The Boids are Flying!

A Look at Population Models and the Code of Agent-Based Modeling

Alec Foote

Whitman College
Spring 2016

Abstract: Differential equations can model the competition between species fairly well. But what if we modeled populations with moving shapes instead? Join me as I take you on a journey into the world of boids (that is, digital triangle birds), as they eat food, reproduce, and might even generate an effective simulation of population interference!

Contents

Contents ii

1 Introduction 1

2 Smooth Curves: Differential Equation Modeling 2
The Example: Birds . . . 2
Constructing an Equation . . . 2
Exponential Growth . . . 3
Environmental Threshold . . . 3
Equilibrium . . . 4
Competition . . . 5
Examining Competition . . . 5
Calculations in Theory . . . 6

3 The Digital World: Agent-Based Modeling 10
An Object . . . 10
Foodstuffs . . . 11
The Life of the Boid . . . 12
Boid Values . . . 13
Environment . . . 14
Virtual Realities . . . 15
Optimizations in Doughnut World . . . 16
The Presentation of the Thing . . . 19
What Could Have Been . . . 19

4 Spreadsheets and Graphs: Boid Data Analysis 21
Conversions . . . 21
The Control Boid . . . 21
Stable Population . . . 23
Actual Data . . . 23
Test 1: Single Population, Variable Start . . . 23
Test 2: Two Populations, Variable Starts . . . 24
Test 3: Movement Tests . . . 25
Test 4: Food Till Birth Tests . . . 26
The Possibility of Predicting Populations . . . 27

5 Models and Reality: Interrelations **28**
To the Math . . . 28
A Differential Equations Graph Party . . . 30
To the World . . . 32

6 Conclusion **33**

A Chart of Eigenvalues and Equilibrium Points **34**

B Chart of Boid Values for One Species **35**

C Processing Code: The Utilized Version of Boids **36**

Bibliography **53**

Chapter 1

Introduction

In our various roles as mathematicians, scientists, and humans, we try to examine the world and make sense of it. Oftentimes this leads us into the world of predictive models. One important situation in which such models can help us is the studying of populations: How populations grow and change over time, in response to factors such as population size, environment, and competition with other species.

This paper will examine two such population models: The math-based differential equation model, and the more computer-generated agent-based model.

We shall begin our exploration by putting together the differential equation model and seeing how it works. We will then move on to examine an agent-based model and the code that drives it. With that established, we will analyze some example simulations and relate them back to our mathematical models and to the real world.

Chapter 2

Smooth Curves: Differential Equation Modeling

This first model is based entirely around mathematical ideas and concepts—it is simple, but in many cases effective.

The Example: Birds

In investigating these population models, I will be looking at, as an example, two different types of birds: green birds, which are our initial focus, and red birds. These two species will stand in for practically any two similar but competing species in the real world.

These birds will have different quantities and qualities depending on what we want them to do and represent. We will denote the population of green boids by the function over time g, while we will denote the population of red birds by the function r.

Now, onward into the world of modeling.

Constructing an Equation

Simply put, when a population is bigger, it will (barring any growth restrictions like food shortages or environment size) grow *faster*. The rate of change will grow in proportion to this population size.

Having a bigger population developing even faster certainly does make intuitive sense: There would be lot more birds to make babies together, and as the number of birds making babies grows, the growth rate would continue to rise.

Thus, the easiest way to express this relation is with this differential equation:

$$\frac{dg}{dt} = \epsilon g$$

where g is our function of green bird population, $\frac{dg}{dt}$ represents the change in green bird population over time, and ϵ (which must be greater than 0, else the growth rate would shrink as the population size rises, which is contrary to our assumptions) denotes the *intrinsic growth rate*

Definition 1 *Intrinsic Growth Rate: a number which describes the linear proportion between the population size and the population growth rate, taking into account no other factors.*

The basic model assumes the proportion is linear because that is the easiest model to use and describe, and it generally describes population growth well.

Exponential Growth

We know that a differential equation including such a linear relation between population size and growth rate only has one solution: $g = g_0 e^{\epsilon t}$, where g_0 is the initial population size.

This is known as the exponential equation, so we say that a population of our green birds, at least within this model, grows exponentially—given these ideal conditions.

So let us look at situations where those conditions are less than ideal.

Environmental Threshold

As I said, a bird population growing exponentially forever would not make any sense in the context of the real world. There is not enough space for all those birds, and there definitely are not enough resources.

Thus, at some point, these green bird population will reach what we call the *environmental threshold*.

Definition 2 *Environmental Threshold: a number that describes the upper limit of a growth rate given environmental constraints like size and resource availability.*

Essentially, the threshold is a measure of how much a population will interfere with its own growth—resources become scarcer relative to the number of birds, and the population growth ceases to be exponential. This effect is felt more heavily the larger the population grows.

So how can we express that in our differential equation?

We add more expressions.

We know this slowing of population growth happens proportionally to the size of the population, so our environmental threshold must be related to the current population size.

Let us first take our original equation and replace the coefficient with a function $h(g)$ instead, so: $\frac{dg}{dt} = h(g)g$.

When the population is small enough, $h(g)$ is approximately our original constant ϵ, but it must also decrease as the population grows. We will introduce another coefficient, σ, to describe how the growth rate changes based (once again, linearly) on population.

We end up with: $h(g) = \epsilon - \sigma g$, making our expression

$$\frac{dg}{dt} = g(\epsilon - \sigma g)$$

which is also known as *logistic growth*—the rate increases steadily as it goes along, much like exponential growth, before gradually leveling off.

These values for a intrinsic growth rate and environmental threshold must be found experimentally, through observation and data collection (which we will examine more later on).

Equilibrium

As I said, a logistic growth curve eventually approaches some level point and stays there. This is known as *equilibrium*.

Definition 3 *Equilibrium: a population size that causes the population growth rate to be* 0.

This point can easily be found in our differential equation: wherever $h(g)$ is 0, that population size rests at equilibrium, as this means that the rate of change is also 0.

(As a note, the derivative is 0 when population is 0, too. This makes sense; with no population there is no capacity for growth. So, population zero is another place of equilibrium. It is simply not a very interesting one).

On the graph of a curve of population versus population growth, these are known as *critical points*—x-axis intersections. In a sense, the rate of change "wants" to be 0, so it will "tend" towards the equilibrium population, where it will then have no reason to change.

From our equation $\frac{dg}{dt} = g(\epsilon - \sigma g)$, we can see that when $g = \frac{\sigma}{\epsilon}$:

$$\epsilon - \sigma \frac{\epsilon}{\sigma} = 0$$

so $\frac{dg}{dt} = 0$, so point $g = \frac{\epsilon}{\sigma}$ is a second (non-trivial and more interesting) critical point.

A population larger than equilibrium would have a negative growth rate (the environmental threshold overcoming the intrinsic growth—the population size beyond equilibrium is not sustainable by the environment), and anything smaller would have a positive growth rate, before settling at our critical point. On a more conventional time versus population graph, this is our asymptote—in real world terms, leveling out.

Competition

Now, to complicate the model further, we introduce the second species of birds: The red birds, population r. In the real world, this could represent the sudden introduction of a new species to one environment, or comparing multiple environments containing different types and numbers of types of birds.

Our approach in constructing the newly complicated equation is remarkably similar to that of adding the environmental threshold. The greater the red bird population, the smaller the green bird population growth can be, as the red birds are taking up space and consuming resources.

(All of this can apply just as well the other way, with red bird population growth being affected by green bird population).

Thus, let us introduce a third variable into all of this: α. This will represent the *competition coefficient*.

Definition 4 *Competition Coefficient: a number that describes the limitations of one population that is coexisting and fighting for the same resources with a separate population.*

Thus, $\epsilon - \sigma g - \alpha r$ becomes the new $h(g)$ of this situation, so:

$$\frac{dg}{dt} = g(\epsilon - \sigma g - \alpha r)$$

is our new differential equation.

The red bird population can also be described with their own equation, with an intrinsic growth rate, environmental threshold, and competition coefficient based on the presence of green birds; these values also depend entirely on the properties of these red birds, and, depending on factors describing the species, can vary wildly from the green bird species variables.

Examining Competition

With the presence of the red birds, the equilibrium state of green birds is going to be even lower than before—provided, of course, the equilibrium state is not simply $r = 0$, in which case the green birds will proceed as they would if they had been a single population all along.

In cases of competition, extinction is very possible. In fact, this can vary depending on what the initial conditions are: Sometimes extinction is practically inevitable.

Say the green birds are a lot more efficient at surviving than red birds. Maybe they require fewer resources to give birth, or red birds do no live as long. If the green birds start with a

higher population, it is very likely that they will drive all of the red birds out—be that out of the given environment, or out of life entirely.

It is possible, however, that if the red birds start with a high enough population, there will generally be enough of them to survive and achieve a different equilibrium with a smaller population of green birds, even though the green birds may have been more effective.

This depends on the interactions between the expressions σg and αr. If σg is big enough that the environmental threshold alone brings about stability, the red birds might be squeezed out, but if αr is big enough to make an impact, then the red birds may remain at some level.

Thus, we can construct a different sort of graph—green birds versus red birds. In the prior example, this graph would have multiple critical points, and the graph would tend towards separate outcomes depending on what the starting point was.

(Again, a critical point that always exists is $(0, 0)$—a stable population always exists at the point when nothing is alive, which once again proves itself monumentally dull).

There could also exist situations where red birds could actually flourish in the presence of green birds (in the case of a parasite or some such creature), but for now we will focus on two birds competing over the same outside resources.

Calculations in Theory

In setting up the systems of equations for the competition model, the issue in trying to apply the model now becomes: How does one actually solve such a system? Often the differential equations become non-linear systems, which are much more difficult to solve than one would like.

To circumvent this, we focus on a critical point, and reduce our field of view to the part of the system that contains only that critical point and no others. This is then termed an *isolated critical point*, and we use a technique known as *local linearization* to approximate it as a much more easily calculated linear system.

The Jacobian

Take two differential equations $x' = f(x, y)$ and $y' = g(x, y)$. Let (a, b) be an equilibrium point, so that $f(a, b) = 0$ and $g(a, b) = 0$. To linearize the system, we want to find the matrix A that solves the equation

$$\mathbf{z}' = A\mathbf{z}$$

where $\mathbf{z}$ is the vector

$$\begin{bmatrix} x \\ y \end{bmatrix}$$

and $\mathbf{z}'$ is the vector

$$\begin{bmatrix} x' \\ y' \end{bmatrix}$$

To find the matrix A we use to linearize a system (remember, we are considering only the space around the point (a, b) such that the non-linear system would approximate a linear one), we take multiple partial derivatives and develop what is known as The Jacobian Matrix:

$$\begin{bmatrix} f_x(a,b) & g_x(a,b) \\ f_y(a,b) & g_y(a,b) \end{bmatrix}$$

We can then take this matrix and plug in the various critical points to get multiple matrices A for each critical point vector $\mathbf{z}$, for which we can then find eigenvalues.

