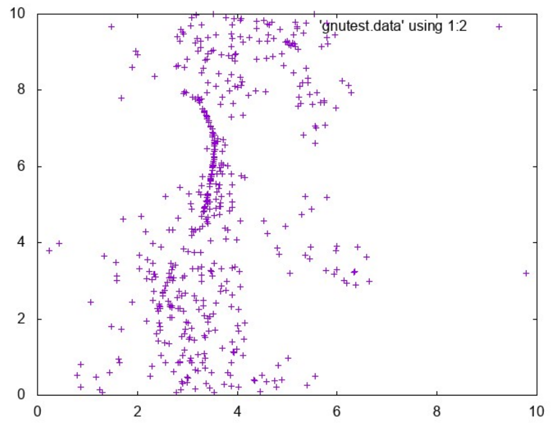

Fig 4.40 Final Positions of Particles

N = 500, L = 10, $\mu = 0.01$.

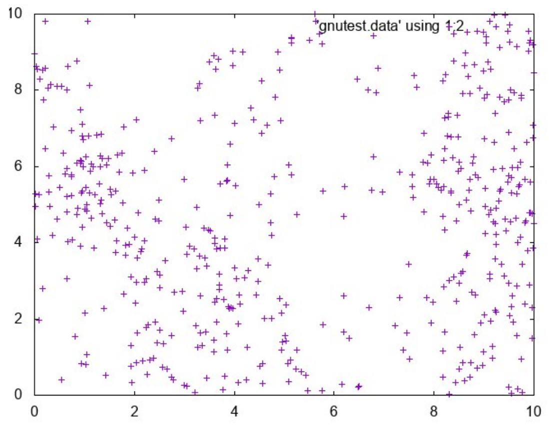

Fig 4.41 Final Positions of Particles

N = 500, L = 10, $\mu = 2$.

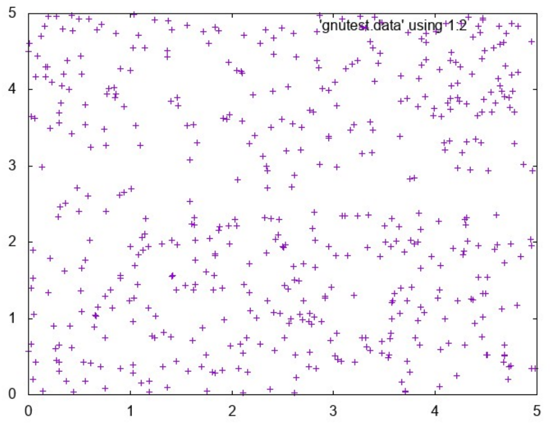

Fig 4.42 Final Positions of Particles

N = 500, L = 5, $\mu = 0.01$.

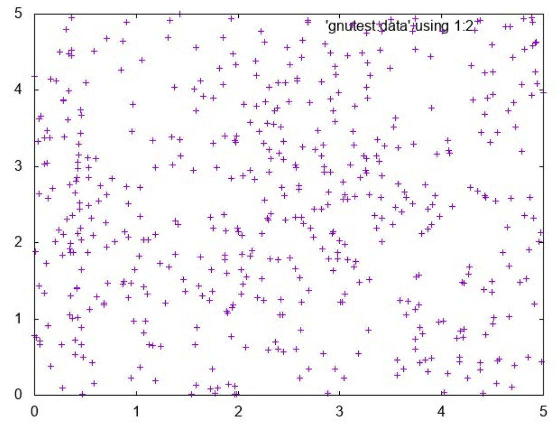

Fig 4.43 Final Positions of Particles

N = 500, L = 5, $\mu = 2$.

The animated plots of these models are uploaded and linked as appendices 36, 37, 38, 39, 40 and 41. Note that 500 is the upper limit for particles in these models hence, these models will have slightly slower performance.

For a low noise amplitude and a large boundary, the particles form large groups similar to previous models. These groups combine in 2-3 large groups as seen in figure 4.38. When the boundary is decreased to 10 the particles begin as two "streams" that flow opposite to each other before combining head on causing one large group that moves together with a long line of particles where these streams collided. This can be clearly seen in figure 4.40. At the lowest boundary size, the particles move ordered together as one group.

In the model of high noise and a large boundary, many particles form a few large clusters, but some still move independently. As the clusters of particles are larger than previous models (with the same parameters), they seemingly stay together instead of frequently splitting. When the boundary is decreased in size, most of the particles move together. Frequently however, some clusters break off and move independently before reconnecting. This is especially prominent when large amounts of particles move across a boundary. At the smallest boundary size, all particles move in the same approximate direction which doesn't change much in comparison with the smaller particle models. Similar to the 200-particle model, the particles freely move "side to side" while moving in the same direction overall.

Limited Neighbourhood

For a further model, a restriction is added to each particle limiting the neighbourhood size to 3. This restriction could be due to limited visibility of other particles either with a particle only able to process the direction of velocity of 2-3 neighbours at a time or limited visibility from the environment. As mentioned in the code breakdown chapter, this addition was done by adding a break statement to the for loop for when the neighbourhood reaches a size of 3. These limited neighbourhood models used a noise amplitude of 2 and a boundary size of 5. The result of these models will be compared to the previous models with the same noise amplitude and boundary size as follows:

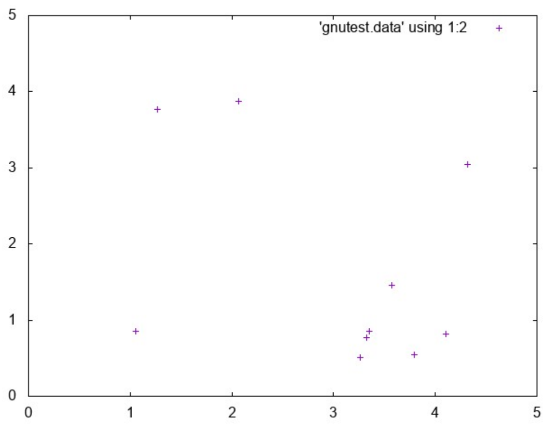

Fig 4.9 (from page 23) Final Positions of Particles

N = 10, L = 5, $\mu = 2$.

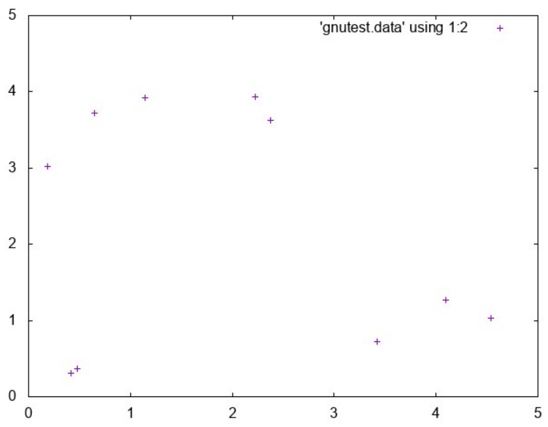

Fig 4.44 Final Positions of Particles

N = 10, L = 5, $\mu = 2$. (Limited Neighbours)

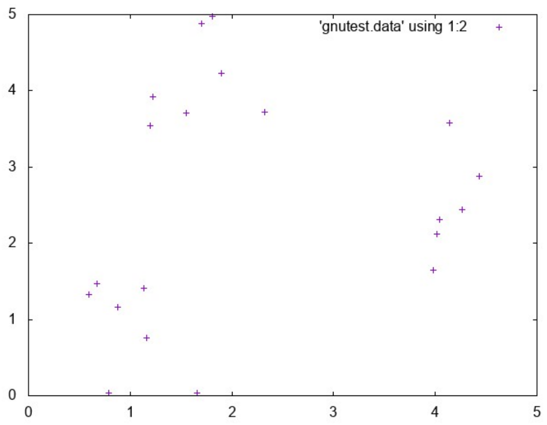

Fig 4.16 (from page 25) Final Positions of Particles

N = 20, L = 5, $\mu = 2$.

