Fall 12-4-2019

Individual Based Model to Simulate the Evolution of Insecticide Resistance

William B. Jamieson
University of Nebraska - Lincoln, wjamieson@huskers.unl.edu

INDIVIDUAL BASED MODEL TO SIMULATE THE EVOLUTION OF INSECTICIDE RESISTANCE

by

William B. Jamieson

A DISSERTATION

Presented to the Faculty of

The Graduate College at the University of Nebraska

In Partial Fulfilment of Requirements

For the Degree of Doctor of Philosophy

Major: Mathematics

Under the Supervision of Professor Richard Rebarber and Professor Brigitte Tenhumberg

Lincoln, Nebraska

December, 2019

INDIVIDUAL BASED MODEL TO SIMULATE THE EVOLUTION OF INSECTICIDE RESISTANCE

William B. Jamieson, Ph.D.

University of Nebraska, 2019

Adviser: Richard Rebarber and Brigitte Tenhumberg

Insecticides play a critical role in agricultural productivity. However, insecticides impose selective pressures on insect populations, so the Darwinian principles of natural selection predict that resistance to the insecticide is likely to form in the insect populations. Insecticide resistance, in turn, severely reduces the utility of the insecticides being used. Thus there is a strong economic incentive to reduce the rate of resistance evolution. Moreover, resistance evolution represents an example of evolution under novel selective pressures, so its study contributes to the fundamental understanding of evolutionary theory.

Insecticide resistance often represents a complex interplay of multiple fitness tradeoffs for individual insects. Resistant individuals tend to suffer significant decreases in fitness when no insecticide is present, resulting in non-resistant individuals having the tendency to outcompete resistant ones in areas with no insecticide. In the use of standard modeling practices, difficulties arise when trying to incorporate these complexities in a fashion which facilitates the simulation of the model and analyzing the results. Individual based models (IBMs) are one approach to overcoming these difficulties by leveraging modern computational techniques and modern computer power. In an IBM each member of the population is simulated to follow a set of stochastic rules, which includes rules about the behaviors and interactions of individuals. We propose to apply an IBM approach to modeling the evolution of insecticide resistance

in an insect species population.

The fall armyworm is an economically damaging pest which has recently become invasive in Africa, India, and China. A common type of insecticide used control fall armyworms is *Bacillus thuringiensis* (Bt). We hypothesize that individuals that are resistant to Bt grow at slower rates than their counterparts. This creates a strong fitness disadvantage when Bt is not present because the fall armyworms are cannibalistic, where smaller individuals have a large disadvantage. Thus we use our IBM to explore the nature of the fitness trade-offs between resistance and growth rate in order to understand how it could be exploited to lessen the rate of resistance evolution in the species.

DEDICATION

For my wife Jessie, and in loving memory of my two best friends Taz and Rocket. For without any of you, I would not have gotten this far in life.

ACKNOWLEDGMENTS

Below, I attempt to acknowledge all those who have contributed to my success in completing my doctorate in mathematics, for which this document is the capstone. I could not have made it this far without the support of many people and institutions, and I cannot possibly acknowledge them all.

To begin, I credit most of my success to my parents, my mother Dr. Mary Jane Jamieson and my father Robert Gordon Jamieson. They taught me to think critically and rationally, supported me through thick and thin, and most importantly believed in my success. Together they are my heroes because they did everything in their power to get me here, and acted as role models for what I wanted to achieve and how I wanted to be. Their love and support was incalculably important to me in getting this far. I owe so much to their support that it is impossible to express in words; I truly would not have made it this far without them.

I would also like to acknowledge the role that my wife Dr. Jessie Deering Jamieson played in getting me through my doctorate in mathematics. We supported each other through both of our doctorate programs (both in mathematics), acting as each other's partners in all things mathematical and real world. She has acted as my sounding board, greatest supporter, and largest helper. I would especially like to acknowledge her role in editing this document, for I am not a word smith and she knows how to fix all my smithing errors. Jessie has acted as my rock through all of the troubles, false starts, failures, and successes in this process. I am truly grateful to have such a wonderful partner.

Without all of my close friends, I would not have made it this far. I would like to include all of my close graduate student friends including Matt Reichenbach and Nathan Poppelreiter, and my three closest office mates: Cory Wright, Wei Hui, and

Jesse Moeller. Some of my closest friends include all of my friends from my undergraduate years: Erin Middlemas, Chance Brown, Brandon Sexton, Matt Cannon, and Olivia Miller. I also cannot forget all of my close friends from both my ETSU REU and my Auburn REU, their are too many to list; however, I would like to specifically thank Dr. Bill Kay for telling me to not give up when I needed it the most. Finally, there are my three closest friends Ethan Duncan, Taz, and Rocket. Ethan is the closest person I ever had to being an actual sibling. While Taz and Rocket, though only dogs, were the closest individual friends I had in the deepest and darkest times; I feel their absence from my life everyday and without the joy they brought to my life, this process would have ended much sooner; thus I dedicated this endeavor to their memory.

Next, I would like to acknowledge all of my undergraduate instructors at East Tennessee State University, especially those in the Department of Mathematics and Statistics and the Department of Physics and Astronomy. Without the support of both of these faculties I do not think I would have made it through my undergraduate education, let alone my graduate one.

In particular, I would like to acknowledge several of these faculty members of the Department of Mathematics and Statistics individually. First, I would like to acknowledge Dr. Robert Gardner for being an awesome mentor who was always willing to indulge my interests and for introducing me to my wife Jessie. I would also like to acknowledge my first REU mentor, Dr. Anant Godbole who believed in me enough to admit me to his REU and show me how to really do mathematics; I learned more from him than anyone else about how to think, write, and act as a mathematician. A glaring omission would be Dr. Teresa Haynes, who advised my undergraduate thesis in mathematics. I cannot express how much she taught me about life and how much her belief and support in me has made my success possible. Dr. Jeff Knisley needs

special mention for sparking my interests in using computers and computation to solve applied mathematics problems, and for always acting as someone to tell me what the next big thing in mathematical computing is. Finally, I would like to thank Dr. Edith Seier for teaching me real statistics, which I thought I would never need in order to do mathematics. Her time series analysis class, which she permitted me into taking because I had no statistics background, has been by far the most useful mathematics course I took as an undergraduate in pursuit of the research presented here.

In the Department of Mathematics and Statistics, I would like to thank Dr. Mark Giroux and Dr. Beverly Smith for their support of someone who did not know if they wanted to be a scientist or a mathematician. I would like to especially thank Dr. Donald Luttermoser for teaching me the arts of Fortran programming, which lead me to so many opportunities that brought me to my thesis topic. And finally, Dr. Richard Ignace and Dr. Gary Henson for teaching me how to think like a scientist.

There are still others outside of my university education. I need to thank Dr. Peter Johnson of Auburn University for hosting me in my second REU and acting as a massive supporter to my endeavors. Dr. James Baranuk of Georgetown University for being an awesome scientist, friend, and supporter who gave me my first real opportunity to do something outside a classroom. And finally, Dr. Thomas Clune of the NASA Goddard Space Flight Center, who chose me to be his intern. Dr. Clune changed my life forever by helping me fall in love with doing science with hard mathematics and teaching me how to be a real computer scientist.

I would like to acknowledge the support of the University of Nebraska-Lincoln. Specifically, this includes all of the faculty members who I took classes from or were willing to give me advice; additionally, the graduate secretary Marilyn Johnson's support cannot be ignored. I especially thank my dissertation committee members: Dr.

Glenn Ledder, Dr. Huijing Du, and Dr. Ana Meria Vélez in addition to both of my wonderful advisors: Dr. Richard Rebarber and Dr. Brigitte Tenhumberg. My committee has been wonderful and supportive. I specifically thank Dr. Adam Larios and Dr. Petronela Radu for your aid in helping me win the National Science Foundation Graduate Research Fellowship. Moreover, I need to thank the Holland Computing Center for allowing me to use its extensive resources free of charge. Also, I would like to thank the UNL Counseling and Psychological Services, and Dr. Brigham Scott for giving me the mental/emotional support I needed to make it through my PhD program.

Finally, I need to acknowledge the support of each of my advisors. Richard's support and patience have been so wonderful for me to have. He has been a clear rock of stability and a wonderful mentor in my journey to this point. Brigitte has taught me more than anyone else how to be a biologist and think like a biologist. She has been a guiding force shaping how I approach my work and in how I want to move forward in my career. Both of your support and continual motivation have helped me not give up and succeed in this doctorate dissertation. I hope that both of you have enjoyed working with me as much as I have with you, and that if possible we can continue to work together, as I will miss working with both of you.

GRANT INFORMATION

William B. Jamieson was supported by National Science Foundation Graduate Research Fellowship under Grant No. DGE-104100. Any conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

This work was completed utilizing the Holland Computing Center of the University of Nebraska, which receives support from the Nebraska Research Initiative.

Table of Contents

List of Figures

List of Tables

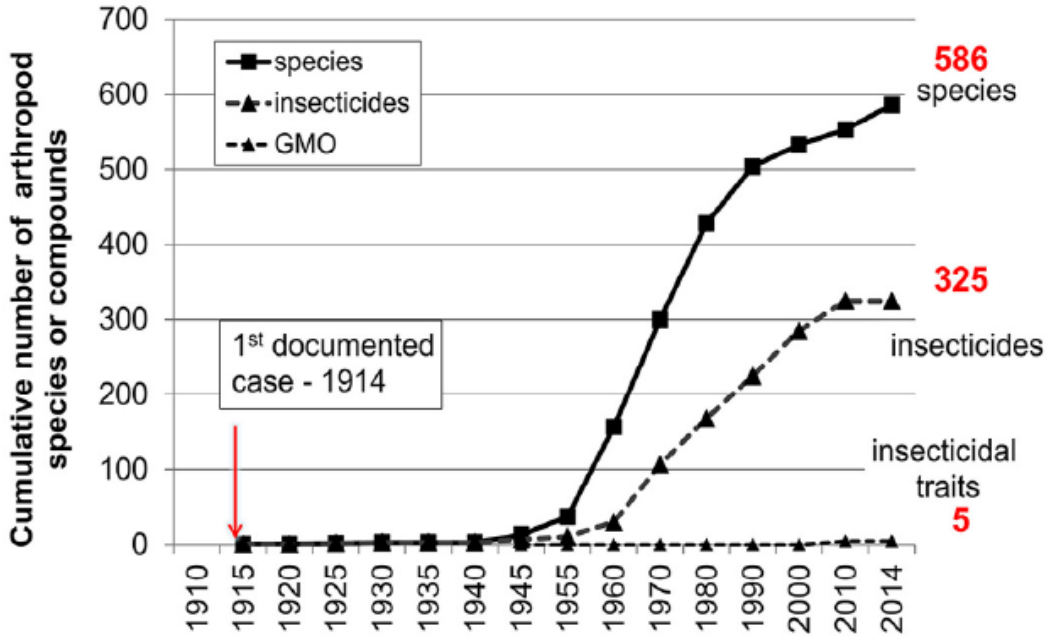
Chapter 1

Introduction

The use of insecticides over the last century has greatly improved agricultural productivity [71]. By 2050 it is anticipated that the human population will exceed nine billion; in order to sustainably meet the agricultural demands of feeding this population, it is anticipated that effective insecticides will be required [57]. However, insecticide use imposes strong selection pressures on pest insects. Darwinian principles assert that if sufficient genetic variation and selective pressures exist for a population, then the population will adapt to the new pressures through evolutionary changes [31]. It follows that insecticide use is likely to lead to evolution of insecticide resistance in insect populations.

In 1914 the first case of insecticide resistance in an insect species was observed; one hundred years later there were at least 586 different insect species with documented insecticide resistance [107], see Figure 1.1 This rapid rise in resistance would not be problematic if pharmaceutical and agribusiness companies could continually develop effective new insecticides before old ones begin to experience failures due to evolution of resistance; however, it is likely that the rate of resistance evolution to insecticides outpaces the development of effective replacements [30, 99]. Thus there is a need for strategies to reduce the rate of resistance evolution to insecticides in order to prolong the utility of current insecticides [21]. Moreover, evolution of insecticide resistance

Figure 1.1: Cumulative Increase in Number of Insecticide Resistant Species [107]



represents an example of rapid evolution under novel selective pressures, so the study of insecticide resistance evolution contributes to our fundamental understanding of general evolutionary theory [57].

Insecticide resistance represents a multifactorial phenomena depending on an array of factors related to the species and insecticide in question. The resistant strains of a species evolve through the survival and reproduction of individuals which carry mutant genomes that provide mechanisms for surviving exposure to the insecticide [21]. These mutations come in several varieties including: alterations to biochemical target sights [37, 57], changes to detoxification processes [8, 40], physiological process adjustments [9, 39], behavioral changes [128], and altered internal environments [46]. Herbivorous insects evolve resistance to natural allelochemical toxins using the same mechanisms although on much slower time scales [57]. The contrasting rapid timescales of insecticide resistance evolution, derived from field studies, are most likely

due to the extreme mortality (almost 100%) on susceptible individuals imposed by commercial insecticides. Clearly, this will force any resistance related traits to rapidly increase to near fixation (most individuals have resistance genes).

Often when insecticide use ceases for a population, resistance begins to rapidly decline, suggesting resistance is selected against in the absence of the insecticide [71]. Hence, this suggests that there are fitness costs to resistant individuals when compared to susceptible individuals. These costs come in several forms, such as overexpression of resistance-conferring genes changing the resource allocation balance of an organism, or the modification of target-sights (of the insecticide) which can substantially reduce metabolic efficiency [63]. Often these costs reduce insect development times and/or size, which can be especially detrimental in cannibalistic species [13]. So the evolution of resistance to insecticides can yield fitness trade-offs between when individuals are exposed to the insecticide and when they are not.

Evolution of resistance to insecticides has been modeled in many different ways. Typically, these models use common types of population models which have been altered to include genotype information using the Hardy-Weinberg equations [43]. Most of these models use deterministic techniques such as systems of differential, difference, or algebraic equations [18, 23, 33, 84, 91, 112]. However, these techniques become intractable for more complex situations, thus Stratonovitch *et al.* [110] suggested the use of individual based based models for evolution of resistance to insecticides in order to capture complexities that make analysis of deterministic techniques intractable, see Section 1.1 for a discussion of individual based models.

Indeed, mathematical models of biological populations using deterministic techniques can get extremely sophisticated [24, 35]; however, to capture many aspects of evolution there is a need to include stochastic effects. Stochasticity can be incorporated into many of these deterministic models with varying degrees of ease

and success, but complex evolutionary often require extremely complex distributions. This creates issues when simulating these stochastic models because drawing from complex distributions can be both computationally expensive and results in errors due to the pseudo-random number generators employed by computers. To solve this Otto and Day [82] suggest switching to an individual based modeling approach to limit the stochastic complexities involved with implementing complex stochastic models.

Our goal is to produce a general-purpose model to simulate the evolution of resistance to an insecticide which results in fitness trade-offs between resistant and susceptible genotypes. To accomplish this goal, we choose the economically relevant fall armyworm exposed to insecticide-producing transgenic-crops. We model this using an individual based model to capture the complex interdependencies within the fall armyworm's behavior that may effect the evolutionary processes. Note that we developed this individual based model to be easily generalizable to other situations with evolution of insecticide resistance.

1.1 Individual Based Models

Historically, most scientific models of ecological phenomena describe populations of organisms. Mathematically, these models are typically based on systems of algebraic or ordinary/partial differential equations. This approach often requires simplifying assumptions so that the resulting mathematical models have tractable solution and/or analysis methods. These simplifications impose limits upon the complexity of phenomena which can be modeled in a useful fashion.

Individual based models (IBMs), also called *agent based models*, are one way to circumvent some of these limitations by modeling ecological phenomena through descriptive modeling of organisms and then allowing many organisms to interact, result-

ing in the *emergence* of ecological phenomena. IBMs produce solutions in a tractable fashion by taking advantage of modern computational techniques and modern computational power to simulate all of the organisms together in an interactive manner. Thus *IBMs are models where individual organisms, or agents, are described as unique and autonomous entities that usually interact with each other and their environment locally*, Railsback and Grimm 2012 [96]. IBMs model populations so that each individual within the population is represented by a distinct and independent agent. In doing this, we reduce our modeling from a global outlook to a local outlook because individuals in a population can typically only have local effects on their surroundings. That is, interactions among individuals tend only to occur between pairs (or possibly small groups) of individuals, while individuals can only affect the small (local) portion of the environment they occupy.

We reserve the term *agent* to refer to the composite model corresponding to the individuals we are modeling, *i.e.* agents refer to the collection of models and variables that form autonomous entities within our IBM. This means that IBMs are discussed through the descriptions of the agents involved; specifically, agent descriptions involve describing the processes within the IBM as actions taken by the agents or events that happen to agents. *Behaviors* are actions taken by agents, such as growing or moving. Note that behaviors include actions taken by agents which have external effects, like producing new agents, and actions taken by agents which have internal effects, such as growing. To contrast, *events* are things that happen to agents, such as death, being attacked by another agent, or being born.

Thus IBMs for ecological systems are created by describing agents in terms of all the relevant properties, behaviors, and important events for each individual organism. It is important that each of these agents be built from a purely local perspective such that each resulting agent is independent (except through interaction) from all other

agents. Additionally, the IBM needs an environment which both contains the agents and any external factors. Typically, the environment contains the agents in a manner which facilitates interactions between agents; often this is achieved by organizing the agents spatially, such as with a grid. External factors are those things outside of a description of an agent; they can be global such as time of year, or localized such as the amount of food near an agent.

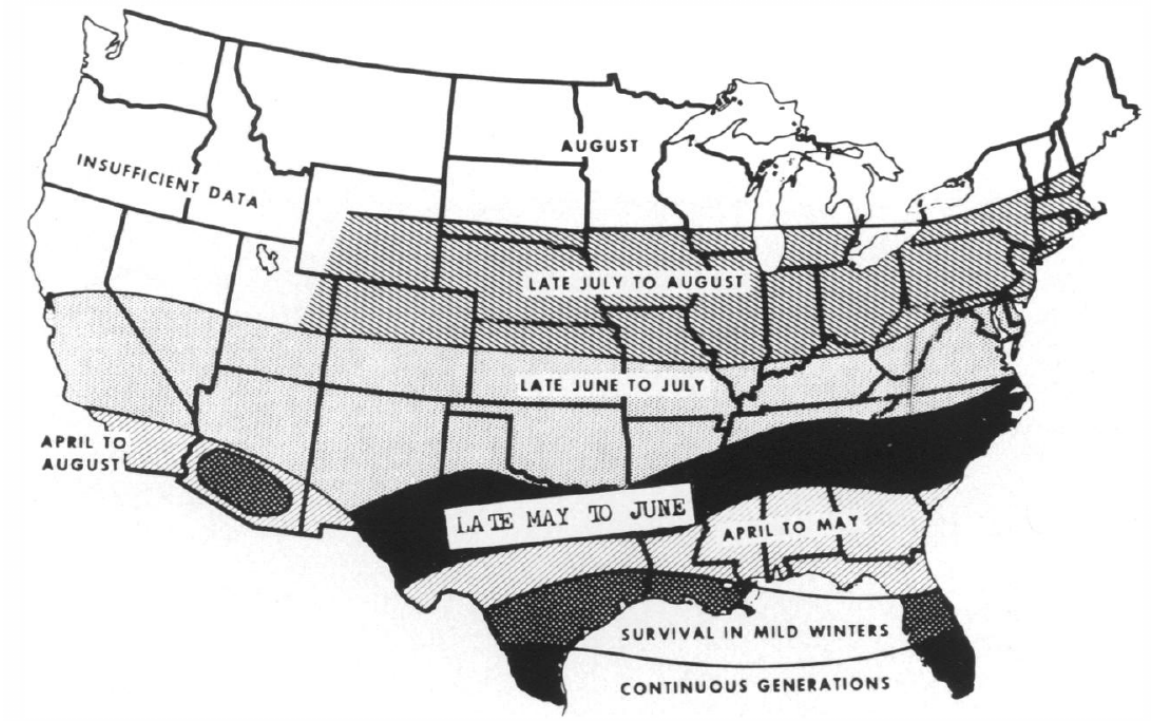
Organisms change in response to other organisms and their environment; moreover, organism's responses tend to be governed by local events. IBMs are a useful way to approach incorporating such mechanisms into a cohesive model [53]. We can then study the emergence of global behaviors which result from all of the local modeling. Thus IBMs make good candidates for modeling the ecological phenomena which involve adaptation and mediated by organism-organism interactions, which tend to be difficult to capture in more classical approaches. Hence, our problem will benefit from being modeled by an IBM.

1.2 Fall Armyworm

Our organism of interest is *Spodoptera frugiperda* (J.E. Smith, 1797), which is commonly known as the fall armyworm. The fall armyworm is an insect species of the order Lepidoptera and family Noctuidae, meaning the fall armyworm is a species of moth. Fall armyworms are common pest insects found throughout the United States that are capable of large-scale crop damage [7]. Their propensity for causing crop damage together with their appearance in the northern portions of the United States in late summer are what give them their common name.

The species overwinters in regions where temperatures rarely drop below 50°F (10°C), mainly in more tropical regions such as the southern reaches of Florida and

Figure 1.2: Fall Armyworm Migration in the United States [106]



Texas in the United States. This is because the fall armyworm lacks a diapause mechanism to survive in areas of colder year-round climate [122]. However, fall armyworm adults are strong flyers [77], which allows them to make use of the prevailing wind currents to seasonally spread northward across the United States and Southern Canada; see Figure 1.2. Fall armyworm populations begin to establish themselves in the Midwestern United States in the late summer, and so by the start of fall large-scale crop damage begins to appear in these areas due to fall armyworms.

The fall armyworm is a particularly destructive pest because the final instars[1] consume relatively large amounts of foliage. Since the relatively tiny (and so harder to detect) young larvae tend to go undetected until the population matures, this results in large amounts of crop damage appearing *overnight* [48]. Indeed, Luginbill 1928 [69] reports that over 77% of the foliage consumed by a fall armyworm larva occurs during

[1] An *instar* is a developmental phase between each moult of an insect's exoskeleton.

its final instar, while less than 2% is consumed by in the first three instars. Moreover, established populations of fall armyworms will rapidly shift across landscapes as they defoliate different areas[2], making it difficult to mitigate large-scale crop damage once damages from fall armyworms start to become evident.

Fall armyworms have polyphagous feeding habits, meaning that they feed on a wide variety of substrates. Indeed, the fall armyworm has been documented feeding on at least 353 different host species of plant [76]. They do exhibit some feeding preferences, in particular they appear to prefer grasses in the Poaceae family, such as corn, sorghum, bermudagrass, and crabgrass [89]. There is also a subspecies of fall armyworm which specializes on rice and related crops [55, 79]. However, when pressed they seem to feed on any leafy green plant [69]. Moreover, the fall armyworm exhibits strong cannibalism [25] even when abundant vegetation is present.

The migratory and polyphagous feeding habits of the fall armyworm make it an economically important pest insect throughout the tropical and sub-tropical regions of the Americas, see Figure 1.3[3]. Moreover, these factors make the fall armyworm a dangerous possible invasive species. In 2016, the fall armyworm invaded and spread throughout sub-Saharan Africa [28, 51], leading to an estimated 20% to 60% of corn yield loss over the first two years of the invasion [42]. The fall armyworm is now considered an invasive pest species in sub-Saharan Africa [75, 78]. In 2018, a fall armyworm invasion was established in India [38] and has continued to spread [27]. By 2019, the fall armyworm was reported to infest regions in China [125]. The fall armyworm is expected to continue to spread throughout eastern China and southeast Asia due to the prevailing wind patterns [67].

[2]This is where the common name *armyworm* originates. The armyworm insect species all share the behavioral trait that they will rapidly move across landscapes, invading new areas to defoliate.

[3]Slanted lines indicate year-round population presence, while the checkered pattern indicates presence only during the summer months.

Figure 1.3: Native Distribution of the Fall Armyworm [62]

Therefore, the fall armyworm is an economically important pest insect and investigations into pest and destruction mitigation techniques are of high importance. Techniques to control the establishment of a fall armyworm in cropland in order to mitigate damage involve the use of insecticides. However, as we will see below, this enforces evolutionary pressures on the fall armyworm populations to adapt to these interventions. We are interested in modeling this adaptive response in order to understand how we can prolong the usefulness of these insecticides.

Figure 1.4: Fall Armyworm Egg Mass [22]

1.2.1 Life-Cycle

The fall armyworm life-cycle completes in 30 (in summer) to 90 (in winter) days, depending on the local temperature. The number of generations in a locale depends largely on when in the year adults begin to appear, which correlates with temperature (see Figure 1.2). In lower latitudes the fall armyworm has upwards of four generations, while higher latitudes can have as little as one or two [22, 106]; typically, in most of the United States there are three generations.

As members of the Lepidopteran order (butterflies and moths), the fall armworm experiences four distinct life-stages during its life cycle: egg, larva, pupa, and adult. The duration of life-stages varies mainly due to temperatures, although there is some evidence that the types of food resources available also have some effect [70, 93].

1.2.1.1 Eggs

Fall armyworm eggs are approximately $0.4mm$ in diameter and are deposited in clumps of 100 to 200 eggs, called egg masses, which the female covers with scales [22] (Figure 1.4). Females usually deposit their egg masses on the undersides of leaves; however, as adult density increases they become less and less discriminating in where they deposit their eggs [106]. Eggs hatch in 2 to 4 days during the summer months,

Figure 1.5: Fall Armyworm Larvae [22]

(a) Newly hatched fall armyworm larva (b) Mature fall armworm larva

with most eggs in an egg mass hatching at almost the same time.

1.2.1.2 Larvae

Newly hatched larvae (caterpillars) are less than $1.7mm$ in length (Figure 1.5a). Larvae then progress through 6 instars as they mature. Throughout this development period the larvae will continually consume plant foliage until pupation occurs. In the 6^{th} instar, larvae will have grown to near $34.2mm$ length (Figure 1.5b) and will drop to the ground to pupate in the nearby soil [22, 106].

Cannibalism in the larval population plays a crucial role in fall armyworm populations [12, 13]. Larval cannibalism is so prevalent that despite the large reproductive output of adults, population densities on individual plants decrease so that typically only one late instar larva can be found per plant [45]. It is suspected that the prodigious cannibalism rates of the fall armyworm help limit the effects of predation[4] and intra-species competition on the late-stage larvae [26].

[4]Chapman *et al.* [26] found evidence that noticeable feeding damage to crop plants attracts natural predators, so they theorized that cannibalism helps to reduce damage below these levels.

Figure 1.6: Fall Armyworm Pupa [15]

1.2.1.3 Pupae

Once larvae have fully grown, they drop to the ground, burying themselves 1 to 3 inches deep, depending on soil conditions, in order to pupate (Figure 1.6). Adults then emerge from pupae after 7 to 37 days depending on the soil temperature. Note that this emergence is highly correlated with the number of days that soil temperature remains above 10°C [124]. Therefore, the fall armyworm's year round habitats are restricted to the more tropical regions in Figure 1.3.

1.2.1.4 Adults

Fall armyworm adults typically emerge from their pupal cases after dark, where they then cling to plants and inflate their wings; once their wings are inflated, they move off to begin feeding. After their first night of life adults fall into a normal cycle of behavior, where just after dusk adults begin finding suitable host plants for feeding. Then after sunset females begin calling mates by positioning themselves near the top of plants and emitting sex pheromones. The distances that males respond from depends highly on wind conditions and temperature; typically, she waits for two or

Figure 1.7: Adult Fall Armyworms [22]

(a) Female (b) Male

more males to respond. Since females only mate once per night males tussle over who gets to mate with her; however, once she has selected a mate the rejected males fly off to find other females while the selected male follows her for most of the night [106].

Note that fall armyworm adults are strong fliers and so many adults will use prevailing wind currents to migrate long distances to new habitats. Adults typically live 2 to three weeks [22]; however, once females begin oviposition, their fecundity drops rapidly to near zero after about 10 days [89].

1.2.2 Bt Insecticides

Bacillus thuringiensis (Bt) is a Gram-positive soil-dwelling bacteria which produces a wide variety of parasporal crystals that have pathogenic activity in insects [36]. In particular, insecticides derived from Bt are particularly effective against insects in the order Lepidoptera; however, there are Bt insecticides that affect other orders of insects. Bt insecticides have been developed into effective liquid sprays, with trade names such as DiPel and Thuricide; recently, advances in genetic engineering have allowed for the development of transgenic crop plants which produce Bt insecticides internally. Transgenic crops engineered to produce Bt include: tobacco, potato, corn, cotton, and soybeans [66].

Bt's pathogenic mechanism for insects operates via parts of the parasporal crystals, called *δ-endotoxins*. When ingested, a δ-endotoxin gets inserted into the membranes of the cells which form the lining of the gut. This disrupts the activities of these cells, which eventually paralyzes the digestive tract and causes pores to form. This causes the insect to stop eating, which results in it starving to death [101].

Bt insecticides are generally effective against the fall armyworm [105], so they are a common means of preventing crop damage from this pest. However, Bt and several related species of bacteria are known to colonize some fall armyworm digestive tracts. Thus a subset of the fall armyworm population has evolved resistance to these effects from some types of δ-endotoxins [59, 111]. However, resistance to δ-endotoxins is not widespread in the tropical year round populations, so it is theorized that it has some fitness costs. Some of these fitness costs have been shown to be some type of growth inhibition [34, 116]. This negatively effects resistant fall armyworms because during cannibalism it causes disproportionately more mortality in smaller fall armyworms [12, 13]. These fitness trade-offs will be assumed as part of our model.

1.2.3 Summary of Important Characteristics

Recall that our goal is to study the adaptation of the fall armyworm to the presence of transgenic Bt-crops. First, we will need to appropriately capture the life history/life-cycle of the fall armyworm. Next, we need to include genetics along with methods to allow fitness differences due to genotype. These differences should include differences in survival in Bt environments and differences in organismal growth rate. By including genetics we also need to model the inheritance of genetic characteristics into the next generation. Finally, we will include cannibalism because differences in growth rate will help facilitate the fitness trade-offs of resistance. We hypothesize that cannibalism together with these trade-offs will help control the emergence of resistance to Bt.

Chapter 2

Individual Based Model for the Fall Armyworm

This chapter is intended to explain the major components and features of our IBM. We begin by modeling the relevant components of the fall armyworm's biology (Section 2.1) and then assemble these models into an IBM (Section 2.2).

2.1 Component Models and Their Baseline Parameters

Here we discuss modeling all of the components of the fall armyworm's biology that we need in order to assemble our IBM. These are genetics, initial biomass, growth, cannibalism, biomass consumption, movement, reproduction, development, survival, and migration.

2.1.1 Genetics

Our goal is to understand the effects of Bt exposure on the evolution of Bt-resistance in the fall armyworm population. Hence, resistance must be genetically conferred to individuals. Thus we assume resistance is controlled by a single gene which has two alleles: susceptible (S) and resistant (R). This is well supported by the study of the ABCC2 gene in the fall armyworm by Flagel *et al.* [47], which identified mutations of the ABCC2 gene that confer resistance to Bt insecticides.

Table 2.1: Expected Proportions of Child Genotypes

Parental Cross	SS	SR	RR
SS × SS	1.0	0.0	0.0
SS × SR	0.5	0.5	0.0
SS × RR	0.0	1.0	0.0
SR × SR	0.25	0.5	0.25
SR × RR	0.0	0.5	0.5
RR × RR	0.0	0.0	1.0

There are three possible combinations of alleles (termed *genotypes*): fully susceptible (SS), heterozygous (SR), and fully resistant (RR). Under Mendelian rules of genetics, we assume the resistant gene will independently assort. That is, for each parent, each of its copies (alleles) of the resistant gene are equally likely to be passed to the child. This leads to the genotype proportions for offspring in Table 2.1. In Figure 2.1 we report the distribution of relative proportions of genotypes of parental crosses as simulated by the IBM. One can see that for a large number of trials we will recover the distribution of child genotypes that we expect from a given parental cross.

Genotype affects agent behaviors by changing the parameters used to determine traits. Suppose that X is a parameter value concerning some genotype dependent trait; we focus on finding/fitting values for X_{SS} and X_{RR}, which are the parameter values for the SS and RR genotypes respectively. Stoner [108] proposed applying Falconer's *degree of dominance* formula [43] to quantify the effects of insecticide resistance alleles. This means that the parameter value for the SR genotype is given by:

$$X_{SR}(d) = \frac{X_{RR} + X_{SS}}{2} + d\frac{X_{RR} - X_{SS}}{2}, \tag{2.1}$$

Figure 2.1: Relative Proportions of Child Genotypes

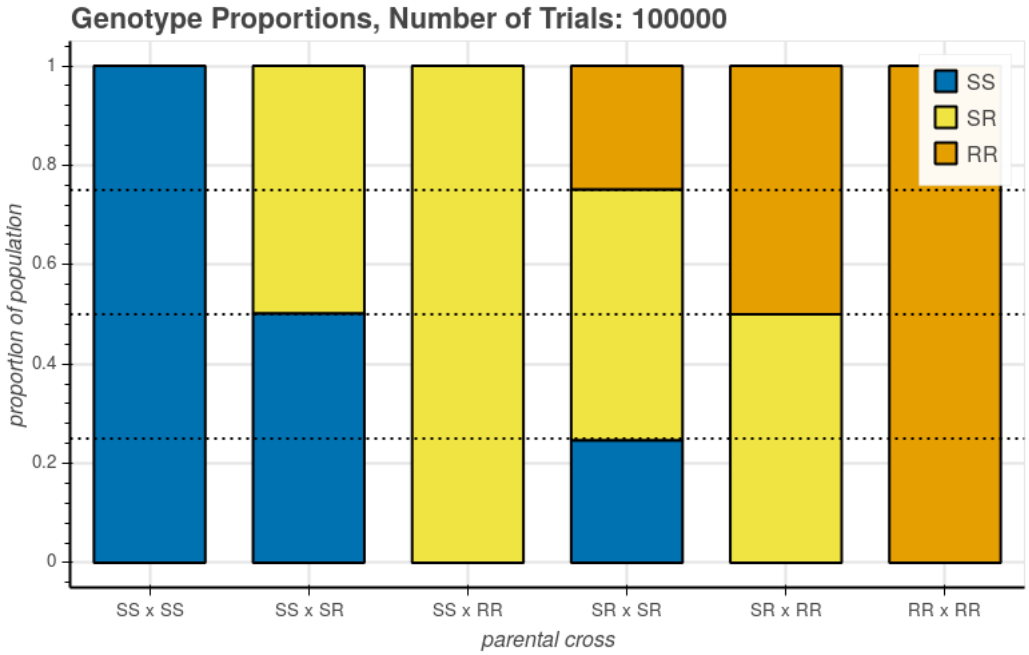


Table 2.2: Dominance Data, D

Data Range	Data Mean	Sources
$[0, 0]$	0.0	Niu *et al.* [80], Vélez *et al.* [117]
$[0.3, 0.19]$	0.11	Bernardi *et al.* [14], Niu *et al.* [80], Storer *et al.* [109]
$[0.24, 0.35]$	0.29	Farias *et al.* [44], Niu *et al.* [80]

where $-1 \leq d \leq 1$ is the degree of dominance. Observe that when $d = 1$, we get that $X_{SR}(1) = X_{RR}$, which is equivalent to saying that the allele R is *dominant*. Similarly, when $d = -1$ we have $X_{SR}(-1) = X_{SS}$, so this is the case when R is *recessive*. When $d = 0$ we get the case $X_{SR}(0) = \frac{X_{SS}+X_{RR}}{2}$, which is often called *incomplete dominance*.

In the literature, a myriad of different methods were used to estimate values similar to d. The methods used in fall armyworm studies (see [14, 44, 80, 109, 117]) to quantify the dominance of the resistant allele are described by Bourguet *et al.* [20].

These methods of estimating dominance use a modified version of equation (2.1):

$$X_{SR}(D) = X_{SS} + D\left(X_{RR} - X_{SS}\right), \tag{2.2}$$

where $D = \frac{d+1}{2}$ and $0 \leq D \leq 1$. This change to D moves the domain for dominance to $[0, 1]$ instead of $[-1, 1]$ for d. Notice that manipulation of D will be one of our primary modes of investigation; namely, we use the data means found in Table 2.2.

2.1.2 Initial Biomass

Biomass plays a central role in agent behaviors within our IBM, affecting growth, cannibalism, development, and survival. Thus, we require a closely related set of models for the initial biomasses of agents, one for each life-stage. Note that these models will be closely related to each other and to the biomass growth model described in Section 2.1.3.

2.1.2.1 Initial Egg Biomass

Fall armyworm females lay eggs in large clumps of eggs called *egg masses*. Typically, each female lays several egg masses over her reproductive lifetime, so we will model each of these egg-mass-laying events rather than the production of each individual egg within the egg mass. This will be modeled using the data for entire egg masses in Table 2.3.