Eigenvalues

If you will recall some linear algebra, eigenvalues and vectors are found by solving the equation $A\mathbf{v} = \lambda\mathbf{v}$, where $\mathbf{v}$ is the eigenvector and λ is the eigenvalue. Eigenvalues in this scenario can be used as a tool to identify the type of equilibrium point we are working with in the given system.

The solutions to this are only non-trivial when $|A - \lambda I| = 0$ (where I is the identity matrix). We start by defining

$$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

so we see that

$$A - I\lambda = \begin{bmatrix} a - \lambda & b \\ c & d - \lambda \end{bmatrix}$$

and thus we can calculate that the magnitude $|A - I\lambda| = 0 = ad - a\lambda - d\lambda + \lambda^2 - cb$.

We define the $Tr(A)$ as the trace of A which is the sum of the diagonal, $a + d$, and the $det(A)$ as the determinant of A which is, in general form, $ad - bc$. Thus, we can simplify this to $\lambda^2 - Tr(A)\lambda + det(A) = 0$.

This can be solved by the quadratic formula to become

$$\lambda = \frac{Tr(A) \pm \sqrt{\Delta}}{2}$$

where $\Delta = (Tr(A))^2 - 4det(A)$ is the ***discriminant***, which can be easily used to determine how many and what sort of eigenvalues there are.

Types of Equilibrium Points

Let us investigate exactly what types of equilibrium points there are, connected to the eigenvalues. This will at first be based on the trace, determinant, and discriminant Δ. This will be a textual representation of the points; for a more graphical look, see the Poincare diagram below. This diagram represents points in a Trace vs. Determinant plane.

For the first cases, let us assume that Δ is positive. (On the diagram, this is the area underneath the parabolic curve). Here, there are two real eigenvalues.

If the determinant is negative, the equilibrium point is always a saddle point: The slope vectors point into the point in two directions, and out in all others. Two positive eigenvalues.

If the determinant is zero and the trace is negative, there is a line of points of stability: All vectors approach the line at various points. One negative, one zero eigenvalue. If the trace is positive, it is the opposite: a line of unstable fixed points, where vectors are repelled from points on the line. One positive, one zero eigenvalue.

If the determinant is positive and the trace is negative, then the point is a sink: Every vector runs toward the point from every direction. Two negative eigenvalues. If the trace is positive, then the point is a source: Every vector points away from the point in every direction. One positive, one negative eigenvalue.

Now let us consider the situations where Δ is 0. (On the Poincare diagram, this is represented with the parabolic curve, where $det(A) = \frac{1}{4}(Tr(A))^2$). Here there is one eigenvalue.

If the trace is negative, the point is still a sink, but acts slightly differently: Vectors do not run directly to the point, but curve in more sharply, like a pinwheel. This is a degenerate sink. Single negative eigenvalue. The same is true if the trace is positive, being a degenerate source: The vectors pinwheel out. Single positive eigenvalue.

If both the determinant and trace are zero, nothing moves. This is uniform motion. Single zero eigenvalue.

Now to where Δ is negative. This is where there is where we have two imaginary eigenvalues. Here the determinant is always positive. (On the diagram, this is the area above the parabolic curve). Here there are two imaginary eigenvalues.

If the trace is negative, the point is a spiral sink: The vectors spiral in to the point and do not escape. Complex eigenvalues with a negative real component. If the trace is positive, the point is a spiral source: The vectors spiral out away from the point. Complex eigenvalues with a positive real component. If the trace is zero, the point is a center: Vectors make circles around the point, but do not approach. Complex eigenvalues, no real component.

For a more visually accessible summary table of the relationship between eigenvalues and equilibrium points, please see the appendices.

I will be going through a numerical example near the end of this paper pertaining to the data gathered in agent-based modeling simulations.

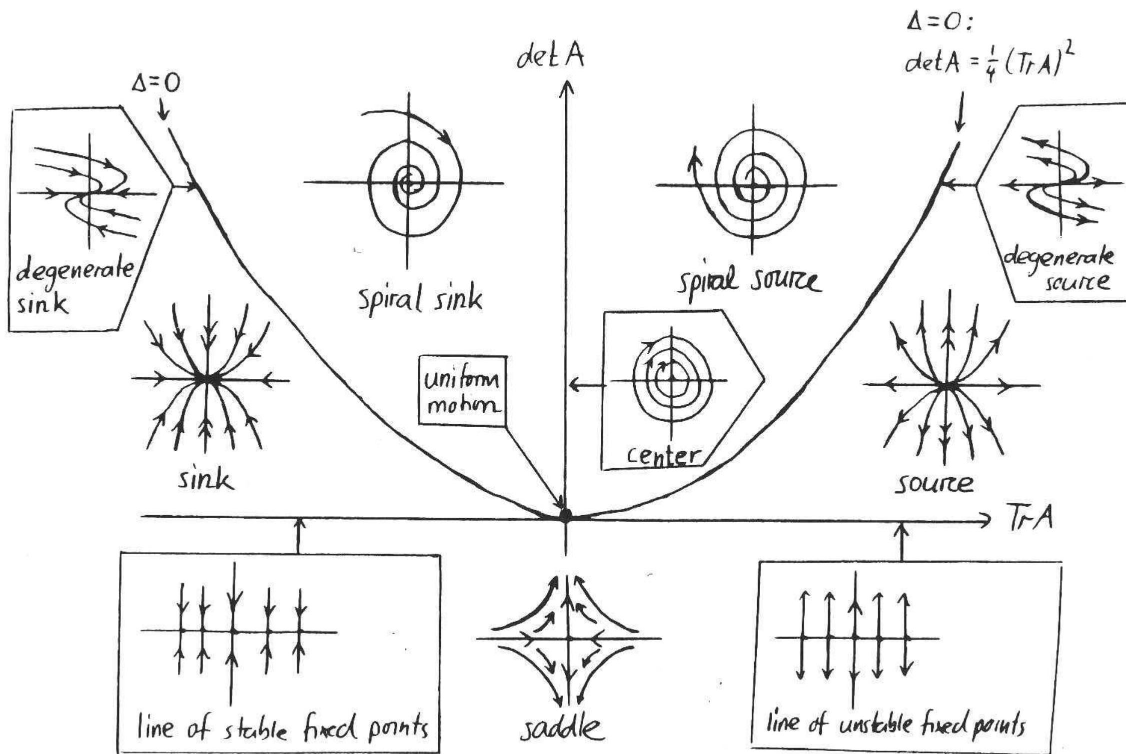


Figure 1: Poincare Diagram—“Phase Portraits” being Equilibrium Points

Chapter 3

The Digital World: Agent-Based Modeling

The differential equations model is as its base a model based on mathematical curves—but populations are not represented perfectly by smooth curves in relation to time. They are composed of individuals, and change comes from individual discrete temporal events: Births and deaths.

An *Agent-Based Model* is computer-based model meant as an alternative model of population change.

Definition 5 *Agent-Based Model: a model in which autonomous agents are given individual programming and allowed to operate independently within a digital environment.*

Agents are not directed by any outside forces; they are fed stimuli and react in certain ways depending on how they are programmed. Thus, populations are composed of individuals acting independently, which is a more complicated system to figure than the differential equations model, less smooth, and potentially more accurate.

An Object

I constructed my agent-based models in a language called Processing, which is an object-based language.

Definition 6 *Object: a digital construct within a simulation that has both data and functionality.*

Data is a any value or string of characters that an object can store to define its individual identity and behavior. *Functionality* is any function or process an object can run to enact that behavior and any motion.

Each object is defined by a *class*, which acts as a mold for each object—for example, a class would define the shape of an object, and how long it takes until it reproduces, etcetera. But

each individual object contains data that tells it where it is, how fast it is currently moving, how many resources it has consumed, and so on.

The passage of time in Processing is determined in *frames*—one iteration through the running draw program is one frame.

In the context of Agent-Based Modeling, the object is the agent. The agents are given tendencies and values, then are released into the environment to operate as they will.

In this case, our objects of choice are known as *boids*.

Definition 7 *Boid: an artificial bird-like object shaped liked an arrowhead that wants to seek food and avoid other boids to varying determined degrees.*

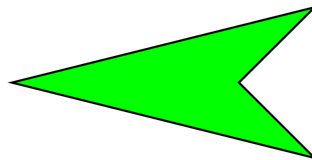


Figure 2: A green boid, facing left

The word "boid" is evidently a corruption of the word "bird," so our examples now become green boids and red boids—their simple arrowhead shape represents how they are simplified bird-like agents, instead of real complex birds (and, for modeling purposes, that is okay. It would take a lot more computing power to draw a lot of realistic birds).

Foodstuffs

There is another object present in the simulations: The Foodstuffs. These are not agents, however, as they cannot move or take any actions. They are a representation in the area of resources that the boids want and need to consume.

Figure 3: A foodstuff

These foodstuffs are generated randomly: Whenever one is eaten, another is generated elsewhere in the environment.

The Life of the Boid

As a note, the initial placement of boids is also random. This randomness eliminates any odd properties from predetermined placement, and any trends caused by some particular random placement can be eliminated by running multiple trials if one is so inclined—and you really can, as I have included my entire boid code in the appendix.

In this simulation, the only action a boid can take is changing velocity—its direction and its speed. We therefore give each boid two functions: Seek and Separate.

Separate

The boids want to separate from each other, which I first introduced as a way to keep the boids from overlapping completely, as that was unsightly and unrealistic (nothing can exist in the same place at the same time as something else).

Later this became a way of creating "flocking." If boids want to avoid other species of boids more than they want to avoid their own species, they will tend to stick together in flocks—much like birds in the real world tend to.

This flocking is an example of *emergent behaviors*, that is, patterns that were never specifically programmed to happen that nevertheless emerge. This can be an important aspect of agent-based modeling. Changing one variable might affect other aspects of the simulation about which one would not have thought.

A different flocking function could be written that would allow the boids to seek other similar boids as well, but this was unnecessary for the scope of this simulation.

Seek

In regards to the first function: The boids seek the nearest foodstuff—once they get close enough (in this case, 10 pixels, though that can be bigger or smaller depending on the needs of the model and general size of boids), the food is considered consumed and is removed.