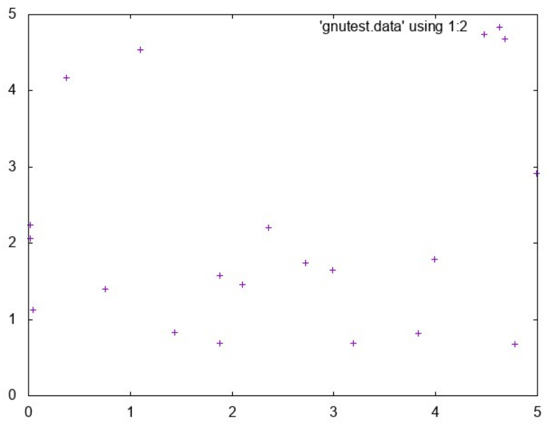

Fig 4.45 Final Positions of Particles

N = 20, L = 5, $\mu = 2$. (Limited Neighbours)

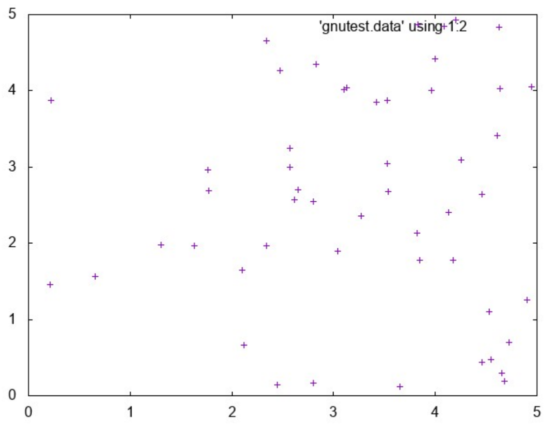

Fig 4.22 (from page 27) Final Positions of Particles

N = 50, L = 5, $\mu = 2$.

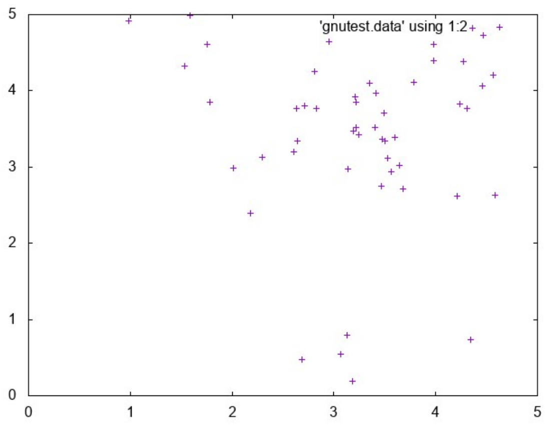

Fig 4.46 Final Positions of Particles

N = 50, L = 5, $\mu = 2$. (Limited Neighbours)

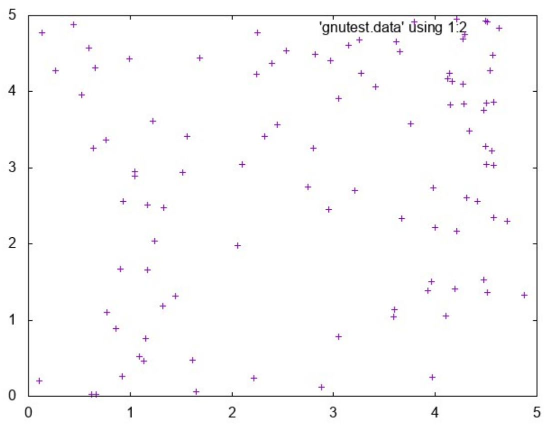

Fig 4.29 (from page 29) Final Positions of Particles

N = 100, L = 5, μ = 2.

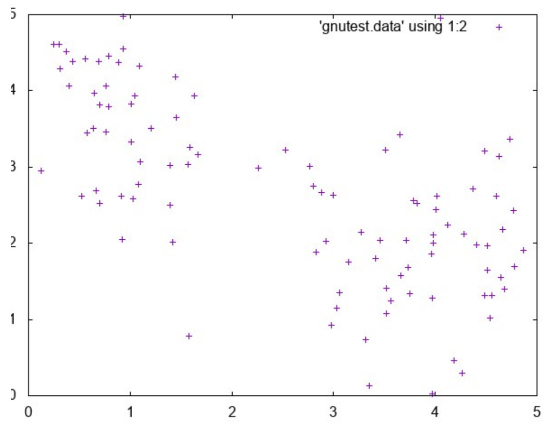

Fig 4.47 Final Positions of Particles

N = 100, L = 5, μ = 2. (Limited Neighbours)

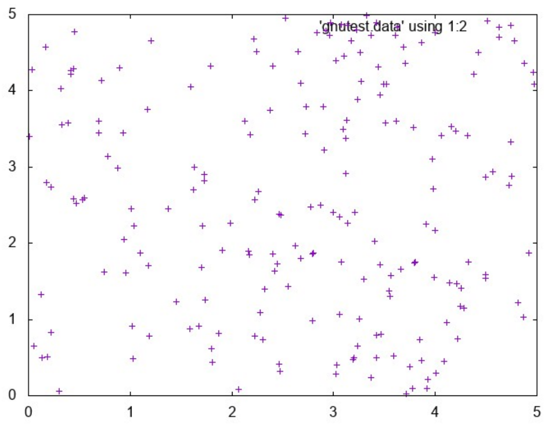

Fig 4.36 (from page 31) Final Positions of Particles

N = 200, L = 5, μ = 2.

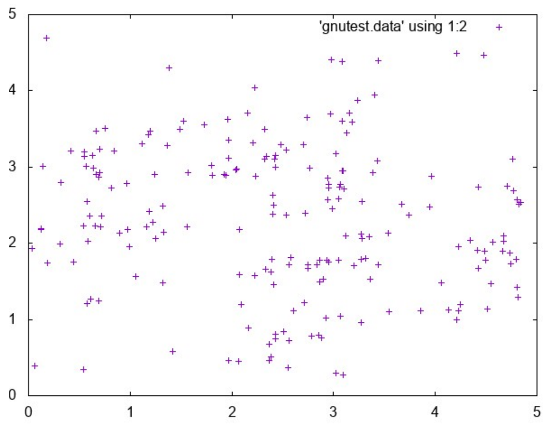

Fig 4.48 Final Positions of Particles

N = 200, L = 5, μ = 2. (Limited Neighbours)

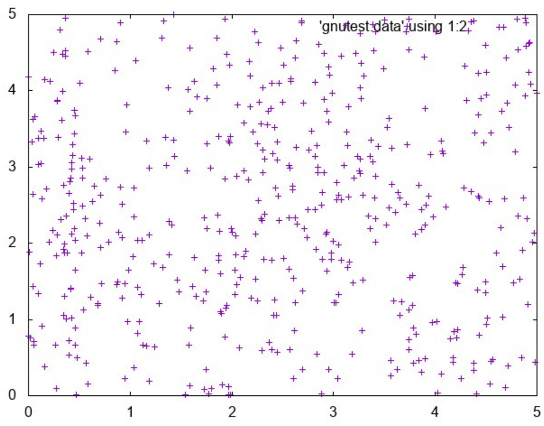

Fig 4.43 (from page 33) Final Positions of Particles

N = 500, L = 5, μ = 2.