The number of eggs N is drawn from Poisson distribution with mean $\lambda_{0,\text{egg}}$, and a total mass for the egg mass M is drawn from a normal distribution with mean $\mu_{0,\text{egg}}$ and standard deviation $\sigma_{0,\text{egg}}$; see Table 2.4 for values. Then, each constituent egg in the egg mass is assigned a biomass of M/N.

We assume that Bt-resistance is associated with a reduction in individual egg

Table 2.3: Egg Mass Experimental Data [123]

Trial	Fall Armyworm Strain	Mean Mass (mg)	Mean No. Eggs	Survival
1	corn × corn	10.4	164.4	0.91
	corn × rice	14.3	112.8	0.81
	rice × rice	12.8	269.6	0.95
	rice × corn	14.9	222.1	0.81
	$F_1(r \times c) \times F_1(r \times c)$	14.7	129.2	0.74
	$F_1(c \times r) \times F_1(c \times r)$	10.8	143.2	0.96
2	corn × corn	12.0	205.1	0.97
	corn × rice	11.3	197.2	0.99
	rice × rice	11.2	156.0	0.87
	rice × corn	14.4	276.4	0.88
	$F_1(r \times c) \times F_1(r \times c)$	10.3	154.7	0.98
	$F_1(c \times r) \times F_1(c \times r)$	13.7	200.2	0.90

mass. Hence, we use a $\mu_{0,\text{egg}}$ value for the RR genotype that this 80% smaller than the one for the SS genotype.

2.1.2.2 Initial Larva Biomass

For analysis and parameter fitting, we need to simulate the IBM starting with larvae directly; consequentially, we need to a model for the initial biomass of a larva. This is accomplished by scaling the total biomass distribution for egg masses by the average number of eggs in an egg mass. Hence, the initial biomass for larvae is sampled from a normal distribution with mean $\mu_{0,\text{larva}} = \mu_{0,\text{egg}}/\lambda_{0,\text{egg}}$ and standard deviation $\sigma_{0,\text{larva}} = \sigma_{0,\text{egg}}/\lambda^2_{0,\text{egg}}$ in Table 2.4.

2.1.2.3 Initial Mature Biomass

Pupae and adults do not change in biomass, including when pupae develop into adults; therefore, the same model may be used to initialize both of these agent's biomass. To create this model, we simulate many agents from the egg stage through the larval

Table 2.4: Initial Biomass Parameters

Parameter	Genotype	Value	Description
$\lambda_{0,\text{egg}}$		185.9	Mean number of eggs
$\mu_{0,\text{egg}}$	SS	12.6mg	Mean mass of egg mass
	RR	10.1mg	
$\sigma_{0,\text{egg}}$	SS	1.69	STD in mass of egg mass
	RR	1.08	
$\mu_{0,\text{larva}}$	SS	0.0676mg	Mean mass of new larva
	RR	0.0541mg	
$\sigma_{0,\text{larva}}$	SS	4.902×10^{-5}	STD in mass of new larva
	RR	3.137×10^{-5}	
$\mu_{0,\text{mature}}$	SS	156.588mg	Mean mass of pupa/adult
	RR	144.893mg	
$\sigma_{0,\text{mature}}$	SS	20.855	STD in mass of pupa/adult
	RR	21.255	

stage until pupation using the models from Section 2.1.3 and Section 2.1.8.2. This results in the distribution found in Figure 2.2. From this we calculate the mean $\mu_{0,\text{mature}}$ and standard deviation $\sigma_{0,\text{mature}}$ in Table 2.4, so that biomass is normally distributed.

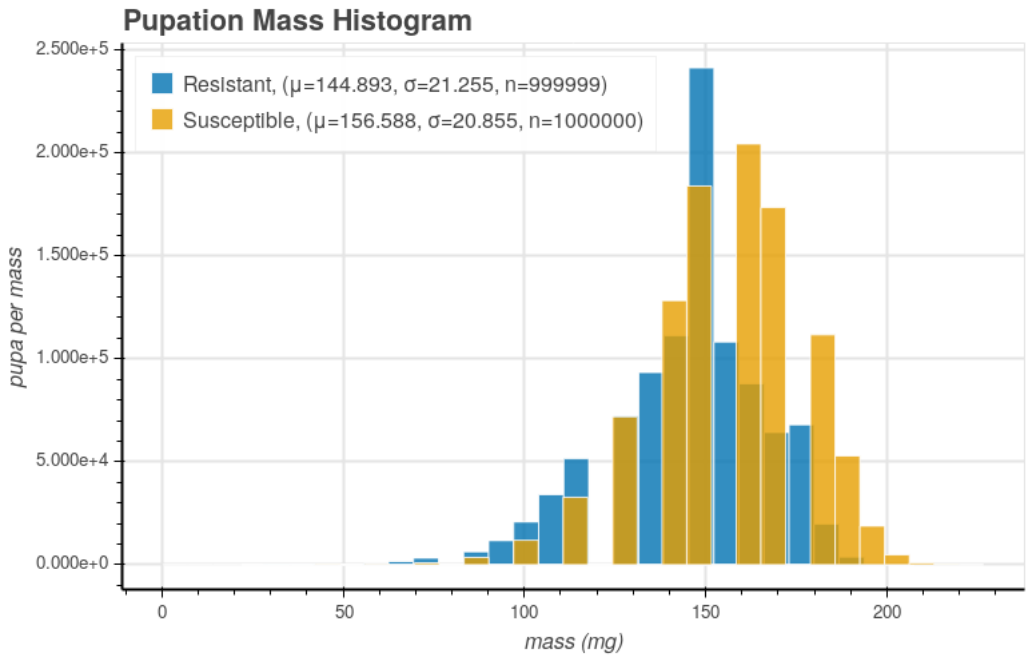
2.1.3 Biomass Growth

Biomass is one of the principal factors affecting how agents behave and interact [65]. Indeed, a primary difference between different genotypes is the difference in growth rates in biomass.

We have chosen to use the West *et al.* [120] model for biomass. This model has been shown to be widely applicable to a diverse collection of organisms [50, 100, 121], see Figure 2.3 for examples.

The West *et al.* model is derived by first considering the cost to an organism to maintain each of its current cells, and the cost of creating new cells and how this total

Figure 2.2: Simulated Distribution of Pupation Biomasses



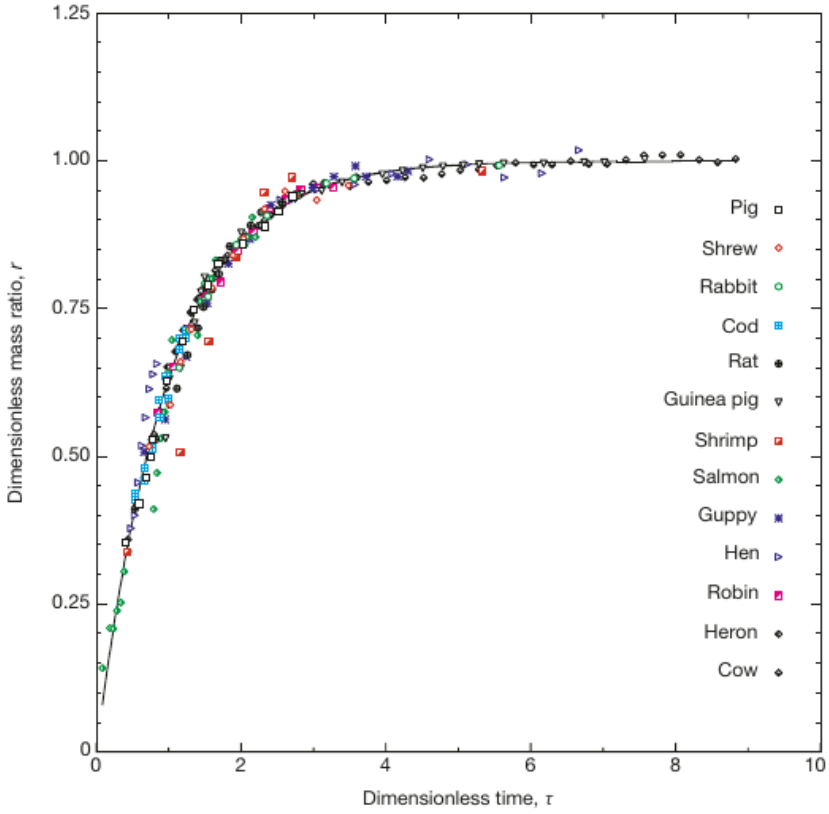
amount of energy must be balanced by the incoming rate of energy. Assuming that all of the cells within an organism require approximately the same amount of energy, we can express this relationship as:

$$B = NB_{\text{cell}} + E\frac{dN}{dt},$$

where B is the total incoming rate of metabolic energy to the organism, B_{cell} is the metabolic rate of a single cell, E is the metabolic cost of creating a new cell, and N is the number of cells in the organism at time t. Thus if B_{cell} and E are independent of the total mass of the organism $m = m_{\text{cell}}N$, then we can rewrite this as

$$\begin{aligned} m_{\text{cell}}B &= mB_{\text{cell}} + E\frac{dm}{dt} \\ \implies \qquad \frac{dm}{dt} &= \frac{m_{\text{cell}}}{E}B - \frac{B_{\text{cell}}}{E}m, \end{aligned} \tag{2.3}$$

Figure 2.3: West *et al.* [120] Model Applied to Several Species



where m_{cell}/E is the growth constant, and B_{cell}/E is the maintenance cost constant. For our model we assume that B is related to the amount of food in the form of biomass consumed by a larva in each step, as this relates directly to the amount of incoming energy.

The key to the West *et al.* model, is that for a given taxon of organisms one can find a constant B_0 so that the resting metabolic rate of the organism can be approximated by

$$B = B_0 m^{\frac{3}{4}}, \tag{2.4}$$

which gives rise to the ordinary differential equation

$$\frac{dm}{dt} = \alpha m^{\frac{3}{4}} - \beta m. \tag{2.5}$$

This equation approximates the mass of an organism at a given time, where $\alpha = B_0 m_{\text{cell}}/E$ is the growth rate in terms of input mass and $\beta = B_{\text{cell}}/E$ is the maintenance cost. The fractional power is key to the wide applicability of this model to many different organisms and is derived from metabolic studies [17].

2.1.3.1 Maximum Consumed Biomass

Consumption of biomass is simulated for each larva; this requires us to define a maximum consumed biomass model in order to facilitate biomass consumption modeling. This is modeled using

$$G_{\max}(m) = m^{\frac{3}{4}} \tag{2.6}$$

because it must be related to equation (2.4).

2.1.3.2 Growth

We base our model on equation (2.3) and equation (2.5) to arrive at

$$\frac{dm}{dt} = \alpha G - \beta m, \tag{2.7}$$

where α, β are the fit constants from the West *et al.* model, G is the current amount of biomass consumed by the larva, and m is the current biomass of the larva.

Typically, equations like equation (2.7) are approximated step to step using an ordinary differential equation stepping method (where G is given by the IBM at each

step). These methods are approximating solutions to an initial value problem of the form:

$$\begin{cases} \dfrac{dm}{dt} = f(t, m) \\ m(t_0) = m_0. \end{cases}$$

Conventionally, a model like equation (2.7) in an IBM would be approximated step to step using Euler's method:

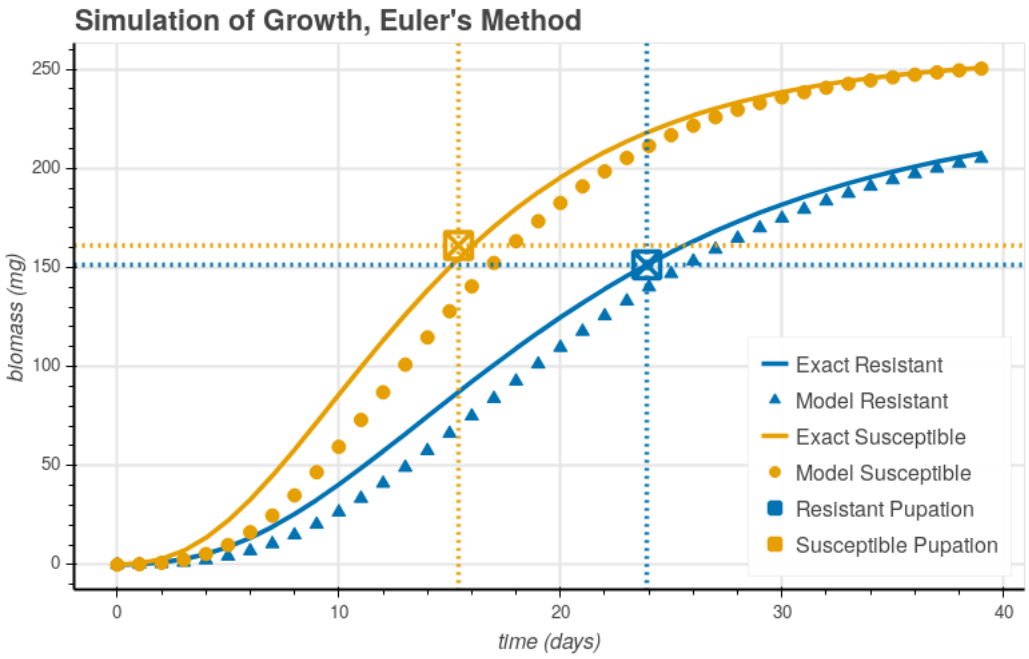
$$t_{n+1} = t_n + \Delta t \qquad\qquad m_{n+1} = m_n + \Delta t f(t_n, m_n),$$

where Δt is the time step size [52]. This is because we only have a measurement of G from one step to the next step, while other stepper methods usually take partial predictive steps and combine the results into a complete step. However, during initial experiments, it was found that in order to accurately recover the solution to the West *et al.* model under *ad. libitum* consumption, we needed a step size smaller than a natural time-scale for the IBM; therefore, when we use the natural time-scale we obtain poor results as demonstrated in Figure 2.4.

This leads us to use a slight modification to the commonly used Runge-Kutta 4 (RK4) method:

$$t_{n+1} = t_n + \Delta t \qquad\qquad m_{n+1} = m_n + \frac{1}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right),$$

Figure 2.4: Approximation of the West *et al.* Model Using Euler's Method



where

$$
\begin{aligned}
k_1 &= \Delta t f\left(t_n, m_n\right), \\
k_2 &= \Delta t f\left(t_n + \frac{\Delta t}{2}, m_n + \frac{k1}{2}\right), \\
k_3 &= \Delta t f\left(t_n + \frac{\Delta t}{2}, m_n + \frac{k2}{2}\right), \\
k_4 &= \Delta t f\left(t_n + \Delta t, m_n + k_3\right),
\end{aligned}
$$

and Δt is the time step size [52]. Note that here we have to take predictive steps $k1, k2, k3, k4$ some of which require information about G at times different than that of a complete step, which is the only information we get from the IBM.

To use RK4 in our model we need a way to approximate values for G at each of these predictive steps. Therefore, we modify equation (2.7), so that within the RK4 framework G can be (approximately) determined. We need only consider advancing

one step, so assume G_{food} is the biomass consumed by the larva over the step and m_{current} is the larva's biomass during the step. Using equation (2.6) we may define the fixed value

$$r = \frac{G_{\text{food}}}{G_{\max}(m_{\text{current}})} = \frac{G_{\text{food}}}{m_{\text{current}}^{\frac{3}{4}}}.$$

We can then approximate G for each of the predictive steps by assuming that G is a function of m, that is

$$G(m) \approx rG_{\max}(m) = rm^{\frac{3}{4}}.$$

Now we can use this and equation (2.7) to obtain the system:

$$\begin{cases} \dfrac{dm}{dt} = \alpha r m^{\frac{3}{4}} - \beta m \\ m(0) = m_{\text{current}}; \end{cases} \tag{2.8}$$

for which, we can apply a single step of RK4 to grow m for the next step of the IBM. Indeed, if we examine the growth of a larva with an *ad. libitum* food supply and appropriately calculated parameters from Section 2.1.3.3, we can see in Figure 2.5 that the solution points closely approximate the exact solution to the equation using the model's step size.

2.1.3.3 Growth Model Parameters

West *et al.* provides the continuous solution to equation (2.5):

$$\left(\frac{m}{M}\right)^{\frac{1}{4}} = 1 - \left[1 - \left(\frac{m_0}{M}\right)^{\frac{1}{4}}\right] \exp\left(-\frac{\alpha t}{4M^{\frac{1}{4}}}\right), \tag{2.9}$$

Figure 2.5: Approximation of the West *et al.* Model Using Runge-Kutta 4 Method

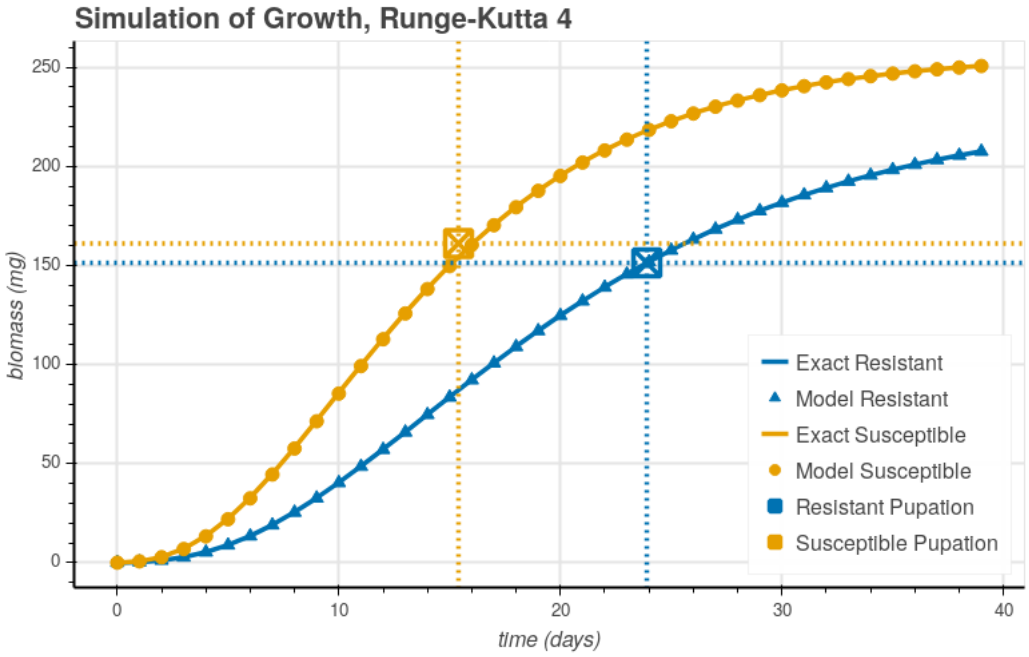


Table 2.5: Data Points for Fitting the West *et al.* Model

Genotype	time±SEM days	mass±SEM mg	Sample Size	Source
SS	8.0 15.6 ± 0.2	54.4 ± 3.7 156.0 ± 5.1	40	Veenstra *et al.* [114]
RR	10.0 24.3 ± 0.4	37.4 ± 3.1 145.4 ± 5.9	40	Prasifka *et al.* [93]

where m_0 is the initial biomass, $\beta = \frac{\alpha}{M^{\frac{1}{4}}}$, and M is the asymptotic maximum biomass for the organism.

We use $m_0 = \mu_{0,\text{larva}}$ from Table 2.4. Thus we need only two data points to estimate α, M in equation (2.9). As a starting point for susceptible genotypes's growth we use data from Veenstra *et al.* [114]. While Prasifka *et al.* [93] measures larval growth on suboptimal food sources, so we use this study's data as starting point for the resistant genotype's growth because we are hypothesizing that resistant genotype larvae will have delayed growth which is similar to a sub optimal food source. Note that there

Table 2.6: Biomass Growth Parameters

Parameter	Genotype	Value	Description
α	SS	2.070	Growth Rate
	RR	1.420	
β	SS	0.517	Maintenance Cost
	RR	0.365	
M	SS	256.4	Maximum Biomass
	RR	229.7	

is a later study by Vélez *et al.* [116] which reports data on both genotypes; however, its data places the growth curves too close together to be useful.

Ultimately we are estimating the maximum growth as opposed to just growth because we have variation up to $G_{\max}(m)$ of consumed biomass, and that $G_{\max}$ is used in the estimation of the parameters for growth. Thus growth will be bounded above at every point by our parameter fits using equation (2.5). Therefore, we assume the largest masses measured at the smallest times within the data intervals listed in Table 2.5 because this will give us the largest amount of growth step to step.

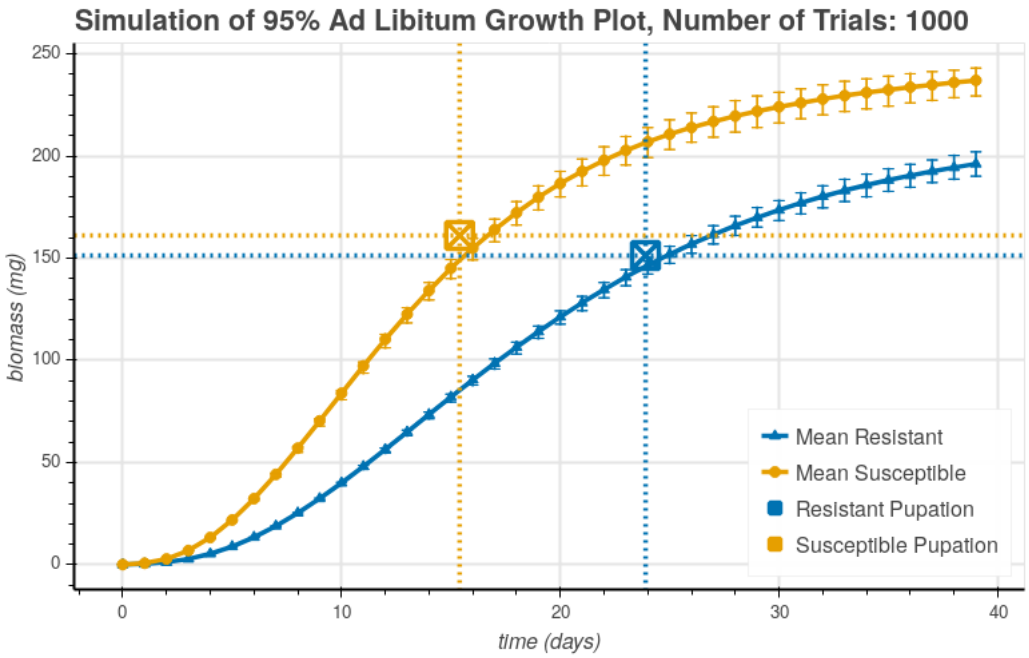
Using m_0 and the data in Table 2.5, we apply SymPy's [74] to equation (2.9). This results in fits for α, β, M as displayed in Table 2.6.

If we allow for initial variations in the initial mass of larvae, and an almost *ad. libitum* food supply (Section 2.1.5), the model nicely converges to the correct growth curves as in Figure 2.6a. Moreover, if we now additionally introduce variations in the food supply using models in Section 2.1.5, we start see delays in growth as evidenced by Figure 2.6b.

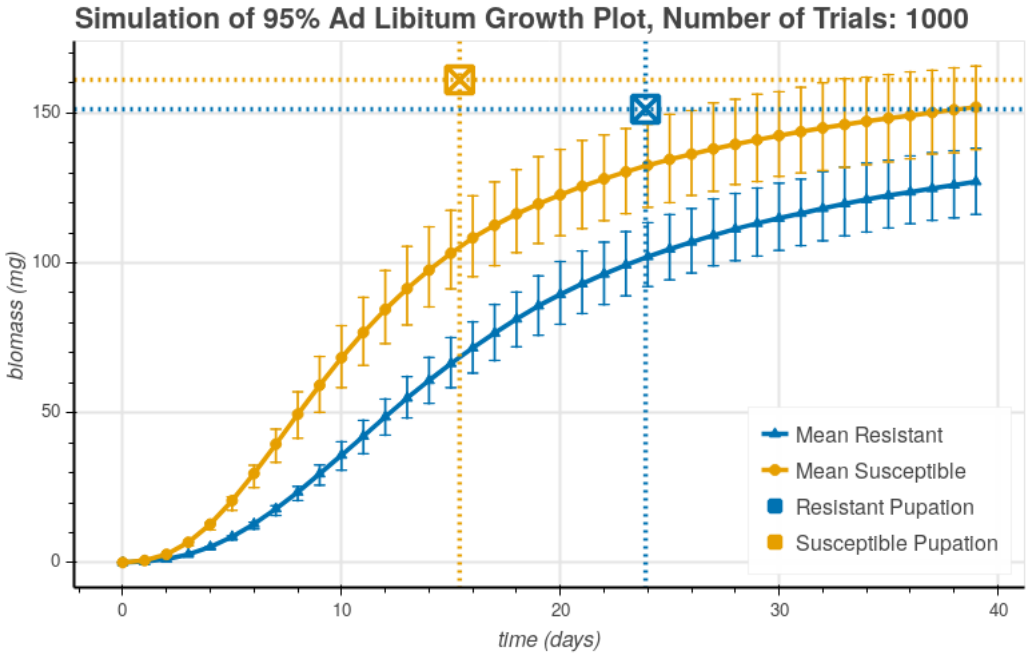
2.1.4 Cannibalism

Cannibalism represents the principal means of direct interaction among agents. Recall that we hypothesized SS genotype larvae grow faster than RR genotype larvae.

Figure 2.6: Stochastic Simulations of the West *et al.* Model



(a) 0.95 *ad. libitum* food supply



(b) 0.80 *ad. libitum* food supply

Furthermore, we assume larger larvae are more likely to prevail in cannibalistic interactions and larvae will always prevail in such interactions over eggs. With these observations in mind, we will model cannibalism.

2.1.4.1 Cannibalistic Encounters

Note that larvae move around within a grid, which is intended to model physical space (Section 2.2.1.1). For a cannibalistic interaction to occur, a larva must encounter some other agent. The maximum distance over which these interactions can occur is called the radius. To simplify our assumptions for the encounter model, we assume that grid-cells are sized so they are the largest area over which a larva can be expected to have a cannibalistic interaction during a single foraging step. This reduces the interaction radius to 0; that is, cannibalism will only happen between agents in the same grid-cell. Therefore, our encounter model is modeling how likely a larva is to encounter another agent within a given grid-cell. Note specifically that cannibalism only involves the egg masses and larvae within a grid-cell.

We choose to model the interaction of agents within a given grid-cell using an "ideal gas model" [58]. In these models, we assume the agents are particles moving randomly as if they are "gas" particles in a two dimensional plane. To begin, assume that all agents in a cell with area A have the same average speed v. We can then find, by integrating over all angles relative to a fixed agent, the average speed of any other agent relative to the fixed agent to be $4v/\pi$. Now suppose that the encounter detection distance is D; that is, the distance at which another agent must be within in order for an encounter to occur. Then the strip of of detection space swept out by any agent has width $2D$. Thus, relative to a fixed agent, every other agent sweeps out an area of $8Dv\Delta t/\pi$ over time Δt. If there are N agents, then collectively these agents have covered an area of $8NDv\Delta t/\pi$. Next, the number of these strips that

can cover a fixed individual within the cell's area follows a Poisson distribution, with mean given by

$$\lambda = \frac{8NDv\Delta t}{\pi A}. \tag{2.10}$$

Hence, the encounter rate for an agent is given by $\frac{8NDv}{\pi A}$. Since we are working with a Poisson distribution it follows that the probability that a given individual has no encounters is:

$$P(N) = \exp(-\lambda) = \exp\left(-\frac{8NDv\Delta t}{\pi A}\right). \tag{2.11}$$

So for a larva in a given grid-cell containing N other agents the probability that any encounter occurs is $1 - P(N)$, where $P(N)$ is given by equation (2.11).

The encounter rate λ in equation (2.10), has four assumed constants for our model, D, v, Δt, and A. We will group all of these constants together into a single parameter

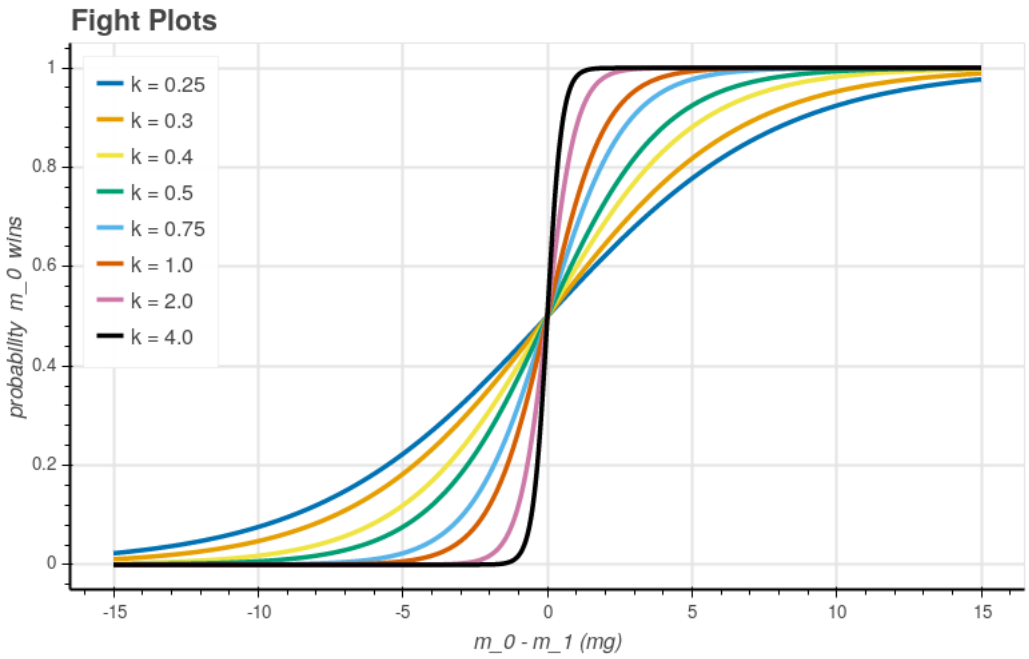
$$\rho_c = \frac{Dv\Delta t}{A} \tag{2.12}$$

which will control how much cannibalism can occur in a given forage step. Note that Δt depends on the foraging step, while A is dependent on the grid; meaning, ρ_c depends on the time and space scales chosen.

2.1.4.2 Cannibalistic Outcomes

When one larva attempts to cannibalize another larva, the outcome of the interaction is unknown because it is possible for either larva to triumph. This requires us to model the outcomes of cannibalistic encounters. Kokko proposed a model [64] for the direct competition of two similar organisms which differ in "size". Namely, if m_0, m_1

Figure 2.7: Cannibalistic Outcome Model for Various Values of k



are biomasses for two larvae then the probability that the larva with biomass m_0 is triumphs in over the larva with biomass m_1 can be modeled as

$$P(m_0, m_1) = P_L + \frac{P_H - P_L}{1 + \exp\left(-k(m_0 - m_1)\right)}, \tag{2.13}$$

where P_H is the probability of success if $m_0 \gg m_1$, P_L is the probability of success if $m_0 \ll m_1$, and k controls the slope of the transition from P_L to P_H. For our situation we expect that $P_L = 0$ and $P_H = 1$ because when a larva is much larger than the larva it is encountering we expect that larva to always win, such as when a larva near its pupation biomass encounters a newly hatched larva. Thus the only parameter needed for this model is k which controls the steepness of the 0 to 1 transition.

Hence, for a larva, with biomass m_0, approaching another larva, of biomass m_1, the outcome of a cannibalistic encounter is modeled using a Boolean choice with

Table 2.7: Percent Cannibalism Data [95]

Host	# Tests	# Larvae per Test	% Cannibalism	STD
Bean Plant	29	10	38.6	18.5
Corn Plant	31	10	17.7	16.3
Bean Disks	16	10	35.6	15.9
Corn Disks	16	10	32.6	17.3

probability given by:

$$P(m_0, m_1) = \frac{1}{1 + \exp\left(-k(m_0 - m_1)\right)} \tag{2.14}$$

between triumphing or not. In Figure 2.2, we see that the standard deviation in larval biomass is between 10 and 25; meaning that our transition should take place over an interval this wide. Examining Figure 2.7 we can see k needs to be between 0.4 and 0.75, so we use $k = 0.5$.

2.1.4.3 Fitting Cannibalism Parameters

Cannibalism is often understood at a population level as opposed to at an individual level. In Raffa's 1987 study [95] percent cannibalism was measured for the fall armyworm. This was accomplished by placing 10 third instar larvae in an enclosure for 4 days, during which deaths due to cannibalism where counted. We will estimate an initial value for ρ_c by replicating Raffa's experiments within our simulation. To do this we will assume our previous baseline parameter values and only adjust ρ_c, while replicating the conditions used, then comparisons with Raffa's data in Table 2.7 allow us to empirically estimate a value for ρ_c.

To replicate this experimental setup within our IBM's simulation we need additional information. First, we need the mean and standard deviation in biomass for

Table 2.8: Larval Instar Duration (days) Data [88]

Diet	1^{st}	2^{nd}	3^{rd}	4^{th}	5^{th}	6^{th}
Corn	3.3	1.7	1.5	1.5	2.0	3.7
Cotton	3.7	2.8	3.0	4.4	4.8	7.2
Soybean	3.2	2.7	2.3	2.6	2.9	9.9

third instar fall armyworms under our growth conditions. We can then use these in our initial biomass model for larvae to initialize the larvae for our experiment. To find the initial biomasses, note that Pitre and Hogg [88], reported the average instar durations, see Table 2.8. From this we estimate the third instar begins after 7.7 days (8 steps of the IBM). Thus in a similar fashion to how we found Figure 2.2, we simulate many larvae growing from hatching to this time under *ad. libitum* conditions and without any other behaviors or other processes, see Figure 2.8. Second, we establish a grid size for the cannibalism by iteratively adjusting grid sizes until we recovered ranges which included the values in Table 2.7, resulting in a 10 by 10 grid.

In examining the results of these simulation experiments in Figure 2.9, we find that $\rho_c = 0.07$ recovers the percent cannibalism from Raffa's study resembling our IBM. Also, note that there are no differences among genotypes and that percent cannibalism measurements are density limited for large values of ρ_c (Figure 2.9a).

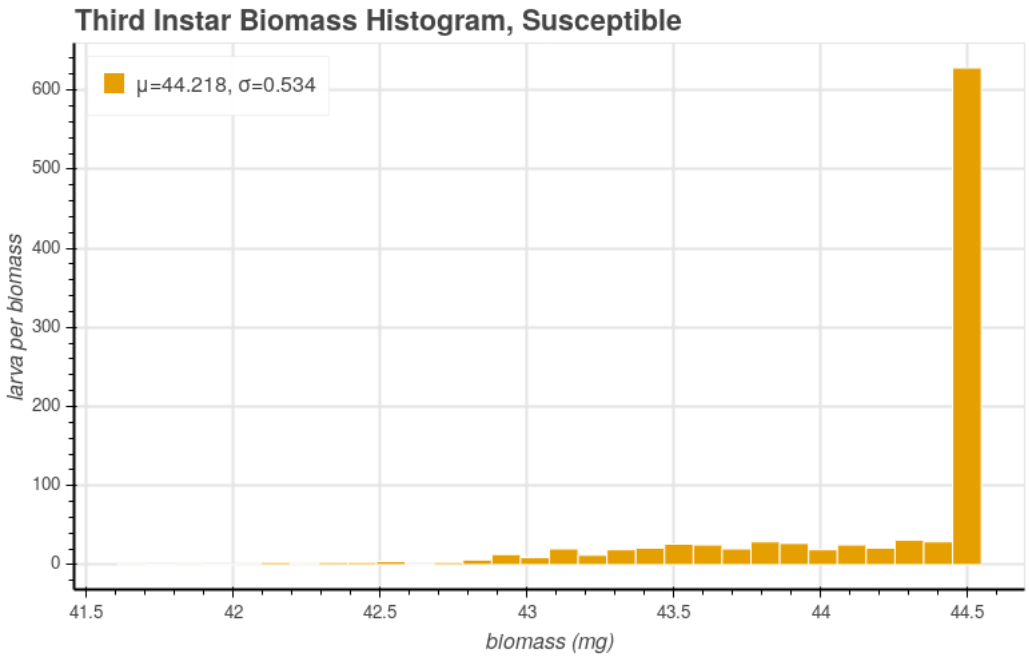
2.1.5 Biomass Consumption

Larvae consume biomass from two sources: plant biomass and cannibalism victims.