Birth and Death

Food also serves as the fuel for reproduction. Unlike most birds, boids reproduce asexually. Given some determined amount of food consumed, a boid will split into two different younger boids. Thus, birth essentially kills a boid, but the population increases by one.

Boids have two other methods of dying, apart from reproduction. Every time a boid is generated, it is given two timers: A starvation timer and an aging timer. If either timer reaches some determined time limit, the boid will die.

The starvation timer resets every time a boid eats food, but nothing resets the age timer, so if a boid has not reproduced before reaching that determined time limit, even if it has eaten some amount of food, it will disappear from the simulation, leaving nothing behind.

Boid Values

Each boid has 12 different values that describe how it lives and operates, which can be variable from species to species, but never boid-to-boid within one species. Please see the appendices for a simpler chart. Here I will describe them in more detail.

The first three of these boids values are simply color values; Red, Green, Blue. These are a visual indicator for someone watching the simulation to tell the types of boids apart, and are thus an aesthetic choice by the user (as stated, for example I primarily used green and red boids).

The next value for boids is the starting population. A greater starting population can mean greater survivability, as discussed previously—the species is not crowded out from resources. However, if the population is too big and the resources too scarce, it can crowd itself out and each boid can die of old age or starvation before any manage to eat enough to give birth.

Next we have two related values: Starvation and Lifespan. These are the two values that tell the boid when to die after not eating or since being born, respectively.

The next two values involve the separate function: selfinterference (SI) and otherinterference (OI). This is how much a boid wants to avoid other boids, in pixels. As stated, to generate flocking behaviors, OI is typically set higher than SI.

The next, foodtillbirth, tells the boids how many pieces of food it needs to eat to give birth to two new boids.

(As an aside: Connected to this variable, there is another value that describes a boid, which is its radius. This governs the size of the boid. The radius for each boid starts at some predetermined startingsize, and increases to some maxsize as it eats more food—though as it eats enough food to become that maxsize, it instead gives birth to two new boids and disappears. This change in size is another purely aesthetic choice, but it can be rather fun to generate terrifyingly gigantic boids).

The last three values, maxforce, maxspeed, and transitiontime are also interrelated: All three tell a boid how it can move. The former essentially tells the boid how well it can turn. It is a measure of agility and acceleration ability. A fast boid will miss food if it cannot turn well. The second value gives the maximum speed of a boid—a slow boid might die before it ever reaches the food. The transitiontime tells a boid how many frames it will be until it can reach that maxspeed since eating a piece of food. This subsection will explain this aspect in greater detail.

Changes in Speed Function

It did not make sense to me that a boid would always want to be going at its maxspeed. When you eat, do you immediately sprint after the next meal? I certainly do not, and nor do birds,

so boids do not either. I determined that their speed should slow considerably immediately after eating, before increasing again as hunger returns.

Previously, a boid's maxspeed was determined linearly from how long it had been since it had eaten, multiplied by some constant. As boids had no real maximum speed, it was difficult to compare between species, and it was unrealistic. No animal simply gets faster and faster, no matter how hungry it might be.

The curve with the properties I needed that was the easiest to understand and program was a simple sinusoid. This would start at zero, start increasing, increase maximally at the halfway time until it would reach max speed, and speed up more slowly until it reached a maxpoint with derivative zero.

I created a piecewise function–after the suiusoid reaches the top of its curve, it then connects to a constant maxspeed function, which avoided any jarring movement or acceleration.

This maxspeed holds until a boid eats a piece of food. The speed is then reset to 0, but immediately begins increasing sinusoidally again. The length of this sinusoidal curve is determined by transitiontime, so a boid with a higher transition time will take longer to reach its maximum speed.

As a note, this change in programming increases competition between even boids of the same type. Previously, the hungriest boid (the one in most need of food) was also necessarily the fastest, and thus the most likely to eat the food. Now, however, there is a cap on this speed—any boid past its transition time will be equally fast.

For example, if a boid has a transition time of 100 frames, and a starvation time of 1000, a boid with a hunger timer of 101 since eating will have just as much chance of getting to a piece of food as a boid with a timer of 999 (right on the cusp of starvation), depending on the relative positions. Thus, the neediest boids no longer have a better change of eating the food.

This decreases the efficiency of the boids, but makes them a little more realistic—they certainly starve more now.

Environment

The digital environment operates on three numbers: Height and width, which determine the size of the area, and numfoods—the number of the pieces of food that exist at any time. More food generally means more boids.

When I began constructing the digital area for the boids, I put up rigid walls the boids could not pass through. This always kept the boids visible (previously they would fly off the screen before then begrudgingly returning, sometimes after quite a long while).

This worked, but it caused a behavior I called "skating." Programming-wise, I stopped the boids at the walls by setting the velocity perpendicular to the impacted wall back to zero and disallowing the boid's position to change in that direction. This caused boids to bunch up on the walls as they tried to avoid each other, skating along the wall until they could find a moment to escape (generally this was when they finally got hungry enough).

So, to avoid this phenomena, I formed the boid world into a torus of sorts: When the boids reached the walls, they would be sent to the opposite wall—thus, the top and bottom wrap around to each other, forming a tube whose edges are the right and left walls, which themselves also connect, forming a torus.

This, however, presented some problems regarding the seek and separate functions: the boids became incredibly shortsighted whenever they were near one of the walls (walls which should, from their perspective, not exist).

If a boid was at the bottom of the screen and a piece of food at the top, the boid could simply fly down and get to the food quickly. But it only checked where the food was relative to the distances across the area, not through the walls, and flew all the way up instead—they could not see to the correct places on the doughnut world.

So how to make the boid seek a piece of food just beyond the wall? Or separate itself from boids that would be nearby on an actual torus, but do not appear to be so near?

Virtual Realities

First I need to talk about parallel universes.

Definition 8 *Parallel Universes: a parallel universe (PU) is a virtual replica of the objects in the area, shifted to a different nearby zone, which can be identified by the program but not seen by the user.*

The parallel universes come in grids—eight parallel universes surround the single original universe (OU).

The boids proper exist only in the center area (representing the real world), but they can sense replicas of boids and foodstuffs in the virtual realities. Thus, the boid near the bottom does not sense the food near the top, but it does sense that there is a piece of food down below the border—that this food is virtual does not affect the boid's behavior. The program does not render this food, but the boid can sense the shifted marker of foodstuff. Thus, it flies down, wraps around to the top where it then senses the real food as the nearest, thus continuing to move in the direction it was already going, and appearing as if it were heading for that real food the whole time.

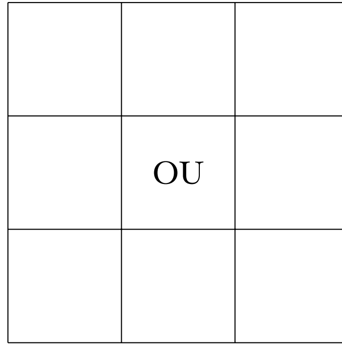


Figure 4: The parallel universe grid

This is accomplished by iterating through not only every object, but other copies of that object—a boid senses not just food in the real world, but also senses replicas that it treats as real and will pursue.

This also applies to the separate function. The boids sense each real boid and virtual copies of the boids and avoid each one as they should.

And so, boids can properly see the whole doughnut. Let us look at how this was accomplished programming-wise in a bit more detail.

Optimizations in Doughnut World

Two sections of programming changed majorly when the boids were placed on a torus.

Separate

The separate function works by iterating through every other boids' position and determining whether the distance to that position is less than the desired separation (variable depending on SI and OI). If it is, that position is added to the vector force applied to the boid at the end of the separate. This was done for all boids.

When I first put the boids on a torus, I had the program iterate through not just every boid, but every virtual boid as well—meaning the program had to do nine times as many distance calculations for each boid. When there were 50 or 100 boids, this was not a huge issue, but with 200 or 400 boids slowdown was evident, and with 1000 boids the program went at a painful crawl.

In place of this, I wrote a new function, "replicate," which is fed a position and a string of one or two letters that then returns a vector to the position of the virtual boid in the quadrant as determined by the string, as seen below.

(Capital letters make a virtual boid with a greater position value, while lower case letters send it back. It is worth it to note that greater x-values send you right, while greater y-values send you down).

xy	y	Xy
x	—	X
xY	Y	XY

Figure 5: The separate function key

This is called in the boid's separate function only when a boid close enough to a wall—within the distance of desiredseparation to the wall. If the presence of a boid could have any impact, a virtual boid is created. If the boid is near two walls, three virtual boid vectors are fed into the calculations—two past the opposite walls, and one in the appropriate diagonal quadrant.

If the boid is not near enough to any walls, then, no virtual copy is generated, which speeds up calculation immensely (unless you have a fringe case where every boid is near enough to the borders, but this seems unlikely, and would not be permanent in any case).

Thus, the program only has to calculate boids within a limited area, represented as a blue square in the figure.

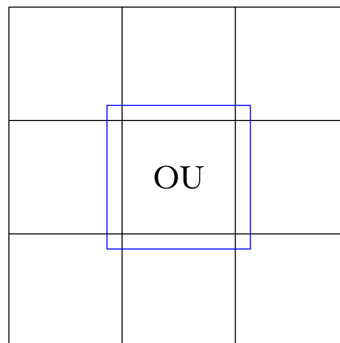


Figure 6: The calculated universe

Ignoring the rest speeds up the calculations immensely, making our system of PUs that much more efficient. Admittedly, this is not terribly important for the boids (even if every

frame took an entire day to render, they would still move forward in time), but certainly helps in real world time when attempting to run simulations with large flocks of them.

NearestFoodstuffs

For the nearestFoodstuffs function on the doughnut, I wrote a nearestToZero function, which took in three inputs and returned the one nearest to zero (a sort of absolute value without removing any negative signs).

This could then be used to calculate the nearest food, real or virtual—it found the nearest x-distance and the nearest y-distance, and then I just had to put these values into the distance formula to find the location of the nearest food: $\sqrt{(xdistance)^2 + (ydistance)^2}$.

This function iterated through every foodstuff to make sure it had found the closest one, then returned the index of the food—that is, the number the machine had internally assigned to the food on the foodstuffs list. The main program then took the index and iterated through the real food and each of its eight virtual copies for a second time to determine which was closest.

This repetition was unnecessary.

So, I made the nearestFoodstuffs function return a vector (known in Processing as a PVector). This is not a typical Calculus xy-vector, however; the relationship between the two contained values is a bit more complicated.

The first number of the vector is still the index of the food. Giving the main program the index makes it very easy to remove the correct foodstuff once the boid has consumed it.