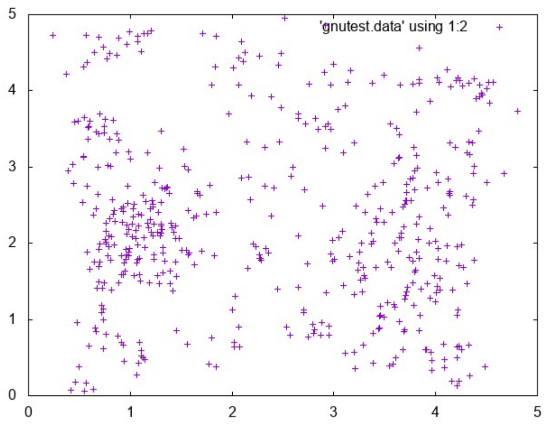

Fig 4.49 Final Positions of Particles

N = 500, L = 5, μ = 2. (Limited Neighbours)

The videos of the limited neighbour models are uploaded and linked as Appendices 42, 43, 44, 45, 46 and 47 respectively. The differences between these new models and the previous models can be clearly seen in the images. The models with lower numbers of particles contain systems of particles that form small compact groups moving together with rapidly changing direction. When the number of particles increases, the particles form much larger groups. However, these groups are much more compact then the non-restricted models. The 200 and 500 particle restricted models expand on this with most particles moving around as one compact group rapidly changing direction and occasionally separating into small groups. Unlike the non-restricted models this group is more compact, and the direction of the group rapidly changes even with a large number of particles. This movement is notably similar to the observed movement of flocks of birds moving in a tight group rapidly changing direction.

Multiple Particle types

The final set of models contain systems with two types of particles. These two different types have two different absolute velocities. As mentioned in the code breakdown this was achieved by adding a second absolute velocity variable and splitting the particles so that odd particles had one absolute velocity and even particles had the other absolute velocity. These systems had a noise amplitude of 2 and a boundary size of 5 and will be compared to the original systems with these parameters.

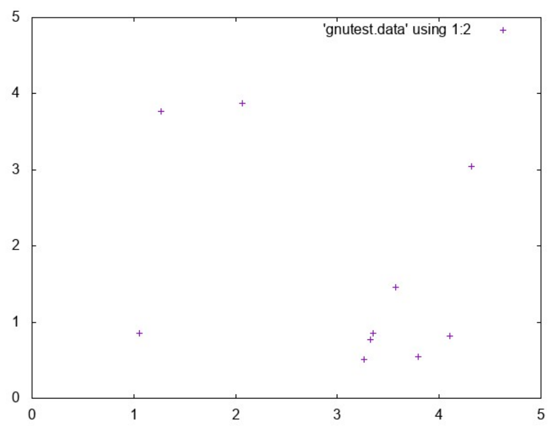

Fig 4.9 (from page 23) Final Positions of Particles

N = 10, L = 5, μ = 2.

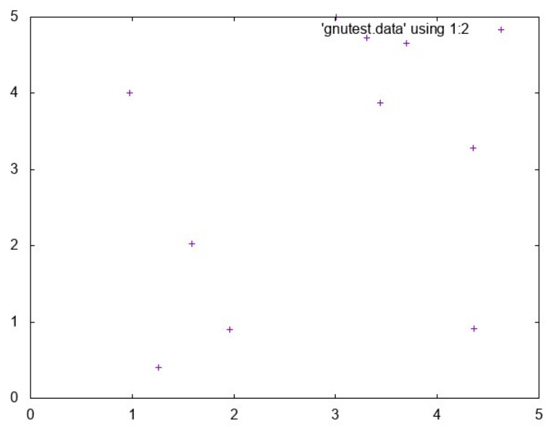

Fig 4.50 Final Positions of Particles

N = 10, L = 5, μ = 2. (Two types of Particle)

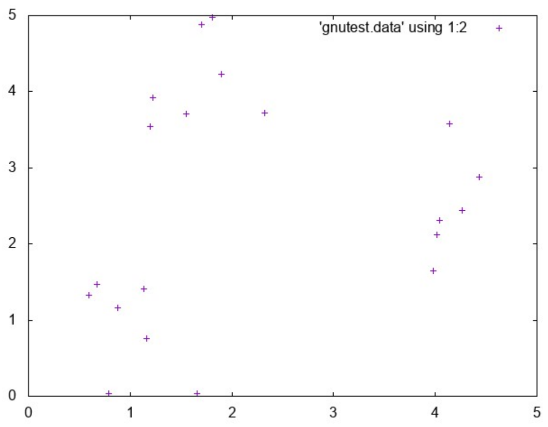

Fig 4.16 (from page 25) Final Positions of Particles

N = 20, L = 5, $\mu = 2$.

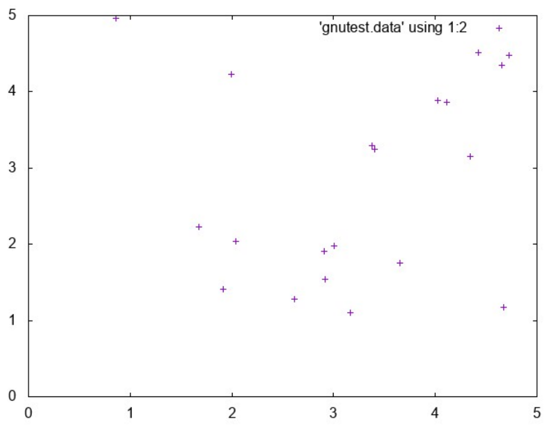

Fig 4.51 Final Positions of Particles

N = 20, L = 5, $\mu = 2$. (Two types of Particle)

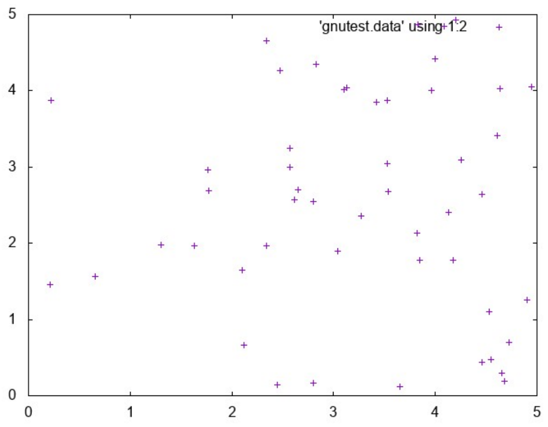

Fig 4.22 (from page 27) Final Positions of Particles

N = 50, L = 5, $\mu = 2$.

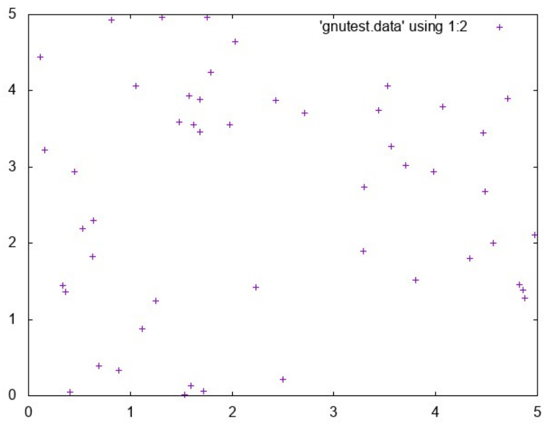

Fig 4.52 Final Positions of Particles

N = 50, L = 5, $\mu = 2$. (Two types of Particle)

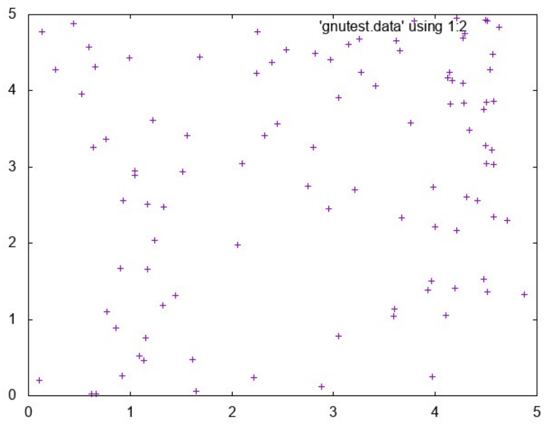

Fig 4.29 (from page 29) Final Positions of Particles

N = 100, L = 5, $\mu = 2$.