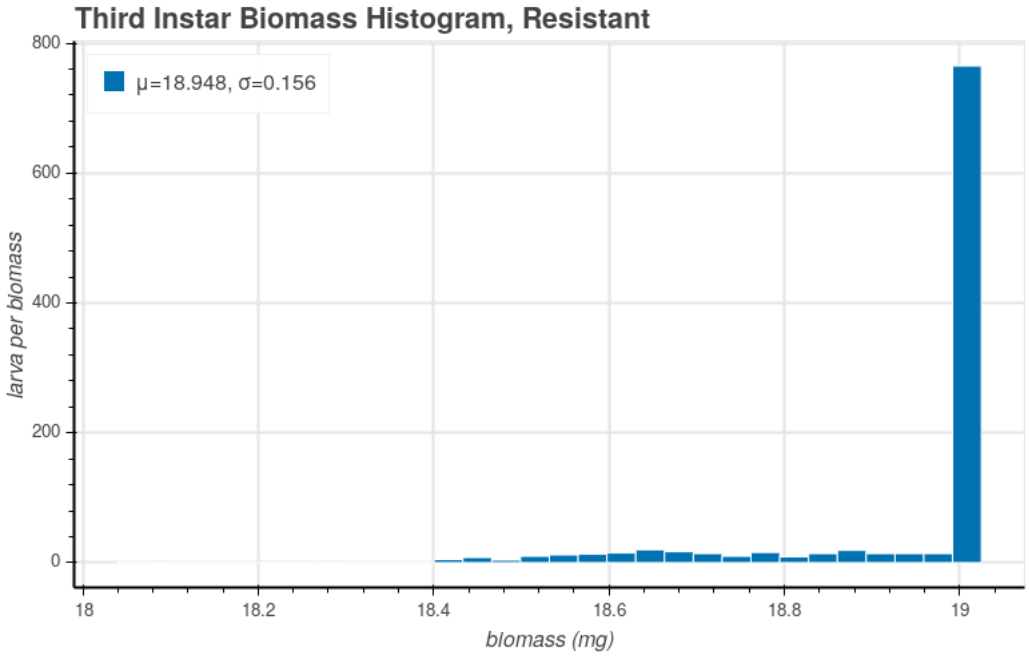
2.1.5.1 Plant Biomass Consumption

The main source of biomass for larvae comes from the environmental substrate, which is the plant that the larva lives upon. Under the idealized conditions larvae will fill its gut; that is, it will consume the amount of biomass governed by equation (2.6). In

Figure 2.8: Simulated Distribution of Third Instar Fall Armyworm Biomass

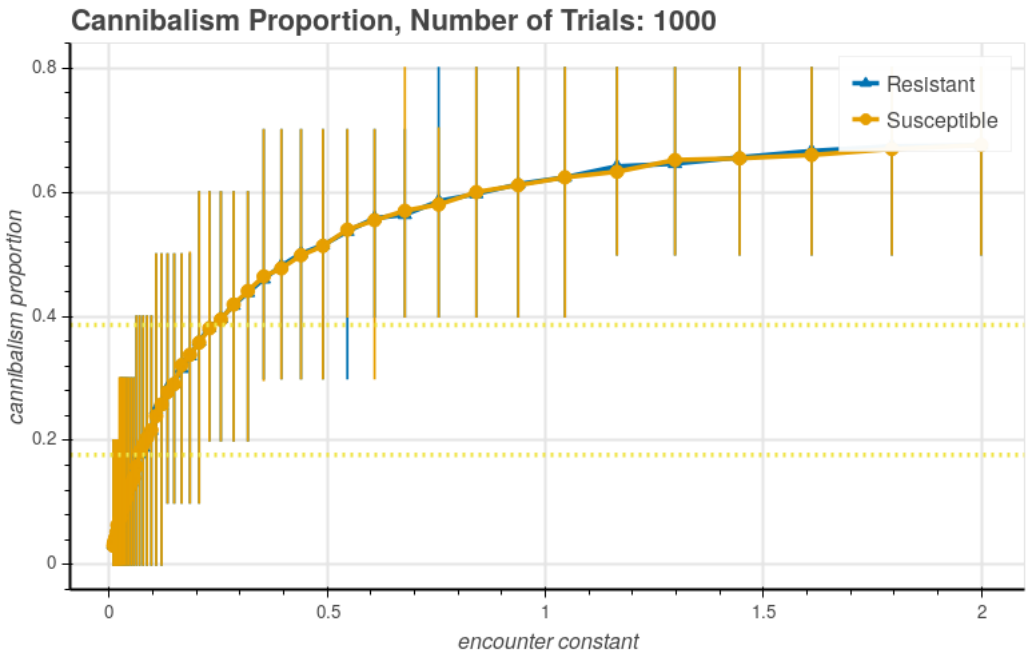


(a) SS genotype

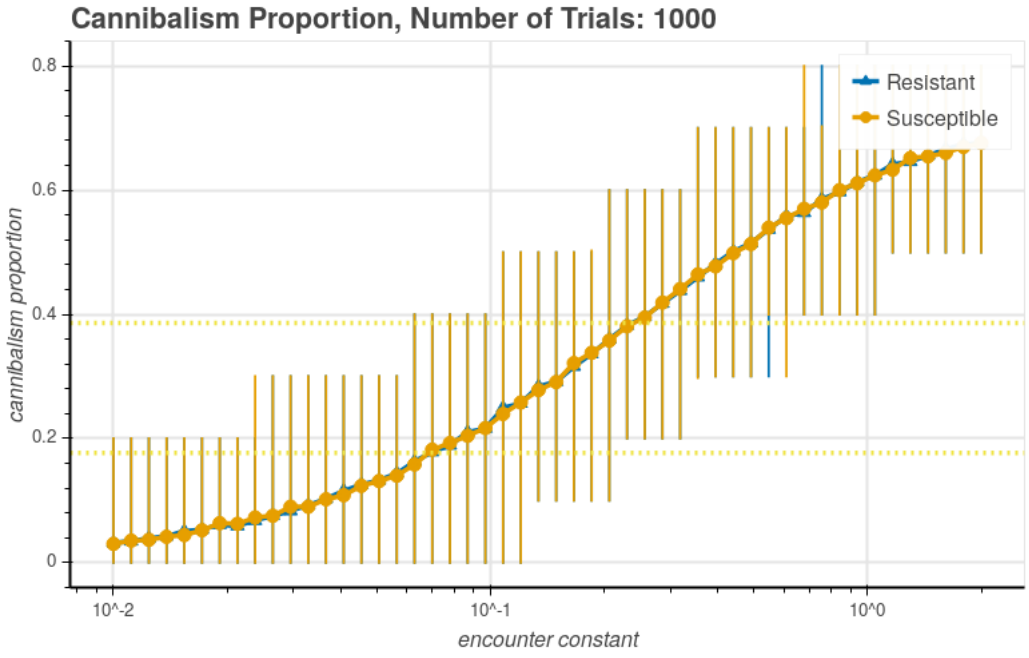


(b) RR genotype

Figure 2.9: Cannibalistic Encounter Constant Simulations



(a) ρ_c vs % cannibalism



(b) $\log(\rho_c)$ vs % cannibalism

these *ad. libitum* conditions, biomass consumption is modeled based on equation (2.6), where:

$$G(m) = \frac{1}{t_f} G_{\max}(m) = \frac{m^{\frac{3}{4}}}{t_f}, \tag{2.15}$$

where t_f is the number of repeats of the forage per step.

If instead we are in a food scarce environment, then larvae cannot gain $G_{\max}$ amount of food. This is modeled by assuming that there is a factor $0 \leq \theta \leq 1$, which defines how scarce food is to the larva. In this case we assume that the amount of food available, at each repeat of foraging, to be normally distributed with mean found using equation (2.15) and θ,

$$\mu_{\text{scarce}}(m) = \theta G(m) = \frac{\theta m^{\frac{3}{4}}}{t_f} \tag{2.16}$$

and standard deviation σ_{scarce}. Thus consumption in a food scarce environment will be modeled via sampling from a normal distribution defined by μ_{scarce} and σ_{scarce}. Note that extreme starvation occurs when $\theta = 0$ and approximately *ad. libitum* consumption occurs when $\theta = 1$.

2.1.5.2 Cannibalism Biomass Consumption

Larvae also incorporate biomass from victims of successful cannibalism events. This requires us to model the biomass they can consume from the victim. Since, we have t_f possible foraging repeats, we assume the larvae consumes biomass from the victim following the *ad. libitum* amount in equation (2.15). However, we are constrained by the total amount of biomass of the victim, m_{victim}, as an upper bound on the consumed amount. Hence the amount of biomass consumed by a cannibalism encounter is given

by

$$G(m, m_{\text{victim}}) = \min\left[\frac{m^{\frac{3}{4}}}{t_f}, m_{\text{victim}}\right]. \tag{2.17}$$

Note that we can impose a scarcity θ (distinct from the previous one) on this just as we did for the environmental biomass by multiplying G by θ to define $\mu_{\text{scarce}}(m, m_{\text{victim}})$. However, to be consistent with the m_{victim} upper limit on biomass we need to instead draw specifically from a truncated-normal distribution (with standard deviation σ_{scarce} possibly distinct from the previous value), with truncation upper bound given by m_{victim}. Typically, we will assume that cannibalistic consumption is of the *ad. libitum* form given by equation (2.17).

2.1.6 Movement Models

Movement plays a role in cannibalism for larvae and reproduction for adults. We make use of the same basic model for both larvae and adults; however, note that the parameters may be different. Here we employ a Lévy flight [97, 98] model for movement on a grid. Lévy flights are similar to random walks, with the difference being that in a Lévy flight agents are not bound to a fixed movement distance. Recall that in a random walk, an agent randomly selects a new grid-cell at random from among all the grid-cells at some fixed radius (typically 1) away from its current grid-cell. In a Lévy flight, by contrast, an agent draws a radius from some distribution, which typically is heavy-tailed (this is unnecessary in practice [81]), and then chooses a random grid-cell from those grid-cells at the selected radius. One can think of a Lévy flight as moving some distance drawn from a distribution in a uniformly random direction instead of moving in a uniformly random direction a fixed distance.

Lévy flights are often used to model predator movements or foraging movements

Table 2.9: Movement Parameters

Parameter	Agent	Value	Description
x_m	Larva	1	Peak distance
	Adult	1	
a	Larva	10	Shape of tail
	Adult	1	

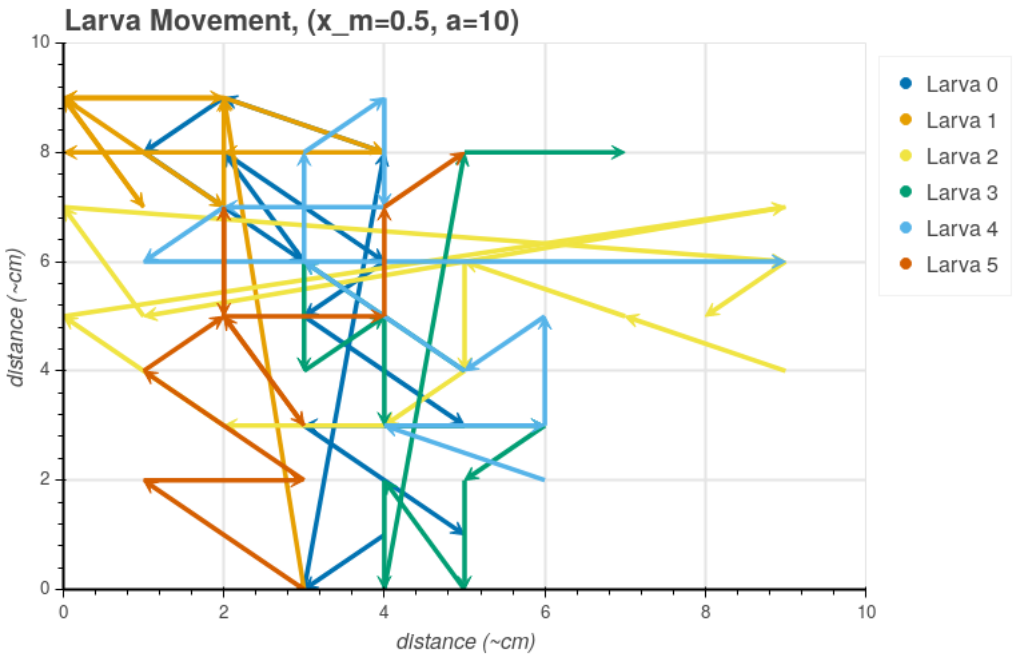
because they tend to create clumps of short distance movements followed by occasional long distance movement (see Figure 2.10), which is why we use Lévy flights. The larvae tend to move around in one area for a time exhausting the food supply and then jump to a new locale, while adult females will tend to search several nearby plants for suitable places to lay eggs and then jump to another area.

Classically, a Lévy fight is based on a Pareto distribution [90], which is what we use for the underlying distribution in both cases. Typically, Pareto distributions are parameterized by the peak x_m and the shape factor a. Note that just as with the cannibalism encounter model in Section 2.1.4.1, the ultimate parameters for the Pareto distributions will be tied directly to the grids (Section 2.2.1.1) and the time-scale for movement (Section 2.2.3.1). Moreover, there is little data on the specifics of either type of agent's movement patterns [83, 126, 127]. Thus we have taken liberties with the parameters of the two distributions via adjusting parameters and analyzing plots of movements; see Figure 2.10a for larvae, Figure 2.10b for adults, and Table 2.9 for the resulting parameters.

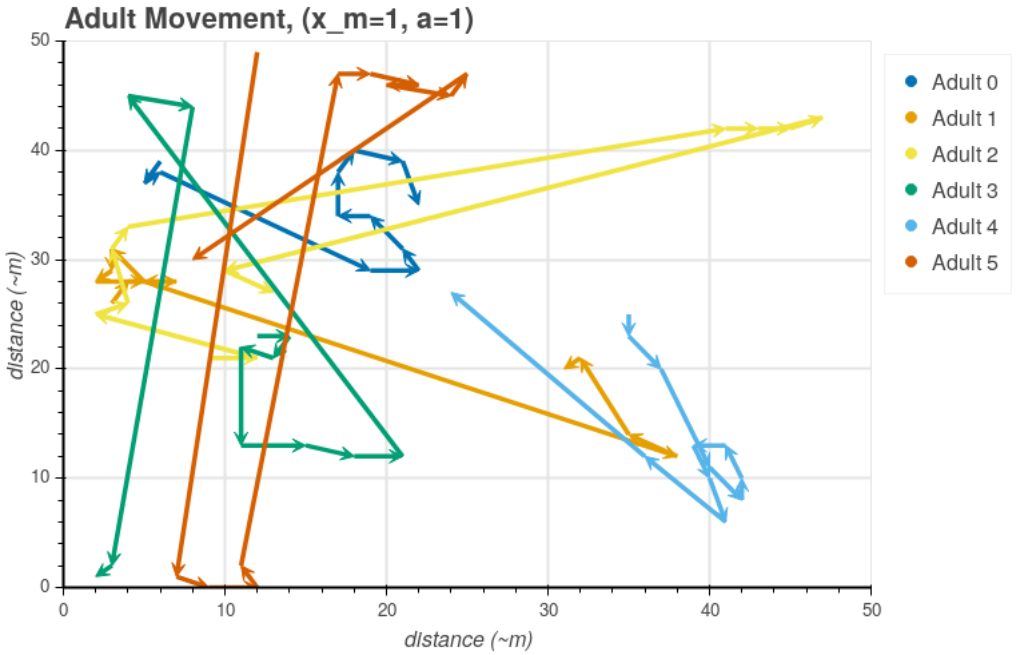
2.1.7 Reproduction

The key to closing the generational cycle within our IBM is reproduction; these are adult mating and female egg mass laying. Mating is an interaction between adult agents of differing genders; therefore, in addition to modeling encounters, we must

Figure 2.10: Simulations of Lévy Flights



(a) Several larva Lévy flights



(b) Several adult Lévy flights

also model the assignment of sex to agents. Egg mass laying is the process from which the next generation of agents emerges.

2.1.7.1 Mate Encounters

Recall we modeled cannibalistic encounters between larvae and other agents using an "ideal gas model" [58]. Here, we again make use of this model, but assume a non-zero radius R. This will not significantly impact the calculations for the model derived in Section 2.1.4.1; in fact, the larger radius only affects the scaling of the encounter rate constant. Thus, we model the probability that a female adult encounters a male adult using the model based on equation (2.11):

$$P(N) = 1 - \exp\left(-\rho_r N\right), \tag{2.18}$$

where N represents the number of male adults within radius R of the female in question and ρ_r represents the mate encounter rate constant. The determination of when an unmated female adult encounters a male adult is made via a Boolean choice with probability given by equation (2.18). If an encounter occurs, then a mate is selected uniformly at random from among those adult males within radius R of the female.

Note that in Sparks's [106] description of the fall armyworm's mating habits it is suggested that after signaling readiness to mate, females generally *wait for at least two males to approach her before she makes a choice of mate.* It is unclear how she chooses; hence, our choice of randomly mating. Moreover, we have little data beyond this observational description to base ρ_r on. Thus we arbitrarily assume to that ρ_r is

the value such that $P(2) = 0.5$ in equation 2.18. Therefore, we have

$$\rho_r = \frac{\ln(2)}{2} \approx 0.347. \tag{2.19}$$

Noting that ρ_r's effects are extremely dependent on the radius R chosen.

Here Sparks observes that males have been found to respond to female calls at up to 40 feet, but the response distances are heavily influenced by environmental factors, such as temperature and wind velocity. Thus we again make an arbitrary guess of $R = 2$ grid units.

2.1.7.2 Agent Sex

Since mating among adult agents occurs between a "female" and a "male", we must model the assignment of sex to agents. Note that sex is only assigned to adult agents because we assume sex only matters during the adult life-stage. We model this with a Boolean choice with probability P_{sex}, between being female or male. Dangal and Huang [34] found that the fall armyworm has a roughly equal distribution of both genders, so we assume $P_{\text{sex}} = 0.5$.

2.1.7.3 Fecundity

Each adult female only produces a limited number of egg masses over her lifetime. Moreover, the number of such egg masses is both limited per day and should decrease over time. Note that we are specifically separating fecundity from oviposition, which is modeled in Section 2.1.7.4. Our data on observed oviposition over time for the fall armyworm comes from Pencoe and Martin [85].

Fecundity is modeled as an exponentially decaying number of egg masses as females age, illustrated in Figure 2.11. So the average number of egg masses produced

Figure 2.11: Oviposition Data [85] Data Plot

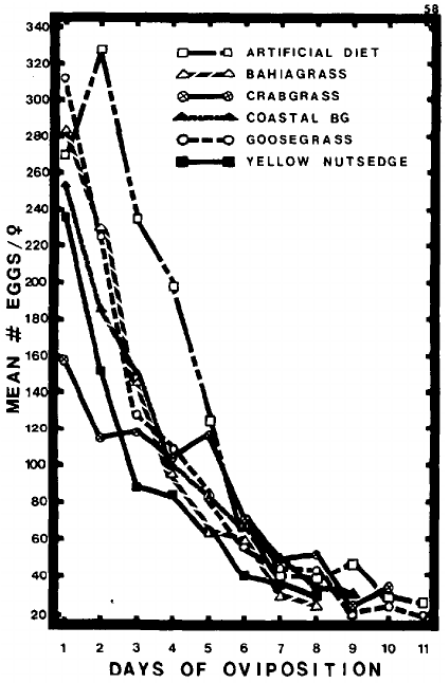


Figure 2.12: Female Fecundity

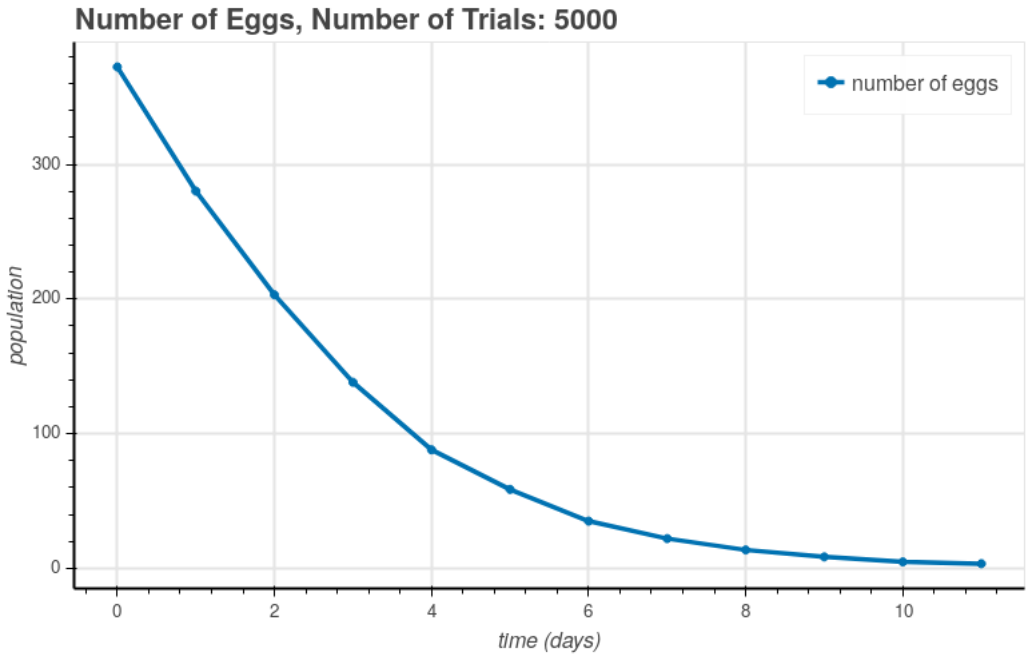


Table 2.10: Fecundity Parameters

Parameter	Value	Description
$N_{\max}$	2	Max # egg masses
ν	0.5	Decay rate

per day N_{egg}, is given by

$$N_{\text{egg}}(t_{\text{age}}) = \frac{2N_{\max}}{1 + \exp(\nu t_{\text{age}})}, \tag{2.20}$$

where t_{age} is the number of model steps (age) since becoming an adult, $N_{\max}$ is the maximum number of egg masses produced on average, and ν is the decay rate.

Using the data in Pencoe and Martin together with the egg mass data from Whitford *et al.* [123] we can empirically estimate $N_{\max}$ and ν, using curves like those in Figure 2.12 (see Table 2.10 for parameter values). Equation (2.20) models the average number of egg masses that an adult female can lay in a given step. Thus fecundity of a female, with age t_{age}, during a given step is modeled by drawing from a Poisson distribution with mean given by $N_{\text{egg}}(t_{\text{age}})$ from equation (2.20).

2.1.7.4 Density and Oviposition

In order to improve the computational tractability of the IBM, we have introduced a density dependent egg laying model.

Adult females will only invest in egg mass laying where the eggs have a high probability of success. Since larvae are highly cannibalistic, high densities of competitors (other fall armyworm larvae and eggs) are detrimental to the ability of the eggs to survive.

Our model is drawn from Louda *et al.* [68], in which the probability of egg laying is regulated by the number of eggs already present. In our case, we assume that the

Table 2.11: Oviposition Density Parameters

Parameter	Value	Description
η	0.45	Scale factor
γ	4	Shape factor

probability of an egg mass being oviposited in a grid-cell depends on the number (density) of other egg masses and larvae already present. Using the Louda *et al.* model:

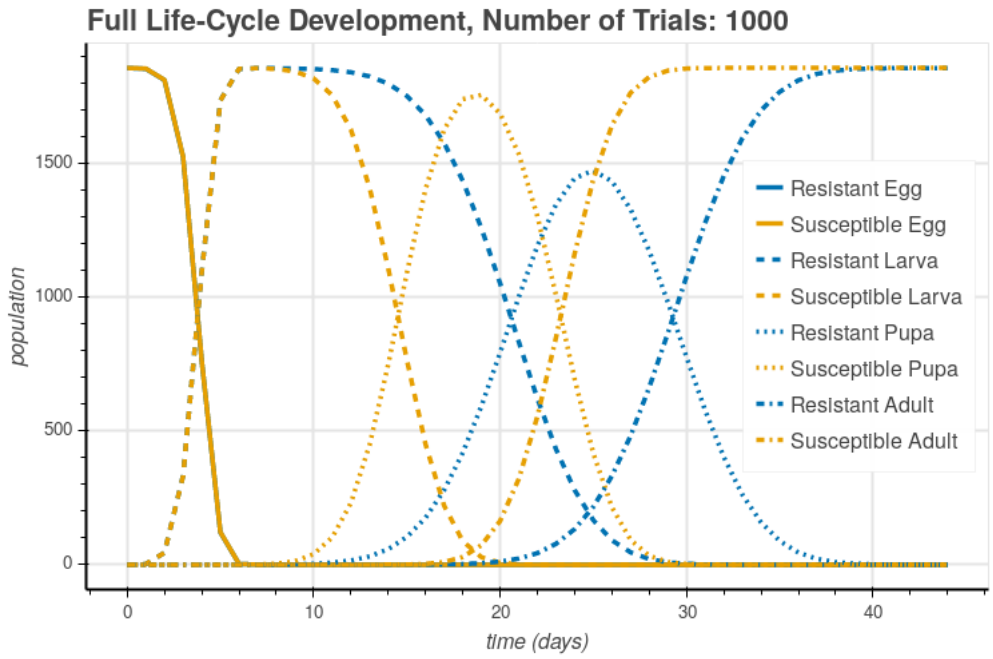
$$P(N) = \exp\left[-\left(\frac{N}{\eta}\right)^{\gamma}\right], \tag{2.21}$$

where N is the number of egg masses and larvae, η is a scale parameter, and γ is a shape parameter, we compute the probability of oviposition using the local population density. For computational reasons we choose η and γ so that on average the number of egg masses per plant approaches 1 rapidly when the population density of fertile females is high. The chosen parameter values are reported in Table 2.11.

2.1.8 Development

Individual fall armyworms progress through a series of four discrete stages: egg, larva, pupa, and adult. In each life-stage, the behaviors, interactions, and tracked properties of the fall armyworm agent change. Thus there are development models: egg-to-larva, larva-to-pupa, and pupa-to-adult. If we combine the development models with the growth model from Section 2.1.3 *ad. libitum*, and start with initial biomasses from Section 2.1.2 we obtain the developmental cycle in the absence of all other behaviors in Figure 2.13.

Figure 2.13: Single Generation of Development



2.1.8.1 Egg-to-Larva and Pupa-to-Adult Development

Both eggs and pupae behave in similar ways; these life-stages are governed by internal processes which require some amount of time to complete, after which the organism develops. Consequentially, we will model development in these life-stages using t_{age}, the number of time steps elapsed since the start of the agent's life-stage. Experimental data on fall armyworms development is given as average amount of time spent in a life-stage. Using this data we generate normal distributions for the time spent in life-stages, and then consider the cumulative probability function (CDF). This CDF is the function of t_{age} whose output is the probability of a developmental transition occurring at t_{age}, formulated as:

$$P(t_{\text{age}}) = \text{CDF}(t_{\text{age}}) = \frac{1}{2}\left[1 + \text{erf}\left(\frac{t_{\text{age}} - \mu}{\sigma\sqrt{2}}\right)\right], \tag{2.22}$$

Table 2.12: Pupa Duration Experimental Data

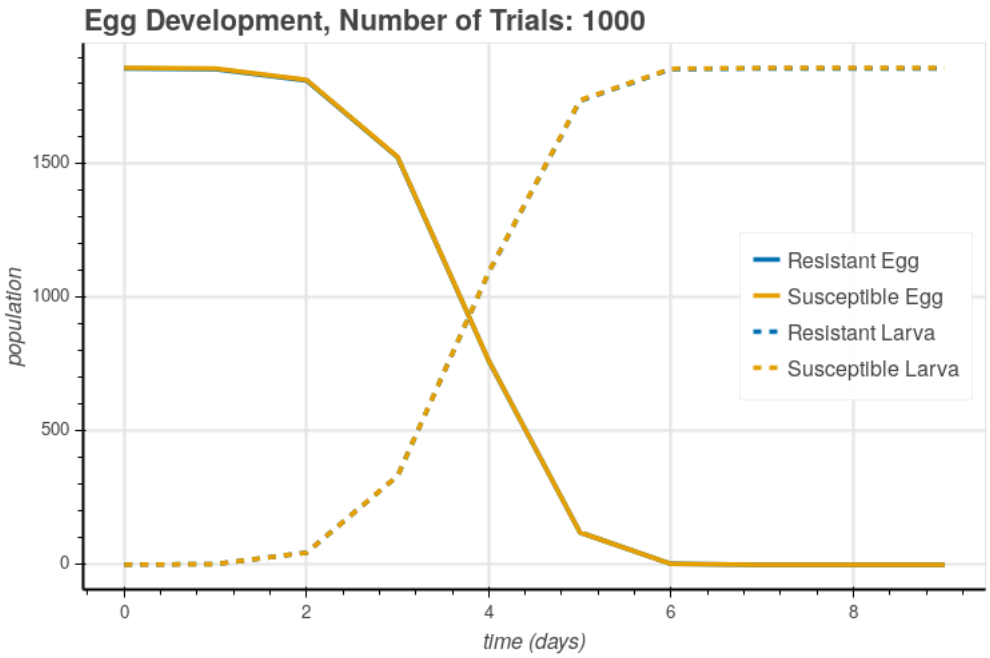
Experiment	Duration (days)	Source
Corn Diet	9.0	Pitre and Hogg [88]
Cotton Diet	10.8	
Soybean Diet	9.6	
DP5415 Male	8.43	Adamczyk *et al.* [1]
DP5415 Female	7.88	
DP 451 B/RR Male	8.25	
DP 451 B/RR Female	7.29	
NuCOTN 33B Male	8.05	
NuCOTN 33B Female	7.20	

where μ is the mean, σ is the standard deviation, and erf denotes the Gauss error function [103]. Hence, the egg-to-larva and pupa-to-adult developmental models can be defined as Boolean choices during each model time step whose probability is given by the $P(t_{\text{age}})$ from equation 2.22, between developing or not.

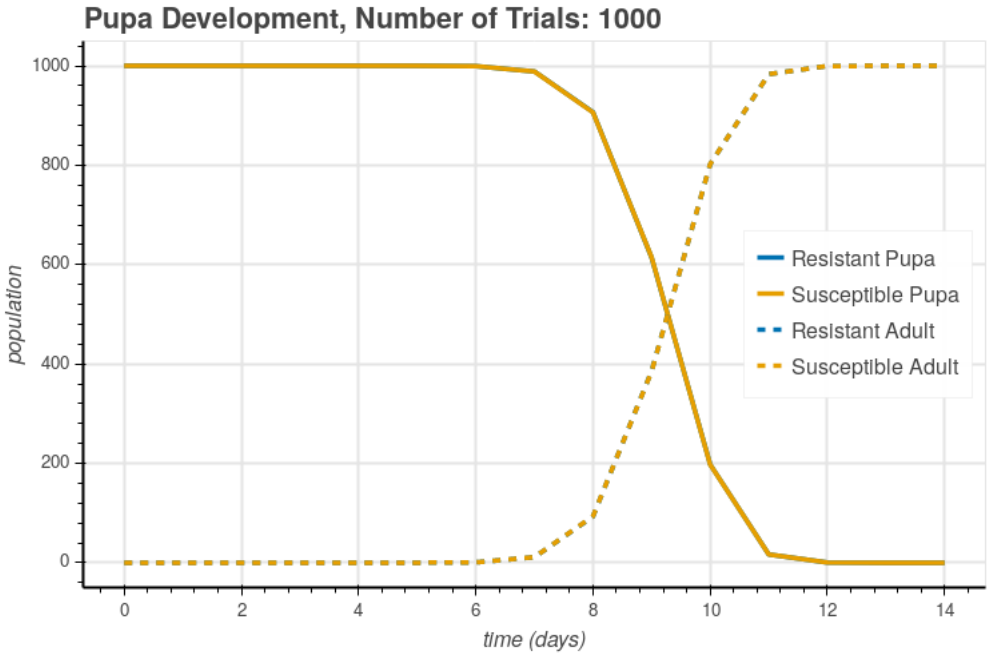
According to a review of fall armyworm biology [106], larvae typically emerge from eggs $2-4$ days after oviposition, giving a life stage duration distribution with mean, $\mu_{\text{egg}} = 3$ days, and standard deviation, $\sigma_{\text{egg}} = 1$. To see the effect of this model on the population over time see Figure 2.14a noting that both the process is genotype independent (so both genotypes should follow the same curve).

Pitre and Hogg [88] and later in Adamczyk *et al.* [1] measure pupal duration, which is summarized in Table 2.12. Assuming each of these experiments represents a single data point, we can find a mean μ_{pupa} and standard deviation, σ_{pupa} for the pupal life-stage duration distribution. Figure 2.14b plots the progression from pupa to adult as time progresses. Note again that this process is genotype independent, so both genotypes should produce the same curve. All resulting developmental parameters are summarized in Table 2.13.

Figure 2.14: Simulations of Egg and Pupal Development



(a) Egg-to-larva development



(b) Pupa-to-adult development

Table 2.13: Developmental Parameters

Parameter	Genotype	Value	Description
μ_{egg}		3 days	Mean egg duration
σ_{egg}		1	STD in egg duration
μ_{pupa}		8.5 days	Mean pupa duration
σ_{pupa}		1.08	STD in pupa duration
μ_{larva}	SS	172.0mg	Mean pupation mass
	RR	176.4mg	
σ_{larva}	SS	32.26	STD in pupation mass
	RR	37.31	

2.1.8.2 Larva-to-Pupa Development

We conceptualize larva-to-pupa development differently from egg-to-larva or pupa-to-adult development. In particular, we consider larva-to-pupa development in terms of gaining enough biomass to pupate as opposed to allowing sufficient time to elapse. This is because larvae are dominated by their biomass growth, so after sufficient growth, the larva is in a position to pupate.

Thus we can begin to model larva-to-pupa development by supposing the pupation biomass of larvae is normally distributed with mean μ_{larva} and standard deviation σ_{larva}. Just as in Section 2.1.8.1, we consider the CDF (equation (2.22)) of this distribution using the biomass m as the independent variable. We define the larva-to-pupa development model to be a Boolean choice with probability determined by this CDF between developing or not. Combining this model with the growth model from Section 2.1.3, we can plot the population of larvae as they pupate in Figure 2.15.