The second number is a new value called the offset, which represents which of the nine quadrants the closest food is in, real or virtual. This allows the main program to much more easily determine the point the boid is supposed to seek.

Essentially, this offset uses a trinary system. It is a number from 0 to 8, all of which represent different quadrants. If the x-coordinate of the nearest food is to the right, the function adds 3 to the offset; if the x-coordinate is to the right, 6 is added. If the y-coordinate is down, 1 is added, and if up, 2.

The program then receives this number: If the number is greater than 5, shift the boid's target to the right; if it is not, but is greater than 2, shift to the left. If the number is 1 modulo 3, shift down, and if it is 2 modulo 3, shift up.

Thus, the program takes the location of the food indexed and shifts the boid's target accordingly, cutting down on the number of calculations necessary.

This is not as huge of an improvement as the separate function was, but it is still an optimization.

8	2	5
6	0	3
7	1	4

Figure 7: The trinary system of food

The Presentation of the Thing

The actual window where the boids fly around is bordered by two bars (actually just shapes, which help cover up the boids as they are teleported across the area when they wrap around), which contain real time information about the total number of boids alive, starved, aged to death, and births, followed by the individual numbers for each species.

This is accomplished by initiating running totals of each of these values for each boid type, and for the total number dead of either type of death, and born. These are then printed on the side. The bottom left corner shows the total number of frames, and is constantly increasing.

These counters also help to print out data to a spreadsheet, once the simulation ends. The data points are printed out every 100 frames, and put into separate cells of the spreadsheet by separating the print out with \u0009, which is Unicode for the tab key. These spreadsheets can then be analyzed and generate graphs and other data sets—which we can then do proper hard science with as we shall soon see.

What Could Have Been

As much as I love these boids, and as effective as they might be in forming a potentially predictive model, there are a couple improvements that could still be made.

First, there is a certain threshold of number of boids (somewhere around 1500 or 2000, depending on boid type) where every piece of food is eaten every frame or nearly every frame. As consuming a piece of food immediately causes another to be generated, these means every frame there is more food.

This is too much food.

This is reflected in the number of boids. The populations grow immensely and show little to no sign of stopping, though I cannot truly test that hypothesis—my computer is not pow-

erful enough to run that simulation with more than 2000 boids at anything faster than one frame every six seconds, which is just not feasible.

Secondly, also involving food: Boids automatically head towards the nearest piece of food, no matter how far away that piece of food might be. It could be more realistic for the boid to head towards the piece of food if it is within some range of sight.

Boids that always know exactly where the food is, no matter how far away it is, act less like birds hunting prey and more like sharks smelling blood in the water.

(So, shoiks?)

Chapter 4

Spreadsheets and Graphs: Boid Data Analysis

In this section, we will set up experiments and analyze the data that this agent-based model gives us.

What can we know about these boids?

Conversions

Our goal here with this agent-based model is to use it to generate data that can then be compared to the differential equations model. To do so easily and effectively, I will use this temporal conversion:

Definition 9 *Frame to Year Conversion:* 100 *Frames* = 1 *year.*

This comes from the fact that the program generates a data point in our spreadsheets every 100 frames, so this seems a reasonable and useful comparison. It at least makes things easier to talk about.

The Control Boid

To more directly use the boids for testing, I had to establish a control boid and control environment to measure all other boids by. This required some guess and check to create a boid that would be easy to use and give me numbers that could be easily measured. If stable population was too small, for example, any deviation seemed huge.

I started with establishing reasonable lengths of time for starvation and lifespan. To ensure speedy tests, I wanted these lengths to be fairly short, and the most important thing was comparison: A boid with a shorter lifespan than starvation would be unable to starve. So, 5 years seemed a reasonable starvation rate, with lifespan set at four times that, at 20 years.

The other temporal variable is transition time, the time it takes a boid to get to its maximum speed. As I was not planning on varying this amount very much in my example tests, I set it fairly low, to one fifth of starvation: 1 year.

The consumption of resources, Foodtillbirth, was also rather arbitrary; 9 pieces of food until birth seemed a healthy but not insurmountable amount, and gave me room enough to play around with it (increasing it and decreasing it) in my tests.

The variables left were spatial variables. I could never figure out a practical real-world value for what space could represent, so everything else—maximum speed, maximum force, and the two separation variables—were determined through running simulations and seeing what worked.

(It is also important to note that there is absolutely no area size or distance involved in the differential equation model—thus, there is no constraint for me to determine a comparison to anything in particular).

Self and other interference are related—OI has to be greater than SI to generate flocking behaviors. A desired pixel length of 20 from like boids let each species cluster together without overlapping all that much, which was aesthetically pleasing, and assigning a value approximately four times that length to the distance kept away from other boids was enough to noticeably create flocking. Thus, the values became 20 and 80.

Maxspeed and maxforce had to be related somehow as well; if speed increases but force does not, the boids become incompetent, flying right past the food without eating it as they are not agile enough to actually reach it. If force increases too much, however, the boids fly erratically, with no regard to previous motion. They also gain an uncomfortable jitter, as they are able to oscillate wildly back and forth in their quest to separate themselves from other boids. Thus, through some trial and error, a ratio of 30 to 1, maxspeed to maxforce, would be my baseline, leaving the control boid starting at 6 and .2.

For the control environment, my area had been 600 pixels by 600 pixels for a long while; it seemed a good dimension, so I saw no reason to change it.

When first testing this boid, the equilibrium seemed to be around 6 to 14 boids. These are not big numbers. I upped the food count from 2 to 10 and the populations grew considerably, to around 100, which was much more workable.

From there, it was simply a matter of testing the control boid, putting a second boid species in, and further altering some variables to see how it all interacted and what data I could generate.

I also limited each test to 200 years, simply for real-life time reasons—boid populations tended to attain stability long before that time.

Stable Population

In order to compare the boids to the differential equation model, I needed some value to compare to that model's equilibrium or critical points.

Definition 10 *Stable Population: a stable population, or equilibrium population, is the average of a range of boid populations that continues for a period of time such that it shows no signs of moving significantly (more than* 20 *boids) out of that range.*

As the food is produced randomly, there is an element of randomness when running these simulations. In one test, the average stable population of a control boid may appear to be 102, while in another it is 110, for example. This can come down simply to lack of luck, or abundance of it, on the part of the boids. If food is generated close enough to eat, then that works out well—for that specific boid (not so much for any others).

The populations also tend to oscillate somewhat, instead of reaching some definite population and forever remaining there—these are ever-changing systems. This must be kept in mind; as much as possible I used my control of 20 boids, but on more than one occasion the exact point this became true was based off of some amount of visual approximation.

Actual Data

Now we move on to what I have been promising: Actual data gathered from boid simulations!

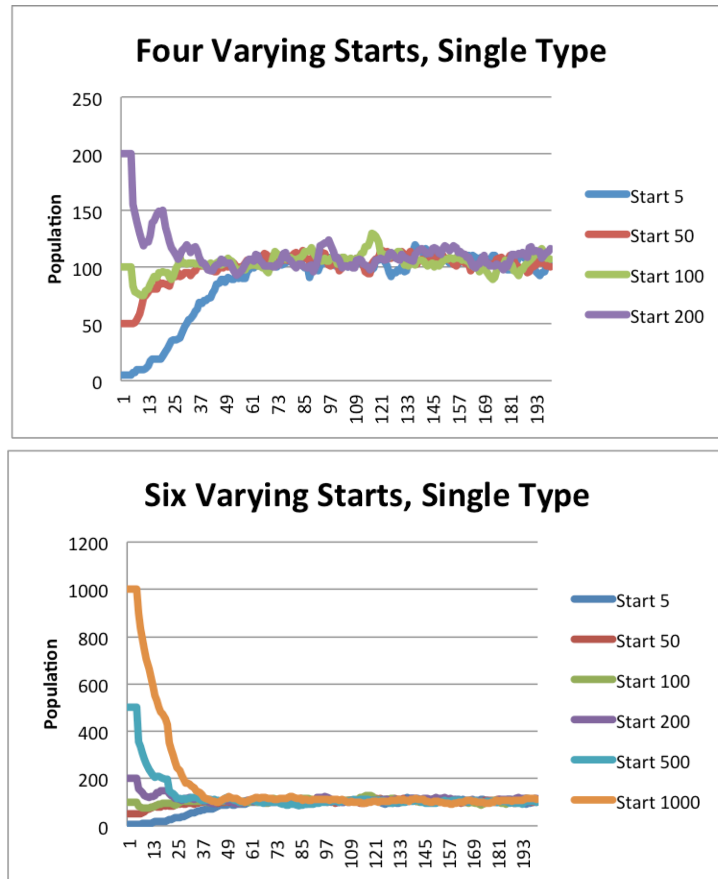
All approximations of boid populations are rounded to the nearest whole number—with the relative impreciseness of this model versus the more mathematical curves, two significant figures seemed appropriate. But we will take what we have and see what we can find.

Test 1: Single Population, Variable Start

For the first test, I had to look at the simplest simulation: One population of control boids (which obviously had to be green). I placed them into the environment with varying starting population values, and saw what could happen.

I tested them starting at population 5, 50, 100, 200, 500, and 1000—smaller numbers, then approximately the stable population, then higher numbers.

I created two graphs: One only including the first four tests to it is easier to see, and one including all six tests including the larger numbers. Here they are:



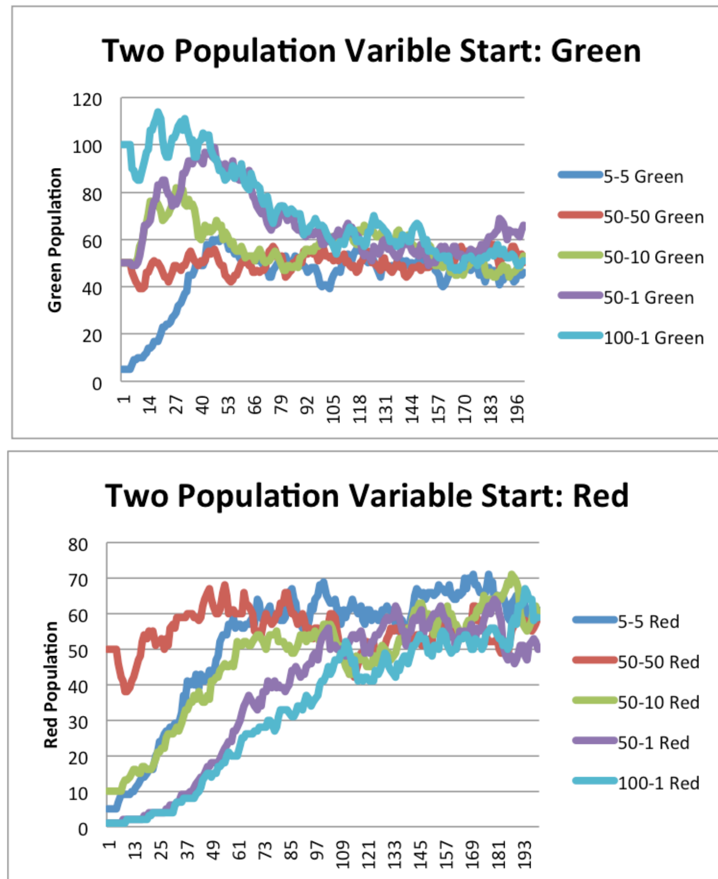
It is easy to see, especially in the second graph, that no matter the starting population, it always tends towards a certain range—in this case, each boid species achieved a definite stable population around year 77, which generated averages of 104, 106, 107, 108, 101, 106, with a total mean of 105.