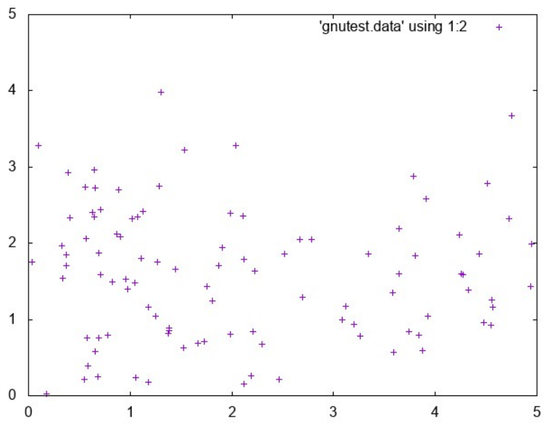

Fig 4.53 Final Positions of Particles

N = 100, L = 5, $\mu = 2$. (Two types of Particle)

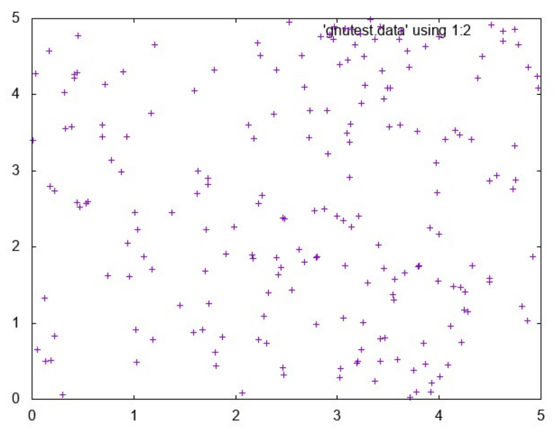

Fig 4.36 (from page 31) Final Positions of Particles

N = 200, L = 5, $\mu = 2$.

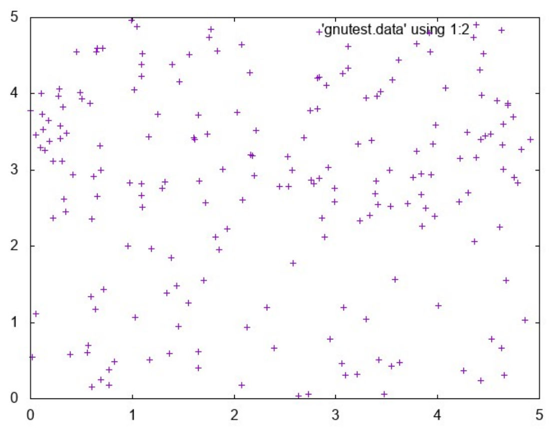

Fig 4.54 Final Positions of Particles

N = 200, L = 5, $\mu = 2$. (Two types of Particle)

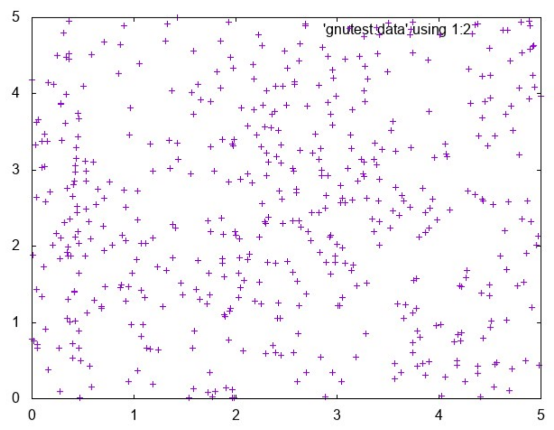

Fig 4.43 (from page 33) Final Positions of Particles

N = 500, L = 5, $\mu = 2$.

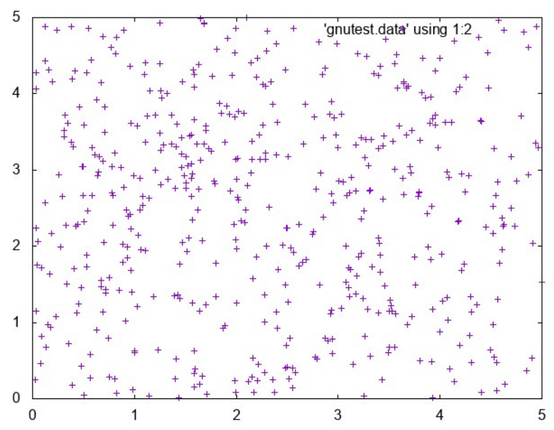

Fig 4.55 Final Positions of Particles

N = 500, L = 5, $\mu = 2$. (Two types of Particle)

The videos for these models are uploaded and linked as appendices 48, 49, 50, 51, 52 and 53 respectively. Each of the models in this set contain systems of particles that seemingly form two groups with a group for each type of particle. However, as these two groups frequently interact we can observe all particles moving in the same approximate direction. As the number of particles increases, this behaviour becomes more apparent.

Comparisons with Nature/Technology

As mentioned in the introduction chapter, there are many real-world examples of systems of self-propelling particles. This includes various artificial examples in addition to many examples throughout nature. One of the most common and intriguing examples in nature is a flock of birds. If we observe the video linked in appendix 53, the movements of the flock of birds are very similar (although 3-dimensional) to the models with limited neighbours. This similar behaviour is prominent in the 200 and 500 particle models linked as appendix 51 and 52 respectively. This is understandable as birds may only be able to perceive the direction of velocity of a limited number of neighbouring birds.

Conclusion

To conclude, this project has achieved its objective to create computational models of self-propelling particles using Vicsek's model. This report has compared the different models with two independent parameters across an increasing number of particles. These models have been expanded upon to include a system with restricted interaction rules and a system more with than one type of particle.

The particles in these systems do exhibit collective behaviour even with a large amount of noise for each particle. This can be in the form of small groups of particles which become larger as the particle density increases. This eventually leads to systems where all particles move in the same direction while still moving or "twitching" independently inside the group.

Systems with two types of particles seemingly move in two separate groups. However, the groups move in the same direction when they interact. When the neighbourhood is limited, the particles form more compact groups and as a large group they move around more freely. This behaviour is comparable to the observed behaviour of flocks of birds in nature.

In the future, this project can be expanded upon with more models for different interaction rules. This could include systems with more than two particles or even two types of particles with a restricted neighbourhood. These models could also be expanded for much larger systems of particles. As it is possible to see in the animated models, the upper limit for these programs is 500 particles. With more powerful equipment, a different compiler or perhaps a different programming language, the number of particles could be much larger.

Acknowledgements

Firstly, I'd like to say a big thank to you my family for all the support of the past 4 years. There have been many times when I've struggled, and the unwavering support has always pulled me through. Even in these last few weeks the constant messages and support have kept up my motivation. I honestly cannot thank you enough.