Recall that larval growth was modeled using data from Veenstra *et al.* [114] to model the SS genotype's growth, while data from Prasifka *et al.* [93] was used to model the RR genotype's growth (see Table 2.5). However, if we directly use this data for the parameters, we do not obtain the distribution defined by the data when

Figure 2.15: Simulation of Larval Development

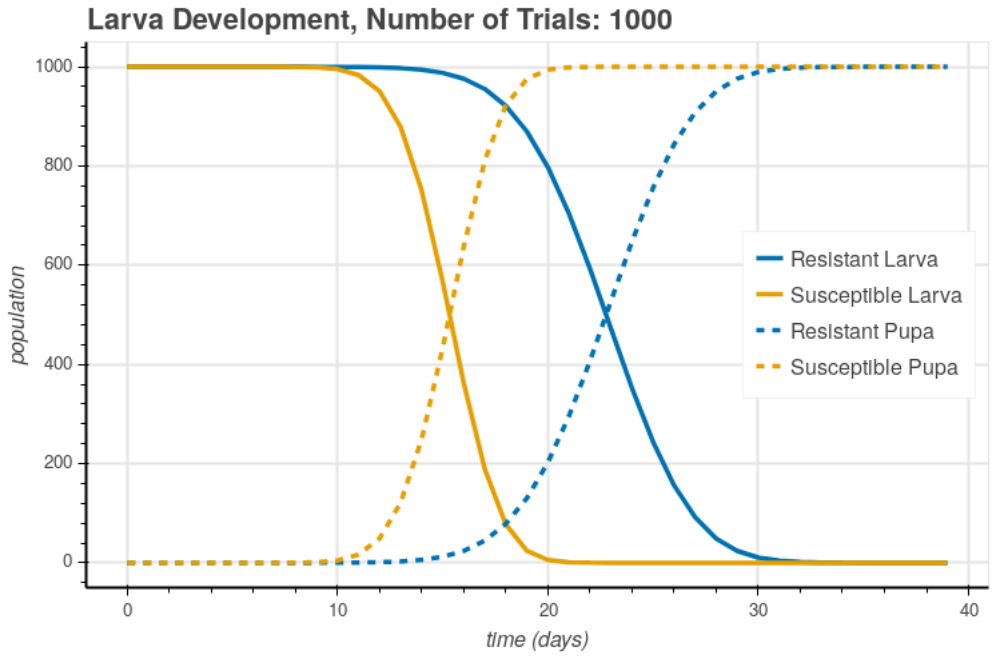


Figure 2.16: Simulated Pupation Distribution With and Without Corrections

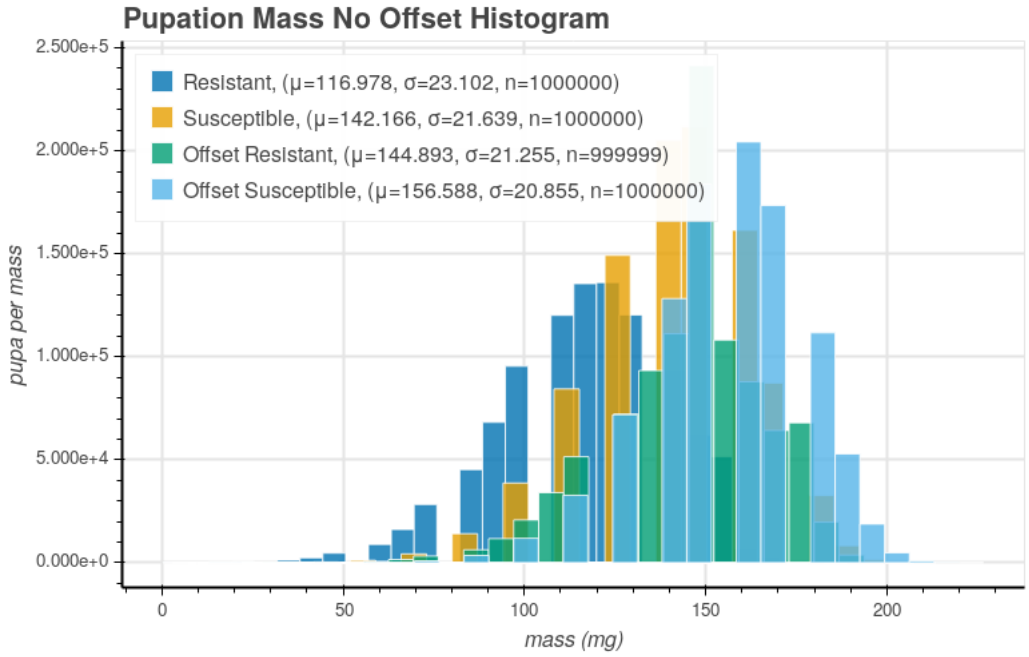
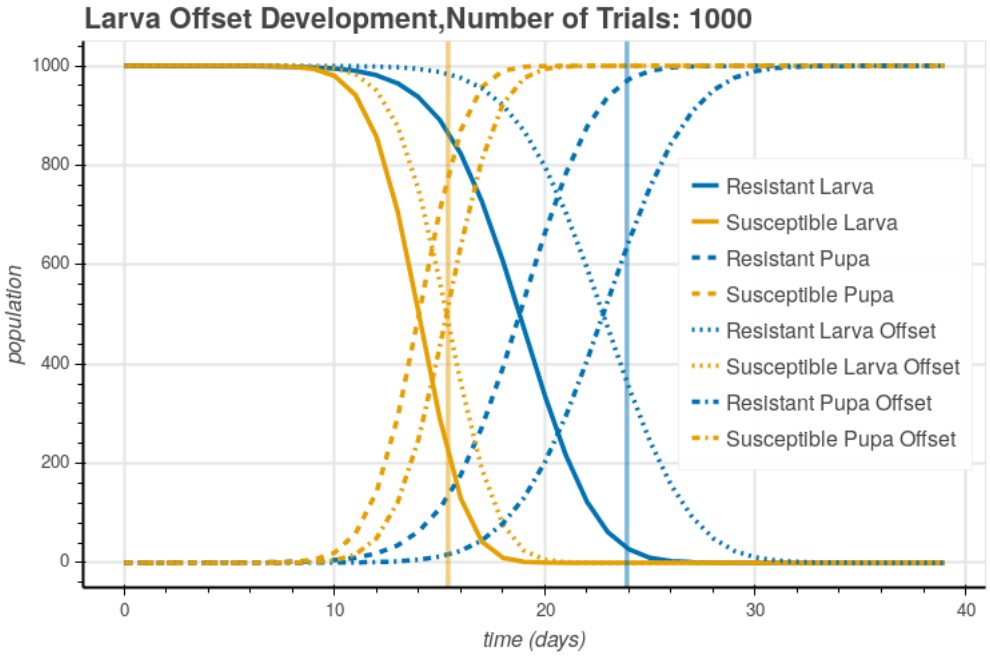


Figure 2.17: Simulation of Larval Development With and Without Corrections



we simulate. Instead we get a distribution shifted to lower values in Figure 2.16.

This discrepancy caused by discretization effects. To correct this process we empirically offset the mean of the model until we recover the original data's distribution as demonstrated by the offset distributions in Figure 2.16. Note that when we plot the corrected and non-corrected simulations together in Figure 2.17, we see the offset parameters recover the pupation time (vertical lines) more closely. Thus we have reasonable confidence in our empirical adjustments, yielding the parameters in Table 2.13. Note that the distribution in Figure 2.2 and development plot in Figure 2.19 use these corrections.

2.1.9 Survival

All populations of organisms have some level of mortality resulting from uncertainties in their lives; thus we need to consider this for the fall armyworm.

2.1.9.1 Egg and Pupal Survival

Egg and pupal survival models apply a *background* mortality rate. We assume this mortality will be independent of genotype, Bt-state, and biomass. So there is a fixed probability P_{step} of survival to the next step of the model. Thus survival is modeled using a Boolean choice with probability P_{step} between survival to the next step or death.

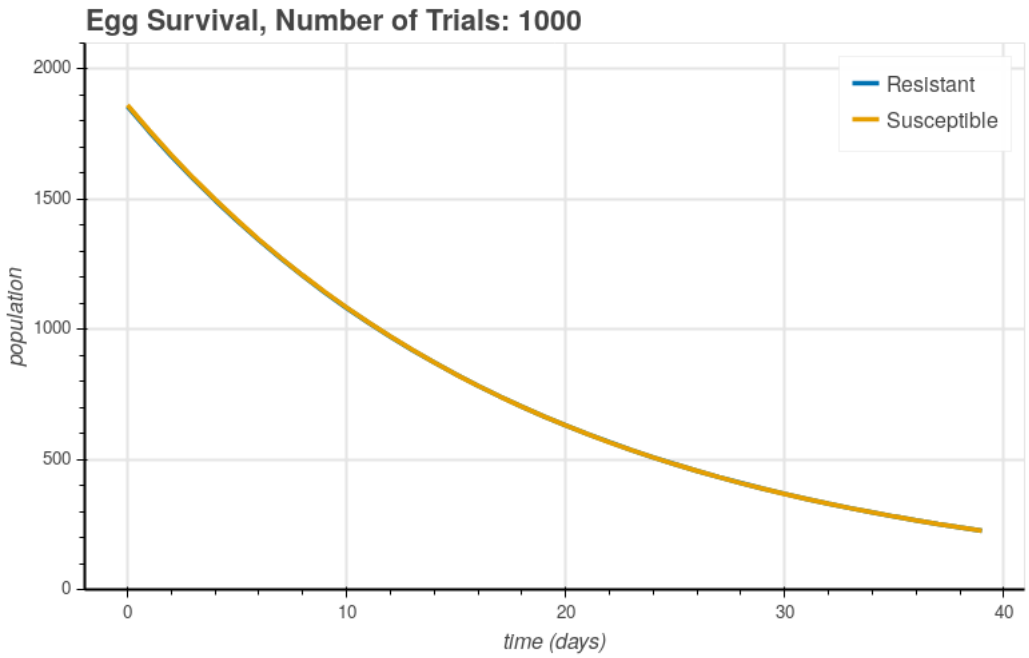
Data for survival for these stages is typically collected as total stage survivals, P_{total}; thus we need to convert P_{total} into P_{step}. Using t_{stage}, the average time that an individual spends in a given life-stage, together with P_{total} we can find P_{step}. To do this, suppose P_{total} is the probability that an agent survives to step n. Since probabilities are multiplicative and $t_{\text{stage}} \approx n$:

$$P_{\text{total}} = \prod_{j=1}^{n-1} P_{\text{step}} = P_{\text{step}}^{n-1},$$

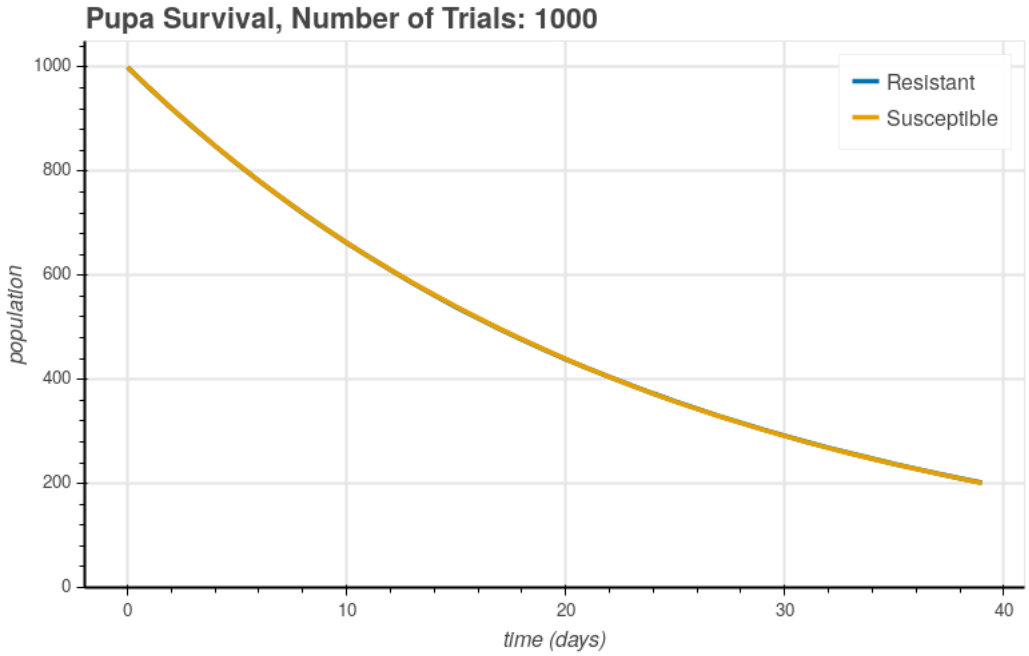
$$\implies \qquad P_{\text{step}} = P_{\text{total}}^{\frac{1}{1-t_{\text{stage}}}}. \tag{2.23}$$

Whitford *et al.* collected data on P_{total} for eggs. (Table 2.3). Moreover, adult ecolsion, adult emergence from pupa, was measured by the Hardke *et al.* study (Table 2.15), which can be thought of as P_{total} for pupae. Furthermore, in Table 2.13 we calculated μ_{egg} and μ_{pupa} to be the average time spent in these respective life-stages, giving us t_{stage}. Thus for eggs and pupae we can average the experimental results to find an average P_{total} and use this together with t_{stage} in equation (2.23) to calculate P_{step} (Table 2.14). Simulating these models results in the curves illustrated in Figure 2.18, again note that the model is independent of genotype which means the curves will be the same.

Figure 2.18: Simulations of Egg and Pupal Survival



(a) Egg survival



(b) Pupa survival

Table 2.14: Survival Parameters

Parameter	Life-Stage	Genotype	Bt	Value	Description
P_{total}	Egg			0.898	Total probability
	Pupa			0.735	
	Larva		non-Bt	0.821	
		SS	Bt	0.003	
				0.340	
				0.672	
		RR	Bt	0.821	
t_{stage}	Egg			3.0	Time spent in life-stage
	Pupa			8.5	
	Larva		non-Bt	19.95	
		SS	Bt	15.6	
		RR	Bt	19.95	
	Adult			10.18	
P_{step}	Egg			0.947	Daily probability
	Pupa			0.960	
	Larva		non-Bt	0.990	
		SS	Bt	0.663	
				0.929	
				0.973	
		RR	Bt	0.990	
	Adult			0.902	

2.1.9.2 Larval Survival

Larva survival is where we incorporate the effects of Bt. In practice, this amounts to finding values of P_{step} for each of the genotypes when the Bt-state is true and another value of P_{step} when the Bt-state is false (the genotype only decreases the effects of Bt and so has no effect in its absence).

Again, we use equation (2.23) together with the stage durations, $\mu_{\text{egg}} = t_{\text{stage}}$, from Table 2.5 to convert P_{total} into P_{step}. General data on survival of larvae in non-Bt environments can be found in the studies of Bernardi *et al.* [14] and Hardke *et al.* [56] (Table 2.15). Survival of both genotypes on Bt crops was also measured by

Table 2.15: Non-Bt Corn Survival Data

Experiment	Survival	Ecolsion	Source
RR	0.80 ± 0.04		Bernardi *et al.* [14]
S♂R♀	0.80 ± 0.04		
S♀R♂	0.85 ± 0.04		
SS	0.75 ± 0.04		
Winnsboro, LA, 2007	0.66 ± 0.116	0.57 ± 0.095	Hardke *et al.* [56]
Winnsboro, LA, 2008	0.88 ± 0.061	0.79 ± 0.076	
Winnsboro, LA, 2009	0.91 ± 0.031	0.85 ± 0.040	

Bernardi *et al.* and by Storer *et al.* [109]. Additionally, data on SS genotype survival on Bt-crops was measured by both Hardke *et al.* and Adamczyk *et al.* [2]. All of the survival data on Bt-crops are summarized in Table 2.16.

For simplicity, we assume the RR genotype larvae will have the same survival probability in the presence or absence of Bt. Hence, we can use the data from Table 2.15 and Table 2.16 to calculate an average total survival probability for all larvae on non-Bt crops and resistant larvae on Bt crops. The values of P_{total}, t_{stage}, and P_{step} for equation (2.23) in both cases can be found in Table 2.14. Notice that the survival data for the susceptible on Bt-crops appears to fall into three groups: low probabilities $P_{\text{total}} \approx 0.003$, intermediate probabilities $P_{\text{total}} \approx 0.340$, and high probabilities $P_{\text{total}} \approx 0.672$. This results in three levels of survival in our later simulations. In Figure 2.19 we can see the effects, of Bt, genotype, and the different survival conditions.

2.1.9.3 Adult Survival

For adults, survival is used to regulate longevity t_{stage}; however, this value represents the average time until death occurs not the average time of the measurement. Hence, we find a value for P_{step} such that the longevity for adults is t_{stage}. Note that $p =$

Table 2.16: Bt Survival Data

Experiment	Genotype	Survival	Source
MON 89034	RR	0.75 ± 0.04	Bernardi *et al.* [14]
	SS	0.00 ± 0.00	
Cry1F	RR	0.978 ± 0.027	Storer *et al.* [109]
	SS	0.005 ± 0.013	
Cry1F Winnsboro, LA, 2007	SS	0.367 ± 0.186	Hardke *et al.* [56]
Cry1F Winnsboro, LA, 2008	SS	0.313 ± 0.057	
Cry1F Winnsboro, LA, 2009	SS	0.644 ± 0.029	
DP 5415	SS	0.695	Adamczyk *et al.* [2]
NuCOTN 33B	SS	0.678	

Figure 2.19: Simulation of Larval Survival

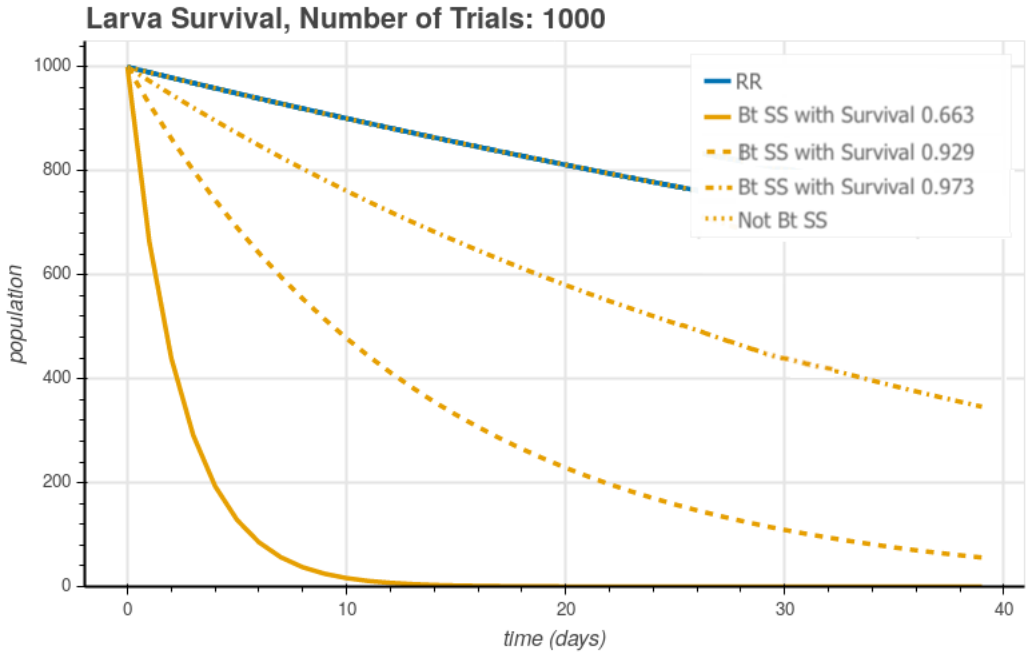


Table 2.17: Adult Longevity Data [85]

Experiment	Longevity (days)
Artificial diet	10.8 ± 4.2
Goosegrass	11.9 ± 4.6
Coastal bermudagrass	9.9 ± 3.4
Large crabgrass	9.9 ± 3.9
Bahigrass	9.3 ± 2.5
Yellow nutsedge	9.3 ± 2.7

$1-P_{\text{step}}$ is the probability that the adult dies during a step. Therefore, the probability that the adult dies on the n^{th} step is given by

$$P(X = n) = (1-p)^{n-1}p,$$

where X is the random variable representing the number of steps before death occurs. The expected value for X is by definition:

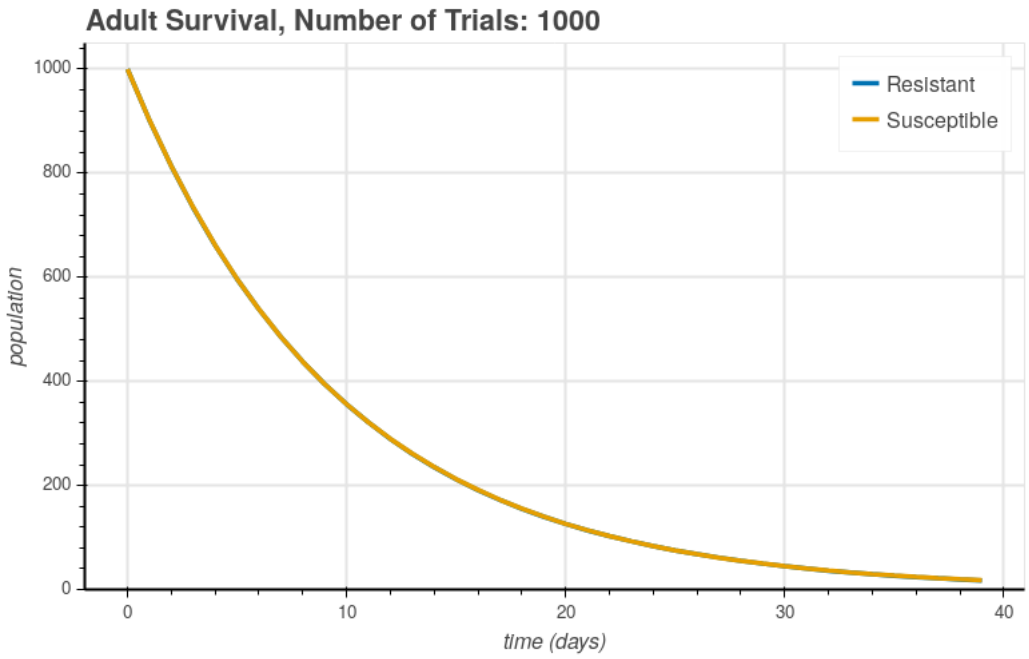
$$E(X) = \sum_{n=1}^{\infty} nP(X = n) = \frac{p}{1-p}\sum_{n=1}^{\infty} n(1-p)^n = \frac{p}{1-p}\frac{1-p}{\big(1-(1-p)\big)^2} = \frac{1}{p}.$$

Note that $t_{\text{stage}} = E(X)$, resulting in:

$$P_{\text{step}} = \frac{t_{\text{stage}} - 1}{t_{\text{stage}}}. \tag{2.24}$$

Pencoe and Martin measured the average longevity of non-migrating adults (Table 2.17). Applying equation (2.24) to this data, we obtain P_{step} (Table 2.14). This process can be simulated to obtain the curves shown in Figure 2.20; nothing that, this is a genotype independent process.

Figure 2.20: Simulation of Adult Survival



2.1.10 Migration

Migration models add (immigrate) new adults into the IBM, or remove (emigrate) adults from the IBM.

2.1.10.1 Immigration

Immigration refers to the addition of adult agents into the IBM. We introduce this phenomena to prevent genotypes from going extinct. During each step, the IBM draws a number N from a Poisson distribution with mean λ and creates/adds N adults randomly with a given genotype (the one we are preventing extinction of). Note that we assume immigration occurs before emigration.

2.1.10.2 Emigration

Emigration refers to removal of adult agents from the model to reduce egg creation by adults because egg creation is computationally expensive.

For our migration model, we assume the population of adults agents is normally distributed with mean μ and standard deviation σ. Next, define a CDF from this distribution in terms of N, the number of adults in the IBM. Then during each step in a random order, every adult makes a Boolean choice between emigrating or staying with probability given by this CDF evaluated with the current value of N.

This emigration model acts as a negative feedback on the population, enforcing a carrying capacity of approximately μ on the adult population. Setting this carrying capacity on adults limits the simulation population to computationally tractable levels.

2.2 Individual Based Model Structure

Structurally, we divide the IBM into four components: environment, agents, schedule, and behaviors. This process was inspired by the pattern-oriented-modeling (POM) approach proposed by Grimm *et al.* [54]. Our presentation of the IBM's structure is based loosely on the ODD (Overview, Design concepts, and Details) protocol proposed by Polhill *et al.* [92] which was later revised and extended by Piou *et al.* [87] to fit within the POM approach. Note that we did not entirely follow any one of these guiding methodologies, but instead used a hybrid of them together with our development philosophy (Appendix A).

2.2.1 Environment

The environment forms the component of the IBM composed of the processes that do not directly involve agents but rather describe or manage the external information. Here we will consider the environment to be a spatial organization together with environmental variables and migration. The spatial organization is summarized in Figure 2.21 and described in detail throughout Section 2.2.1.1. Note that the reference to *Bt-state* is related to the environmental variables, see Section 2.2.1.2. Finally, the migratory forces are discussed in Section 2.2.1.3. Furthermore, Figure 2.21 also includes a conceptual summary of the agent information organization which is expanded upon in Section 2.2.2.

2.2.1.1 Spatial Organization and Grids

We have a three-tiered spatial organization, as illustrated by Figure 2.21, because to describe the fall armyworm's behavior properly we require multiple length-scales. For convenience, we will name these tiers (levels) after real environmental features that are descriptive of these length-scales; these are: field, plant, and leaf.

The field-level refers to the global length-scale, which is the scale that encompasses the entire model. Within this level we encounter our first spatial grid, the *field-level grid*, which organizes space into *local environments*. These local environments are plants (space occupied by juvenile stages of fall armyworm) which are defended or undefended by insecticides (see Section 2.2.1.2). As illustrated in Figure 2.21, we decided to use a toroidal hexagon tiled grid for the field-level grid. We choose a toroidal grid to eliminate possible boundary effects, which may introduce modeling artifacts. This choice is reasonable if we assume the grid is so large that organisms are unlikely to be effected by boundary conditions. In practice, this means we chose

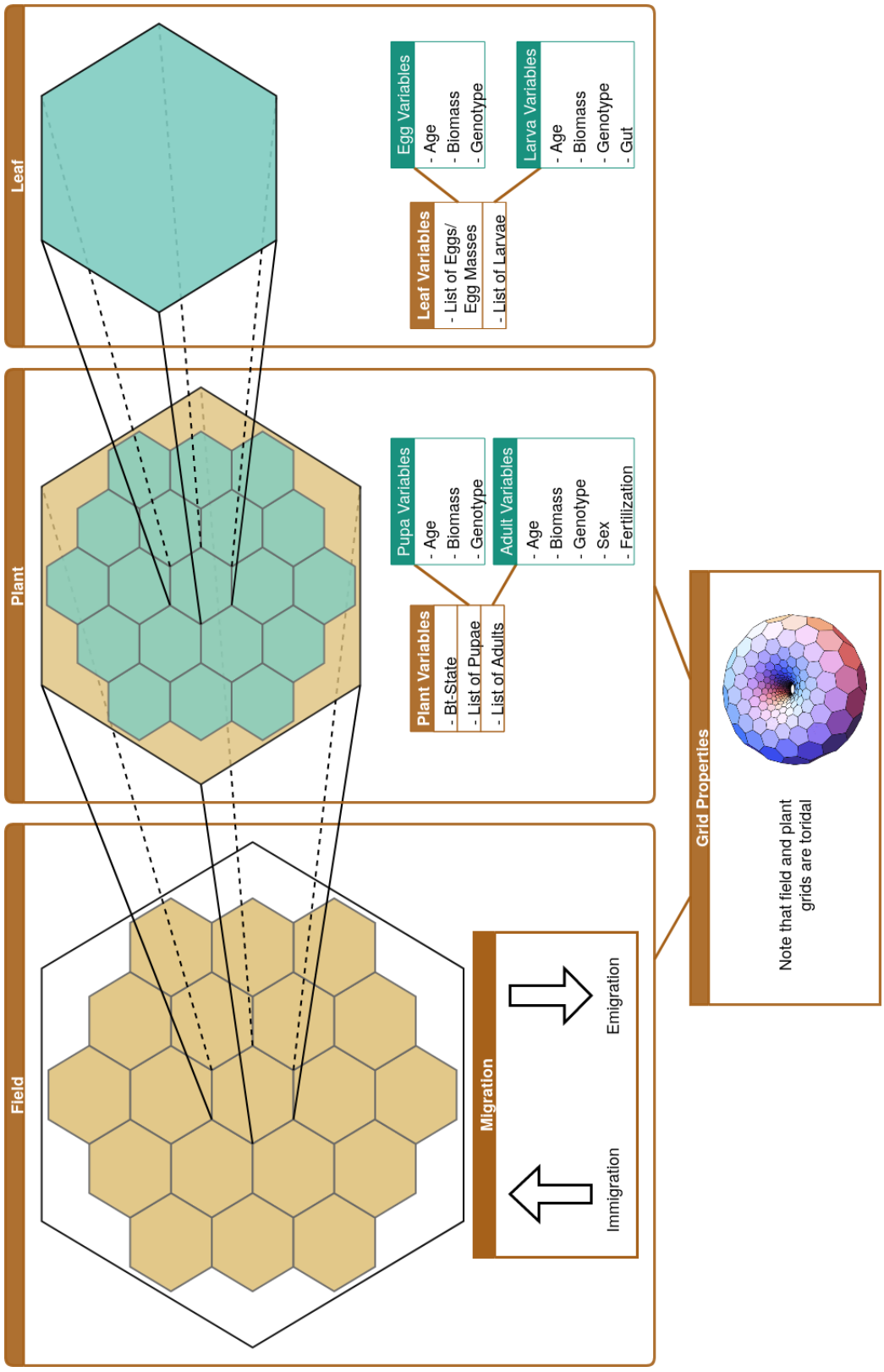


Figure 2.21: Model Environment

Table 2.18: Spatial Grid Parameters

Grid	Type	Height	Width	Toroidal
field-level	hexagonal	50	50	True
plant-level	hexagonal	10	10	True

the largest grid that remains computationally tractable[1], namely a 50 tile by 50 tile grid (Table 2.18). A hexagonal grid was chosen so that we could allow movement in a more realistic fashion from grid cell to adjacent grid-cell that more closely (than a Cartesian grid) approximates a real world pattern because radii out from a given grid cell form circles instead of straight lines that we would see in a square grid. The work of Birch *et al.* [16] indicates that, for the above reasons, hexagonal grids are the most effective grid style for biological models while maintaining the biologically appropriate length-scales.

The plant-level refers to the maximum length-scale experienced by a fall armyworm larva. Since crop plants, such as corn, are large relative to larvae and tend to be spread out we may consider one plant (or small cluster of plants) to be the local environmental area for a larva. As shown in Figure 2.21 each plant level environment can be thought of as the interior of a field-level grid-cell. Note that we will assume that larvae do not cross between the different grid-cells of the larger field-level grid. We make this assumption because, even though there is probably some transfer between clusters of crop plants in reality, the process would be extremely complex to

[1]Our methods for representing these grids and associated distances are extremely memory intensive for large grids. The amount of memory needed is $O(n^2)$ (n is the number of grid tiles) which can grow to intractable levels quickly. Indeed, memory use for a 50 by 50 grid was approximately 6GiB while a 100 by 100 grid required more than 32GiB (the memory size of the machine being used). Note that more granular analysis was not performed because the computational time of pre-computing these grids was also $O(n^2)$ with the 50 by 50 grid taking approximately one day of computation on 16 cores, while the 100 by 100 grid took over a week with the same resources. This memory footprint is simply too large to be usable for large numbers of simulations. Hence, a compromise of 50 by 50 was chosen because this would allow us to saturate (use all CPUs and almost all memory) the available computational resources.

model in practice for only limited gains. Each plant level environment is then divided into a *plant-level grid*, the second grid of our IBM. Once again, this grid is a toroidal hexagon tiled grid. The particulars of this grid were chosen for the same reasons as for the field-level grid. However, this time a smaller intermediate size was chosen for the grid size; namely, a 10 tile by 10 tile grid (Table 2.18). We chose this grid size to allow the IBM to remain consistent with the results of an import set of experimental results, see Section 2.1.4.3.

Finally, we have the leaf-level, which is a single grid-cell of the plant-level grid. These cell's dimensions can be considered to be the natural area over which an average larva influences its greater local environment on the foraging-timescale from Section 2.2.3.1. Here we do not apply a grid, instead we invoke particle interaction models (described later in Section 2.1.4.1) to model cannibalistic agent interactions within this space in the given time and spatial scales.

2.2.1.2 Insecticide Environment

The primary environment variable in the IBM relates to the presence or absence of a Bt insecticide, which we term *Bt-state* in Figure 2.21. Recall that we are modeling the adaptive (evolutionary) response of the fall armyworm to transgenic Bt crop plants. There are several different types of transgenic modifications to produce Bt (Section 1.2.2); each of these modifications produce different forms of Bt insecticides and/or different levels of insecticide effects. We will simplify Bt insecticide to a Boolean, Bt-state, for Bt present/defended (True) or Bt absent/undefended (False).

Farmers are required to plant a portion of their fields with a non-Bt variation of the crop plant in order to postpone the evolution of resistance to Bt insecticides in pests [104, 115, 118]; this portion is often called the *refuge*. Thus our insecticide environment will have local areas which contain Bt varieties and local areas (refuges)

which contain non-Bt varieties. This is accomplished in the environment by assigning grid-cells of the field-level grid their own value of the Bt-state variable, as noted by Figure 2.21.

We approach the construction of a refuge by assumiung that the field-level grid will be divided so that one part represents the refuge and the other the non-refuge. Note for this we can adjust the relative proportion of refuge, P_{Bt}, by adjusting where this divide occurs. Observe that manipulation of the P_{Bt} parameter represents one of our primary methods for creating different experimental scenarios. We will use five values for P_{Bt}: 1.0, 0.3, 0.2, 0.1, and 0.0 in our simulations because these values represent the both the most economically interesting refuge sizes and the extremes of all Bt or no Bt.

2.2.1.3 Migration

We include migration in this IBM as means of limiting population sizes (to improve computational tractability) and the elimination of extinction events due to stochastic fluctuations. We limit population sizes by *emigrating* reproductive adults out of the IBM when population densities become large (Section 2.1.10.2). On the other hand, the biologically relevant initial populations typically have a low percentage of Bt-resistant fall armyworms (often $< 1\%$). The nature of the stochastic processes involved in the IBM can easily lead to the extinction of Bt-resistance. Thus we assume some individuals *immigrate* into the IBM in order to mitigate this possibility as discussed in Section 2.1.10.1.

2.2.2 Agents

Agents are modeling descriptions of single organisms, which consist of descriptive variables for the organism together with models of the life processes that make-up the

Table 2.19: Common Agent Variables in Source Code

Variable	Type	Description	Value(s)
`unique_id`	string	Model identifier	`'init_0'`, ...
`agent_key`	string	Agent type organizer	`'egg'`
			`'egg_mass'`
			`'larva'`
			`'pupa'`
			`'adult_female'`
			`'adult_male'`
`location`	list	Location storage	`[0, 1, 2]`, ...
`alive`	Boolean	Agent activity state	`True`
			`False`

organism. When modeling the fall armyworm one considers the agents as modeling proxies for individual organisms; so in particular, they provide the vehicles by which information is stored and changed within the IBM.

Recall that there are four life-stages for the fall armyworm: egg, larva, pupa, and adult. We will create agents for each of these life-stages. We chose not to model individual larval instars because this would amount to adding several more additional types of similar agents (one for each of the six instars). Note that eggs are laid in large masses, so we create the egg mass agent to represent these masses because interactions with eggs are really interactions with these masses of eggs.

For technical reasons, all agents have a unique identifier string and an agent type identifier, which are detailed in Appendix B.2. All agents also keep track of their location and a Boolean for whether they are alive or dead due to technical limitations.

The life-stage agents track a few additional core variables: genotype, age, cause of death, and biomass. We explored the precise nature of the agent's genotype in Section 2.1.1. Age refers to the number of steps that an agent has spent in its current life-stage. Additionally, each agent records its cause of death to be tallied for later analysis.

Table 2.20: Common Life-Cycle Agent Variables

Variable	Type	Description	Value(s)

`genotype`	string	Agent genotype	`'susceptible'`
			`'heterozygous'`
			`'resistant'`

`age`	integer	Steps since start of life-stage	`0`, `1`, ...

`death`	string	Cause of death	`'alive'`
			`'cannibalism'`
			`'starvation'`
			`'survival'`

`biomass`	float	Current biomass	`0.1`, `1.1`, ...

The biomass of an agent is akin to the mass and/or size of an actual organism, but with a key distinction that it does not decrease (irreversible mass [49]). Note that biomass is highly correlated with adult fecundity [85].

2.2.2.1 Eggs and Egg Masses

Eggs are simple agents that exist to model the egg life-stage portion of the fall armyworm's life cycle, so they do not have any additional variables to those in Table 2.19 and Table 2.20. Indeed, eggs are mainly distinguished by having egg-specific survival (Section 2.2.4.5) and development (Section 2.2.4.6) behaviors.

The egg mass agents are more complex. They exist as a distinct form of agent to help facilitate cannibalism and reproductive interactions. Larval cannibalism of eggs is sensible as an interaction with an egg mass, see Section 2.2.4.1. While reproduction produces and tracks the local population of eggs using egg masses, see Section 2.2.4.2. Therefore, egg masses have one additional variable to those in Table 2.19, namely a dictionary[2] of the constituent eggs.

[2] In python, a dictionary is an associative array consisting of key value pairs, where keys are unique. Often these may be called pointer arrays, where the key acts as a pointer to the value. Here the for an egg, `unique_id` acts as a key and the egg acts as the value. One can consider this as a list, but for efficiency purposes in python this is faster at the expense of memory.

Table 2.21: Larval Agent Specific Variables

Variable	Type	Description	Value(s)
`plant_gut`	float	Biomass of plant eaten	0.1, 1.1, ...
`egg_gut`	float	Biomass of eggs eaten	0.1, 1.1, ...
`larva_gut`	float	Biomass of larvae eaten	0.1, 1.1, ...
`starve`	Boolean	If larva is starving	`True` `False`

2.2.2.2 Larvae

In addition to the values in Table 2.19 and Table 2.20, larval agents have more tracked values (summarized in Table 2.21): plant biomass, egg biomass, larva biomass, and starvation. Larvae not only consume plant biomass, but they also cannibalize eggs and other larvae. Biomass from each of these food sources is tracked separately; however, the total amount of plant, egg, and larval biomass consumed by a larval agent is given by G (Section 2.1.3.1). Recall that biomass is irreversible (it cannot decrease), so we assume that larvae will die whenever circumstances create situations where biomass can shrink. To facilitate this we introduce the starvation Boolean, so that these circumstances can be noted by the IBM and dealt with appropriately.

Larval agents have complex behaviors, which are: growth, forging, movement, development, and survival. In particular, the foraging, movement, and survival behaviours all depend on information from the larva's environment. Foraging (Section 2.2.4.1) depends interactions with other agents contained in the local environment. While movement (Section 2.2.4.3) depends on the spatial information provided by the local environment. Survival (Section 2.2.4.5) requires information from the environment about the Bt-state. By contrast, growth (Section 2.2.4.4) and development (Section 2.2.4.6) are related to the larva's internal dynamics; mainly, the interplay between the consumed biomass and its current biomass.

Table 2.22: Adult Agent Specific Variables

Variables	Type	Description	Value(s)
`num_eggs`	integer	Number of egg masses to lay	1, 2, ...
`mate`	string	Id for mate	`male_0`, ... `None`

2.2.2.3 Pupae

Like eggs, pupae are simple agents because, like biological eggs, biological pupae have little external interaction with their outside environment. Unlike eggs, the fall armyworm pupates in the soil; meaning that, the pupa is not vulnerable to cannibalism. We account for the larvae moving off the plants and into the soil for pupation by changing the grid-level tracked by pupae, meaning that pupae switch from the plant-level to the field-level. Thus pupal agents are just like egg agents except with distinct models of behavior (see Section 2.1.9.1 and Section 2.1.8.1). Indeed, pupal agents only need to track the values in Table 2.19 and Table 2.20.