Thus, the stable population of our control boid is approximately 105.

Test 2: Two Populations, Variable Starts

For my second test, I added a second population of boids—the red boids arrived. Here I tested variable starting population numbers, to see if two groups of boids would also even out to approximately the same numbers given different starting points.

I tested, going green-red, the numbers 5-5, 50-50, 50-5, 50-1, 100-1. Here are the graphs of each boid species separately:



Here 114 years was the definite stable point for all populations. The average population for green are 48, 51, 54, 57, 57, with a total average of 53—all approximately the same.

For red the stable populations are approximately 63, 53, 57, 55, 51, with a total average of 56—again, all approximately the same, and additionally quite similar to green's. Thus, no matter what the starting points, these two identical boid species are going to come to nearly enough the same equilibrium population.

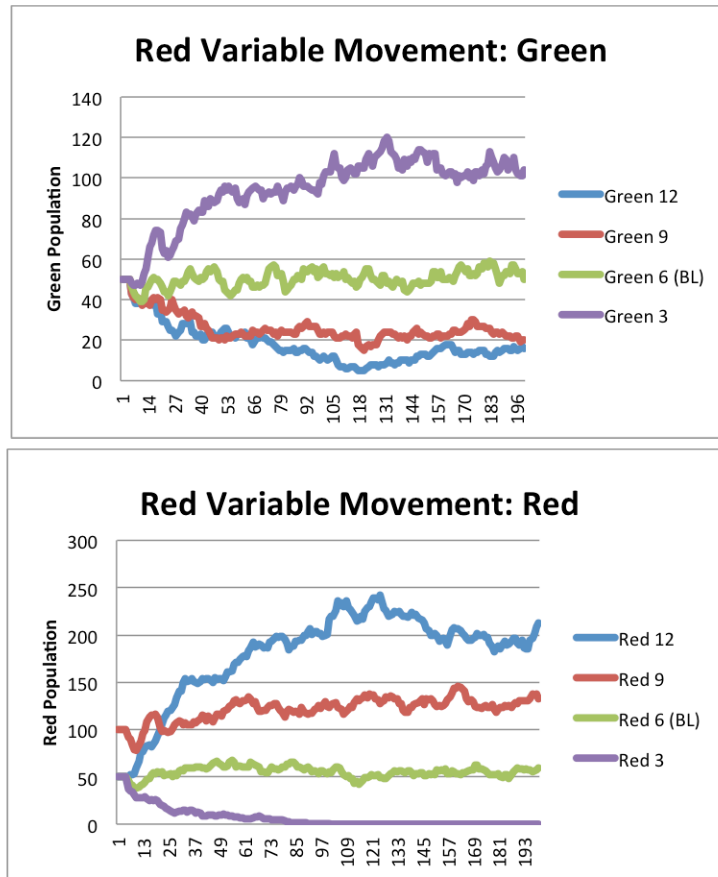
Test 3: Movement Tests

For my third test, I set the control boid start to 50, and kept that constant.

I varied the speed and force of the red boids, keeping the ratio between speed and force constant and varying them within that framework. Along with the baseline of maxspeed = 6.0 pixels/frame, I also tested 3.0, 9.0, and 12.0.

I attempted to test 24.0, but the red boids expanded too rapidly, reaching a population of 2000 within 10 years. I had to shut that one down.

Otherwise, here are the results:



Here we can see that increasing the speed gives the boids a drastic improvement in stable population numbers. Most obviously this can be seen in the slower boid: At speed 3.0, they were wiped out completely by the speed 6.0 green boids.

Otherwise, 99 years was definitely stable point:

At speed 9.0, red boids had a stable population of 209 (while green boids had a population of 12—poor show, green).

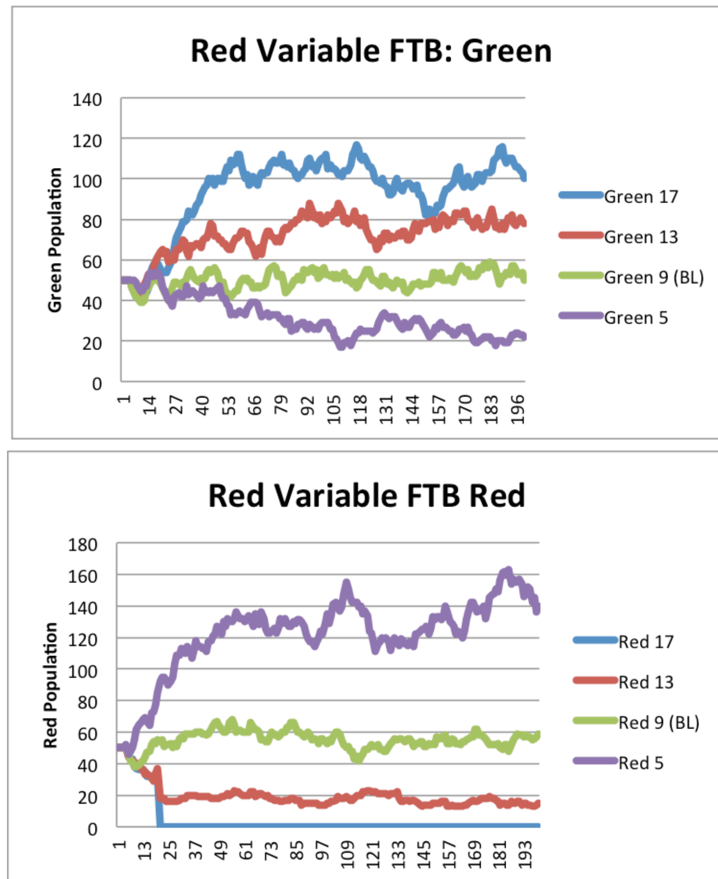
At speed 12.0, red boids had a stable population of 129 (green boids had 23—better).

And at speed 6.0, as we have already seen, the boids have equal populations, somewhere around 55.

Test 4: Food Till Birth Tests

For this test, I kept all things constant except foodtillbirth. Along with the baseline of 9, I tested intervals of 4: the red boids were given FTB values of 17, 13, 5, and 1. FTB 1, however, once again blew the population up over 2000 and I had to dump the whole mess into the digital ether.

And here once again, the graphs:



Here, a lower FTB greatly increases a stable population. Once again, we have another extreme result: AT FTB 17, the red boids were wiped out by the FTB 9 green boids.

The other average stable populations, after year 80:

At FTB 13, the red boids had a stable population of 17 (green boids had 78).

At FTB 5, red boids had a stable population of 133 (green boids had 25).

And the baseline once again is approximately the same.

The Possibility of Predicting Populations

If I were to run more trials testing every variable, I could potentially create a mathematical way to predict what a certain boid species' stable population would be, given every value, possible presence of other species, and the environmental state.

With the number of variables involved, however, and the approximating that would have been necessary, this proved to be beyond the scope of this project. This makes sense, given the fact that the purpose of the agent-based model is to more easily represent detailed simulations that would be difficult to compute accurately.

Chapter 5

Models and Reality: Interrelations

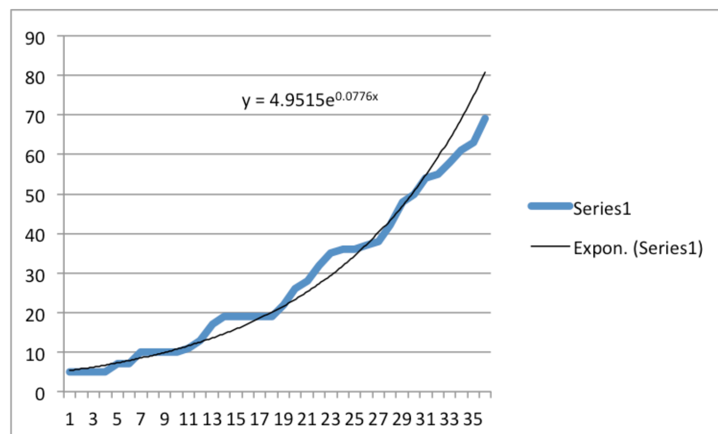
After studying the mathematics and examining the generated data, it is time to put it all together and see how the agent-based model can compare to both the differential equations model and the real world.

To the Math

Now that I have all of the stable populations in each of these cases, I need only find the intrinsic growth rate to be able to calculate everything for each of a given model to represent it with differential equations.

As the stable population represents the place where the derivative is 0, I can first plug numbers into equations for a single population to find the environmental threshold ($0 = \epsilon - \sigma g$), and following this I can take the numbers for each of the two population models and find the competition coefficient ($0 = \epsilon - \sigma g - \alpha r$)—provided I can use an exponential model on the lower ranges of the single population test to find that initial ϵ.

Luckily, using those spreadsheets it can easily be shown that the start 5 population growth from year 2 to year 35 is approximately exponential, as demonstrated by this graph:



(To explain the year range: The first five years skew the graph a bit—no boid has reached their starved point at that time yet, so there is no death, and there is also little to no growth, as each boid starts as an unfed baby. So year one can be quietly ignored, as year two gives a better approximate curve and better starting point. This means that for the population size itself based on time, we are actually dealing with times $t-1$, but for our differential equation, in which is not directly a variable, this still serves as a good approximation).

Given that we are attempting to approximate

$$\frac{dg}{dt} = \epsilon g$$

we rewrite it as

$$g = g_0 e^{\epsilon t}$$

where g_0 is, as before the starting population of $g(0) = 5$.