Secondly, I would like to thank my supervisor Dr Andrei Zvelindovsky. You have been incredibly supportive and enthusiastic throughout this project. I could not have asked for a better supervisor

I'd also like to thank Matt Barrett, Jack Murphy, Chloe Page, Josh Edwards, Eloise Bowden for putting up with these past couple of years (this year especially). Matt especially over these last few weeks helping keep my motivation up.

I also need to thank the whole Premier Inn team past and present for the continued love and support throughout this degree. Adam Washington, Jordan Stow, Mike Atter, Katie Eyre, Rhogan Cartwright, Bogdan Sofronie, Ben Wash, Holly Thornton, Holly Parkin, John Redpath, Johnathon Stevens, Morgan Bennet-Martin, Constance Yau, Bev and the list goes on. I could not have picked a greater group of people to have got to have known over these past couple of years. Thank you for putting up with me I will really miss you all.

I'd also like to thank Callum King, Jordan Anderson, Deva Donofrio, Nick DeBoer, Alec DeBoer, James Lehane, and Edward Bageman, who have been fantastic friends for many years and especially throughout this course.

I’d finally like to thank the INB staff. This project has been very computer intensive and would not be possible without the 4th year room continuously running.

Appendices

Appendix 1: Program with Standard Interaction Rules

```
#include "stdafx.h"
//#include <iostream>
#include <fstream>
#include "gnuplot.h"
#include <random>
#include <windows.h>
#include <math.h>
#define N 50 //set the number of particles

//using namespace std;

int main()
{
	Gnuplot plot; //Use of external Gnuplot library
	ofstream myfile;
	myfile.open("gnutest.data"); //open data file to export to

	const long double PI = 3.141592653589793238; //define constant PI
	const double L = 10; //boundary size
	double x[N], y[N], theta[N], xvel[N], yvel[N]; //arrays for each x and
y coordinates, velocities and direction
	double vel = 0.05; //absolute velocity of each particle (constant)
	double r = 1; //interaction radius
	double deltat = 1; //time interval
	double maxt = 1000; //max time period
	double noise = 2; // noise amplitude
	double xvelsum = 0; //sum of velocity x component
	double yvelsum = 0; //sum of velocity y component
	double xvelavg = 0; //average x velocity
	double yvelavg = 0; //average y velocity
	double non = 0; //number of neighbours inside the interaction radius
	double avgtheta = 0; //average theta of neighbourhood
	double difference = 0; //variable for distance between particles



	// random number distribution
	random_device rd;
	mt19937 e2(rd()); //Mersenne Twister 19937 generator
```

```
        uniform_real_distribution<> dist(0, 1);


        for (int i = 0; i < N; i++) //intialise each particle
        {
                x[i] = L*dist(e2); //intial x = L * (random variable between 0
and 1)
                y[i] = L*dist(e2); //intial y = L * (random variable between 0
and 1)
                theta[i] = 2 * PI*(dist(e2) - 0.5); //intial theta = 2 * PI *
(random variable between 0 and 1)
                xvel[i] = 0; //initialise x velocity
                yvel[i] = 0; //initialise y velocity
                myfile << x[i] << "\t" << y[i] << endl;
        }

        myfile.close();
        plot("set xrange [0:10]"); //sets x range in Gnuplot
        plot("set yrange [0:10]"); //sets y range in Gnuplot
        plot("plot 'gnutest.data' using 1:2\n"); //plot the data file
        plot("set term png");
        plot("set output 'outputstart.png' "); //outputs a PNG file with the
starting random distribution
        plot("replot");
        plot("set term win"); //sets Gnuplot back to the window format



        for (double t = 0; t < maxt;) //loop for each timestep up to max time
        {
                myfile.open("gnutest.data");
                for (int i = 0; i < N; i++)
                {
                        xvel[i] = vel*cos(theta[i]); //update velocity x
component
                        yvel[i] = vel*sin(theta[i]); //update velocity y
component
                        x[i] += xvel[i] * deltat; //update x coordinate
                        y[i] += yvel[i] * deltat; //update y coordinate
        //Checks if any particle is outside the boundary
              if (x[i] < 0)
              {
                     x[i] = L + x[i];
              }
              if (x[i] > L)
              {
                     x[i] = x[i] - L;
              }
              if (y[i] < 0)
              {
                     y[i] = L + y[i];
              }
              if (y[i] > L)
              {
                     y[i] = y[i] - L;
              }

       }

       //output and plot these positions
       for (int i = 0; i < N; i++)
       {
              myfile << x[i] << "\t" << y[i] << endl;
       }
       myfile.close();
       plot("plot 'gnutest.data' using 1:2\n");



       for (int i = 0; i < N; i++)
       {
              //reset all variables used in loop
              xvelsum = 0;
              yvelsum = 0;
              non = 0;
              xvelavg = 0;
              yvelavg = 0;
              avgtheta = 0;
```

```
                    for (int j = 0; j < N; j++)
                    {

                            difference = 0;
                            difference = sqrt(pow((x[j] - x[i]), 2.0) +
pow((y[j] - y[i]), 2.0)); //distance between particles
                            if (i == j) //if they are the same particle add
the velocities to the sums
                            {
                                    xvelsum += xvel[i];
                                    yvelsum += yvel[i];
                                    non++; //increase size of the
neighbourhood
                            }
                            else if (difference < r) //if the different
particles are in the radius add the velocities to the sums
                            {
                                    xvelsum += xvel[j];
                                    yvelsum += yvel[j];
                                    non++; //increase size of the
neighbourhood
                                    //cout << "Particles interact" << endl;
//test notification if the particles are close enough together
                            }
                    }
                    xvelavg = xvelsum / non; //calculate average x velocity
                    yvelavg = yvelsum / non; //calculate average y velocity
                    avgtheta = atan2(yvelavg, xvelavg); //calculate average
theta
                    //cout << "average theta = " << avgtheta << endl; //Used
to test the calculation when constructing
                    theta[i] = avgtheta + noise*(dist(e2) - 0.5); //update
theta

            }

            t += deltat; //increase timestep
            Sleep(12); //add 12 millisecond delay to allow gnuplot to keep
up

    }
```

```
	plot("set term png");
	plot("set output 'outputend.png' "); //Output the final result as a PNG file
	plot("replot");



	return 0;
}
```