2.2.2.4 Adults

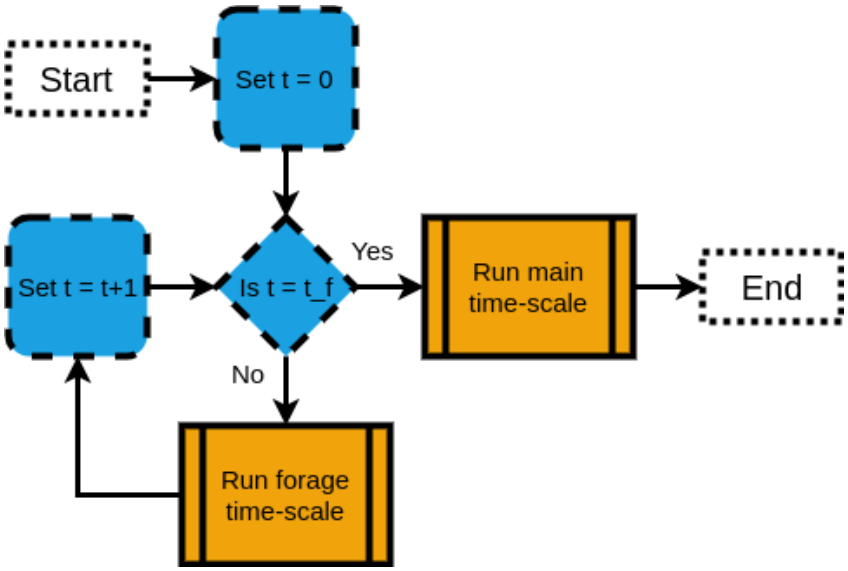
Adults are complex biologically; however, some simplification is necessarily, in particular we do not model foraging and biomass consumption for adults. Although this behavior is likely important to the adult fall armyworm's ability to function, we assume during movement of the adult agent, that enough feeding implicitly occurs.

Adult agents, like larval agents, interact with other agents, in the form of mating and density dependent egg laying. To accomplish this we need two variables in addition to those in Table 2.19 and Table 2.20; namely (as summarized in Table 2.22): number of egg masses and its current mate. Note that the `agent_key` variable is modified by adults to carry the gender instead of using another variable for efficiency purposes.

Table 2.23: Summary of Time-Scales

Time-Scale	Unit	Repeats	Description
Forage	hour	12	time-scale of agent interaction
Main	day	1	time-scale of agent life processes

Figure 2.22: Flow Chart of Model Schedule



Again like larval agents, adult agents have several behaviors: reproduction, movement, and survival. Both reproduction and movement require interactions with the environment. Reproduction (Section 2.2.4.2) is the primary behavior for an adult agent; like cannibalism for larvae, the adult females require information from their local environment through which they select mates and determine if they can lay eggs. While movement (Section 2.2.4.3), as noted for larvae, is the transfer from location to location in the field-level grid. The survival behavior (Section 2.2.4.5) simply regulates how long adults live on average, while also enforcing a background death rate due to any factor.

Table 2.24: Overview of Flow Charts

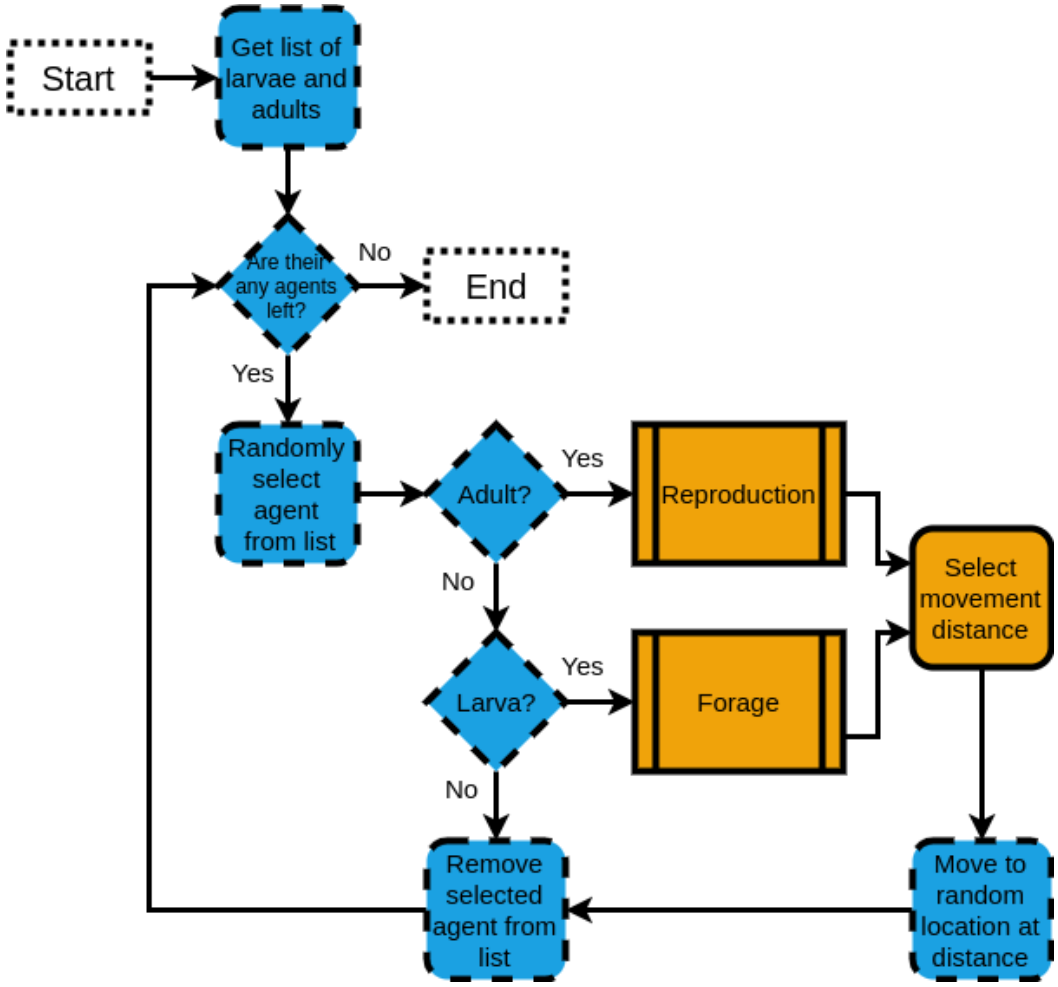
Color	Boarder	Shape	Description
Blue	Dashed		Process/Decision performed by IBM
Orange	Solid		Requires modeling from Section 2.1
Orange	Double Vertical		More detailed flow chart is given
White	Dotted		Start or end
Any	Any	Square	Process performed
Any	Any	Diamond	Decision made

2.2.3 Scheduling

The schedule dictates the order that events or processes occur in. Our IBM incorporates two time-scales for its processes, namely a forage time-scale (forage-scale) and a main time-scale (main-scale). Starting with the main-scale, this represents the of development. Developmental measurements for the fall armyworm are reported in terms of the number of days, so the main-scale step size is taken to be one day. The forage-scale represents the time-scale which individual's foraging, reproductive, and/or movement behaviors occur. Larvae are active throughout the day while the adults are active at night [106]. We assume for simplicity that both larvae and adults will have 12 hours of activity divided in to 1 hour blocks each step. However, both larvae and adult processes are preformed at the same time as a computational simplification; hence, we repeat the forage-scale steps, $t_f = 12$ times for every main-scale step. It follows that the IBM has hours (two distinct 12 hour periods computed simultaneously) within each single day period to form its steps, as summarized in Table 2.23. We can describe the model's schedule with the flow chart in Figure 2.22[3], where the meaning of each box's color/shape is detailed in Table 2.24.

[3]All flow charts were created using `draw.io` [3].

Figure 2.23: Flow Chart of Forge-Scale Sub-Schedule



2.2.3.1 Forage-Scale

This scale only involves the *active* agents, namely the larvae and adults. The behaviors these agents take are foraging (Section 2.2.4.1) for larvae, reproduction (Section 2.2.4.2) for adults, and movement (Section 2.2.4.3) for both. The flow chart in Figure 2.23 summarizes how these behaviors are scheduled.

This sub-schedule begins by randomly selecting a larva or adult, which then performs some behaviors; repeating until all the larval and adult agents have completed their respective behaviors. The behaviors performed for larvae are foraging followed

by movement, while adults perform reproduction then movement. Notice that the movement could happen before the forage/reproduction behavior, but we chose the reverse arbitrarily (either order is equivalent as we initialize to random locations and movement is a random process, see Section 2.1.6).

2.2.3.2 Main-Scale

The main-scale sub-schedule involves: growth (Section 2.2.4.4), survival (Section 2.2.4.5), development (Section 2.2.4.6) agent reset (Section 2.2.4.7), and migration (Section 2.2.1.3). The flow chart in Figure 2.24 collectively summarizes how all of these processes are scheduled.

We begin by selecting an agent at random from the eggs, larvae, pupae, and adults, which then proceeds to perform some behaviors. Egg and pupal agents both perform a survival event followed by development, and then ready (reset) themselves for the next step. While, adult agents perform a survival event followed by readying for the next step. Finally, larval agents grow, then perform a survival event, followed by development, and then get ready for the next step. Once we have cycled through all of the agents, the adult migration occurs. Migration follows the other agent behavioral processes because it is density dependent.

2.2.4 Behaviors

Behaviors are the processes or events that involve agents in order to facilitate modeling of some biologically relevant process.

2.2.4.1 Foraging

Cannibalism and biomass consumption are the processes which accumulate biomass for agents; hence, they are only applicable to larval agents. These are assumed to act

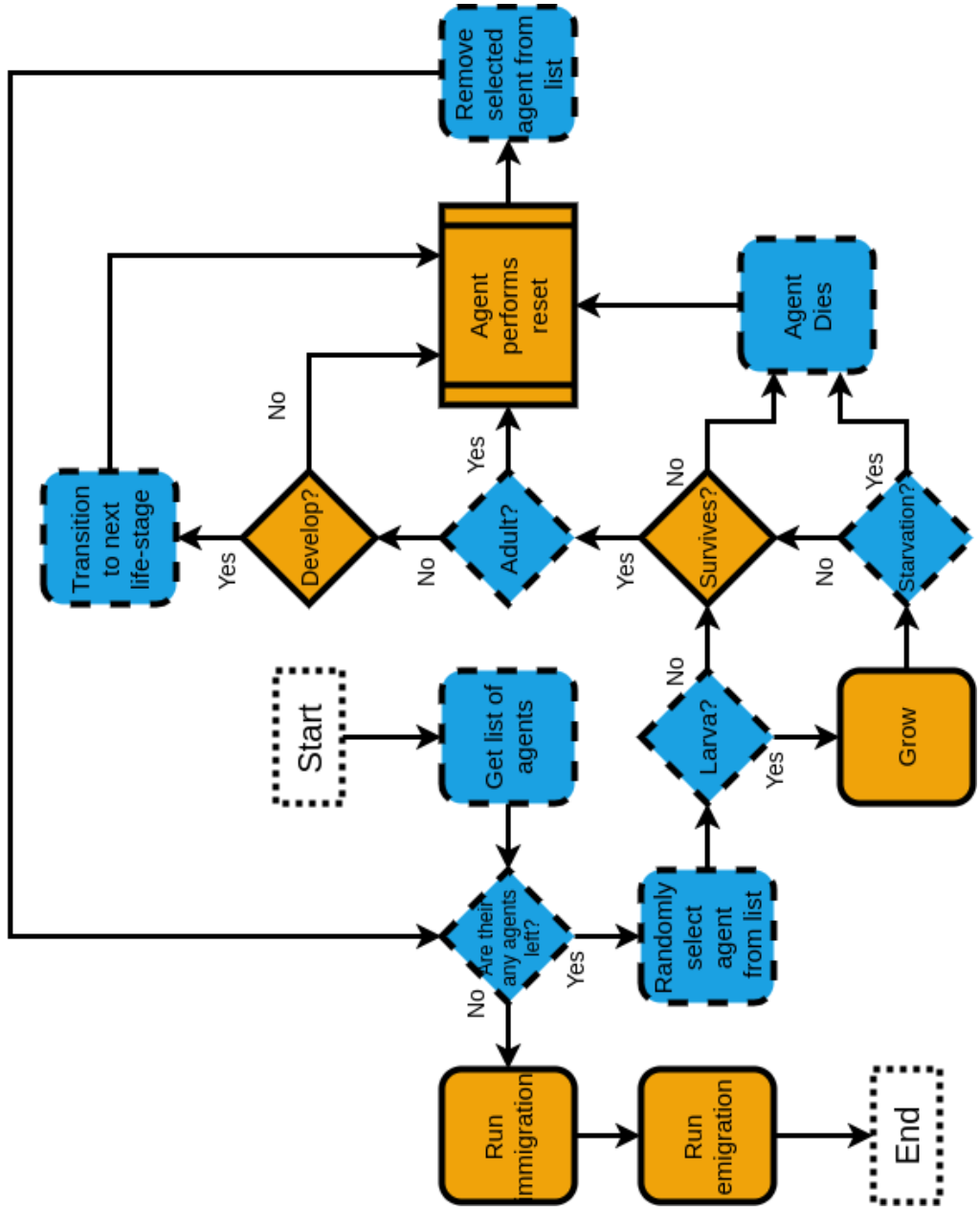


Figure 2.24: Flow Chart of Main-Scale Sub-Schedule

only within the leaf-level (Section 2.2.1.1) on the forage-scale (Section 2.2.3.1).

Cannibalism is the set of processes (Section 2.1.4) involving a larva consuming another agent's biomass. Occurrences of cannibalistic interactions within grid-cells must be modeled, which is accomplished using "idea gas" encounter models detailed in Section 2.1.4.1. Note that the agents which can be cannibalized are the egg masses and larvae contained within the larva's current grid-cell. So a larva risks being killed and consumed by its cannibalism target (Section 2.1.4.2).

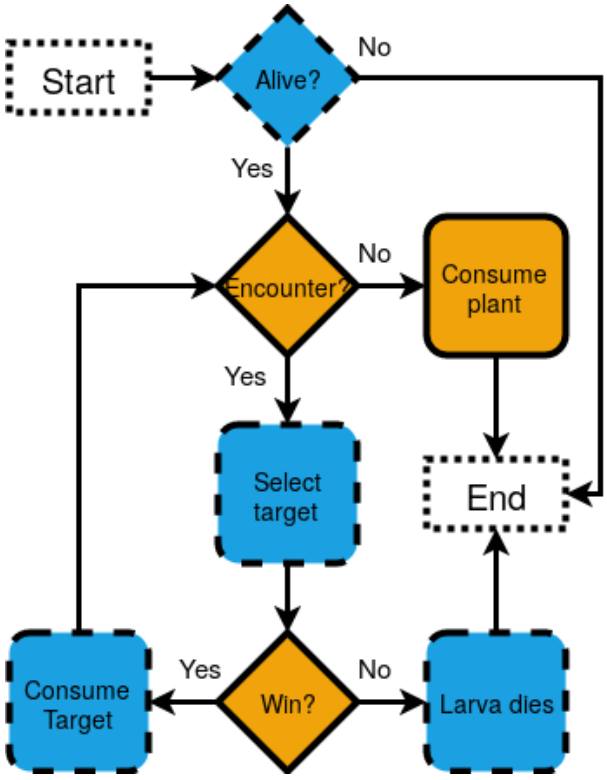
Biomass consumption describes larval biomass intake. One source of biomass are the targets of successful cannibalism interactions, while the other source of biomass is the local plant environment. These different methods of biomass consumption are discussed fully throughout Section 2.1.5. Note that egg masses handle accounting processes to change amounts of biomass consumed into egg deaths. This is accomplished by egg masses providing biomass information to the consumption model as the sum of all the biomasses of the constituent eggs (all with the same biomass, m_{egg}). The consumed biomass, m_{consumed}, is then converted to the number of eggs, N_{death}, to be cannibalized by

$$N_{\text{death}} = \left\lceil \frac{m_{\text{consume}}}{m_{\text{egg}}} \right\rceil . \tag{2.25}$$

The egg mass then kills and removes N_{death} of its constituent living eggs uniformly at random.

Cannibalism and biomass consumption for larvae together form the larger *foraging* behavior for larvae noted in Section 2.2.3.1; wherein, a larva decides if and how it will consume biomass from different sources, as summarized by the flow chart in Figure 2.25. Foraging begins by checking the larva's `alive` Boolean before continuing. After this check, the larva applies the encounter model from Section 2.1.4.1

Figure 2.25: Flow Chart of Foraging Behavior



to determine if it can encounter another agent. If an encounter occurs, the larva uniformly at random selects from the egg masses and other larvae in the grid-cell a cannibalism target. The outcome of the cannibalistic interaction is then determined; by this we mean, if the target is an egg mass the larva automatically wins; if however, the target is another larva the win/loss of the cannibalizing larva is determined via the outcome model in Section 2.1.4.2. If the larva wins, then the target is consumed by the consumption model in Section 2.1.5.2, after which the cannibalism encounter-fight-consume process repeats. If instead the larva loses, then the larva dies and is consumed (using the model in Section 2.1.5.2) by the target larva instead, completing the foraging behavior. Whenever no encounters occur in a cannibalism cycle, the larva then consumes environmental plant biomass as detailed in Section 2.1.5.1, completing the behavior.

2.2.4.2 Reproduction

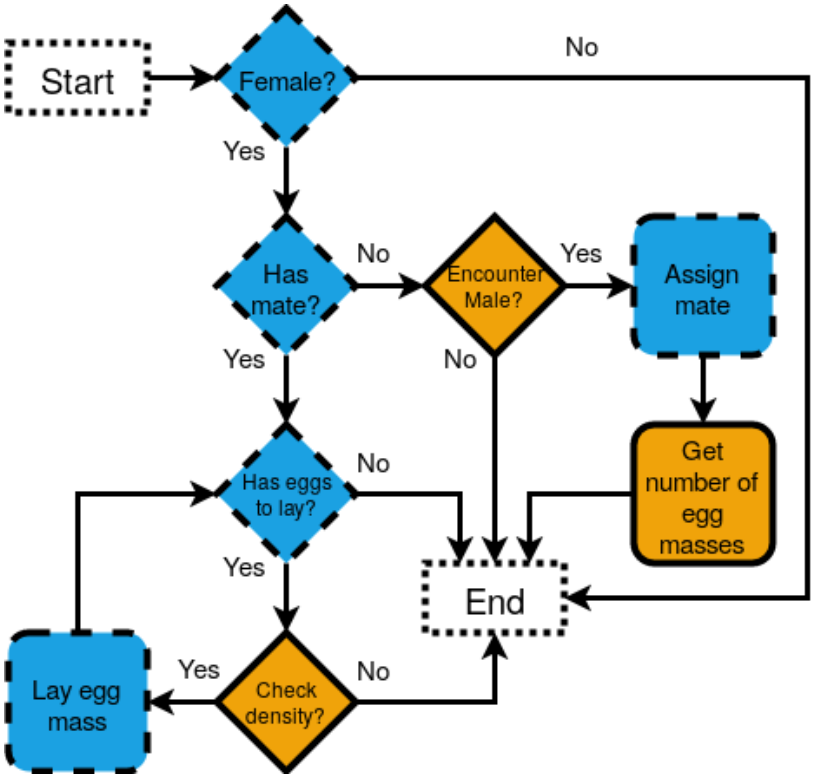
The reproductive behaviors are those which model mating and the production/laying of new eggs. These behaviors take place within the field-level grid, where egg masses are deposited at random within the plant-level grid of the current field-level grid-cell (Section 2.2.1.1).

First, the mating component consists of female adults finding an adult male to mate with. In a similar fashion to cannibalism encounters, this process is mediated by an encounter model (Section 2.1.7.1). Mates are selected uniformly at random from a pool (construction of which is also described in Section 2.1.7.1) of available male adults.

Egg laying encompasses the rest of the reproduction behavior; here, mated females attempt to lay egg masses. This process is regulated by two models: a fecundity model (Section 2.1.7.3) and a density model (Section 2.1.7.4). The fecundity model controls how many egg masses can be laid by the female in a given day, while the density model regulates egg laying success. Egg masses are then created with masses determined by an initial mass model (Section 2.1.2.1) with genotypes determined by the female's genotype and her mate's genotype (Section 2.1.1).

Summarized by the flow chart in Figure 2.26 are the combination of mating and egg laying which together form the reproduction behavior. Reproduction begins by checking if the adult is female, as all reproductive processes are considered from the female agent perspective. The female then determines if she has a current mate, if she does not have a mate, she beings the mating branch. During mating, she first determines if there are available mates (Section 2.1.7.1). If there are mates, she is assigned a mate from the pool of available mates uniformly at random and then sets `num_eggs` using the fecundity model in Section 2.1.7.3 completing this instance

Figure 2.26: Flow Chart of Reproduction Behavior



of reproduction. Whenever she is unsuccessful in finding a mate, the instance of reproduction ends. If a female has already mated, she begins the egg laying branch instead. First, she checks if she has any egg masses left to lay, if so she attempts to lay an egg mass where success is determined by the density model of Section 2.1.7.4. When successful, she lays an egg mass and cycles to the beginning of the egg laying branch. Whenever she is unsuccessful in laying an egg mass or runs out of egg masses to lay, the instance of reproduction ends.

2.2.4.3 Movement

The movement behavior is the process by which an agent moves between grid-cells in the appropriate grid. Larval and adult agents are the only types of agent which can move. Moreover, larval agents move within a local plant-level grid while adults

move within the global field-level grid. Note that we assign location randomly within appropriate grids when creating or initializing agents.

Both larval and adult agents move within their respective grids via the same process, summarized after reproduction/forage behaviors in Figure 2.23. First, the agent selects a distance to move using the process described in Section 2.1.6 and then uniformly at random selects from the grid-cells in the appropriate grid which are the given distance away from the current grid-cell. The agent is then transferred *instantly* to the new grid-cell. This transfer is instant and does not simulate the movement through any intervening grid-cells; doing otherwise would add large amounts of complexity.

2.2.4.4 Growth

The interplay between biomass and genotype is the principal driver of larval behavior. Together they effect every aspect of larval behavior; however, genotype is fixed while for larvae biomass changes.

The growth behavior is the mechanism by which biomass changes for all agents; however, we assume that biomass only changes for larvae. The amount of growth for a larva is governed by the models in Section 2.1.3, while the initial conditions for growth are determined by the models in Section 2.1.2.

In particular, the growth model acts on the larva once per model-step (see Section 2.2.3.2) using the amount of biomass consumed by that larva during the step together with its current biomass and genotype. The result is a possibly larger larva, although if the larva is found to shrink in size the behavior instead sets the starvation state to true. The entirety of growth behavior is summarized in Figure 2.24, after the yes branch for `larva?`. We can see here that this behavior acts on larvae by growing them and then checking the starvation result. If starvation occurs, then the agent

dies. Else it continues to the survival behavior.

2.2.4.5 Survival

The survival behavior is the process by which agents account for sources of mortality other than cannibalism and starvation. Note that for larvae, we invoke different levels of survival for each genotype in a distinct Bt-state by assuming that the genotype and Bt-state effect the survival model (Section 2.1.9.2). The survival for fall armyworms where reported as average proportions surviving a life-stage, whose duration is measured in days or the average lifetime of adults in days. Consequentially, we assume that survival takes place on a daily (main-scale) time-scale. Thus we consider the probability of survival from day-to-day, as discussed in Section 2.1.9.

As survival takes place in the main-scale, this behavior is summarized within the flow-chart in Figure 2.24. Here survival is handled for all agents (just after larval agents grow) as choice of surviving or not. If the agent fails to survive, it then dies. Else it continues through the rest of the main-scale sub-schedule. We place survival between growth and development so that starvation will have been accounted for by larval agents, while determining survival before developmental occurs.

2.2.4.6 Development

Developmental behaviors refer to the methods by which individuals transition from one life-stage to the next. The IBM implements this by creating a *new* agent in the next stage and copying the relevant information from the current agent into the new agent; after which, the current agent is replaced by this new agent. The process then continues with the new agent. Effectively, this means when agents develop to their next life-stages, they do so immediately. Although there are possible delays and other processes involved in this transition, we make this immediate change as

a simplification to a complex situation. Note that development also takes place on the main-scale as life-stage developments act as a major demarcation points in life history of the fall armyworm. Furthermore, note that only eggs, larvae, and pupae can develop because adults do not have a next life-stage to develop into.

Thus the development behavior mainly requires us to model how an agent makes the *decision* to transition to the next life-stage. This is described in Section 2.1.8 and relies primarily on generating a daily probability of developing. Note, when pupal agents transition to adults we assign them a gender, as described in Section 2.1.7.2.

We can see the development behavior within the flow chart of Figure 2.24. It is the processes, following survival whereby egg, larval, and pupal agents test if they develop. When the they test true, they pass through the transition process described above; otherwise, they continue through the rest of the main-scale sub-schedule.

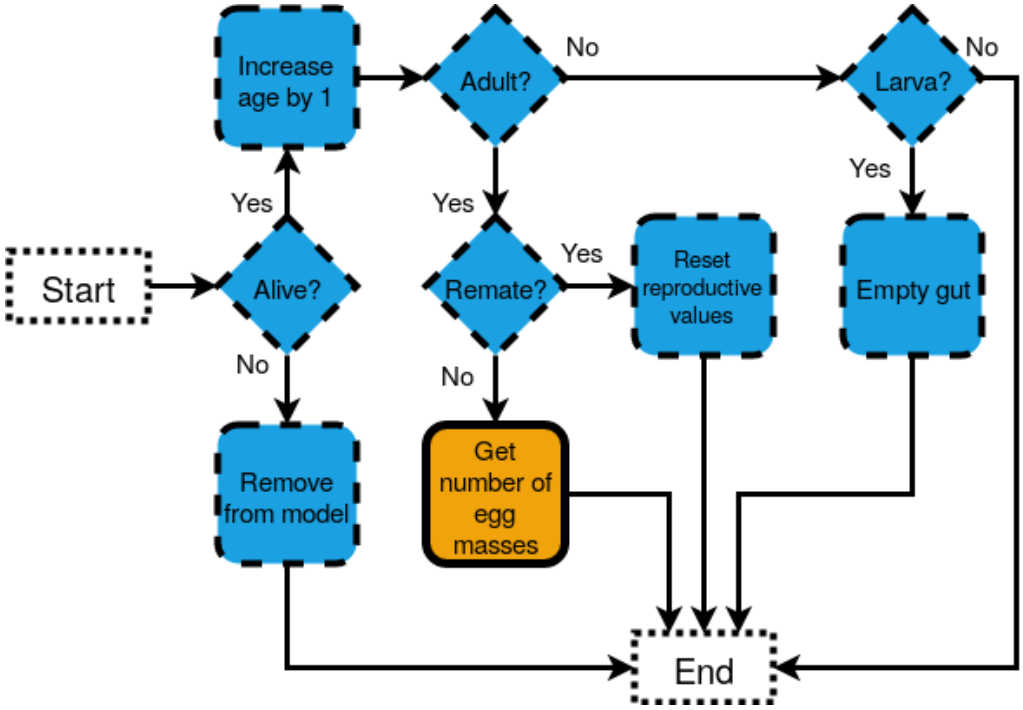
2.2.4.7 Resetting Agents

The agent reset behavior is a behavior added to allow for accounting of items that need to be measured and/or reset before the next step. Data measurements are tallied implicitly by the IBM as agents pass through this reset behavior. Moreover, during this behavior ages are incremented and step-to-step tracking values for agents are reset. Finally, during this behavior, agents that have died are from the model.

For adults, we specifically need to reset a few values. First, we have a global model variable that determines if females ever re-mate. When this variable is true, during reset they set their `mate` variable to `None` and `num_eggs`$= 0$. Otherwise, they run there fecundity model to update their value for `num_eggs`. While for larvae, the gut variables: `plant_gut`, `egg_gut`, and `larva_gut` are all reset to zero values.

This reset process can be found in Figure 2.27. Here we can see the removal of possible deceased agents followed by increasing the agent's internal age. This is

Figure 2.27: Flow Chart of Resetting an Agent



followed by the reset processes described for adults and larvae above.

Chapter 3

Analyzing Short Term Behavior

In this chapter we explore the effects of parameters on the system dynamics over small time periods. We do this to both analyze the local (in time) sensitivity of the IBM's results to parameter estimates and check that the desired emergent effects (such as selection pressures) occur. Specifically, we explore the effects of changes to growth, cannibalism, reproduction, and survival.

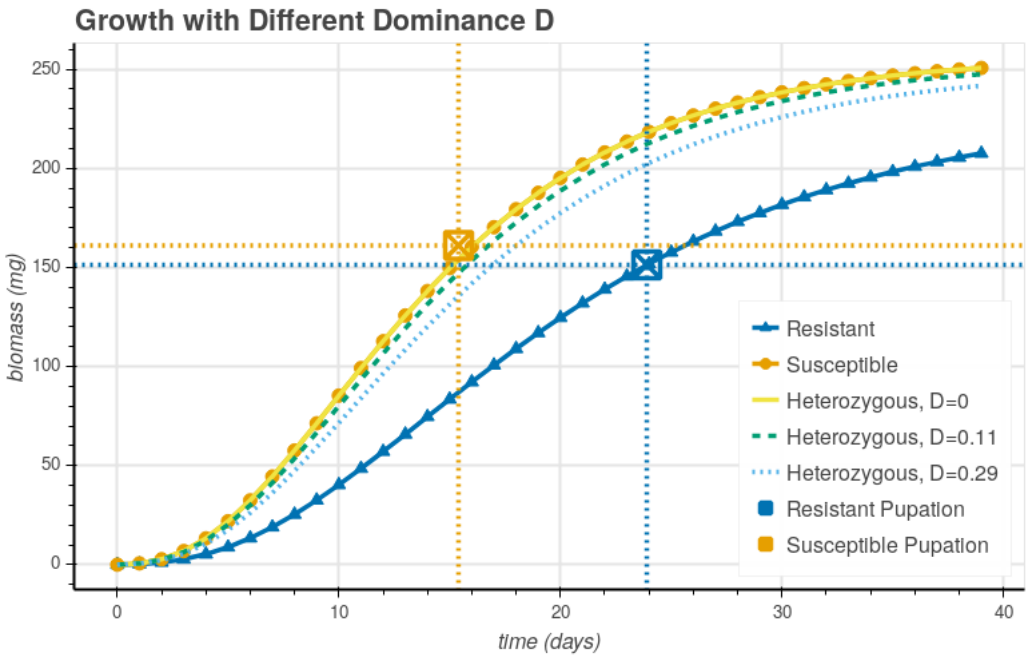
3.1 Growth

Recall that in Section 2.1.3 we based growth for larvae on equation 2.5. This equation has two free parameters α and β, which were specified in Table 2.6. Our goal will be to determine how changes in the values of α and β result in changes in growth.

3.1.1 Dominance

Recall that we are assuming growth depends on the larva's genotype. This means we specify values of α (growth rate) and β (maintenance cost) for each of the SS and RR genotypes, but apply equation 2.2 to get values for the SR genotype. Note that in Table 2.2 we found three different values for the dominance D needed in equation 2.2. Thus in Figure 3.1 we demonstrate the effects of these values of D on larval growth,

Figure 3.1: Plots of Growth for Varying Dominance Factors



assuming that α and β are the baseline values we calculated in Table 2.6.

Notice that when $D = 0$ (complete dominance) we expect that values of α and β for the SR genotype to be the same as the SS genotype, which is exactly what we see. Moreover, observe that the dominance values are biased toward the SS genotype, so we get the growth curves to be close to the SS genotype. Note that if $D = 0.5$, we would get a growth curve which splits the difference between the SS and RR curves.

3.1.2 Growth Rate

In Section 2.1.3 we called α the *growth rate*. This is because if $\beta = 0$, we would have exponential growth with α as the growth rate constant; hence, α controls the growth rate of a larva. It follows that increasing/decreasing the value of α while holding β fixed should increase/decrease the rate of growth observed (and the asymptotic maximum). Figure 3.2 demonstrates this for both of our baseline β values. Note that

the scales of the two plots are different, and that the vertical and horizontal lines represent the pupation times and masses we use to calculate our parameters.

3.1.3 Maintenance Cost

In Section 2.1.3 we called the constant β the *maintenance cost*; that is β is the average energy cost of maintaining a cell. Thus βm, where m is the biomass of the larva, represents the energy cost of maintaining the larva current biomass. Hence, we expect there to be an inverse relationship between changing the value of β and the effect on the biomass growth. We can see this effect clearly in Figure 3.3; for both values of fixed α, increasing β decreased the growth of larvae.

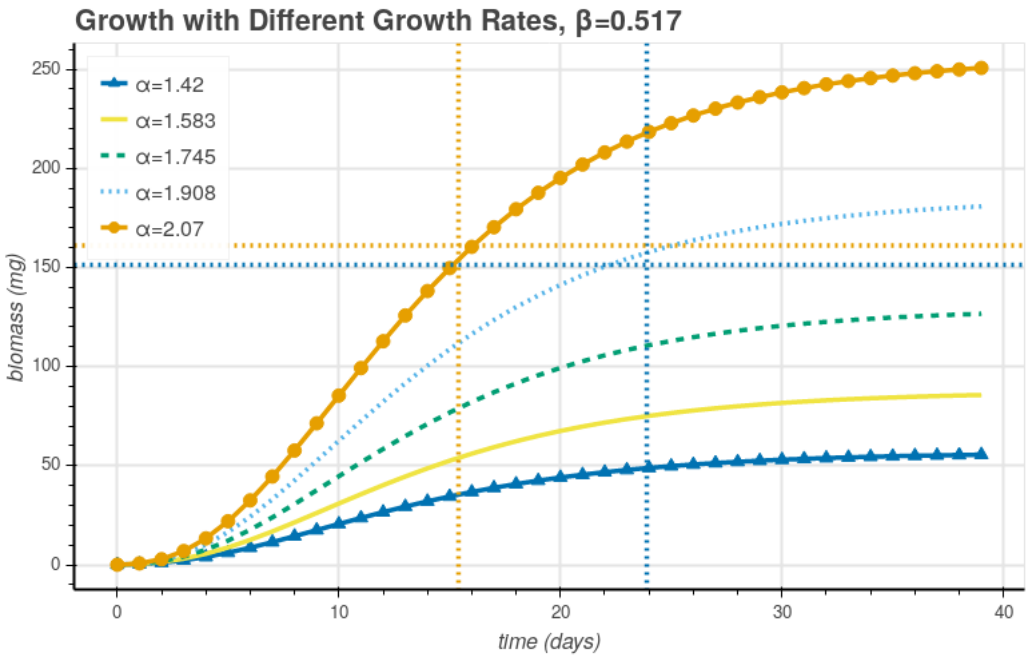
3.2 Cannibalism

Cannibalism represents the main mechanism by which susceptible larvae can outcompete their resistant counterparts. Thus we explore how cannibalism is affected by our parameter choices, and we see that we actually impose selection against resistance when we isolate cannibalism from other effects.

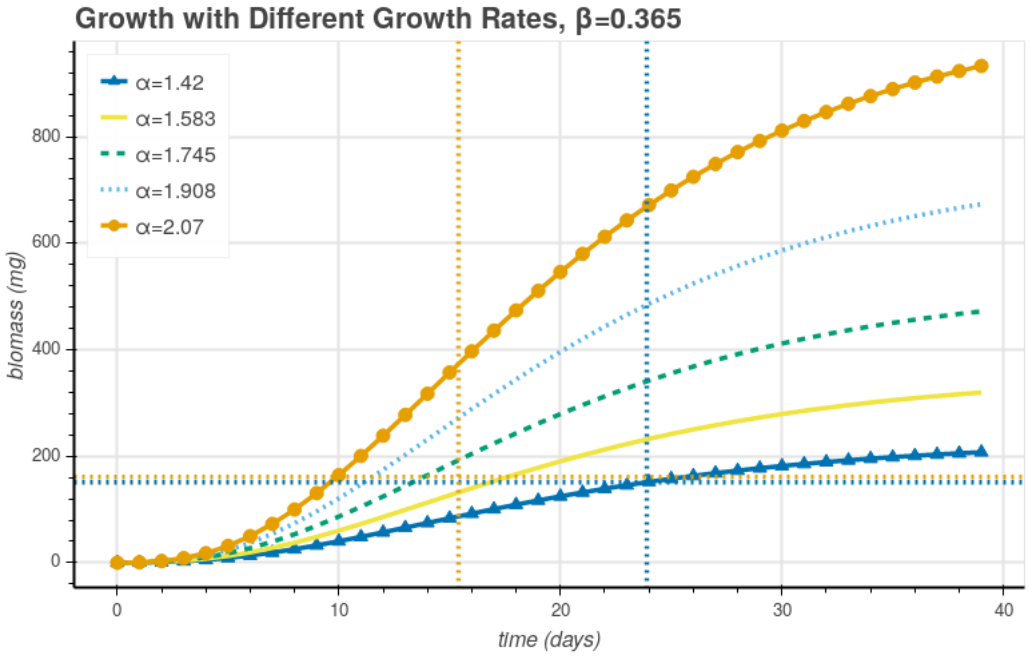
3.2.1 Movement

It is possible for changes in movement patterns to have downstream effects on cannibalism. However, when we performed simulations using many combinations of movement parameters for larvae, we found no effect on the frequency of resistance in the population. This should be expected based on our derivation of the movement in Section 2.1.6 because we essentially assume random movement without any bias toward other larvae or food sources.