Using the excel graph and functions shown above, the approximate equation we get is

$$g = 4.9515e^{0.0776t}$$

which, while slightly different due to approximations, gives us an initial value very near to our wanted g_0, so we can comfortably use this ϵ value of 0.0776 to calculate a differential equations model!

Since we found the stable population of a single type of boid to be 105, we can see from $0 = 0.0776 - 105(\sigma)$ that the environmental threshold is $\sigma = .000759$. Further using that, we see that the stable population of two species of identical boids is about 55 each, so $0 = .0776 - 55(.000759) - 55(\alpha)$, so our competition coefficient $\alpha = .000652$. Thus, our differential equations for green boids and red boids are:

$$\frac{dg}{dt} = g(0.0776 - 0.000759g - 0.000652r)$$
$$\frac{dr}{dt} = r(0.0776 - 0.000759r - 0.000652g)$$

Return of the Jacobian

Returning to our discussion of the Jacobian matrix and eigenvalues, we define functions G and R with $g' = G(g, r)$ and $r' = R(g, r)$, where our equilibrium point is $g = r = 55$, or $(55, 55)$.

The partial derivative $G_g = -0.001518g - 0.000652r + 0.0776$, while the partial derivative $G_r = -0.000652g$. The two partial derivatives for R are the same with g and r swapped.

Thus, we have our matrix A:

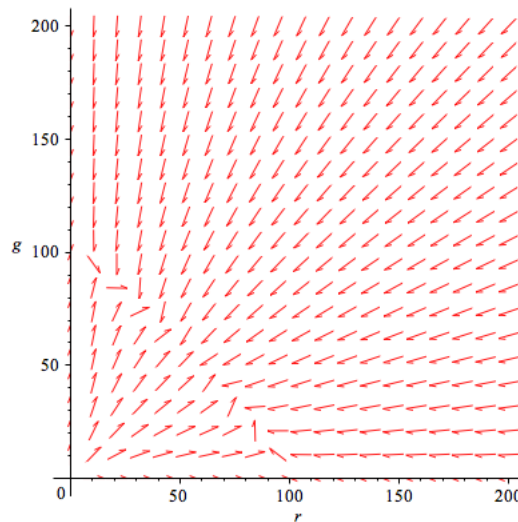
$$\begin{bmatrix} G_g(55,55) & R_g(55,55) \\ G_r(55,55) & R_r(55,55) \end{bmatrix} = \begin{bmatrix} -0.04175 & -0.04175 \\ -0.03586 & -0.03586 \end{bmatrix}$$

With this, we see that $Tr(A) = -0.07736$, $det(A) = 0$, where $\Delta = .005985$. Thus, our eigenvalues are -0.1547 and 0: One negative and one zero eigenvalue. Thus, we would predict a line of stability around one equilibrium point, which is consistent with our graph, as we shall shortly see. We could repeat this with any other equilibrium points.

And that is the basis for a predictive mathematical model. If we wanted a better approximation for our values, we would run several different trials to generate more data, which we would then statistically evaluate more thoroughly.

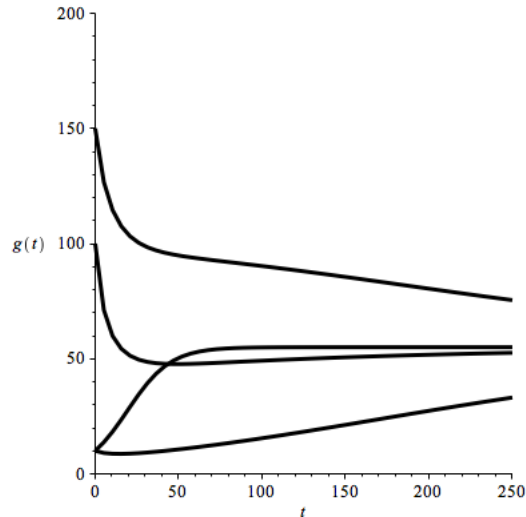
A Differential Equations Graph Party

So to check over and compare, here are some generated graphs of the differential equations based on these values, for two congruent species:



This is a graph of population r versus population g. Here, we can clearly see the arrows as the graph tends to move towards the line of stability and the equilibrium point directly in the middle; in this case, with any amount of red boids and green boids, that amount is approximately 55 boids for each population. However, we also see on the edges that if one population is the sole type of boid in the environment (so, $r = 0$ or $g = 0$) the amount of that sole species would tend to be around 100, also as predicted.

(As well, if both populations are 0, there are no tendencies to go anywhere else, because there are no boids. This is also as stated before, and as stated before it is possibly the least interesting part of the graph: It is not going anywhere at all).



This second graph compares the population of green boids to time.

The topmost line represent starting conditions of $g(0) = 150$ and $r(t) = 10$.

The line that starts second from the top represents $g(0) = 100$ and $r(0) = 150$.

The third line represents $g(0) = 10$ and $r(0) = 10$.

The bottom line represents $g(0) = 10$ and $r(0) = 150$.

As we can see, the equilibrium values are approximately 55 for each population, no matter the starting conditions, which makes sense—these population amounts are what we used to calculate the environmental threshold and competition values, after all.

But what is most interesting is that the equilibrium is approached much more slowly than the agent-based model demonstrated, especially when there is a large difference in starting population sizes. This can be explained with a fairly simple concept: Inefficiency.

The differential equations model assumes that populations will be as efficient as possible—the most members of the population getting the most food for the greatest amount of time. Thus, when red starts high and green starts low, or vice versa, one population will dominate for a lot longer.

For the boids, this is not at all the case. Boids are opportunistic little things, and as discussed under the change in speed function section, they are even less efficient than they used to be—many more boids starve than would in a more fair system.

In the agent-based model I have constructed, a population will approach equilibrium far more quickly in the boid world than the differential equations model, even without a lot of trans-population competition, because boids are highly individualistic and difficult—they compete heavily even with themselves.

Thus, when the smooth model is taken away and all that is left are individuals, things can get very messy indeed.

To the World

As stated, the differential equation model is not the most accurate representation of the boid world—but what matters in the end is the real world. The point of these models is not the data points or the boids, but the real life birds flying around us. How can we more accurately predict and understand them?

As demonstrated using data from the simulation, our Greek letter differential equation constants (or at least usable estimates of them) can be found with graphical curves, calculating stability, and lines of best fit. This gives us a place to start.

But the agent-based model must be tinkered with a bit more to give accurate findings.

First off, no birds I know live on a doughnut. This was a decision I made to give the boids a limitless but finite area in which to live. Birds in real life have far more physical areas to exist in, often far larger and more varied.

Thus, a better agent-based model than mine would factor in terrain and area, making it more than a blank white space.

In addition, the resources would have to have more definite sources, and would be more than just food. Birds need more than that—shelter, mates, water, and so on, would all be represented in a more complicated ABM. As touched on under the "What Could Have Been" section in chapter 3, the way the boids seek food is entirely unrealistic; birds have a much more limited scope than that.

As far as the values ascribed to the boids themselves, this would have to rely on running simulations and comparing the data generated to real world data and adjusting. Not enough boids dying of old age in relation to real birds? Up the foodtillbirth. Too many starving? Either up the resource count, or raise the starvation time—these birds are hardier than you thought.

More basic changes could also be made; for example, more new boids could be generated at birth, if the bird tends to have more young—and the older boid does not necessarily have to disappear (though this would be accurate if you were modeling something like salmon, who do tend to die upon spawning).

More generally, the life cycle could be made much more apparent (baby boids being reliant on their parents for food until a certain age, for example).

The world of agent-based modeling, while starting off as a very basic idea (let individuals go and do what they want to), can be made very complicated very quickly. In comparison to the potential for detail—and *especially* in comparison to the real world—my simulations here are barely getting started.

Chapter 6

Conclusion

The point of this paper was not to come to any grand sweeping conclusions about the nature of the real world or of the realm of mathematics, but instead serve as an introduction to what agent-based modeling might be capable of, and how we can compare it to one of the simpler models out there.

Our original red birds and green birds will never act like curves, nor will they act exactly like their respectively colored boids even as we complicate the simulations, but we can approximate and subsequently try to use those approximations to interpret and predict the world around us, and understand how we and other forces impact these living systems.

Though, according to my model, the world can be so cruel and heartless when your only goal is "eat all the food you can and avoid anyone different even more than you usually avoid things."

So, as much as I love those dopey little primary-colored arrowheads, perhaps there is a lesson in there after all.

Be smarter than a boid.

Thank you for reading.

Appendix A

Chart of Eigenvalues and Equilibrium Points

Number and Types of Eigenvalues	Type of Equilibrium Point
two positive	saddle point
one negative, one zero	line of attracting points
one positive, one zero	line of repelling points
two negative	sink
one positive, one negative	source
single negative	pinwheel sink
single positive	pinwheel source
single zero	uniform motion (zero)
two complex with negative real components	spiral sink
two complex with positive real components	spiral source
two complex with no real component	center

Appendix B

Chart of Boid Values for One Species

Variable	Definition
red	red color value
green	green color value
blue	blue color value
start	starting population
starvation	how long since eating till a boid will starve
lifespan	how long since birth till a boid will die of old age
selfinterference	how far a boid prefers to be from those of its own species
otherinterference	how far a boid prefers to be from those of other species
foodtillbirth	how many pieces of food must be eaten until birth
maxspeed	speed limit
maxforce	speed change and turning limit
transitiontime	time till a boid's maxspeed reaches maximum