Appendix 2: Limited Neighbourhood Program

```
#include "stdafx.h"
//#include <iostream>
#include <fstream>
#include "gnuplot.h"
#include <random>
#include <windows.h>
#include <math.h>
#define N 50 //set the number of particles

//using namespace std;

int main()
{
	Gnuplot plot; //use of external Gnuplot library
	ofstream myfile;
	myfile.open("gnutest.data"); //opens data file

	const long double PI = 3.141592653589793238; //define constant PI
	const double L = 10; //boundary size
	double x[N], y[N], theta[N], xvel[N], yvel[N]; //arrays for each x and
y coordinates, velocities and direction
	double vel = 0.05; //absolute velocity of each particle (constant)
	double r = 1; //interaction radius
	double deltat = 1; //time interval
	double maxt = 1000; //max time period
	double noise = 2; // noise amplitude
	double xvelsum = 0; //sum of velocity x component
	double yvelsum = 0; //sum of velocity y component
	double xvelavg = 0; //average x velocity
	double yvelavg = 0; //average y velocity
	double non = 0; //number of neighbours inside the interaction radius
	double avgtheta = 0; //average theta of neighbourhood
	double difference = 0; //variable for distance between particles


	// random number distribution
	random_device rd;
	mt19937 e2(rd()); //Mersenne Twister 19937 generator
	uniform_real_distribution<> dist(0, 1);
	for (int i = 0; i < N; i++) //intialise each x and y
	{
		x[i] = L*dist(e2); //intial x = L * (random variable between 0 
and 1)
		y[i] = L*dist(e2); //intial y = L * (random variable between 0 
and 1)
		theta[i] = 2 * PI*(dist(e2) - 0.5); //intial theta = 2 * PI * 
(random variable between 0 and 1)
		xvel[i] = 0; //initialise x velocity
		yvel[i] = 0; //initialise y velocity
		myfile << x[i] << "\t" << y[i] << endl;
	}

	myfile.close();
	plot("set xrange [0:10]"); //set Gnuplot x range
	plot("set yrange [0:10]"); //set Gnuplot y range
	plot("plot 'gnutest.data' using 1:2\n"); //plot data to Gnuplot
	plot("set term png");
	plot("set output 'outputstart.png' "); //outputs the starting random 
distribution
	plot("replot");
	plot("set term win"); //sets gnuplot to window format



	for (double t = 0; t < maxt;) //loop for each timestep up to max time
	{
		myfile.open("gnutest.data");
		for (int i = 0; i < N; i++)
		{
			xvel[i] = vel*cos(theta[i]); //update velocity x 
component
			yvel[i] = vel*sin(theta[i]); //update velocity y 
component
			x[i] += xvel[i] * deltat; //update x coordinate
			y[i] += yvel[i] * deltat; //update y coordinate


			//Checks if any particle is outside the boundary
			if (x[i] < 0)
```

```
                {
                        x[i] = L + x[i];
                }
                if (x[i] > L)
                {
                        x[i] = x[i] - L;
                }
                if (y[i] < 0)
                {
                        y[i] = L + y[i];
                }
                if (y[i] > L)
                {
                        y[i] = y[i] - L;
                }

        }

        //output and plot these positions
        for (int i = 0; i < N; i++)
        {
                myfile << x[i] << "\t" << y[i] << endl;
        }
        myfile.close();
        plot("plot 'gnutest.data' using 1:2\n");


        for (int i = 0; i < N; i++)
        {
                //reset all variables used in loop
                xvelsum = 0;
                yvelsum = 0;
                non = 0;
                xvelavg = 0;
                yvelavg = 0;
                avgtheta = 0;



                for (int j = 0; j < N; j++)
                {
```

```
                                difference = 0;
                                difference = sqrt(pow((x[j] - x[i]), 2.0) +
pow((y[j] - y[i]), 2.0)); //distance between particles
                                if (non == 3)
                                {
                                        break; //Particles can interact with a
limited number of neighbours
                                }
                                else if (i == j) //if they are the same particle
add the velocities to the sums
                                {
                                        xvelsum += xvel[i];
                                        yvelsum += yvel[i];
                                        non++; //increase size of the
neighbourhood
                                }
                                else if (difference < r) //if the different
particles are in the radius add the velocities to the sums
                                {
                                        xvelsum += xvel[j];
                                        yvelsum += yvel[j];
                                        non++; //increase size of the
neighbourhood
                                        //cout << "Particles interact" << endl;
//test notification if the particles are close enough together
                                }
                        }
                        xvelavg = xvelsum / non; //calculate average x velocity
                        yvelavg = yvelsum / non; //calculate average y velocity
                        avgtheta = atan2(yvelavg, xvelavg); //calculate average
theta
                        //cout << "average theta = " << avgtheta << endl; //Used
to test the calculation when constructing
                        theta[i] = avgtheta + noise*(dist(e2) - 0.5); //update
theta

                }

                t += deltat; //increase timestep
                Sleep(12); //add 12 millisecond delay to allow gnuplot to keep
up
```

```
        }
        plot("set term png");
        plot("set output 'outputend.png' "); //output final result to a PNG
file
        plot("replot");



        return 0;
}
```

Appendix 3: Program with Multiple Particles

```
#include "stdafx.h"
//#include <iostream>
#include <fstream>
#include "gnuplot.h"
#include <random>
#include <windows.h>
#include <math.h>
#define N 50 //set the number of particles

//using namespace std;

int main()
{
	Gnuplot plot; //use of external Gnuplot library
	ofstream myfile;
	myfile.open("gnutest.data"); //opens data file

	const long double PI = 3.141592653589793238; //define constant PI
	const double L = 10; //boundary size
	double x[N], y[N], theta[N], xvel[N], yvel[N]; //arrays for each x and
y coordinates, velocities and direction
	double vel1 = 0.05; //absolute velocity of particle type one
(constant)
	double vel2 = 0.10; //absolute velocity of particle type two
(constant)
	double r = 1; //interaction radius
	double deltat = 1; //time interval
	double maxt = 1000; //max time period
	double noise = 2; // noise amplitude
	double xvelsum = 0; //sum of velocity x component
	double yvelsum = 0; //sum of velocity y component
	double xvelavg = 0; //average x velocity
	double yvelavg = 0; //average y velocity
	double non = 0; //number of neighbours inside the interaction radius
	double avgtheta = 0; //average theta of neighbourhood
	double difference = 0; //variable for distance between particles



	// random number distribution
	random_device rd;
	mt19937 e2(rd()); //Mersenne Twister 19937 generator
	uniform_real_distribution<> dist(0, 1);


	for (int i = 0; i < N; i++) //intialise each particle
	{
		x[i] = L*dist(e2); //intial x = L * (random variable between 0 
and 1)
		y[i] = L*dist(e2); //intial y = L * (random variable between 0 
and 1)
		theta[i] = 2 * PI*(dist(e2) - 0.5); //intial theta = 2 * PI * 
(random variable between 0 and 1)
		xvel[i] = 0; //initialise x velocity
		yvel[i] = 0; //initialise y velocity
		myfile << x[i] << "\t" << y[i] << endl;
	}

	myfile.close();
	plot("set xrange [0:10]"); //set Gnuplot x range
	plot("set yrange [0:10]"); //set Gnuplot y range
	plot("plot 'gnutest.data' using 1:2\n"); //plot data to Gnuplot
	plot("set term png");
	plot("set output 'outputstart.png' "); //outputs the starting random 
distribution
	plot("replot");
	plot("set term win"); //sets gnuplot to window format


	for (double t = 0; t < maxt;) //loop for each timestep up to max time
	{
		myfile.open("gnutest.data");

		for (int i = 0; i < N; i++)
		{
			if (i % 2 == 0) //odd and even particles have different 
velocities
			{
				xvel[i] = vel2*cos(theta[i]); //update velocity x 
component
				yvel[i] = vel2*sin(theta[i]); //update velocity y 
component
```

```
                    }
                    else
                    {
                          xvel[i] = vel1*cos(theta[i]); //update velocity x
component
                          yvel[i] = vel1*sin(theta[i]); //update velocity y
component
                    }

                    x[i] += xvel[i] * deltat; //update x coordinate
                    y[i] += yvel[i] * deltat; //update y coordinate


                    //Checks if any particle is outside the boundary
                    if (x[i] < 0)
                    {
                          x[i] = L + x[i];
                    }
                    if (x[i] > L)
                    {
                          x[i] = x[i] - L;
                    }
                    if (y[i] < 0)
                    {
                          y[i] = L + y[i];
                    }
                    if (y[i] > L)
                    {
                          y[i] = y[i] - L;
                    }

              }


              //output and plot these positions
              for (int i = 0; i < N; i++)
              {
                    myfile << x[i] << "\t" << y[i] << endl;
              }
              myfile.close();
              plot("plot 'gnutest.data' using 1:2\n");
```

```
		for (int i = 0; i < N; i++)
		{
			//reset all variables used in loop
			xvelsum = 0;
			yvelsum = 0;
			non = 0;
			xvelavg = 0;
			yvelavg = 0;
			avgtheta = 0;


			for (int j = 0; j < N; j++)
			{

				difference = 0;
				difference = sqrt(pow((x[j] - x[i]), 2.0) + 
pow((y[j] - y[i]), 2.0)); //distance between particles
				if (i == j) //if they are the same particle add 
the velocities to the sums
				{
					xvelsum += xvel[i];
					yvelsum += yvel[i];
					non++; //increase size of the 
neighbourhood
				}
				else if (difference < r) //if the different 
particles are in the radius add the velocities to the sums
				{
					xvelsum += xvel[j];
					yvelsum += yvel[j];
					non++; //increase size of the 
neighbourhood
					//cout << "Particles interact" << endl; 
//test notification if the particles are close enough together
				}
			}
			xvelavg = xvelsum / non; //calculate average x velocity
			yvelavg = yvelsum / non; //calculate average y velocity
```

```
                    avgtheta = atan2(yvelavg, xvelavg); //calculate average
theta
                    //cout << "average theta = " << avgtheta << endl; //Used
to test the calculation when constructing
                    theta[i] = avgtheta + noise*(dist(e2) - 0.5); //update
theta

              }

              t += deltat; //increase timestep
              Sleep(12); //add 12 millisecond delay to allow gnuplot to keep
up

       }
       plot("set term png");
       plot("set output 'outputend.png' "); //output final result to a PNG
file
       plot("replot");


       return 0;
}
```