Figure 3.2: Plots of Growth with Varying Growth Rates, α

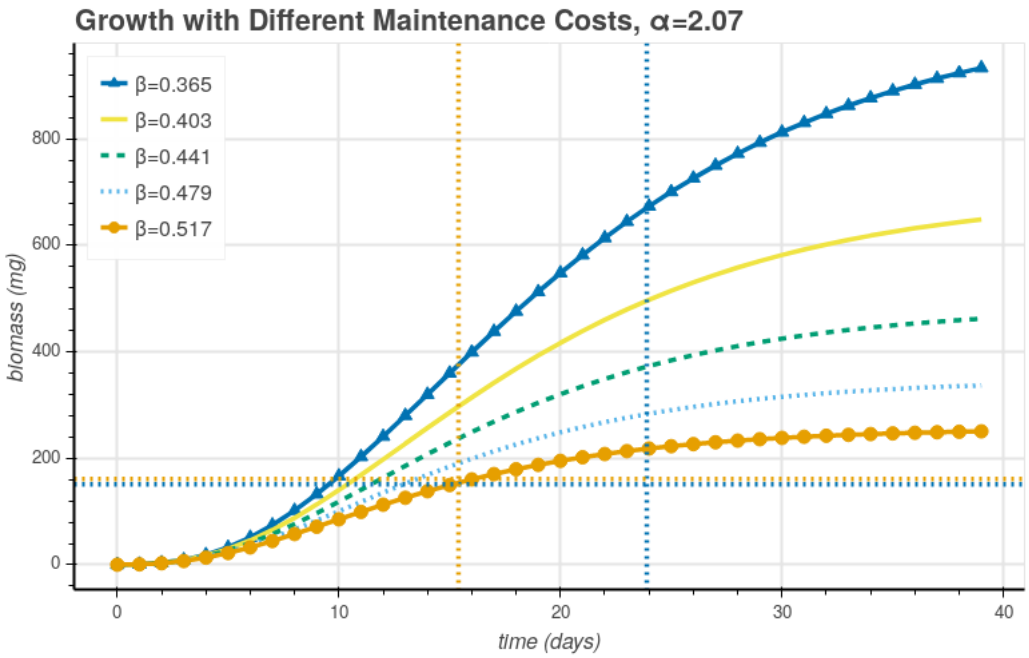


(a) β is from SS genotype

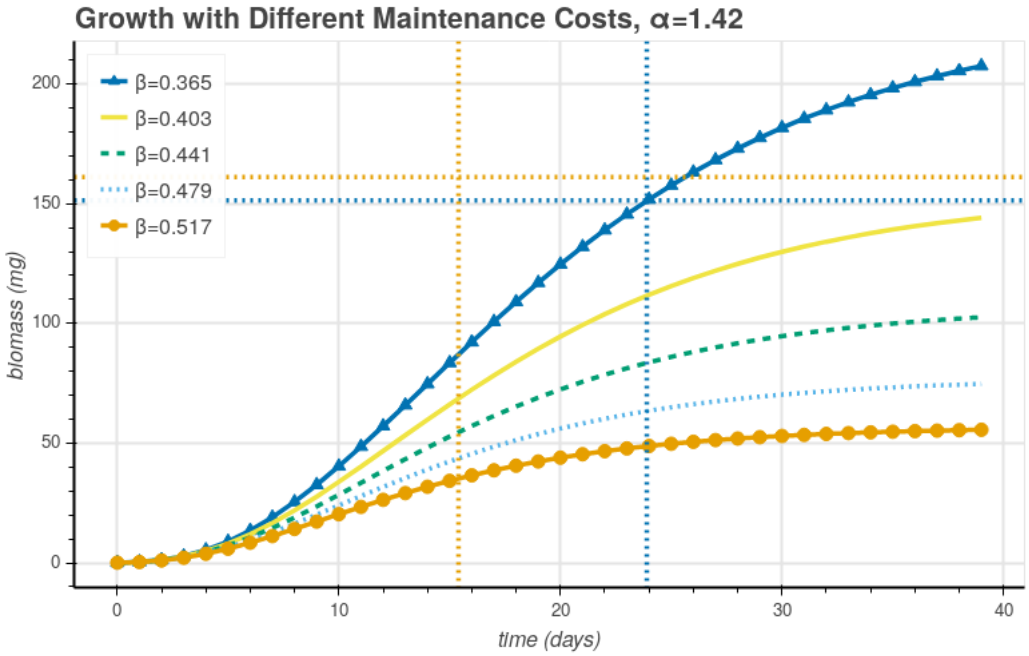


(b) β is from RR genotype

Figure 3.3: Plots of Growth with Varying Maintenance Costs, β

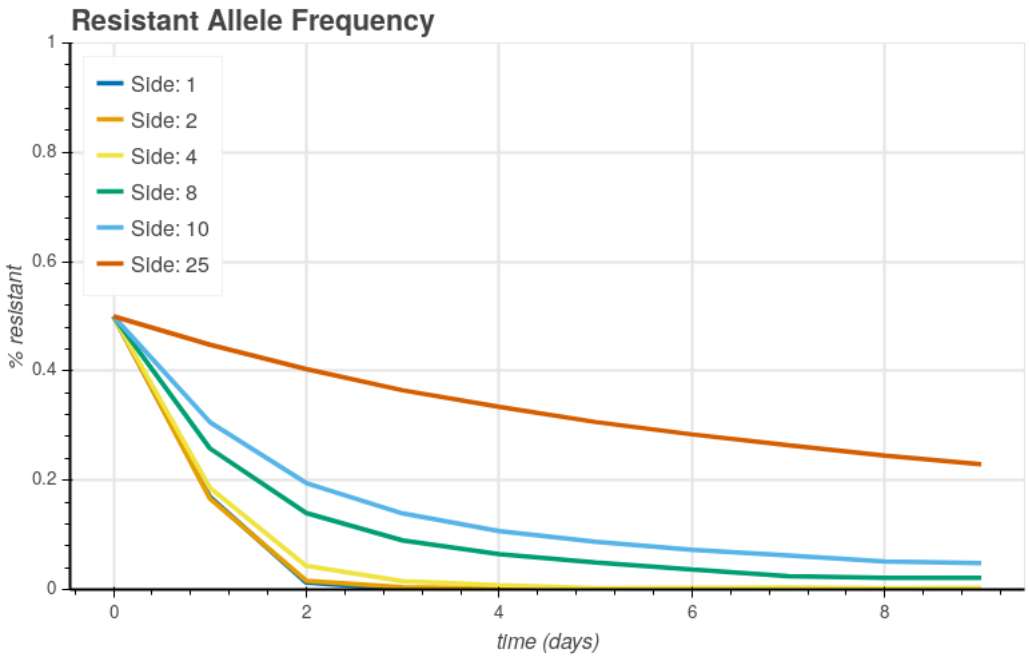


(a) α is from SS genotype



(b) α is from RR genotype

Figure 3.4: Cannibalism Grid Size



3.2.2 Encounters

In Section 2.1.4.1, we derived a cannibalistic encounter model. This model had a single parameter ρ_c, which we noted was the combination of on assumptions about grid-cell area and encounter detection distance. The grid-cell area is directly related to the grid-size of the plant-level grid because in setting the value for ρ_c we are assuming the total area the grid represents is fixed. The encounter detection distance can be thought of as larva aggressiveness because it is measuring the minimum distance that larvae are willing to tolerate other individuals at, meaning the larger this distance is, the more aggressive the larvae are.

3.2.2.1 Grid Size

Changes to grid size should affect the densities of larvae in a local environment. Since the encounter model was density dependent, we expect that increasing the grid size

should decrease the overall amount of cannibalism which occurs. This should result in smaller effects of cannibalism on the R allele frequency.

In Figure 3.4[1] we ran cannibalism on a population with equal numbers of RR and SS genotypes with all baseline parameters and no Bt, only changing the size of the plant-level grid from 1×1 to 25×25. As one can see, the larger this grid becomes the less effect cannibalism has on allele frequencies. Notice that our 10×10 grid choice does not negate the effects of cannibalism in this experiment, nor does it easily drive the R allele to extinction. This means that our 10×10 plant-level grid represents a good first assumption for the grid size.

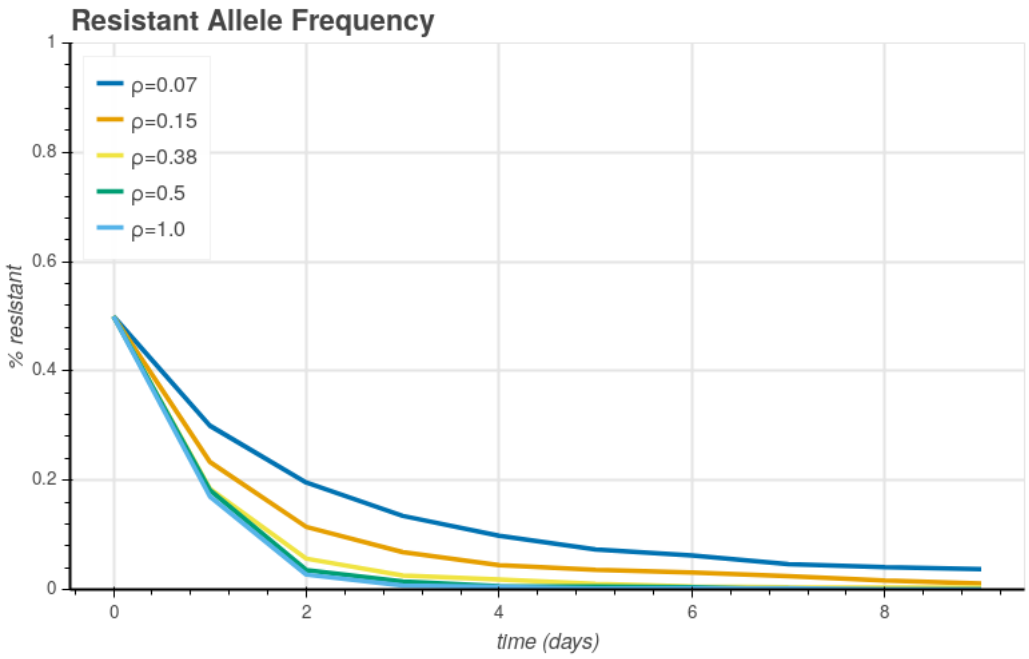
3.2.2.2 Aggressiveness

One way to view the ρ_c parameter in the encounter model is to consider it as a measurement of how aggressive the larvae are in cannibalizing other fall armyworms from the local area. Higher ρ_c means larvae are more aggressive toward other fall armyworms in their local area by increasing the probability that they attack other fall armyworms in their grid-cell. Thus for fixed environmental size and growth parameters, ρ_c controls the amount of selection pressure against the R allele.

In Figure 3.5 we perform the same experiment as we did for the grid size, only instead change the value of ρ_c (as opposed to changing grid-size). The values of ρ_c are the values we will later explore in longer term simulations. Notice that these values have small effects on the R allele frequency. This is because the interval for ρ_c was determined from the experimental data which suggested fall armyworms are aggressively cannibalistic. Hence, we have a range of values for ρ_c that range from some persistence of the R allele to rapid extinction of the allele.

[1]Note that the term "Side" in the plot legend denotes the side length of a square grid, such as "Side: 25" denotes a 25×25 grid.

Figure 3.5: Cannibalism Encounter Rate

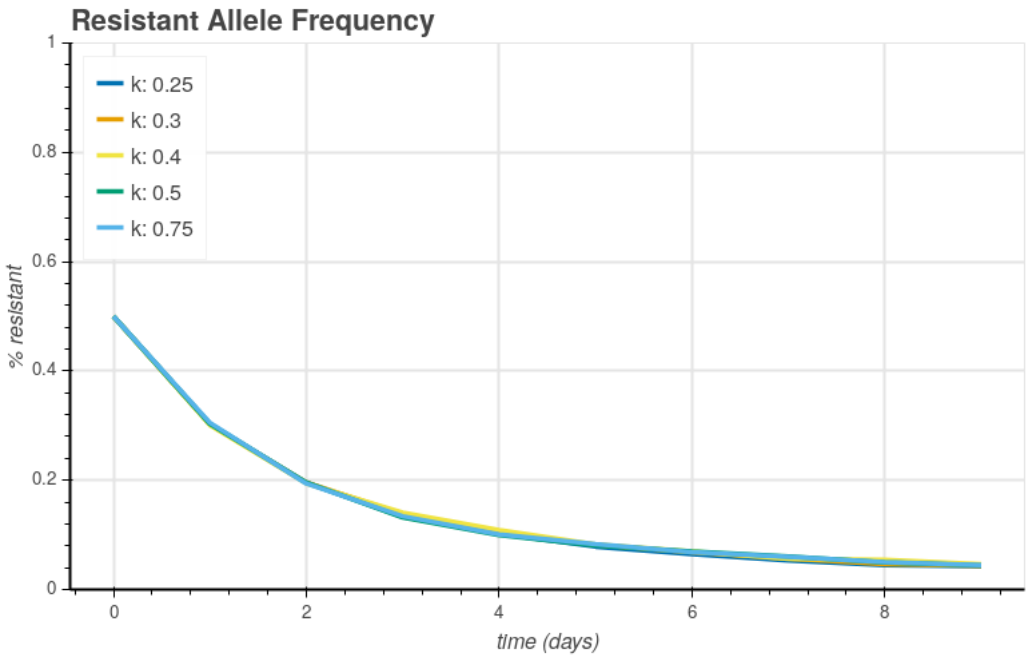


3.2.3 Cannibalistic Outcomes

In Section 2.1.4.2 we created a model for determining the outcome of a cannibalistic event, where we assumed that larger larvae were more likely to succeed over smaller larvae. Moreover, when modeling growth in Section 2.1.3 we assumed that RR genotype larvae will grow slower than SS genotype larvae. This should lead to the emergence of a selection pressure against the R allele.

However, this outcome model required a parameter k, which describes how rapidly a relative size advantage shifts toward the larger larvae. We could not find any data to estimate the value of k, so as noted in Section 2.1.4.2 we made an educated guess based on the variance in the biomasses of pupae. Fortunately, when we alter k, as in Figure 3.6, we find that k has little or no effect on the selection pressure against the R allele. This is likely due to an interaction with growth and how we are experimenting on a single cohort with little difference in hatching times. Moreover, this is the average

Figure 3.6: Cannibalism Outcomes



of 1000 replications the experiment with each parameter combination, meaning that any small effects are being averaged out. Hence, we can conclude that our guess for k is of little practical importance to the larger simulation.

3.2.4 Growth Effects on Cannibalism

When we experimented with altering the parameters for growth, we found a complex effect on R allele frequency which is difficult to capture in a diagram. We suspect this is because our alterations to parameter values failed to produce biologically relevant growth curves for the two genotypes. One pattern observed was when the growth curves of the two genotypes got closer together, the effects on the R allele frequency were decreased. This is expected because the selection pressure is mediated by differences in growth. Indeed, if the growth curves of the genotypes cross, we get extremely strange behavior of the R allele frequency over time. When the two growth curves

cross at time T, we find the trend in the R allele frequency to start varying a lot (reversing or oscillating). Thus we need to be careful to specify biologically appropriate grow parameters in order to get sensible results.

3.3 Reproduction

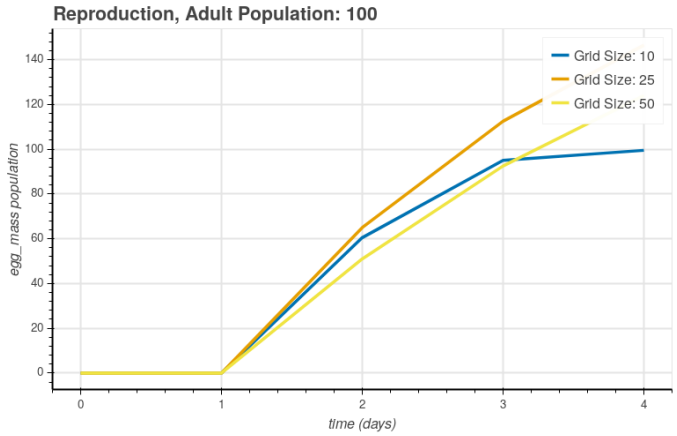
Reproduction in the IBM acts as the mechanism by which the resistance information (R alleles) is transferred from one generation of agents to the next. Recall that in Section 2.1.1 we demonstrated that the IBM can successfully recombine the genotypes of parents to form the genotypes of the offspring. Here we explore two other aspects of this process: preservation of the frequency of alleles from parents to offspring and the density dependence of the numbers of offspring.

3.3.1 Resistance Frequency

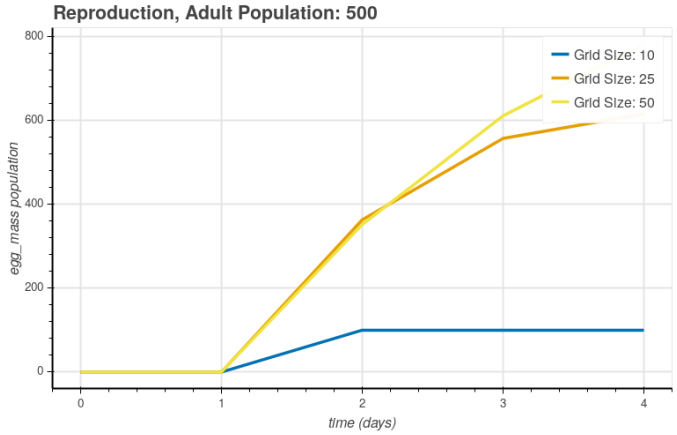
The frequency of R in the adults should approximate the frequency of R in the offspring. This means that the parameters of grid-size, mate radius R, and mate encounter rate ρ_m should not have any effect on the frequency of R in the offspring of a collection of adults.

To test this, we set up a simulation experiment which began with a set number of adults of each genotype. We then changed values of the above parameters while keeping all but that one parameter fixed and allowing reproduction to occur. We found that the frequency of R from the population of adults matched the frequency of R in the resulting egg population (for large numbers of trials). Thus we have high confidence that our IBM correctly performs mating.

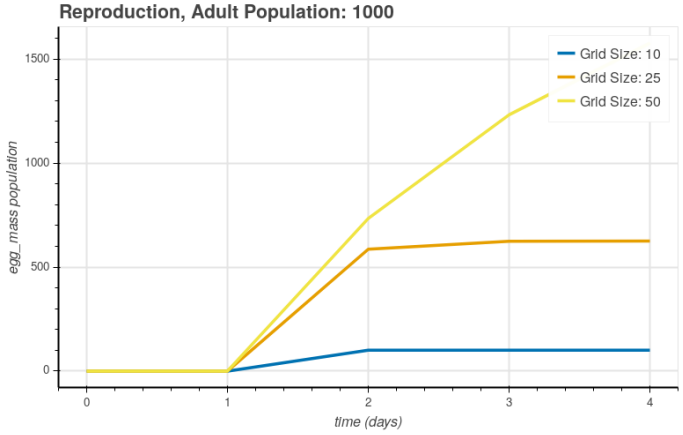
Figure 3.7: Plots of Egg Mass Production for Different Adult Populations



(a) Initial adult population is 100



(b) Initial adult population is 500



(c) Initial adult population is 1000

3.3.2 Density

Note that in Section 2.1.7.4 local density dependent female egg laying was imposed on the adult females. This density dependence only applies to within a single field-level grid-cell; these local dependencies should effectively create a global density dependence. However, this global density dependence will depend on the nature of the field-level grid. Moreover, the rate that the population of egg masses approaches the density limit should depend on the density of reproductive adults.

We set up simulations to test these relationships by adjusting the initial adult population size and the field-level grid size. In Figure 3.7 we report the results of these simulations. One can see that the smaller grids tend to have populations of egg masses asymptotically approaching between 1 and 1.5 times the number of grid-cells, which is precisely what we expect from our density model. Moreover, one can see that larger populations result in growing to the asymptotic limits at faster rates (the 10×10 grid saturates in one step quickly if there are more females than grid cells). Thus we can have reasonable confidence that production of the next generation of agents meets our expectations.

3.4 Survival

Bt mediates our main selection pressure in the IBM; namely, it creates selection against the S allele, resulting in the emergence of resistance evolution. Thus when no other selection pressures are invoked we should expect that the R allele frequency will rise rapidly when Bt is present.

In Section 2.1.9 we modeled Bt selection pressures through a daily probability of survival in the presence of Bt. Over the course of a single developmental cycle from hatching to pupating, we measured the effects of Bt on R allele frequency (see

Figure 3.8: Survival in Different Bt Conditions

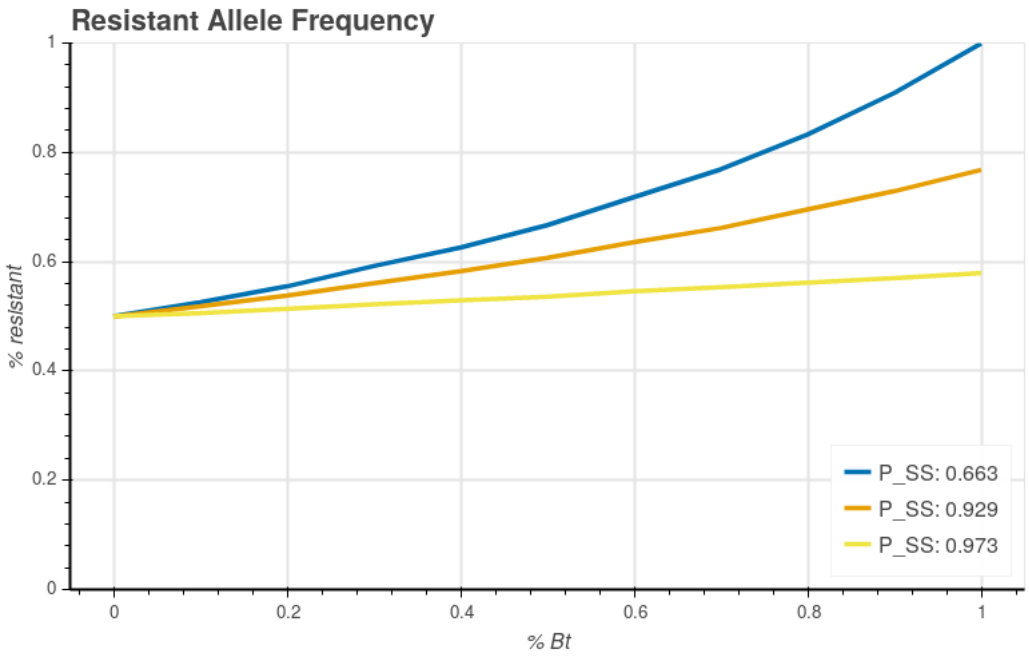


Figure 3.8[2]). One can see that the lowest Bt survival probability results in the susceptible larvae being eliminated in one generation under high percentages of Bt. While when the Bt percentage is lowered results in the creating of local refuge (locally no Bt) areas where extinction does not occur, allowing for the persistence of some susceptible larvae. Intermediate probability survivals allow for a small number of susceptible larvae to pupate in Bt environments. The highest probability is only slightly smaller than the survival probability in non-Bt, meaning we get a modest selection effect.

[2]In the legend `P_SS` denotes the daily survival probability for the SS genotype.

Chapter 4

Multi-Generation Simulations

Here we discuss simulating the evolution of resistance in the fall armyworm over long time periods. Unless specifically stated, we assume the baseline parameters with complete dominance of the S allele. Moreover, simulations span over 4200 days[1] (approximately 11.5 years); as will be shown in the results which follow, the first 200 days of the simulation were cut from analysis because that was the typical time required for the initial transient dynamics to settle.

4.1 Baseline Simulations

First, we explore simulation scenarios which only involve either RR genotype or SS genotype individuals to understand the long term baseline dynamics of our IBM. We will explore four different baseline simulations, each with the baseline parameters. The first simulation is a resistant genotype only simulation, while the others are all susceptible genotype only with Bt levels of 0%, 70%, and 90%. Note that the 70% and 90% Bt simulations will explore two different survival rates for Bt.

[1]We chose to simulate 4200 days because in experimental tests, this was the typical number of simulation days that could be performed for every one day of computational time, and we had a soft limit of 5 days of continuous computational time per simulation.

4.1.1 Convergence

Our IBM is highly stochastic, so we need to replicate the simulation several times in to get statistical averages of the simulation data. To understand the number of replications needed, we need to understand the convergence of the IBM.

To examine convergence we began by replicating every simulation 200 times[2]. To do this suppose that $X_m(t)$ is the mean of m distinct simulations (trials) at time t. Then

$$\int_0^T |X_m(t) - X_{2m}(t)|dt,$$

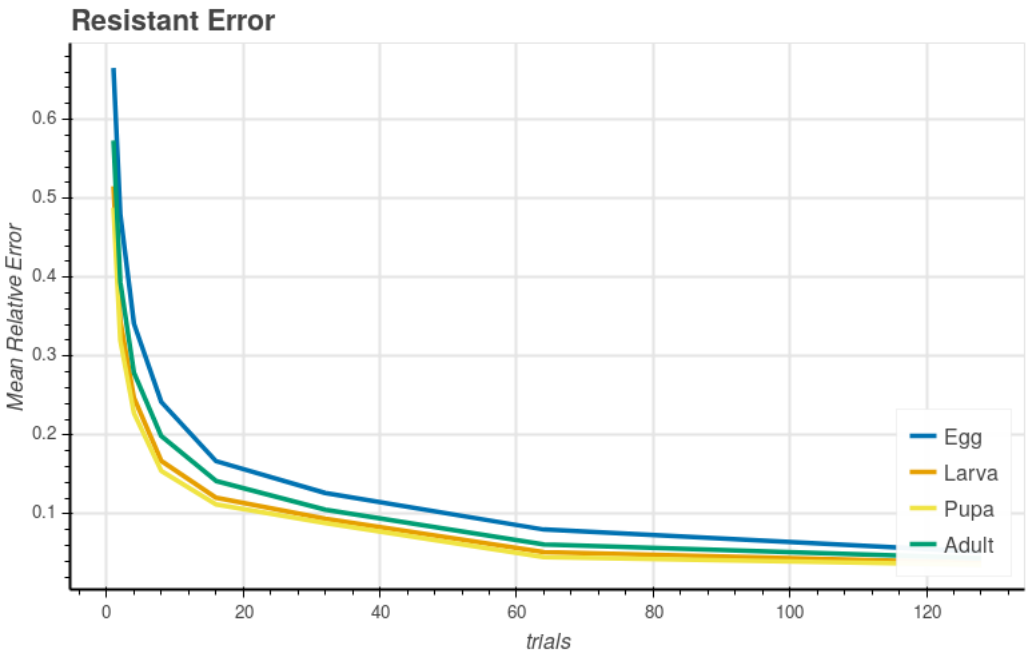
represents the total difference in the averaged simulation time-series when doubling the number of simulations. If $X(t)$ represents the true average time series, dividing by the integral over this series gives the relative error. Since we do not know $X(t)$ we will assume that it is approximately mean of all of our simulations. Thus we have that our relative error is given by:

$$\epsilon = \frac{\sum_{i=0}^{n} |X_m(t_i) - X_{2m}(t_i)|}{\sum_{i=0}^{n} |X(t_i)|}, \tag{4.1}$$

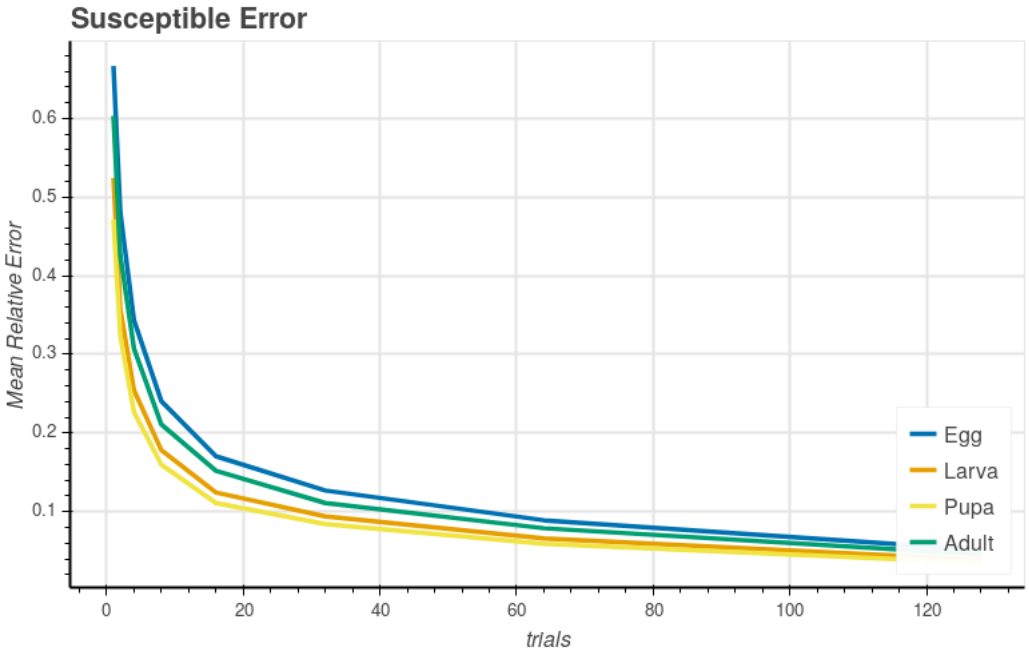
where n is the number of time points in our series. As the number of simulations, m, increases the central limit theorem implies that $|X_m(t) - X_{2m}(t)|$ goes to zero for each t implying that ϵ goes to zero. Figure 4.1 shows the results of computing this error for simulations involving the RR genotype only and SS genotype only with baseline

[2]All simulations were submitted as jobs to the HCC Crane supercomputer in batches of 200 identical simulations; however, due to variance in node performance and simulation performance sometimes a few simulations failed due to computation time limits. These failures were rare with at most 1 or 2 simulations hitting time limits from a given batch, and does not impact the convergence.

Figure 4.1: Baseline Simulation Errors



(a) Resistant (RR) genotype



(b) Susceptible (SS) genotype

parameters and no Bt. We see that in both cases we get a consistent exponential decline in the error to less than 10%, which is reasonable given computational resource limitations[3].

4.1.2 Population Data

The fall armyoworm population data exhibits generational pulses due to the stage structure of the life-cycle. Hence, we expect regular oscillations in the population of any given life stage.

Cryer and Chan [32] suggests that a time series with regular oscillations/variations should be decomposed using a seasonal decomposition. Note that our data contains possible zero values for any given life stage because life stages may be short enough that during a given generation all of the individuals in that life stage transition to the next stage before the individuals in the next generation transition in (all eggs may hatch before the next generation of eggs arise). This means that we statistically decompose the time series into three time series: trend, seasonality, and residual, which can be added[4] together to recover the original series. This decomposition relies on assuming a given seasonal frequency, which in our case is some regular period such as the generation time. The trend is a time series representing the overall trend in the data points; for a stationary (near-constant) time series we expect the trend to be near constant around the average value of the time series. Seasonality represents the periodic function with frequency given by the decomposition frequency that when added to the trend creates the regular pattern found in the time series. Finally, the residual is the time series which represents all the variation in the time series which cannot be accounted for by the trend or seasonal frequency. Residual, in some ways,

[3]Our computational limitations were 5 days of continuous run-time and a maximum of 1000 running simulations.

[4]If we had a non-zero series it would be possible to multiplicatively decompose instead.

can be thought of as noise in the data.

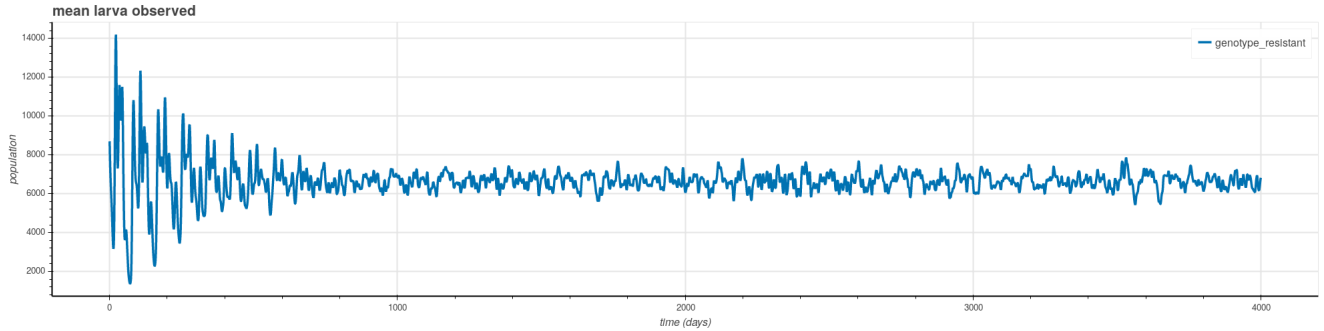
If we decompose the resistant genotype simulation with a 21 day frequency because that is the number of days the average RR genotype larva takes to pupate, then we arrive at the decomposition in Figure 4.2. Note that the seasonal component is not displayed because it is just a periodic function with period 21 days, and so it is difficult to see anything meaningful over the entire time period in our plots. Here we see in the trend what appears to be a regular repeating pattern which is still present from the original observed time series. This means that we may have not chosen the correct seasonal frequency, and that there may be a longer frequency that better explains the data's oscillations.

4.1.2.1 Periodograms

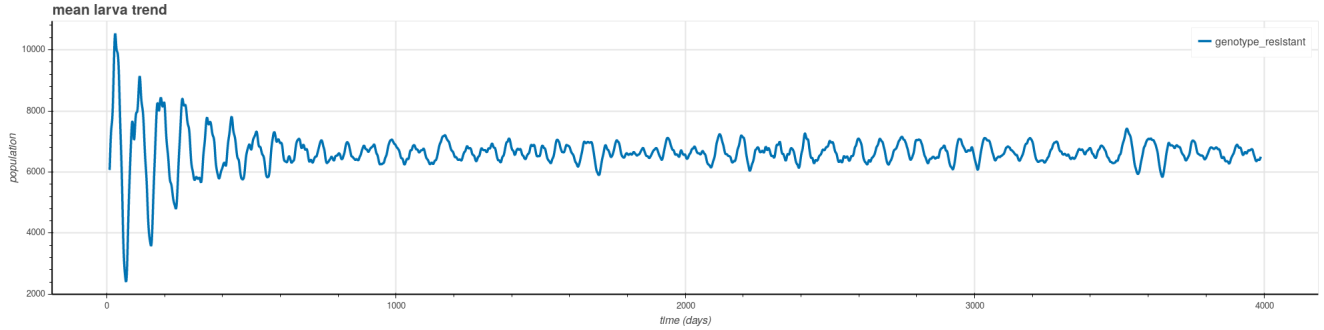
A common method for identifying the prevalent frequencies for a time series is to compute the periodogram of the data. This is calculated from the Discrete Fourier Transform of the time series by squaring all the coefficients of the resulting sequence. This results in a spectrum for the time series data, where the largest peaks denote the most important frequencies.

For example, Figure 4.3 gives the periodogram for the time series in Figure 4.2. This spectrum has three peaks (in decreasing order): 81.633 days, 21.164 days, and 12.308 days. The second largest peak corresponds to the average pupation time for resistant larvae. Note that for a fall armyworm in this simulation we expect that the average time between being laid as an egg to producing off spring is between 32 and 44 days. This suggests that the largest peak corresponds to the length of two generations, which means that generations are mostly correlated with the previous two generations. The third peak is likely the result of the variance generation time (the differences in the number of days it takes to fully mature among different individuals).