Appendix C

Processing Code: The Utilized Version of Boids

This is the entirety of the version of the boid code I used to run my simulations. As I said, there are always improvements to be made and more variables to add, but it served my purposes well. It is fairly well annotated, so if you have any interest in getting further into the gritty details than this paper already has, dive right in.

```
int leftmargin = 55;
int rightmargin = 95; //for fitting text; x-value only (no y margins)
float maxsize = 5.0; //maximum size of boids; purely aesthetic.
//A bit bigger than root(2)*startingsize
//so viewer can better differentiate boids.
float startingsize = 3.0; //starting size of baby boids

int numfoods = 10; //How many foods are generated at start

int totaltime = 20000; //how many frames to run till automatic closing
//(for testing!)
String filename = "lasttests" + ".xls"; //the filename! Can be varied.
//(for testing!)

float nearestToZero(float a, float b, float c) {
  //for looking at food vs. virtual food
  //and finding which is closer
  if (min(abs(a), abs(b), abs(c)) == abs(a)) return a;
  if (min(abs(a), abs(b), abs(c)) == abs(b)) return b;
  return c;
}
```

```
enum Boid_Type { //enumerating each boid type
                 //to allow for easy access to values.
  GREEN_MACHINE(0, 200, 0, 50, 500, 2000, 20, 80, 9, .2, 6.0, 100),
    RED_FED(200, 0, 0, 50, 500, 2000, 20, 80, 9, .2, 6.0, 100),
    BLUE_CREW(0, 0, 200, 50, 500, 2000, 20, 80, 9, .2, 6.0, 100),
    ORANGE_DOORHINGE(200, 155, 0, 50, 500, 2000, 20, 80, 9, .2, 6.0, 100);

  int red;
  int green;
  int blue; //color of boid
  int start; //How many boids at the start
  int starvation; //how long till the boid will starve from hunger
  int lifespan; //the boid lifespan
  int selfinterference; //avoiding self boids
  int otherinterference; //avoiding other boids
  int foodtillbirth; //how many pieces of food to consume until birth
  float maxforce; //how well a boid can turn
  float maxspeed; //maximum speed of boid
  int transitiontime;
  //how long it takes for a boid, based on timer, to get to maxspeed
  //(to get hungry enough)

  int alive;
  int starved;
  int aged;
  int births;

  Boid_Type(int red, int green, int blue, int start,
    int starvation, int lifespan,
    int selfinterference, int otherinterference, int foodtillbirth,
    float maxforce, float maxspeed, int transitiontime) {

    this.red = red;
    this.green = green;
    this.blue = blue;
```

```
    this.start = start;
    this.starvation = starvation;
    this.lifespan = lifespan;
    this.selfinterference = selfinterference;
    this.otherinterference = otherinterference;
    this.foodtillbirth = foodtillbirth;
    this.maxforce = maxforce;
    this.maxspeed = maxspeed;
    this.transitiontime = transitiontime;

    this.alive = 0;
    this.starved = 0;
    this.aged = 0;
    this.births = 0;  //counters initialized for each type
  }
}

class Vehicle { //called to create each boid

  PVector location;
  PVector velocity;
  PVector acceleration; //for each individual boid

  float heading = 0; //used in display
                   //so the boids aren't always facing to the left
  float r; // Additional variable for size of boid
  Boid_Type type;
  int timer; //how long it has been since a boid ate (changes per frame)
  int lifetime; //how long a boid has lived (changes per frame)
  int eaten; //How many pieces of food a boid has eaten

  float speed() {//Replaced simpler maxspeed.
    if (timer > type.transitiontime) return type.maxspeed;
    else return type.maxspeed/2*(1-cos(timer*PI/type.transitiontime));
  }
```

```
Vehicle(float x, float y, Boid_Type type) {
  //the object constructor itself
  r = startingsize;
  acceleration = new PVector(0, 0);
  velocity = new PVector(0, 0);
  location = new PVector(x, y);
  this.type = type;
  timer = 0;
  lifetime = 0;
  eaten = 0;
  type.alive++;
}

void update() { //called each frame to describe the boid
  separate(boids); //Where Separate is called
  velocity.add(acceleration);
  velocity.limit(speed());
  location.add(velocity);
  if (location.x < leftmargin) { //loop when it reaches the margins
    location.x = width - rightmargin;
  }
  if (location.x > width - rightmargin) { //loop it around
    location.x = leftmargin;
  }
  if (location.y < 0) { //more loops
    location.y = height;
  }
  if (location.y > height) { //loops!
    location.y = 0;
  }
  acceleration.mult(0); //reset the acceleration each time
}

void applyForce(PVector force) { // Newton's second law, to move the boid
  acceleration.add(force);
}
```

```
void seek(PVector target) {   // To seek!
  PVector desired = PVector.sub(target, location);
  desired.normalize();
  desired.mult(speed());
  PVector steer = PVector.sub(desired, velocity);
  steer.limit(type.maxforce);
  applyForce(steer);
}

PVector replicate(float positionx, float positiony, String placement) {
  //to generate virtual boid vectors that separate can use.
  //capital letters make virtual boids at greater points
  PVector virtual_boid = new PVector(0, 0);
  if (placement == "X") {
    virtual_boid.x = positionx + width - rightmargin - leftmargin;
  } else if (placement == "x") {
    virtual_boid.x = positionx - width + rightmargin + leftmargin;
  } else if (placement == "Y") {
    virtual_boid.y = positiony + height;
  } else if (placement == "y") {
    virtual_boid.y = positiony - height;
  } else if (placement == "XY") {
    virtual_boid.x = positionx + width - rightmargin - leftmargin;
    virtual_boid.y = positiony + height;
  } else if (placement == "Xy") {
    virtual_boid.x = positionx + width - rightmargin - leftmargin;
    virtual_boid.y = positiony - height;
  } else if (placement == "xY") {
    virtual_boid.x = positionx - width + rightmargin + leftmargin;
    virtual_boid.y = positiony + height;
  } else if (placement == "xy") {
    virtual_boid.x = positionx - width + rightmargin + leftmargin;
    virtual_boid.y = positiony - height;
  }
  return virtual_boid;
```

```
}

void separate(ArrayList<Vehicle> boids) {
  float desiredseparation; //Separation value of boids from each other
  PVector sum = new PVector();
  int count = 0;
  for (int i = 0; i < boids.size(); i++) {
    Vehicle other = boids.get(i);
    if (other.type == type) { //To differentiate boids of different type
      desiredseparation = type.selfinterference;
    } else {
      desiredseparation = type.otherinterference; //boids of other types
    }
    if (other.location.x > leftmargin + desiredseparation &&
      other.location.x < width - rightmargin - desiredseparation &&
      other.location.y > desiredseparation
      && other.location.y < height - desiredseparation) {
      ; //do nothing at all;
        //boid is far enough away from the walls copy is unnecessary
    } else {
      if (other.location.x < leftmargin + desiredseparation) {
        PVector apparentlocation =
          replicate(other.location.x, other.location.y, "X");
        float d = PVector.dist(location, apparentlocation);
          //distance to virtual boid
        if ((d > 0) && (d < desiredseparation)) {
          PVector diff = PVector.sub(location, apparentlocation);
          diff.normalize();
          sum.add(diff);
          count++; //added to count to separate from
        }
        if (other.location.y < desiredseparation) {
          //adds second pvector, diagonal
          PVector apparentlocation2 =
            replicate(other.location.x, other.location.y, "XY");
          float e = PVector.dist(location, apparentlocation2);
```

```
      if ((e > 0) && (e < desiredseparation)) {
        PVector diff = PVector.sub(location, apparentlocation2);
        diff.normalize();
        sum.add(diff);
        count++;
      }
    }
    if (other.location.y > height - desiredseparation) {
      //adds second pvector, diagonal
      PVector apparentlocation2 =
          replicate(other.location.x, other.location.y, "Xy");
      float e = PVector.dist(location, apparentlocation2);
      if ((e > 0) && (e < desiredseparation)) {
        PVector diff = PVector.sub(location, apparentlocation2);
        diff.normalize();
        sum.add(diff);
        count++;
      }
    }
  }
  if (other.location.x > width - rightmargin - desiredseparation) {
    PVector apparentlocation =
      replicate(other.location.x, other.location.y, "x");
    float d = PVector.dist(location, apparentlocation);
    if ((d > 0) && (d < desiredseparation)) {
      PVector diff = PVector.sub(location, apparentlocation);
      diff.normalize();
      sum.add(diff);
      count++;
    }
    if (other.location.y < desiredseparation) {
      //adds second pvector, diagonal
      PVector apparentlocation2 =
        replicate(other.location.x, other.location.y, "xY");
      float e = PVector.dist(location, apparentlocation2);
      if ((e > 0) && (e < desiredseparation)) {
```

```
      PVector diff = PVector.sub(location, apparentlocation2);
      diff.normalize();
      sum.add(diff);
      count++;
    }
  }
  if (other.location.y > height - desiredseparation) {
    //adds second pvector, diagonal
    PVector apparentlocation2 =
        replicate(other.location.x, other.location.y, "xy");
    float e = PVector.dist(location, apparentlocation2);
    if ((e > 0) && (e < desiredseparation)) {
      PVector diff = PVector.sub(location, apparentlocation2);
      diff.normalize();
      sum.add(diff);
      count++;
    }
  }
}
if (other.location.y < desiredseparation) {//second check of this
  //makes either first or third vector, depending
  PVector apparentlocation =
    replicate(other.location.x, other.location.y, "Y");
  float d = PVector.dist(location, apparentlocation);
  if ((d > 0) && (d < desiredseparation)) {
    PVector diff = PVector.sub(location, apparentlocation);
    diff.normalize();
    sum.add(diff);
    count++;
  }
}
if (other.location.y > height - desiredseparation) {//same as above
  PVector apparentlocation =
    replicate(other.location.x, other.location.y, "y");
  float d = PVector.dist(location, apparentlocation);
  if ((d > 0) && (d < desiredseparation)) {
```

```
          PVector diff = PVector.sub(location, apparentlocation);
          diff.normalize();
          sum.add(diff);
          count++;
        }
      }
    }
    float d = PVector.dist(location, other.location);
    if ((d > 0) && (d < desiredseparation)) {
      PVector diff = PVector.sub(location, other.location);
      diff.normalize();
      sum.add(diff);
      count++;
    }
  }
  if (count > 0) { //separate from all added vectors!
    sum.div(count);
    sum.setMag(speed());
    PVector steer = PVector.sub(sum, velocity);
    steer.limit(type.maxforce);