Appendix 4: External GnuPlot Library (From Reference [17])

```
#ifndef GNUPLOT_H_
#define GNUPLOT_H_
#include <string>
#include <iostream>
using namespace std;
class Gnuplot {
public:
	Gnuplot();
	~Gnuplot();
	void operator ()(const string & command);
	// send any command to gnuplot
protected:
	FILE *gnuplotpipe;
};
Gnuplot::Gnuplot() {
	// with -persist option you will see the windows as your program ends
	//gnuplotpipe=_popen("gnuplot -persist","w");
	//without that option you will not see the window
	// because I choose the terminal to output files so I don't want to see
the window
	gnuplotpipe = _popen("gnuplot", "w");
	if (!gnuplotpipe) {
		cerr << ("Gnuplot not found !");
	}
}
Gnuplot::~Gnuplot() {
	fprintf(gnuplotpipe, "exit\n");
	_pclose(gnuplotpipe);
}
void Gnuplot::operator()(const string & command) {
	fprintf(gnuplotpipe, "%s\n", command.c_str());
	fflush(gnuplotpipe);
	// flush is necessary, nothing gets plotted else
};
#endif
```

Appendix 5: Research Plan

Callum Robinson 15589615

Modelling the Invincible

A Brief Description

This project focuses models of self-propelling particles with inspiration from the 1964 Sci-Fi novel ‘The Invincible’ by Stanislaw Lem which described very complex collective behaviour of large systems of individually very simple microrobots/robot-flies. We can see other examples in nature such as flocks of birds and schools of fish as well as artificial examples such as robots and Janus particles.

Systems consisting of self-propelling particles are of great interest for physicists and biologists because of the complex and fascinating phenomenon of the emergence of ordered motion [1]. Many aspects of the observed transition from disorder to ordered motion are not yet fully understood [1].

This project will focus on the mathematical modelling of systems of self-propelling particles using computation methods. This will be done using C to model Vicsek’s model while investigating the effect of different parameters as well as different interaction rules. The behaviours observed can then be compared with the behaviours in nature and technology.

Reasons for Choosing and Connection to Previous Studies

I had great interest in this project for similar reasons as above in that this collective behaviour seen nature is not fully understood and is fascinating. There are also similar concepts of artificial swarm intelligence seen in ‘The Invincible’ in modern science fiction which has always intrigued me. Considering these two points you may understand my interest in the ‘Collective behaviour of self-propelling particles’ seminar by Professor Andrei Zvelindovsky that was included in our ‘Advanced Topics of Mathematics and Mathematics Seminar’ module and hence my interest in this project.

The mathematics for modelling the positions vectors and velocities in these models will uses areas of the Calculus and Mechanics modules as well as link with the Fluid

Dynamics module. The use of C will link with the previous computational modules of Scientific Computing and Numerical Methods.

Survey of Literature and Equipment Needed

For the implementation of the mathematical models in C I will require the computers in the 4th year project room in the Isaac Newton building. The main book I have been provided in this project is 'Collective behaviour of self-propelling particles' by V. I Ratushna (ref [1]). This thesis goes into a lot more detail and in a different direction then what this project will focus on; however, it introduces the behaviours of systems containing self-propelling particles and Vicsek's method.

More research will need to be done on self-propelling particles and different examples. The Wikipedia site for self-propelling particles (ref [2]) provides a very clear overview in addition to the 'SPP model interactive simulation' (ref [3]). Research will also need to be done when implementing the models in C.

Brief Action Plan

The meetings will be every Wednesday morning in the Isaac Newton building unless rescheduled. The following is a rough action plan on the key stages of progression along this project and the week numbers refer to the meeting at which the stage is aimed to be completed.

Week 23	Research into Self-Propelled particles, Vicsek's model and Natural and Artificial examples
Week 25	Basic working implementation of the computation models with different parameters
Week 27/28	Different interaction rules implemented and comparisons of behaviours with observed behaviours in nature and technology

Week 29	Main work on draft report to be completed
Week 30	Draft completed and handed in
Week 33	Adjustments made from feedback from draft and any final results included

References

[1] V. I. Ratushna, Collective Behaviour of Self-Propelling Particles, Leiden: Leiden University, 2007.

[2] "Self-propelled particles," Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/Self-propelled_particles. [Accessed 15 February 2019].

[3] University of Colorado, "Self driven particle model," 2005. [Online]. Available: https://web.archive.org/web/20121014155808/http://

www.colorado.edu/physics/pion/srr/particles/.

Appendices 6-52 Links to Video Models

Note videos are unlisted, have no descriptions and are labelled with small file codes. The videos will also be stored externally can be requested.