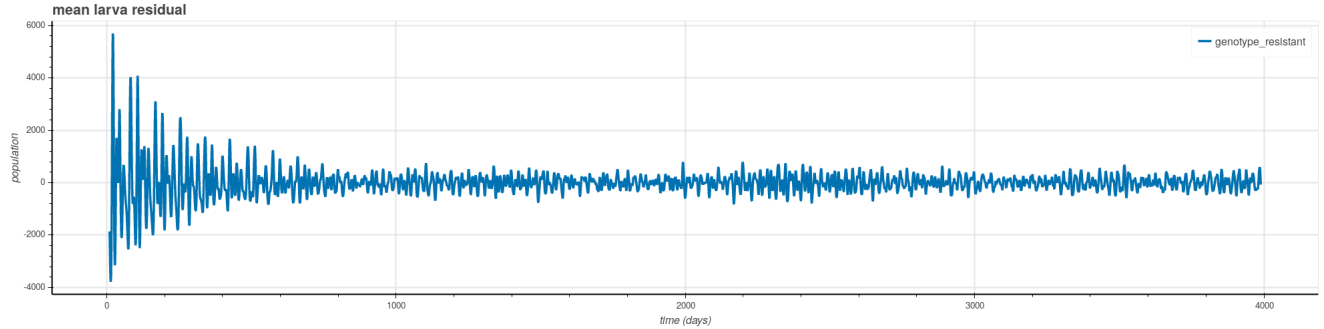
Figure 4.2: Baseline Resistant Time Series Decomposition, Frequency 21 days



(a) Observed

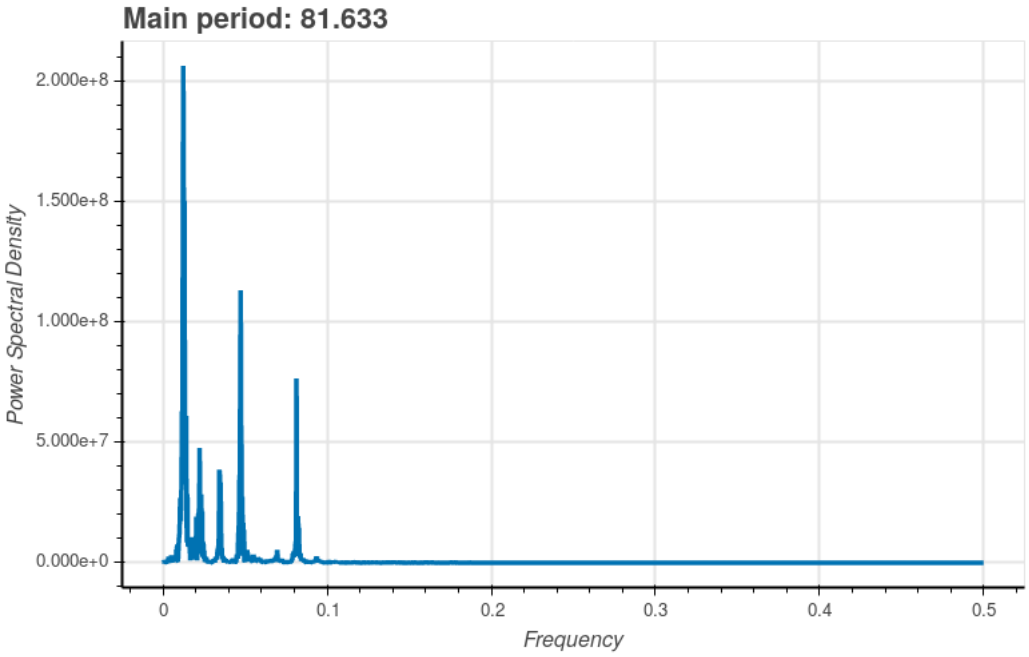


(b) Trend



(c) Residual

Figure 4.3: Baseline Resistant Periodograms



Note that similar analysis of the susceptible genotype with no Bt in Figure 4.4 results in similar estimations for its three main peaks.

In Figure 4.5 and Figure 4.6 we report the periodograms for the other two baseline experiments: susceptible genotype in 70% Bt and susceptible genotype in 90% Bt. All of these periodograms are slight shifts to the Figure 4.4 periodogram due mostly to higher variance in the measured populations due to smaller populations.

4.1.2.2 Baseline Time Series

Using the primary frequencies from each of the periodograms, we performed a seasonal decomposition of each of the baseline time series. In Figure 4.7 we give only the trend plots for these decompositions. Note that in order to present easier-to-read plots, we start the time series at day 1000. Figure 4.7a presents the larval time series; however, this plot is deceiving because it makes it appear as if the population levels of the

Figure 4.4: Baseline Susceptible Periodograms, 0% Bt

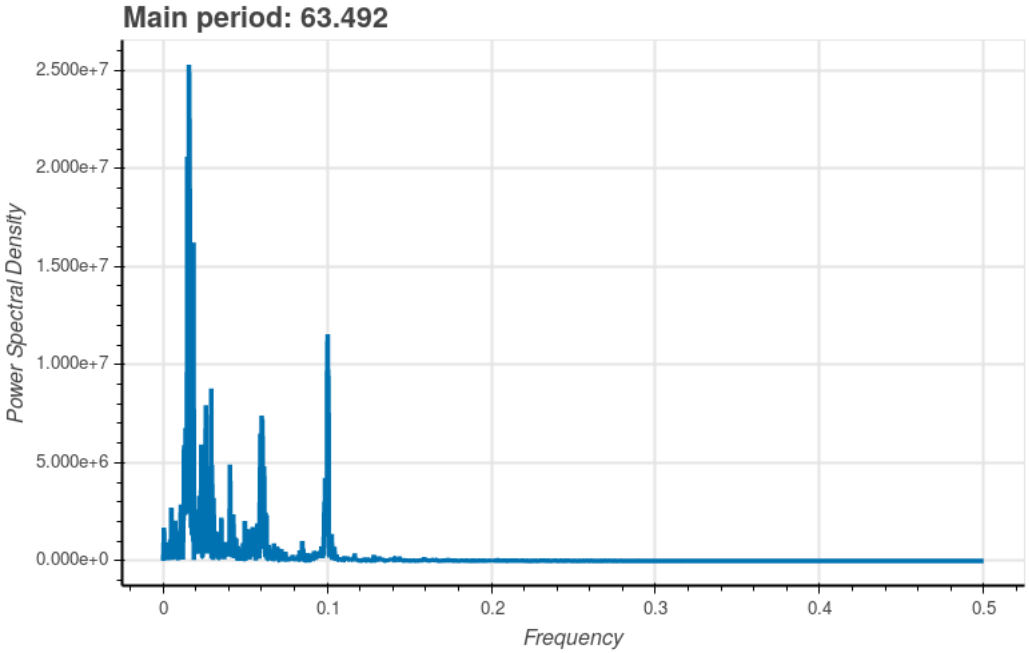


Figure 4.5: Baseline Susceptible Periodograms, 70% Bt

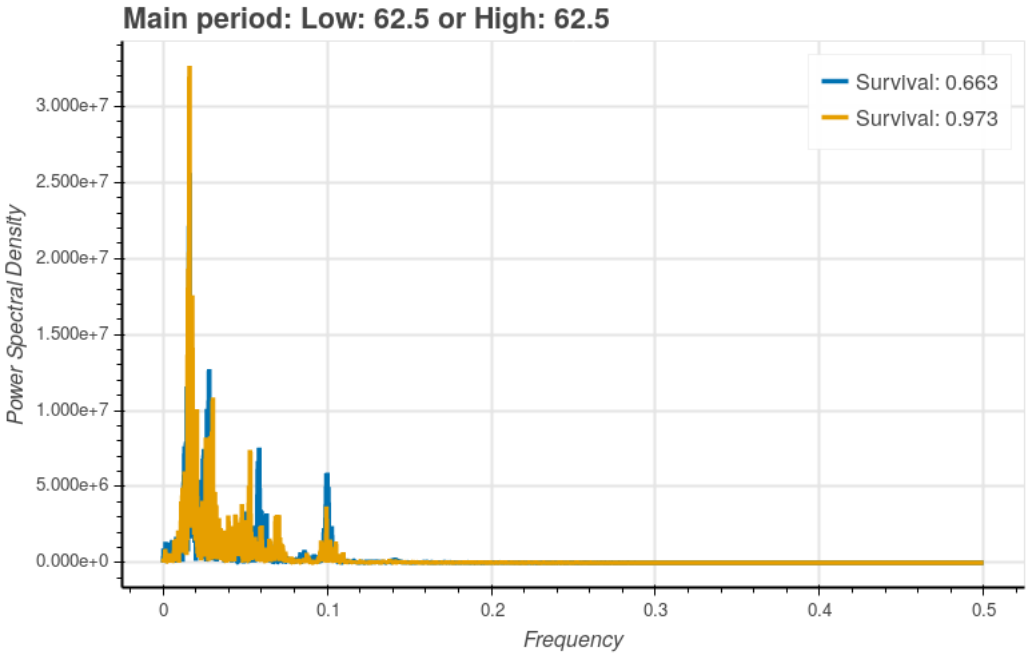
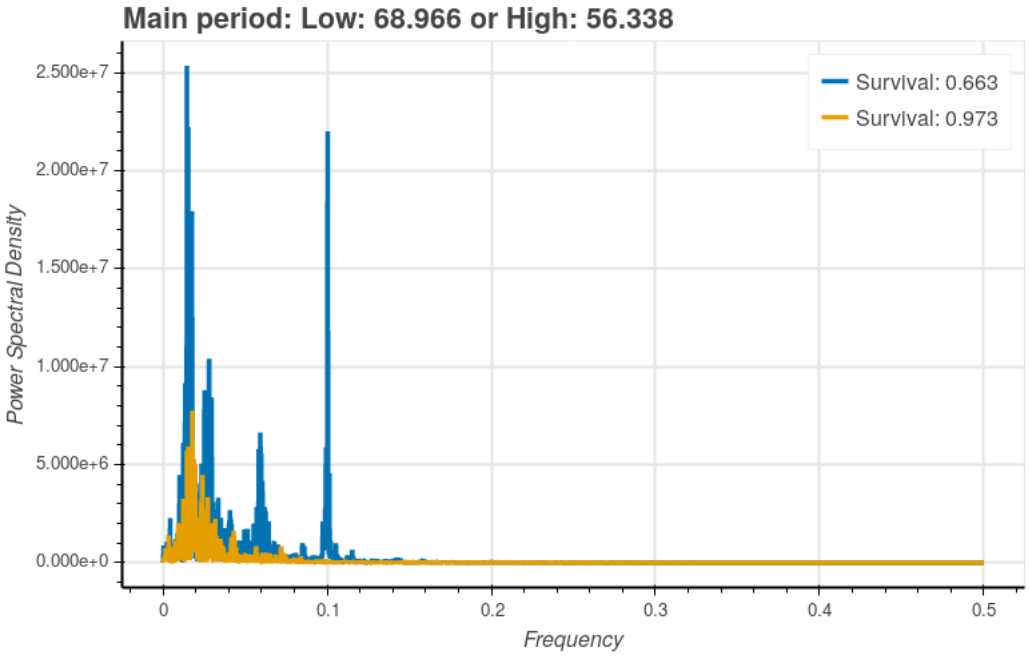


Figure 4.6: Baseline Susceptible Periodograms, 90% Bt



high mortality simulations (susceptible in 70% and 90% Bt with low survival) have significantly larger populations (when we expect the opposite). This is due to the combination of the density dependent egg laying and how we record populations. The high mortality quickly reduces the population of a field-level grid-cell to a level that opens the cell to new egg laying. However, when we scheduled the order of events for the IBM, survival events take place before development events. Since egg survival is independent of Bt, this means that for a single time-step we will measure a newly hatched larva population before the effects of Bt are accounted for. Thus the larva population quickly rises due to all the newly hatched larvae and is recorded for a single time step, and then this population rapidly declines during the next step. These large and unnatural fluctuations in larva measurements are produced across all of the field-level grid-cells at independent times. This results in the constant measurement of egg hatching waves because we are measuring our population globally across the

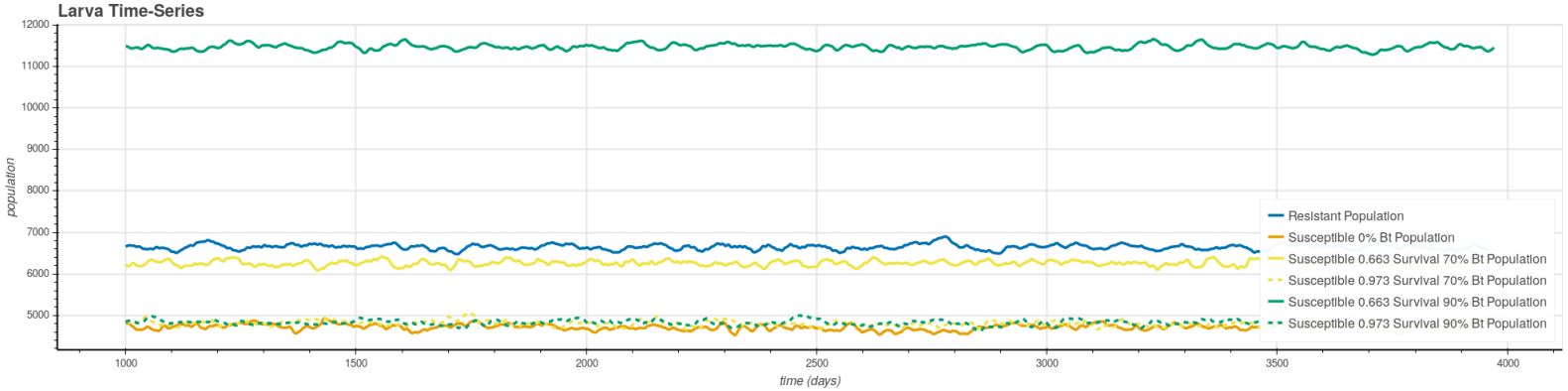
entire grid. A more accurate way is to examine the pupal population time series in Figure 4.7b because the pupal population will not suffer from these brief local explosions in population which are quickly reduced to near zero. One can see that relative to one another, each of these experiments stabilized to reasonable population levels.

4.2 Experimental Simulations

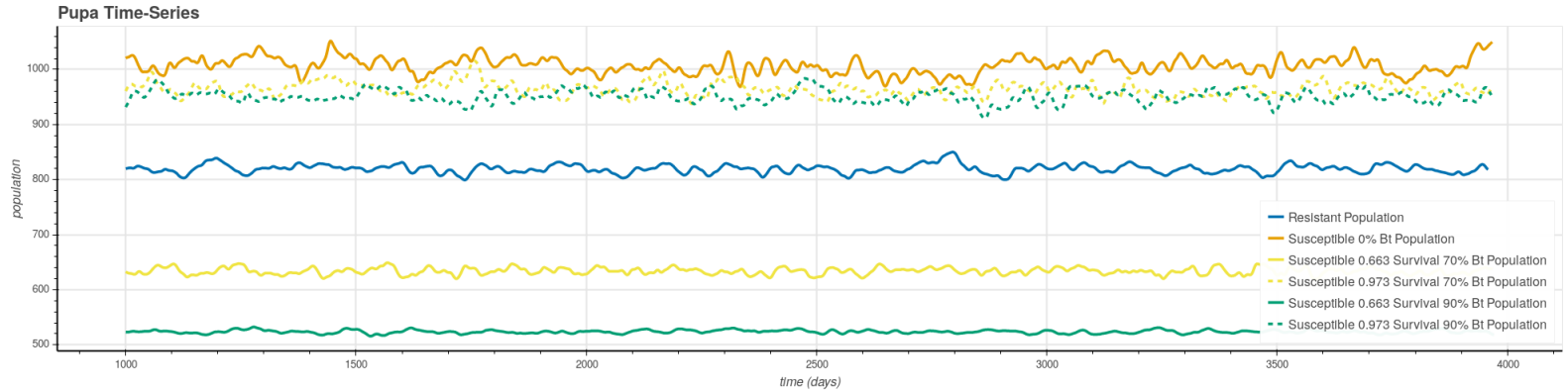
We will explore two sets of simulation conditions: a *low survival* (daily probability of survival for SS genotype in Bt is 0.663) and a *high survival* (daily probability of survival for SS genotype in Bt is 0.974). In both cases, we set the field to be 90% Bt and vary the cannibalism encounter constant ρ_c among the three values found in Section 2.1.4.3. We report the trend from a seasonal decomposition of the pupal data. Whenever a population of a given genotype experiences extinction, the periodogram for the time-series of that population becomes dominated by the extinction; what happens is that the periodogram shifts to the shorter frequencies (larger periods) meaning we no longer get useful information from it. Thus we used the frequencies found from periodograms whenever extinction did not occur. These periodograms closely matched the baseline experiments. For the low survival experiment (Figure 4.8a), the genotypes with a susceptible allele were driven to extinction. Note that the changes in cannibalism affected the average population size that the RR genotype stabilized around. For the high survival experiment (Figure 4.8b) we found an interesting transition; namely, the susceptible allele begins to dominate in the population when cannibalism gets strong enough.

In Figure 4.9, where we examine the proportion of resistance, we can see a similar picture. When survival of Bt is low, we find that resistance quickly evolves to fixation.

Figure 4.7: Baseline Trend Time Series

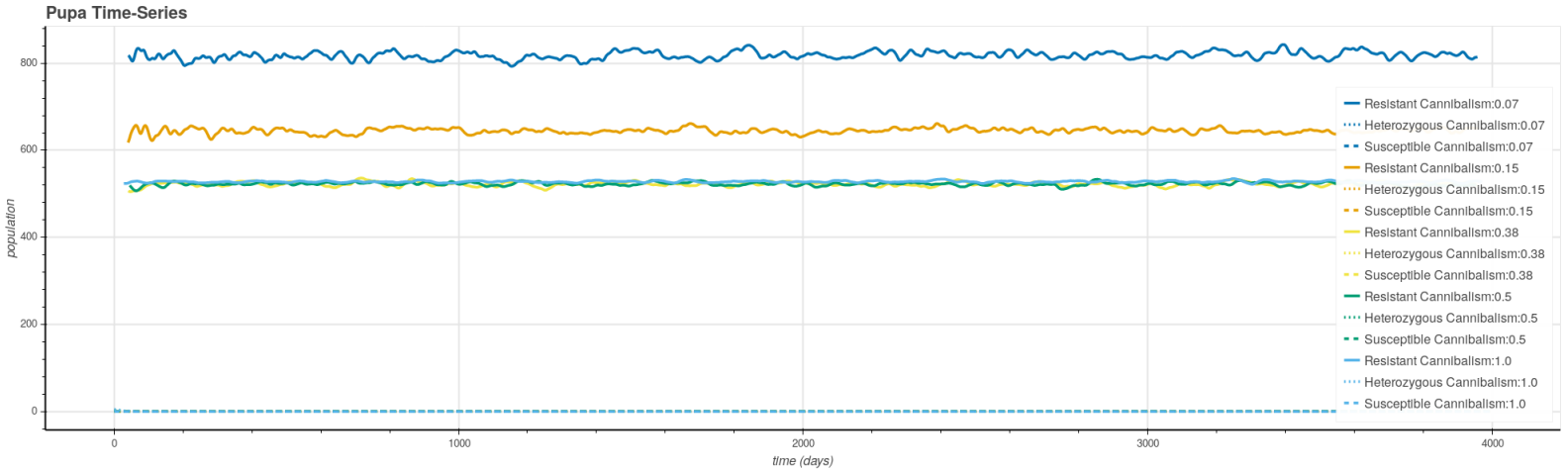


(a) Larva

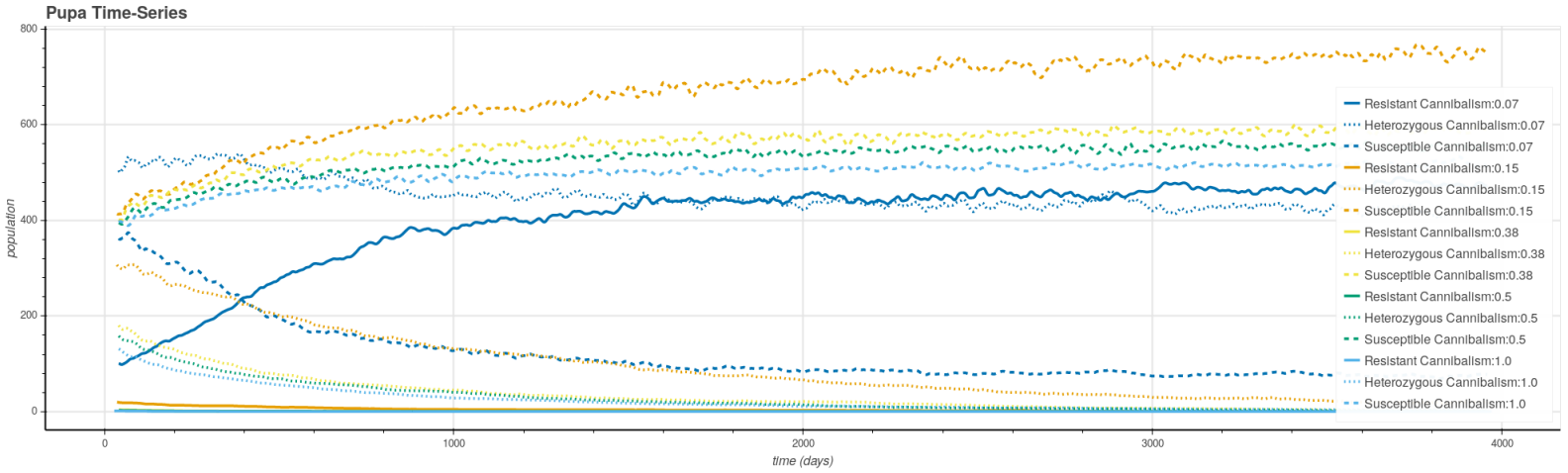


(b) Pupa

Figure 4.8: 90% Mixed Genotype Population Time Series



(a) 0.633 Survival

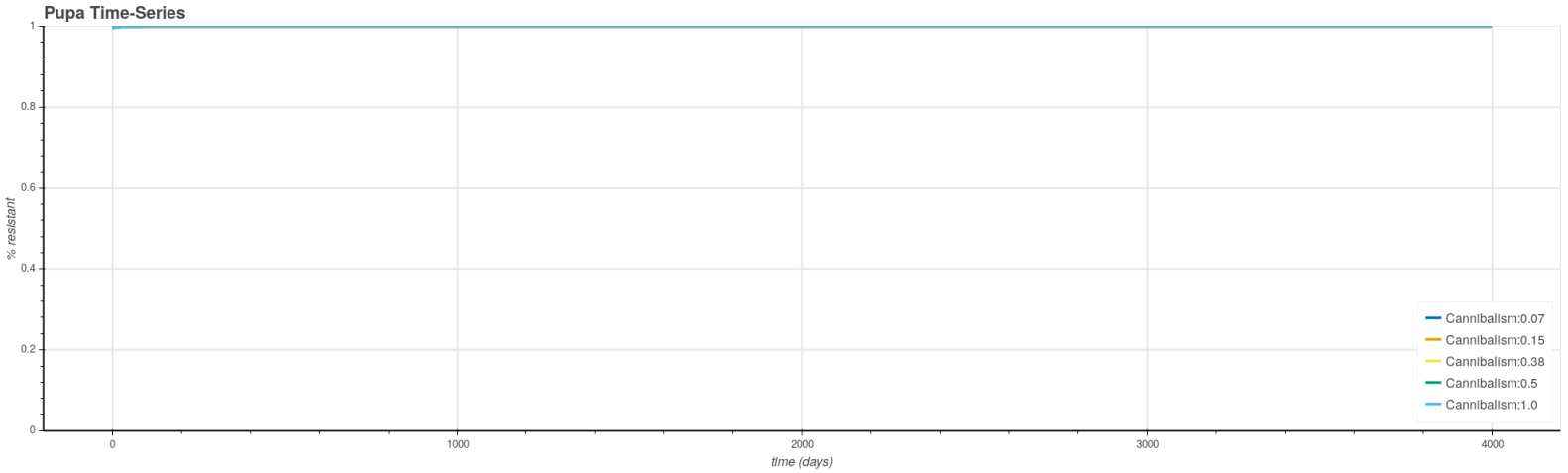


(b) 0.973 Survival

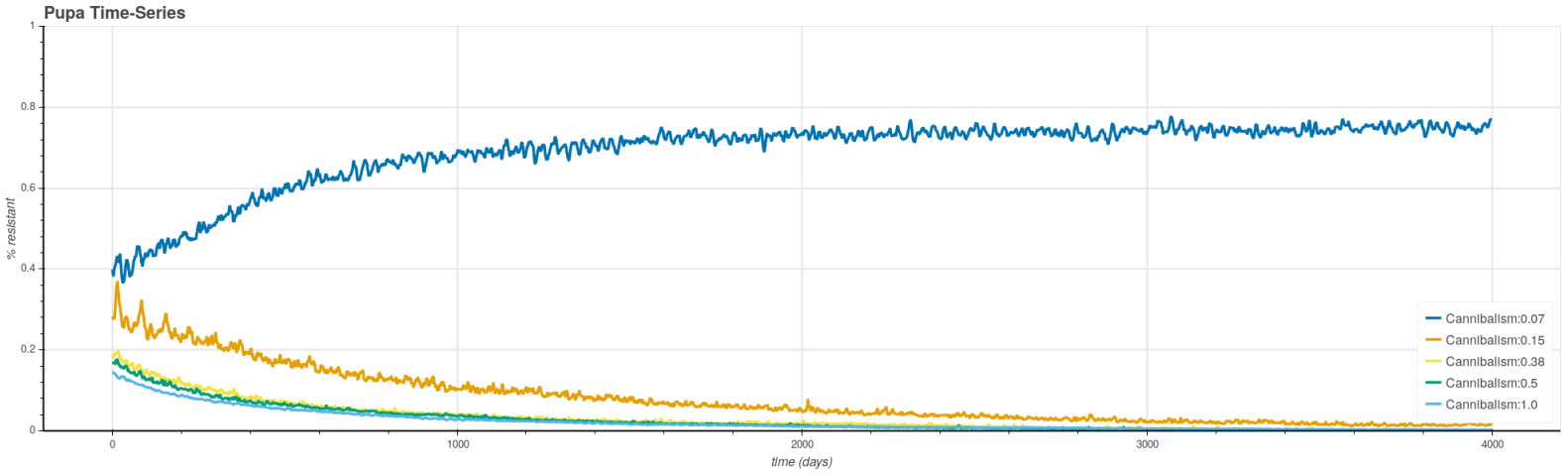
However, if Bt survival is higher, then the effects of cannibalism can result in an intermediate proportion of resistance; mostly RR and SR genotypes with a much smaller SS genotype. This means that more than half the population is susceptible to Bt.

Notice that the cannibalism rate that results in the intermediate mix of genotypes is the cannot rate found from the experimental data. However, the lower survival rate is closer to most of the Bt survival data on the SS genotype we surveyed. This suggests that using a larger refuge maybe able to increase the survival of susceptible individuals to recover the larger average survival for susceptible individuals. To test this idea we ran a third experiment where we considered the low survival case with 70% Bt.. This creates a larger refuge for the susceptible larvae, which may create a similar survival rate to the high survival in 90% Bt. We can see from the results of the experiment in Figure 4.10 that increasing the refuge size does not offset the effects of a low Bt survival rate in the evolution of resistance.

Figure 4.9: 90% Mixed Genotype Resistant Allele Time Series

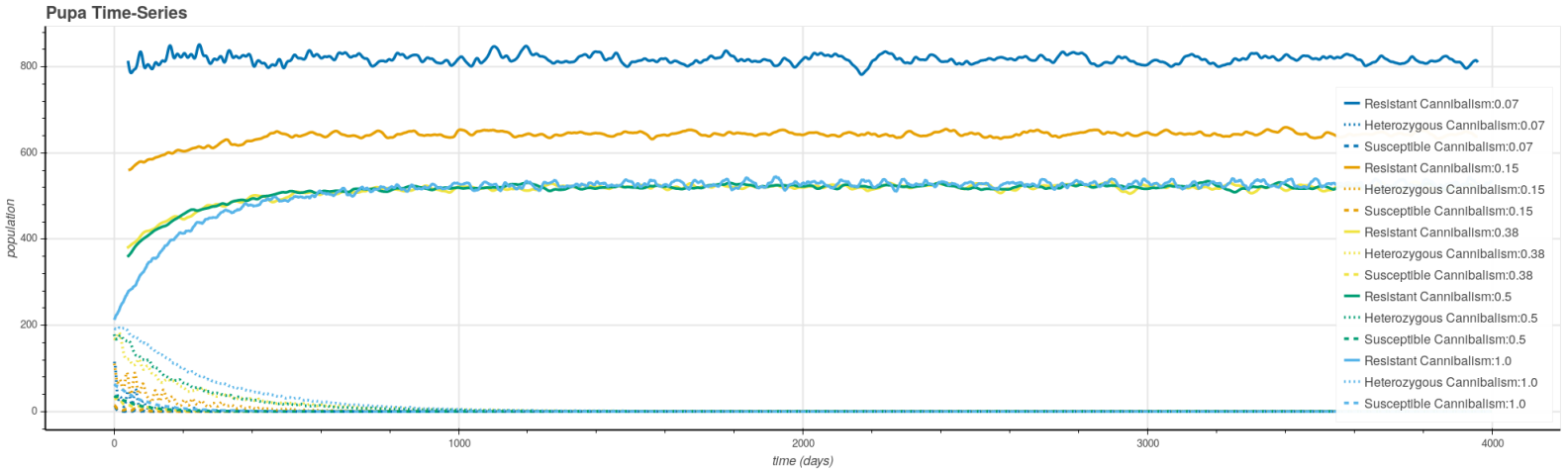


(a) 0.633 Survival

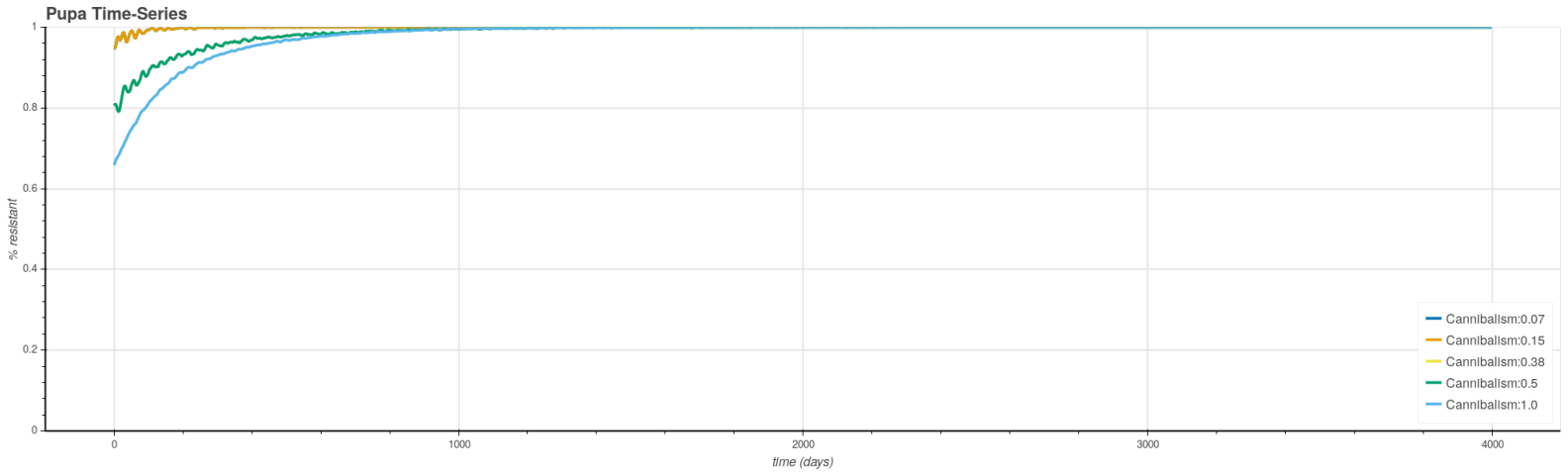


(b) 0.973 Survival

Figure 4.10: 70% Low Survival Simulations



(a) Time series



(b) % resistance

Chapter 5

Further Discussion

In this thesis we modeled the evolution of resistance to Bt insecticides for the fall armyworm. We hypothesized that this evolution would depend on the effects of Bt on susceptible individuals and the effects of Bt resistance on relative growth rates of individuals through cannibalism.

Our IBM created selection pressures in favor of the resistance when Bt was present and selection pressures against resistant individuals through cannibalism. Indeed, if Bt survival of susceptible individuals was large enough, then both the R and S alleles were able to persist in the population. This means that if Bt survival is low, it may be possible to find a refuge size large enough so that both the R and S alleles persist. However, we found that increasing refuge size (even to large refuge sizes for agricultural uses), still resulted in the extinction of the susceptible allele.

We did find that if cannibalism's selection pressure against smaller individuals is strong enough relative to Bt's effects, then it can offset the Bt resistance. This means that to change the level of resistance that evolves in the fall armyworm, it may be necessary to create a means by which cannibalism plays a larger role in the environment. If susceptible individuals grow more rapidly to larger biomasses than we assumed, then cannibalism's selective pressure against resistance will be stronger. Such a situation could be created by introducing multiple forms of Bt, which each

impose different growth costs to resistance making fully resistant individuals much smaller and take longer time periods to grow. A different way to strengthen cannibalism's effects is to spread the refuge throughout the field, instead of our monolithic division of the field into refuge/non-refuge areas. Doing this and allowing individual larvae to change local environments may introduce larger susceptible larvae into Bt environments with resistant larvae present, at which point cannibalism favors the larger larvae. Such an approach requires more detailed information on the toxicity of Bt for different biomass individuals and the relative make-up of a larva's diet. Thus to fully understand cannibalism's effects on resistance evolution we need more information on the precise effects of Bt on individual fall armyworms.

Since fall armyworm infestation in many areas is a seasonal phenomena which does not have a memory of the previous season's population because the eggs and pupae cannot tolerate low temperatures. It may be possible that cannibalism does lessen the rate of evolution of resistance in the small number of generations that occur each season because our model shows a selection pressure against resistance which can slow the rate of resistance evolution. We chose to ignore the early portions of our simulations because we do not have good information on the initial conditions of a fall armyworm introduction to a field. This means that to understand evolution of resistance in these areas of seasonal fall armyworm populations, we will require good data on the initial invasion population size and genotype make-ups. However, we can determine that in areas where the fall armyworm can have a year round presence that resistance evolution will occur rapidly under the conditions commonly observed.

Therefore, we conclude that cannibalism can have some effects on resistance evolution in the fall armyworm, but these effects are weak if non-resistant individuals have low survival rates in the presence of the insecticide. In order to completely understand cannibalism's effects on resistance evolution, we need more detailed in-

formation on how the insecticide affects individuals and more detailed information on the initial populations. We were unable to accomplish these more detailed studies because we ran out of time.

Chapter 6

Future Directions

Our IBM is extremely complex, which limited the extent to which we could investigate different parameter sets. For instance, it would be especially useful to explore the local spatial differences in the resistant allele-proportion, particularly the areas along the divides between the refuge and non-refuge areas. Moreover, a more complete statistical analysis of long-term simulations may yield a more complete and interesting understanding of the results.

Other useful directions include adding the effects of seasonal temperature variation because the fall armyworm is highly sensitive to colder temperatures. This could result in seasonal waves in the fall armyworm population observed in colder areas. Moreover in addition to temperature, it would be informative for modeling efforts to create multiple environment "patches," each patch linked together using the prevailing wind patterns. This would allow modeling of the evolution of resistance to Bt in the fall armyworm at a regional scale, which would be useful for designing pest management programs in areas that have been invaded by the fall armyworm like Africa, India, and China.

The IBM currently does not incorporate many details about the effectiveness of using different types of Bt. Indeed, there are common Bt-insecticide sprays that are used on non-Bt-transgenic crops for insect control. It would be useful to extend

our current IBM to support using insecticide sprays, since these sprays often have interesting temporal behaviors, such as being periodically applied, meaning there is a periodic change in effectiveness. Also insecticide sprays can be used to respond to a population of pests by applying sprays when the density of pests exceeds a threshold. Moreover, insecticide sprays are dose/concentration dependent, which would require changing the Bt-state from a Boolean variable to a continuous variable along with all the related downstream consequences (such as dose toxicity modeling for individuals).

Currently the IBM does not have a feedback between crop damage and the fall armyworm agents. Coupling the IBM to economic damage models would be highly useful for analyzing the trade-offs between interventions for long term pest control problems.

This IBM could easily be modified to suite pests similar to the fall armyworm, such as other insect species in the family Noctuidae or weevils. Many of these species are common economically significant pests (such as the true armyworm), which have similar behavioral patterns and/or effects as the fall arymworm. These species should require only minimal changes (mostly in terms of parameter fits) in order to create working simulations.

Using our IBM's framework, it is possible to model Malaria carrying mosquitoes, which are major public health threats. Mosquitoes have a long history of evolving insecticide resistance [6, 29]. So in addition to controlling the population of mosquitoes for public health reasons, policy makers have to be concerned with the future utility of the insecticides being used. Since our IBM supports extremely flexible environmental setups (see Appendix B) and is generalizable to other species of insect, it is suitable for use in helping identify a proper balance between population control for public health and long-term insecticide utility.

Finally, there are a host of improvements that can be made to the IBM's imple-

mentation. These mostly revolve around restructuring the source code organization to allow for effective parallelization of simulations, which should dramatically improve the computational tractability of larger simulations. These improvements take advantage of using multiple CPUs simultaneously, in order to handle larger numbers of agents.