    applyForce(steer);
  }
}

void display() {// Vehicle is a triangle pointing in the direction of velocity
  if (velocity.mag() != 0) heading = velocity.heading() + PI/2;
  //so it can face a different way when stopped
  fill(type.red, type.green, type.blue); //color of boid
  stroke(0);
  pushMatrix();
  translate(location.x, location.y);
  rotate(heading);
  beginShape();
  vertex(0, -r*2);//top point.
    //Following points are connected with straight lines
  vertex(-r, r*2);
```

```
    vertex(0, r);//dimple!
    vertex(r, r*2);
    endShape(CLOSE);
    popMatrix();
  }
}

ArrayList<Vehicle> boids = new ArrayList<Vehicle>(); //list of boids
ArrayList<PVector> foodstuffs = new ArrayList<PVector>(); //list of foods

int frames = 0; //time!
int starved;
int aged;
int births; //all numbers to display

PFont f; //for displayed font
PrintWriter output; //for flushing data to named file

// Given location, will give index of nearest foodstuffs.
//Replacement for food class
PVector nearestFoodstuffs(PVector location) {
  PVector foodvector = new PVector (0, 0);
  //not a literal vector; x is index of food, y is offset
  float nearest_distance = sqrt(sq(width) + sq(height));
  //setting it to some max
  for (int i = 0; i < foodstuffs.size(); i++) {
    float distancex = nearestToZero(
      location.x - foodstuffs.get(i).x,
      location.x - foodstuffs.get(i).x + width - leftmargin - rightmargin,
      location.x - foodstuffs.get(i).x - width + leftmargin + rightmargin
      ); //closer: real, or two virtual x-spaces?
    float distancey = nearestToZero(
      location.y - foodstuffs.get(i).y,
      location.y - foodstuffs.get(i).y + height,
      location.y - foodstuffs.get(i).y - height
      ); //real or virtual y-spaces?
```

```
    float distance = sqrt(sq(distancex) + sq(distancey)
      );//distance to the nearest food, either real or virtual
    if (distance < nearest_distance) { //set nearest to be a different food
    foodvector.y = 0; //reset to zero to set to a different offset, if necessary
      foodvector.x = i;
      nearest_distance = distance; //to compare
      if (distancex == location.x - foodstuffs.get(i).x -
        width + leftmargin + rightmargin) {
        foodvector.y += 3;
      } else if (distancex == location.x - foodstuffs.get(i).x +
        width - leftmargin - rightmargin) {
        foodvector.y += 6;
      }
      if (distancey == location.y - foodstuffs.get(i).y - height) {
        foodvector.y += 1;
      } else if (distancey == location.y - foodstuffs.get(i).y + height) {
       foodvector.y += 2; //numbers to iterate through in trinary, down below
      }
    }
  }
  return foodvector; //(index of nearest food, offset)
}

void goodnight() { //for shutting down the program and getting data
  output.flush(); // Writes the remaining data to the file
  output.close(); // Finishes the file
  exit(); // Stops the program
}

void setup() {
  size(750, 600); //size of area
  output = createWriter(filename); //use earlier name
  f = createFont("Arial", 16, true); //font!
  for (Boid_Type t : Boid_Type.values()) { //iterate through all boid types
    for (int i=0; i<t.start; i++) {
      boids.add(new Vehicle(
```

```
        floor(random(width - rightmargin - leftmargin) + leftmargin),
        floor(random(height)), t)); //Random placement of boids
    }
  }
  for (int i = 0; i < numfoods; i++) {
    foodstuffs.add(new PVector(
      floor(random(width - rightmargin - leftmargin) + leftmargin),
      floor(random(height))));//initial food placement
  }
}

void draw() {
  background(255); //background color
  stroke(0); //Color of food
  for (int i = 0; i < foodstuffs.size(); i++) { //drawing shape of food
    line(foodstuffs.get(i).x-2, foodstuffs.get(i).y-2,
      foodstuffs.get(i).x+2, foodstuffs.get(i).y+2
      );
    line(foodstuffs.get(i).x-2, foodstuffs.get(i).y+2,
      foodstuffs.get(i).x+2, foodstuffs.get(i).y-2
      );
  }
  for (int i=0; i <boids.size(); i++) { //go through all boids
    Vehicle this_boid = boids.get(i);
    if (this_boid.timer > this_boid.type.starvation) { //starves it
      this_boid.type.starved++;
      this_boid.type.alive--;
      boids.remove(i);
    } else if (this_boid.lifetime > this_boid.type.lifespan) { //ages it
      this_boid.type.aged++;
      this_boid.type.alive--;
      boids.remove(i);
    } else if (this_boid.eaten >= this_boid.type.foodtillbirth) { //births
      boids.add(new Vehicle(
        this_boid.location.x - 1, this_boid.location.y, this_boid.type));
      boids.add(new Vehicle(
```

```
    this_boid.location.x + 1, this_boid.location.y, this_boid.type));
  this_boid.type.births++;
  this_boid.type.alive--;
  boids.remove(i);
} else if (foodstuffs.size() != 0) { //to make sure there is actually food
  PVector nearestFoodstuff = nearestFoodstuffs(this_boid.location);
    //function called
  int nearest_index = int (nearestFoodstuff.x);
  int offset = int (nearestFoodstuff.y);
  PVector nearest_food = new PVector(0, 0);
  nearest_food.x = foodstuffs.get(nearest_index).x;
  nearest_food.y = foodstuffs.get(nearest_index).y;
  if (offset > 5) nearest_food.x -= (width - rightmargin - leftmargin);
 else if (offset > 2) nearest_food.x += (width - rightmargin - leftmargin);
  if (offset % 3 == 1) nearest_food.y += height;
  else if (offset % 3 == 2) nearest_food.y -= height;
    //iterates, trinary. External chart...
  if (sqrt(sq(this_boid.location.x - nearest_food.x) +
    sq(this_boid.location.y - nearest_food.y)
    ) < 10) {
    //10 away from food and they eat it!
    //taking into account all virtual foods too
    foodstuffs.remove(nearest_index); //remove the food
    foodstuffs.add(new PVector(
        floor(random(width - rightmargin - leftmargin) + leftmargin),
      floor(random(height)))
      );//more food place
    this_boid.timer = 0;
    this_boid.eaten++;
   this_boid.r += ((maxsize-startingsize)/this_boid.type.foodtillbirth);
  } else this_boid.seek(nearest_food);
}
this_boid.timer++;
this_boid.lifetime++;
this_boid.update();
this_boid.display(); //end of boid activity for one frame
```

```
}
frames++; //time!
stroke(0);
fill(200, 200, 200); //colors for walls and borders
beginShape();
vertex(0, -1);
vertex(leftmargin, -1);
vertex(leftmargin, height);
vertex(0, height);
endShape();
beginShape();
vertex(width, -1);
vertex(width-rightmargin, -1);
vertex(width-rightmargin, height);
vertex(width, height);
endShape(); //walls!
//Actually shapes placed on the side to cover up boids as they cross borders.

textFont(f, 14); //on-screen text
fill(0);
text(boids.size(), 10, 20);
for (int i = 0; i < Boid_Type.values().length; i++) {
  Boid_Type current = Boid_Type.values()[i];
  fill(current.red, current.green, current.blue);
  text(current.alive, 10, 40 + 20*i);
}

fill(0);
text(frames, 10, height - 10);

starved = 0;
for (int i = 0; i < Boid_Type.values().length; i++) {
  Boid_Type current = Boid_Type.values()[i];
  fill(current.red, current.green, current.blue);
  text(current.starved, width - 30, 40 + 20*i);
  starved += current.starved;
```

```
}
fill(0);
text("Starved:", width - 85, 20);
text(starved, width - 30, 20);

aged = 0;
for (int i = 0; i < Boid_Type.values().length; i++) {
  Boid_Type current = Boid_Type.values()[i];
  fill(current.red, current.green, current.blue);
  text(current.aged, width - 30, floor(height/2) +
      floor(20*(i+1 - Boid_Type.values().length/2.)));
  aged += current.aged;
}
fill(0);
text("Aged:", width - 70,
  floor(height/2) + floor(20*(-Boid_Type.values().length/2.)));
text(aged, width - 30,
  floor(height/2) + floor(20*(-Boid_Type.values().length/2.)));

births = 0;
for (int i = 0; i < Boid_Type.values().length; i++) {
  Boid_Type current = Boid_Type.values()[i];
  fill(current.red, current.green, current.blue);
  text(current.births,
    width - 30, height - 20*(Boid_Type.values().length - (i+1)) - 10);
  births += current.births;
}
fill(0);
text("Births:", width - 73, height - 20*(Boid_Type.values().length) - 10);
text(births, width - 30, height - 20*(Boid_Type.values().length) - 10);
//now for external file output
if (frames == 1) { //boid initial data, labels
 output.println("Area Size" + "\u0009" + (width - rightmargin -leftmargin) +
   " x " + height + "\u0009" + "" + "\u0009" + "Numfoods" + "\u0009" + numfoods
    );
  output.println("Boid Type" + "\u0009" +
```

```
    "Red" + "\u0009" + "Green" + "\u0009" + "Blue" +
    "\u0009" + "Start" + "\u0009" +
    "Starvation" + "\u0009" + "Lifespan" + "\u0009" +
    "Self Itf" + "\u0009" + "Other Itf" + "\u0009" + "FTB" + "\u0009" +
    "Maxforce" + "\u0009" + "Maxspeed" + "\u0009" + "Transition"
    );
  for (int i = 0; i < Boid_Type.values().length; i++) {
    Boid_Type current = Boid_Type.values()[i];
    output.println(current.name() + "\u0009" + current.red + "\u0009" +
      current.green + "\u0009" + current.blue + "\u0009" +
      current.start + "\u0009" +
      current.starvation + "\u0009" + current.lifespan + "\u0009" +
      current.selfinterference + "\u0009" +
      current.otherinterference + "\u0009" +
      current.foodtillbirth + "\u0009" +
      current.maxforce + "\u0009" +current.maxspeed + "\u0009" +
      current.transitiontime
      );
  }
  output.println(" "); //empty line
  output.print("Frames\u0009Alive\u0009" +//labels!
    "Starved\u0009Aged\u0009Births\u0009"
    );
  for (int i = 0; i < Boid_Type.values().length; i++) {
    output.print((i+1) + " Alive" + "\u0009" + //more labels!
      (i+1) + " Starved" + "\u0009" + (i+1) +
      " Aged" + "\u0009" + (i+1) + " Births" + "\u0009"
      );
  }
  output.println();
}
if (frames%100==0) { //how often it spits out data
  output.print(frames + "\u0009" + boids.size() + "\u0009" + //total values
    starved + "\u0009" + aged + "\u0009" + births + "\u0009"
    );
  for (int i = 0; i < Boid_Type.values().length; i++) {
```

```
      Boid_Type current = Boid_Type.values()[i];
      output.print(current.alive + "\u0009" + //individual type values
        current.starved + "\u0009" +
        current.aged + "\u0009" +
        current.births + "\u0009"
        );
    }
    output.println();
  }
  if (frames >= totaltime) goodnight();
}

void keyPressed() { //to shut it down early
  goodnight();
}
```

Bibliography

[1] William E. Boyce, Richard C. Diprima. *Elementary Differential Equations and Boundary Value Problems, Ninth Edition*. Wiley. Grafton, NY, 2008.

[2] Douglas R. Hundley. "Poincare Diagram: Classification of phase portraits in (detA, TrA)−plane." Whitman College, WA, Fall 2012. http://people.whitman.edu/~hundledr/courses/M244F12/M244/PoincareDiagram.jpg

[3] Daniel Shiffman. *The Nature of Code*. Self-published, 2012. http://natureofcode.com/