Appendix 6: Link to 10 Particle Model, L = 25, Noise = 0.01

https://youtu.be/vb1tl5brWUU

Appendix 7: Link to 10 Particle Model, L = 25, Noise = 2

https://youtu.be/rtK4FD49D_o

Appendix 8: Link to 10 Particle Model, L = 10, Noise = 0.01

https://youtu.be/3d8r1MtZakk

Appendix 9: Link to 10 Particle Model, L = 10, Noise = 2

https://youtu.be/6QKg0xmBxHc

Appendix 10: Link to 10 Particle Model, L = 5, Noise = 0.01

https://youtu.be/E7m28rVD1x8

Appendix 11: Link to 10 Particle Model, L = 5, Noise = 2

https://youtu.be/GklrL1Qw2Rs

Appendix 12: Link to 20 Particle Model, L = 25, Noise = 0.01

https://youtu.be/Eb9YOZiNd7s

Appendix 13: Link to 20 Particle Model, L = 25, Noise = 2

https://youtu.be/APbYp7nbI2s

Appendix 14: Link to 20 Particle Model, L = 10, Noise = 0.01

https://youtu.be/ru7Dzda7780

Appendix 15: Link to 20 Particle Model, L = 10, Noise = 2

https://youtu.be/y0NMiaxGRHw

Appendix 16: Link to 20 Particle Model, L = 5, Noise = 0.01

https://youtu.be/XkVU6-p2prw

Appendix 17: Link to 20 Particle Model, L = 5, Noise = 2

https://youtu.be/eDfkaDQYKV4

Appendix 18: Link to 50 Particle Model, L = 25, Noise = 0.01

https://youtu.be/n7uaJTKD_-I

Appendix 19: Link to 50 Particle Model, L = 25, Noise = 2

https://youtu.be/KdzrGCzxGDY

Appendix 20: Link to 50 Particle Model, L = 10, Noise = 0.01

https://youtu.be/Xay0DdTPqQU

Appendix 21: Link to 50 Particle Model, L = 10, Noise = 2

https://youtu.be/IVymcSbM-2I

Appendix 22: Link to 50 Particle Model, L = 5, Noise = 0.01

https://youtu.be/yssvUCIPMbQ

Appendix 23: Link to 50 Particle Model, L = 5, Noise = 2

https://youtu.be/rrVCIPPQCLg

Appendix 24: Link to 100 Particle Model, L = 25, Noise = 0.01

https://youtu.be/rORHHuWsxNs

Appendix 25: Link to 100 Particle Model, L = 25, Noise = 2

https://youtu.be/zc751y4E3fM

Appendix 26: Link to 100 Particle Model, L = 10, Noise = 0.01

https://youtu.be/r5v8275HK8g

Appendix 27: Link to 100 Particle Model, L = 10, Noise = 2

https://youtu.be/zYrPTEU_5J8

Appendix 28: Link to 100 Particle Model, L = 5, Noise = 0.01

https://youtu.be/A-RA0eTqZBk

Appendix 29: Link to 100 Particle Model, L = 5, Noise = 2

https://youtu.be/Pajma35EtM8

Appendix 30: Link to 200 Particle Model, L = 25, Noise = 0.01

https://youtu.be/v6W7GsHHvt0

Appendix 31: Link to 200 Particle Model, L = 25, Noise = 2

https://youtu.be/om7b_GILV1k

Appendix 32: Link to 200 Particle Model, L = 10, Noise = 0.01

https://youtu.be/F3L-Wn3l9hg

Appendix 33: Link to 200 Particle Model, L = 10, Noise = 2

https://youtu.be/TvWloGDU0eU

Appendix 34: Link to 200 Particle Model, L = 5, Noise = 0.01

https://youtu.be/QGY_Iz2Cso8

Appendix 35: Link to 200 Particle Model, L = 5, Noise = 2

https://youtu.be/mvCSFXW7rWc

Appendix 36: Link to 500 Particle Model, L = 25, Noise = 0.01

https://youtu.be/w1F81rbt-rQ

Appendix 37: Link to 500 Particle Model, L = 25, Noise = 2

https://youtu.be/TDr0pA9UaL4

Appendix 38: Link to 500 Particle Model, L = 10, Noise = 0.01

https://youtu.be/egmdIU0_9OY

Appendix 39: Link to 500 Particle Model, L = 10, Noise = 2

https://youtu.be/f0OPCaNqEqE

Appendix 40: Link to 500 Particle Model, L = 5, Noise = 0.01

https://youtu.be/ldWzZ9Qmuco

Appendix 41: Link to 500 Particle Model, L = 5, Noise = 2

https://youtu.be/FGzHp2dwIh8

Appendix 42: Link to 10 Particle Limited Neighbour Model, L = 5, Noise = 2

https://youtu.be/EBLD5WmtZTo

Appendix 43: Link to 20 Particle Limited Neighbour Model, L = 5, Noise = 2

https://youtu.be/LqNVQ3Q9fXo

Appendix 44: Link to 50 Particle Limited Neighbour Model, L = 5, Noise = 2

https://youtu.be/e77GnkAs8Wo

Appendix 45: Link to 100 Particle Limited Neighbour Model, L = 5, Noise = 2

https://youtu.be/_0HuJZBMtXQ

Appendix 46: Link to 200 Particle Limited Neighbour Model, L = 5, Noise = 2

https://youtu.be/zsu2wS8ROHk

Appendix 47: Link to 500 Particle Limited Neighbour Model, L = 5, Noise = 2

https://youtu.be/pIWSAcosIro

Appendix 48: Link to 10 Particle Model with Two Types, L = 5, Noise = 2

https://youtu.be/rW4m5kkbpsg

Appendix 49: Link to 20 Particle Model with Two Types, L = 5, Noise = 2

https://youtu.be/xKLEEcJwtNA

Appendix 50: Link to 50 Particle Model with Two Types, L = 5, Noise = 2

https://youtu.be/q8HnmT1aBO4

Appendix 51: Link to 100 Particle Model with Two Types, L = 5, Noise = 2

https://youtu.be/eFUKxtWSMJw

Appendix 52: Link to 200 Particle Model with Two Types, L = 5, Noise = 2

https://youtu.be/Jsnc_b_oDvA

Appendix 53: Link to 500 Particle Model with Two Types, L = 5, Noise = 2

https://youtu.be/pbf2H5MsY28

Appendix 54: Link to Flock of Birds Video Example

https://www.youtube.com/watch?v=bb9ZTbYGRdc

Bibliography

[1] *Janus Particles*, Wikipedia, Available: https://en.wikipedia.org/wiki/Janus_particles

[2] R.A.Hernandez-Lopez,"Complex networks and collective behaviour in nature", Massachusetts Institute of Technology, Available: http://web.mit.edu/8.334/www/grades/projects/projects10/Hernandez-Lopez-Rogelio/dynamics_2.html

[3] *Self-Propelled Particles*, Wikipedia, available: https://en.wikipedia.org/wiki/Self-propelled_particles

[4] *Sheffield robot swarm exhibits Turing Learning*, The Engineer, Available: https://www.theengineer.co.uk/sheffield-robot-swarm-exhibits-turing-learning/

[5] S.Lem, *The Invincible,* Wydawnictwo MON, 1964 (Polish), Kraków

[6] S.Reid and N.Reid, *Self-driven particle model*, University of Colorado, 2005, Available: https://web.archive.org/web/20121014155808/http://www.colorado.edu/physics/pion/srr/particles/.

[7] *The Invincible*, Wikipedia, Available: https://en.wikipedia.org/wiki/The_Invincible

[8] V.I.Ratushna, *Collective Behaviour of Self-Propelling Particles with Conservative Kinematic Constraints,* Leiden University, Leiden, 2007

[9] *Vicsek Model*, Wikipedia, Available: https://en.wikipedia.org/wiki/Vicsek_model

Links used to help construct the code

[10] Function atan2, Cplusplus, Available: http://www.cplusplus.com/reference/cmath/atan2/

[11] Gnuplot download, SourceForge, Available: https://sourceforge.net/projects/gnuplot/

[12] How do you generate a random double uniformly distributed between 0 and 1 from C++?, Stack Overflow, Available: https://stackoverflow.com/questions/1340729/how-do-you-generate-a-random-double-uniformly-distributed-between-0-and-1-from-c

[13] How to Print Plots from Gnuplot and line types, University of Tennesse, Department of Mathematics, Available: http://www.math.utk.edu/~vasili/refs/How-to/gnuplot.print.html

[14] Use of stdafx.h, Cplusplus, Available: http://www.cplusplus.com/articles/1TUq5Di1/

[15] Using gnuplot, Harvey Mudd College, Department of Computer Science, Available: https://www.cs.hmc.edu/~vrable/gnuplot/using-gnuplot.html

[16] Vicsek Model Simulation, Mathworks, Available: https://uk.mathworks.com/matlabcentral/fileexchange/64208-vicsek-model-simulation

[17] Xingguang Liu, Those Initial Small Steps, Blogspot, Available: http://liuxingguang.blogspot.com/2012/05/how-to-call-gnuplot-from-c.html