Appendix A

Development Philosophy

In this appendix we describe our development philosophy which applies to both the creation of our IBM and its implementation, which occur simultaneously. The implementation of an IBM should feedback into the modeling and methods used in creating the IBM [53], so our development philosophy ties closely into our modeling approaches.

A.1 Test Driven Development in Science

Our software development methodology is based on *Test-Driven Development* (TDD) which was famously laid out in Beck [11][1]. TDD is an approach to software development whose central theme is the systematic creation of computer code in a testable fashion in order to minimize human-caused software bugs.

In TDD, one approaches writing code by first creating tests for the code one wishes to write. Once a test has been written, one then writes code to pass the test. As a simple example, suppose that one wants to write code which divides even numbers by 2 and adds 1 to odd numbers. A test for this could be to list out a bunch of

[1]This software development methodology was proposed as a method of writing maintainable commercial software.

correct examples of this process, say the first 10 integers[2]. Most computer languages now have built-in features or frameworks which facilitate the systematic creation of such tests. One starts by implementing the test or tests; then, implementing code until it satisfies the test with periodic checking of how well the it satisfies the test. For more complex situations, one iteratively creates tests, solving each new test until a resulting program emerges which satisfies/solves our original requirements. Since most tasks we want implement are complex, this forces us to carefully brake down the task into smaller interrelated parts that can each be easily tested. Each of these small tests is called a *unit-test*. This iterative process can result in hundreds of unit-tests[3] which not only verify all of these small parts, but also their assembly into larger and larger components of a more complex piece of software.

Unit-testing the assembly of simple components into more complex components is often accomplished by using *mocks*. Mocks are pieces of software which imitate other pieces of software in a fashion which provides exact constrained responses when used in a test while recording interactions with other pieces of software. For example, it is difficult to test parts of a program that involve random number generators; one can test these processes by mocking the random number generator so that it always outputs particular useful values (such as 0, 1, 2). The particular values output can be used to rigorously create tests of the program's use of the random numbers. The mock can then test its recorded inputs in the actual random number generator in order to verify that the inputs generate the valid random numbers. The abstract idea of mocks is useful when creating IBMs because it allows one to consider complex aspects of a problem as *black-boxes* where we only need to describe its affects not how it accomplishes them. This aids us in constructing insightful thought experiments to

[2]By this we mean correctly identify the results of running this on each of the first 10 integers, such as 1 results in 2, 2 results in 1, 3 results in 4, 4 results in 2, etc.

[3]For example our current version of the IBM requires 490 individual unit-tests

identify the necessary processes that we need to define while ignoring their details. These details can be later analyzed in the context of how to create the effect, without the need for considering the broader context.

Although this process seems like a lot of extra effort, most programmers do some sort logic checking process as part of writing software; TDD just formalizes this process. More importantly, it provides two key additional benefits. First, the formal breakdown process and testing force the programmer to carefully work out all of the logic they need, including considering verification. This means we can have reasonable confidence that the resulting code performs tasks correctly within the constraints of the unit-tests. So if one writes relatively good unit-tests which cover all (or most) of the scope of the tasks that one wants a piece of software to perform at the basic and intermediate levels, and that piece of software passes the tests, then one can have high confidence that the software has no bugs within the scope of the unit-tests. Second, TDD forces programmers to code their check process in a way that preserves it for the future. This is useful because it allows one to quickly demonstrate the bug-free nature (relative to the scope of the tests) at any time. Thus a collection of well designed unit-tests is a way to demonstrate to outside observers the correctness of a program. Moreover, these unit-tests allow the programmer to quickly identify bugs that arise in later updates/changes to a program, which is often a difficult task for complex programs. It has been suggested that this is the greatest advantage of TDD because this significantly reduces the costs (in terms of the amount of time necessary) of maintaining the program through future changes.

In science and mathematics, TDD represents a methodology to write programs so that one has confidence in the validity of the results it produces. Note that by validity here we mean that the program does precisely what the programmer intended (described by the scope of the tests) whether or not their intent correctly captures

the science or mathematics. TDD helps the scientist overcome any logical errors by forcing them to dissect the problems into simple enough concepts that can each be completely described. So even vastly complex processes, which have unknown solutions, can still have their programs logically verified.

It is important for scientists to realize that unit-tests are not intended to check scientific first principles or properties, such as conservation of energy. While these should be analyzed, circumstances like mathematical approximations can create situations where they cannot be *precisely* checked. This leads to the situations where a program's results are close but do not match the expected results. How does one know if these results are "close for correct reasons" (expected approximation errors) or "close for incorrect reasons" (errors due to coding bugs)? Distinguishing these can be difficult in complex scientific programs. Therefore, systematically checking scientific and mathematical computer programs at the implementation level for correctness adds to our confidence that the results fall in the "close for correct reasons" category. Hence, TDD should be used in scientific pursuits as part of the validation and verification process to avoid systematic errors.

For our development process, TDD additionally serve as an additional perspective on the POM approach [54] that is recommended for creating IBMs. TDD forces one to carefully consider all of the parts of the IBM and the biology being modeled, breaking it down into smaller parts, exactly like in POM. However, TDD pushes this breakdown process below the biology to the code function level; looking at the IBM and the biology it is modeling from this perspective gave us insights into how to improve aspects of the model, and more importantly it showed us the limitations imposed by the computer itself.

A.2 Object Oriented Programming and Modularity

Object oriented programming (OOP) is a method of programming which considers everything within a program an *object* [4]. Objects are abstract containers which contain variables together with the functions that manipulate and/or use the data in the variables. Classically functions in programming can be considered objects which manipulate a set of local variables in some fashion to give a result. IBMs are excellent examples of OOP in action because agents are excellent examples of objects; after all, agents are collections of variables that have behaviors which act on or through them. So, in OOP parlance, behaviors form the collection of functions bound to the agent variables.

OOP is a useful programming paradigm because it allows for easy abstraction of complex processes. The resulting abstractions one gets from considering the IBM as an OOP abstraction make describing the biological patterns forming the IBM much simpler. For example, throughout Chapter 2 we referred to agents from multiple, seemingly distinct, perspectives. These perspectives are all the same in the OOP abstraction, only requiring different particulars within the abstract agent object. This fits nicely into the POM approach, which often blurs distinctions among concepts with regard to particular situations.

The natural method for implementation of IBMs is akin to OOP because we consider distinct autonomous agents interacting locally. Implementing this requires one to create collections of nearly autonomous programs and then coordinate them to facilitate interactions. However, the number and state of agents is unknown and changes over the course of running the model. This makes it impossible to directly code every agent on its own; meaning, one must construct abstract agent programs, which falls precisely into OOP objects. OOP extends this to encompass the entire

programming process; consequentially, OOP provides a natural framework to assemble these collections of unknown numbers of programs together into a cohesive whole as required to implement an IBM.

Moreover, OOP efficiently modularizes computer code by wrapping associated computer code into (semi-independent) objects, which can be exchanged for new "versions" when modifying the code. This results in code being simpler to modify because relationships among aspects of the code have been abstracted to the point that distinct behaviors are contained and defined by the objects. So altering objects results in altering the behavior of the program without need to alter the global structure of the program. This moves us toward the goal of producing a general purpose model because OOP's resulting modularity and abstractions makes it simple to identify where changes need to alter things like agent behavior or the environment without having to rewrite most of the computer code. Furthermore, this modularity assists us with encapsulating our unit-tests by allowing us to write tests which are akin to describing a biological situation directly. Finally, considering objects by themselves and together allows us to more carefully and completely unit-test the entire logical scope that we need to cover.

Appendix B

Environment Structuring

Here we detail the implementation of the environment described in Section 2.2.1 together with related technical components. These related components form the agent handling system, which is a purely computational structure to allow the program to logically organize the agents for use by the IBM.

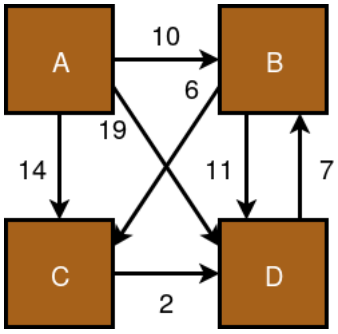
B.1 Space

Recall that we defined a *multi-level* spatial grid in Section 2.2.1.1 for our IBM. In particular, this is structured as grids embedded withing other grids. The implementation of space relies on abstractions of both a grid and this embedding.

B.1.1 Abstraction of Grids

In general, we can consider a grid as a collection of grid-cells which are arranged geometrically, usually as tiles in a plane (see the grids in Figure 2.21). This geometry enforces a relationship between grid-cells by determining which grid-cells are adjacent to which other grid-cells. Thus we can abstract the idea of a grid to a graph, where graph vertices denote grid-cells and graph edges represent the geometric relationships. Typically, graphs derived from geometric grids are called *grid-graphs*; in these graphs

Figure B.1: Example of a Patchy Landscape Graph



one assumes that all adjacent cells are the same distance apart with distance scaled to 1. This common distance between adjacent vertices can then be scaled by weighting each edge. Using these weights one could scale the entire grid by using the same weights for all edges, or one could distort the geometry of the grid by adjusting edge weights individually. Moreover, one could add, remove, or direct edges to change the *topology* of the space. Thus it is possible for any finite weighted directed-graph to form the "grid-graph" for the IBM's space. For example, a patchy landscape where winds prevent individuals from transiting between two locations in one direction but allow transit in the other direction could use a directed graph to represent the directions and then edge weights for the distance, see Figure B.1.

This complete abstraction of the grid was done for several reasons. Chief among them is that hexagonal grids are often difficult to accurately implement because there is not a simple formula for determining adjacency which respects computer arithmetic easily (that is, without requires extensive rounding corrections). One method to implement a hexagon grid is to build a lookup table, which is equivalent to representing a graph. Since we implement the hexagonal grid using lookup tables, it was simple to allow the space to support any lookup table (*i.e.* graph). Indeed, we extend lookup tables, beyond adjacency, to represent all possible distances extending from a vertex

in the graph[1]. The most efficient method for generating these lookup tables given these types of graph is Dijkstra's algorithm [41, 61]. We employ Dijkstra's algorithm to build this lookup table for our hexagonal grids by passing it the corresponding grid graph, so we can support any of the above types of graphs by passing different information. Theoretically, this allows us to support patchy landscapes and/or complex geometry, such as what individual crop plants define. We choose hexagonal grids for our IBM to simplify the problem.

Note that Dijkstra's algorithm is optimal for our use-case; however, large graphs still require extremely expensive initial computations (10 hours to 1 week) because we must apply this algorithm to each vertex in the graph individually. Thus we use a system to pre-compute several spatial configurations, and then save each to python binary file that can be loaded by any simulation to prevent this computation initialization cost.

B.1.2 Agent Location

Agents need to have location addresses within the multi-level grid structure. Note that in graphs vertices (grid-cells) are assigned integer values as labels. Location addresses can then be assigned by listing these grid-cell locations in order of increasing level, *i.e.* outermost grid-cell through to the inner most grid cell in the embeddings. For the space in Section 2.2.1.1, this is organized as: [0 (for organizational purposes), field-level vertex, plant-level vertex], for example `[0, 1, 2]`. Note that when agents do not have a specific coordinate below a given level this list is truncated.

[1]This lookup table takes $O(n^2)$ memory space to store. This ultimately becomes a limiting factor on the grid size used.

B.1.3 Multi-Level Grid

To construct the multi-level grid, we follow logic similar to the location addresses. Instead of building lists we build hierarchical trees, where each level of the tree contains nodes which represent locations at a given level in the multi-level structure, while edges between nodes in different levels represents the embeddings, see Figure B.2. This tree has a single root node 0 at the top representing all of space, with nodes for each field-level grid-cell adjacent to the root. Below these nodes is a node for each grid-cell of the plant-level grid. Hence, location addresses correspond to this tree by allowing us to trace from the root node down to the node representing the location in the address. For example, if one traces the location `[0, 1, 2]` through the tree in Figure B.2, one traces the dashed path depicted. Each node of this tree is also linked to the lookup table for the graph that defines the spatial geometry of the graph it is associated with. Hence, a location address can be used to get the correct spatial geometry around that address at the level the address corresponds to. Technically, there is no limit to how many levels of grids are supported, but there are practical limits such as memory storage of all the lookups[2].

We improve the overhead (in terms of CPU cycles) by flattening this tree into a single Python dictionary by the mapping of location lists to unique dictionary keys. This allows the hierarchical trace to be done in a single step instead of several recursive steps. Again, as with the hexagonal grid choice, supporting "unlimited" numbers of length-scales by embedding grids within grids is a natural extension of our multi-level grid. In particular, this allows for the possibility of having linked patchy landscapes.

[2]We limit the footprint by only creating one graph lookup table for each distinct graph and then using a pointer to link that graph to multiple locations within the hierarchical tree if necessary.

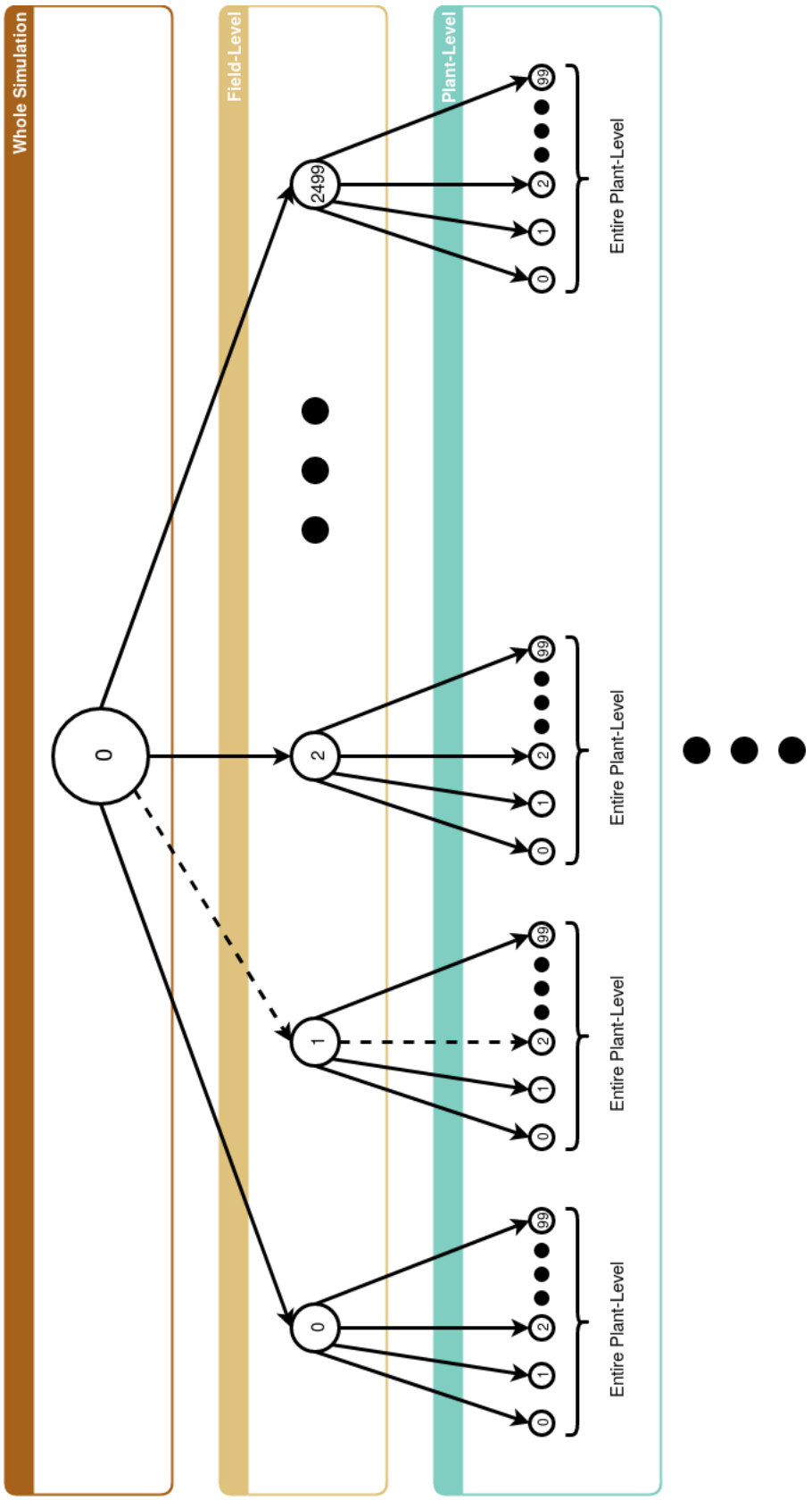


Figure B.2: Hierarchical Lookup Tree

B.2 Agent Handling

The agent handling system shadows the space implementation, so the spatial relations between agents are implicitly tracked. This is important because both the foraging (Section 2.2.4.1) and reproduction (Section 2.2.4.2) behaviors require finding agents which have a specific spatial relationship to another agent. If we did not encode this information implicitly in our methods of storing agents, querying these agents would be computationally expensive due to the need to sort/filter to recover the necessary agents. Moreover, beyond the spatial considerations, we additionally divide the agents by their `agent_key` variable and `alive` variable state.

Agents are stored using the hierarchical tree from Section B.1. Where instead of having a distance lookup table linked at a given node, we have an agent storage mechanism. This allows finding agents related by spatial location to be performed by the tracing process we already discussed. Agent storage is accomplished by a series of nested python dictionaries (extensions of these objects), which for an agent are keyed to `agent_key` then `alive` then `unique_id`. This allows the simulation to quickly get the needed agents by location, agent life-stage, and alive/dead state[3].

We also store agents up and down this hierarchical tree so that accessing agents at different levels of the spatial grid is possible. This is accomplished by adding an agent to the agent storage system at each node as one traverses along the hierarchical tree as we find the agent's location in the tree. Once at an agent's location in the tree it is then added to storage systems located at any node below the location node. The extra overhead of this is negligible when compared with the filtering costs of storing agents only at the lowest levels of the multi-level grid only.

[3]Note that the agents themselves are stored in dictionaries because dictionary comprehensions (lookup, add, remove) in python are an $O(1)$ computation as compared to lists which are $O(n)$ for most of these operations. All of this effort translates to a massive improvements in computational performance, resulting in simulation times dropping by an order of magnitude.

Appendix C

Source Code

This appendix contains all of the computer code needed to create any of the simulations used during this project. Additionally, this computer code is hosted as a software repository on GitHub at the following address: `https://github.com/WilliamJamieson/FallArmyworm_Thesis`. We do not include all of the specific run or analysis files due to their repetition in this appendix; however, they are all contained within the software repository.

This appendix is ordered by the directory structure of the project with file names as section headings, where the section outlines from the series of directories containing the files. Below, we give the main directory tree for the computer code, and subsequent sections will contain the directory trees for the code contained within them.

```
FallArmyworm
├── LICENSE.txt
├── models
├── parameters
├── source
└── test
```

C.1 LICENSE.txt

```
                    GNU GENERAL PUBLIC LICENSE
                       Version 3, 29 June 2007

 Copyright (C) 2007 Free Software Foundation, Inc. <https://fsf.org/>
 Everyone is permitted to copy and distribute verbatim copies
 of this license document, but changing it is not allowed.

                            Preamble

  The GNU General Public License is a free, copyleft license for
software and other kinds of works.

  The licenses for most software and other practical works are designed
to take away your freedom to share and change the works.  By contrast,
the GNU General Public License is intended to guarantee your freedom to
share and change all versions of a program--to make sure it remains free
software for all its users.  We, the Free Software Foundation, use the
GNU General Public License for most of our software; it applies also to
any other work released this way by its authors.  You can apply it to
your programs, too.

  When we speak of free software, we are referring to freedom, not
price.  Our General Public Licenses are designed to make sure that you
have the freedom to distribute copies of free software (and charge for
them if you wish), that you receive source code or can get it if you
want it, that you can change the software or use pieces of it in new
free programs, and that you know you can do these things.

  To protect your rights, we need to prevent others from denying you
these rights or asking you to surrender the rights.  Therefore, you have
certain responsibilities if you distribute copies of the software, or if
you modify it: responsibilities to respect the freedom of others.

  For example, if you distribute copies of such a program, whether
gratis or for a fee, you must pass on to the recipients the same
freedoms that you received.  You must make sure that they, too, receive
or can get the source code.  And you must show them these terms so they
```

```
know their rights.

  Developers that use the GNU GPL protect your rights with two steps:
(1) assert copyright on the software, and (2) offer you this License
giving you legal permission to copy, distribute and/or modify it.

  For the developers' and authors' protection, the GPL clearly explains
that there is no warranty for this free software.  For both users' and
authors' sake, the GPL requires that modified versions be marked as
changed, so that their problems will not be attributed erroneously to
authors of previous versions.

  Some devices are designed to deny users access to install or run
modified versions of the software inside them, although the manufacturer
can do so.  This is fundamentally incompatible with the aim of
protecting users' freedom to change the software.  The systematic
pattern of such abuse occurs in the area of products for individuals to
use, which is precisely where it is most unacceptable.  Therefore, we
have designed this version of the GPL to prohibit the practice for those
products.  If such problems arise substantially in other domains, we
stand ready to extend this provision to those domains in future versions
of the GPL, as needed to protect the freedom of users.

  Finally, every program is threatened constantly by software patents.
States should not allow patents to restrict development and use of
software on general-purpose computers, but in those that do, we wish to
avoid the special danger that patents applied to a free program could
make it effectively proprietary.  To prevent this, the GPL assures that
patents cannot be used to render the program non-free.

  The precise terms and conditions for copying, distribution and
modification follow.

                       TERMS AND CONDITIONS

  0. Definitions.

  "This License" refers to version 3 of the GNU General Public License.

```

```
  "Copyright" also means copyright-like laws that apply to other kinds of
works, such as semiconductor masks.

  "The Program" refers to any copyrightable work licensed under this
License.  Each licensee is addressed as "you".  "Licensees" and
"recipients" may be individuals or organizations.

  To "modify" a work means to copy from or adapt all or part of the work
in a fashion requiring copyright permission, other than the making of an
exact copy.  The resulting work is called a "modified version" of the
earlier work or a work "based on" the earlier work.

  A "covered work" means either the unmodified Program or a work based
on the Program.

  To "propagate" a work means to do anything with it that, without
permission, would make you directly or secondarily liable for
infringement under applicable copyright law, except executing it on a
computer or modifying a private copy.  Propagation includes copying,
distribution (with or without modification), making available to the
public, and in some countries other activities as well.

  To "convey" a work means any kind of propagation that enables other
parties to make or receive copies.  Mere interaction with a user through
a computer network, with no transfer of a copy, is not conveying.

  An interactive user interface displays "Appropriate Legal Notices"
to the extent that it includes a convenient and prominently visible
feature that (1) displays an appropriate copyright notice, and (2)
tells the user that there is no warranty for the work (except to the
extent that warranties are provided), that licensees may convey the
work under this License, and how to view a copy of this License.  If
the interface presents a list of user commands or options, such as a
menu, a prominent item in the list meets this criterion.

  1. Source Code.

  The "source code" for a work means the preferred form of the work
for making modifications to it.  "Object code" means any non-source
```

```
116 form of a work.
117
118   A "Standard Interface" means an interface that either is an official
119 standard defined by a recognized standards body, or, in the case of
120 interfaces specified for a particular programming language, one that
121 is widely used among developers working in that language.
122
123   The "System Libraries" of an executable work include anything, other
124 than the work as a whole, that (a) is included in the normal form of
125 packaging a Major Component, but which is not part of that Major
126 Component, and (b) serves only to enable use of the work with that
127 Major Component, or to implement a Standard Interface for which an
128 implementation is available to the public in source code form.  A
129 "Major Component", in this context, means a major essential component
130 (kernel, window system, and so on) of the specific operating system
131 (if any) on which the executable work runs, or a compiler used to
132 produce the work, or an object code interpreter used to run it.
133
134   The "Corresponding Source" for a work in object code form means all
135 the source code needed to generate, install, and (for an executable
136 work) run the object code and to modify the work, including scripts to
137 control those activities.  However, it does not include the work's
138 System Libraries, or general-purpose tools or generally available free
139 programs which are used unmodified in performing those activities but
140 which are not part of the work.  For example, Corresponding Source
141 includes interface definition files associated with source files for
142 the work, and the source code for shared libraries and dynamically
143 linked subprograms that the work is specifically designed to require,
144 such as by intimate data communication or control flow between those
145 subprograms and other parts of the work.
146
147   The Corresponding Source need not include anything that users
148 can regenerate automatically from other parts of the Corresponding
149 Source.
150
151   The Corresponding Source for a work in source code form is that
152 same work.
153
154   2. Basic Permissions.
```

```
155
156   All rights granted under this License are granted for the term of
157 copyright on the Program, and are irrevocable provided the stated
158 conditions are met.  This License explicitly affirms your unlimited
159 permission to run the unmodified Program.  The output from running a
160 covered work is covered by this License only if the output, given its
161 content, constitutes a covered work.  This License acknowledges your
162 rights of fair use or other equivalent, as provided by copyright law.
163
164   You may make, run and propagate covered works that you do not
165 convey, without conditions so long as your license otherwise remains
166 in force.  You may convey covered works to others for the sole purpose
167 of having them make modifications exclusively for you, or provide you
168 with facilities for running those works, provided that you comply with
169 the terms of this License in conveying all material for which you do
170 not control copyright.  Those thus making or running the covered works
171 for you must do so exclusively on your behalf, under your direction
172 and control, on terms that prohibit them from making any copies of
173 your copyrighted material outside their relationship with you.
174
175   Conveying under any other circumstances is permitted solely under
176 the conditions stated below.  Sublicensing is not allowed; section 10
177 makes it unnecessary.
178
179   3. Protecting Users' Legal Rights From Anti-Circumvention Law.
180
181   No covered work shall be deemed part of an effective technological
182 measure under any applicable law fulfilling obligations under article
183 11 of the WIPO copyright treaty adopted on 20 December 1996, or
184 similar laws prohibiting or restricting circumvention of such
185 measures.
186
187   When you convey a covered work, you waive any legal power to forbid
188 circumvention of technological measures to the extent such circumvention
189 is effected by exercising rights under this License with respect to
190 the covered work, and you disclaim any intention to limit operation or
191 modification of the work as a means of enforcing, against the work's
192 users, your or third parties' legal rights to forbid circumvention of
193 technological measures.
```

```
194
195   4. Conveying Verbatim Copies.
196
197   You may convey verbatim copies of the Program's source code as you
198 receive it, in any medium, provided that you conspicuously and
199 appropriately publish on each copy an appropriate copyright notice;
200 keep intact all notices stating that this License and any
201 non-permissive terms added in accord with section 7 apply to the code;
202 keep intact all notices of the absence of any warranty; and give all
203 recipients a copy of this License along with the Program.
204
205   You may charge any price or no price for each copy that you convey,
206 and you may offer support or warranty protection for a fee.
207
208   5. Conveying Modified Source Versions.
209
210   You may convey a work based on the Program, or the modifications to
211 produce it from the Program, in the form of source code under the
212 terms of section 4, provided that you also meet all of these conditions:
213
214     a) The work must carry prominent notices stating that you modified
215     it, and giving a relevant date.
216
217     b) The work must carry prominent notices stating that it is
218     released under this License and any conditions added under section
219     7.  This requirement modifies the requirement in section 4 to
220     "keep intact all notices".
221
222     c) You must license the entire work, as a whole, under this
223     License to anyone who comes into possession of a copy.  This
224     License will therefore apply, along with any applicable section 7
225     additional terms, to the whole of the work, and all its parts,
226     regardless of how they are packaged.  This License gives no
227     permission to license the work in any other way, but it does not
228     invalidate such permission if you have separately received it.
229
230     d) If the work has interactive user interfaces, each must display
231     Appropriate Legal Notices; however, if the Program has interactive
232     interfaces that do not display Appropriate Legal Notices, your
```

```
    work need not make them do so.

  A compilation of a covered work with other separate and independent
works, which are not by their nature extensions of the covered work,
and which are not combined with it such as to form a larger program,
in or on a volume of a storage or distribution medium, is called an
"aggregate" if the compilation and its resulting copyright are not
used to limit the access or legal rights of the compilation's users
beyond what the individual works permit.  Inclusion of a covered work
in an aggregate does not cause this License to apply to the other
parts of the aggregate.

  6. Conveying Non-Source Forms.

  You may convey a covered work in object code form under the terms
of sections 4 and 5, provided that you also convey the
machine-readable Corresponding Source under the terms of this License,
in one of these ways:

    a) Convey the object code in, or embodied in, a physical product
    (including a physical distribution medium), accompanied by the
    Corresponding Source fixed on a durable physical medium
    customarily used for software interchange.

    b) Convey the object code in, or embodied in, a physical product
    (including a physical distribution medium), accompanied by a
    written offer, valid for at least three years and valid for as
    long as you offer spare parts or customer support for that product
    model, to give anyone who possesses the object code either (1) a
    copy of the Corresponding Source for all the software in the
    product that is covered by this License, on a durable physical
    medium customarily used for software interchange, for a price no
    more than your reasonable cost of physically performing this
    conveying of source, or (2) access to copy the
    Corresponding Source from a network server at no charge.

    c) Convey individual copies of the object code with a copy of the
    written offer to provide the Corresponding Source.  This
    alternative is allowed only occasionally and noncommercially, and
```

```
    only if you received the object code with such an offer, in accord
    with subsection 6b.

    d) Convey the object code by offering access from a designated
    place (gratis or for a charge), and offer equivalent access to the
    Corresponding Source in the same way through the same place at no
    further charge.  You need not require recipients to copy the
    Corresponding Source along with the object code.  If the place to
    copy the object code is a network server, the Corresponding Source
    may be on a different server (operated by you or a third party)
    that supports equivalent copying facilities, provided you maintain
    clear directions next to the object code saying where to find the
    Corresponding Source.  Regardless of what server hosts the
    Corresponding Source, you remain obligated to ensure that it is
    available for as long as needed to satisfy these requirements.

    e) Convey the object code using peer-to-peer transmission, provided
    you inform other peers where the object code and Corresponding
    Source of the work are being offered to the general public at no
    charge under subsection 6d.

  A separable portion of the object code, whose source code is excluded
from the Corresponding Source as a System Library, need not be
included in conveying the object code work.

  A "User Product" is either (1) a "consumer product", which means any
tangible personal property which is normally used for personal, family,
or household purposes, or (2) anything designed or sold for incorporation
into a dwelling.  In determining whether a product is a consumer product,
doubtful cases shall be resolved in favor of coverage.  For a particular
product received by a particular user, "normally used" refers to a
typical or common use of that class of product, regardless of the status
of the particular user or of the way in which the particular user
actually uses, or expects or is expected to use, the product.  A product
is a consumer product regardless of whether the product has substantial
commercial, industrial or non-consumer uses, unless such uses represent
the only significant mode of use of the product.

  "Installation Information" for a User Product means any methods,
```

```
procedures, authorization keys, or other information required to install
and execute modified versions of a covered work in that User Product from
a modified version of its Corresponding Source.  The information must
suffice to ensure that the continued functioning of the modified object
code is in no case prevented or interfered with solely because
modification has been made.

  If you convey an object code work under this section in, or with, or
specifically for use in, a User Product, and the conveying occurs as
part of a transaction in which the right of possession and use of the
User Product is transferred to the recipient in perpetuity or for a
fixed term (regardless of how the transaction is characterized), the
Corresponding Source conveyed under this section must be accompanied
by the Installation Information.  But this requirement does not apply
if neither you nor any third party retains the ability to install
modified object code on the User Product (for example, the work has
been installed in ROM).

  The requirement to provide Installation Information does not include a
requirement to continue to provide support service, warranty, or updates
for a work that has been modified or installed by the recipient, or for
the User Product in which it has been modified or installed.  Access to a
network may be denied when the modification itself materially and
adversely affects the operation of the network or violates the rules and
protocols for communication across the network.

  Corresponding Source conveyed, and Installation Information provided,
in accord with this section must be in a format that is publicly
documented (and with an implementation available to the public in
source code form), and must require no special password or key for
unpacking, reading or copying.

  7. Additional Terms.

  "Additional permissions" are terms that supplement the terms of this
License by making exceptions from one or more of its conditions.
Additional permissions that are applicable to the entire Program shall
be treated as though they were included in this License, to the extent
that they are valid under applicable law.  If additional permissions
```

```
apply only to part of the Program, that part may be used separately
under those permissions, but the entire Program remains governed by
this License without regard to the additional permissions.

  When you convey a copy of a covered work, you may at your option
remove any additional permissions from that copy, or from any part of
it.  (Additional permissions may be written to require their own
removal in certain cases when you modify the work.)  You may place
additional permissions on material, added by you to a covered work,
for which you have or can give appropriate copyright permission.

  Notwithstanding any other provision of this License, for material you
add to a covered work, you may (if authorized by the copyright holders of
that material) supplement the terms of this License with terms:

    a) Disclaiming warranty or limiting liability differently from the
    terms of sections 15 and 16 of this License; or

    b) Requiring preservation of specified reasonable legal notices or
    author attributions in that material or in the Appropriate Legal
    Notices displayed by works containing it; or

    c) Prohibiting misrepresentation of the origin of that material, or
    requiring that modified versions of such material be marked in
    reasonable ways as different from the original version; or

    d) Limiting the use for publicity purposes of names of licensors or
    authors of the material; or

    e) Declining to grant rights under trademark law for use of some
    trade names, trademarks, or service marks; or

    f) Requiring indemnification of licensors and authors of that
    material by anyone who conveys the material (or modified versions of
    it) with contractual assumptions of liability to the recipient, for
    any liability that these contractual assumptions directly impose on
    those licensors and authors.

  All other non-permissive additional terms are considered "further
```

```
restrictions" within the meaning of section 10.  If the Program as you
received it, or any part of it, contains a notice stating that it is
governed by this License along with a term that is a further
restriction, you may remove that term.  If a license document contains
a further restriction but permits relicensing or conveying under this
License, you may add to a covered work material governed by the terms
of that license document, provided that the further restriction does
not survive such relicensing or conveying.

  If you add terms to a covered work in accord with this section, you
must place, in the relevant source files, a statement of the
additional terms that apply to those files, or a notice indicating
where to find the applicable terms.

  Additional terms, permissive or non-permissive, may be stated in the
form of a separately written license, or stated as exceptions;
the above requirements apply either way.

  8. Termination.

  You may not propagate or modify a covered work except as expressly
provided under this License.  Any attempt otherwise to propagate or
modify it is void, and will automatically terminate your rights under
this License (including any patent licenses granted under the third
paragraph of section 11).

  However, if you cease all violation of this License, then your
license from a particular copyright holder is reinstated (a)
provisionally, unless and until the copyright holder explicitly and
finally terminates your license, and (b) permanently, if the copyright
holder fails to notify you of the violation by some reasonable means
prior to 60 days after the cessation.

  Moreover, your license from a particular copyright holder is
reinstated permanently if the copyright holder notifies you of the
violation by some reasonable means, this is the first time you have
received notice of violation of this License (for any work) from that
copyright holder, and you cure the violation prior to 30 days after
your receipt of the notice.
```

```
428
429   Termination of your rights under this section does not terminate the
430 licenses of parties who have received copies or rights from you under
431 this License.  If your rights have been terminated and not permanently
432 reinstated, you do not qualify to receive new licenses for the same
433 material under section 10.
434
435   9. Acceptance Not Required for Having Copies.
436
437   You are not required to accept this License in order to receive or
438 run a copy of the Program.  Ancillary propagation of a covered work
439 occurring solely as a consequence of using peer-to-peer transmission
440 to receive a copy likewise does not require acceptance.  However,
441 nothing other than this License grants you permission to propagate or
442 modify any covered work.  These actions infringe copyright if you do
443 not accept this License.  Therefore, by modifying or propagating a
444 covered work, you indicate your acceptance of this License to do so.
445
446   10. Automatic Licensing of Downstream Recipients.
447
448   Each time you convey a covered work, the recipient automatically
449 receives a license from the original licensors, to run, modify and
450 propagate that work, subject to this License.  You are not responsible
451 for enforcing compliance by third parties with this License.
452
453   An "entity transaction" is a transaction transferring control of an
454 organization, or substantially all assets of one, or subdividing an
455 organization, or merging organizations.  If propagation of a covered
456 work results from an entity transaction, each party to that
457 transaction who receives a copy of the work also receives whatever
458 licenses to the work the party's predecessor in interest had or could
459 give under the previous paragraph, plus a right to possession of the
460 Corresponding Source of the work from the predecessor in interest, if
461 the predecessor has it or can get it with reasonable efforts.
462
463   You may not impose any further restrictions on the exercise of the
464 rights granted or affirmed under this License.  For example, you may
465 not impose a license fee, royalty, or other charge for exercise of
466 rights granted under this License, and you may not initiate litigation
```

```
(including a cross-claim or counterclaim in a lawsuit) alleging that
any patent claim is infringed by making, using, selling, offering for
sale, or importing the Program or any portion of it.

  11. Patents.

  A "contributor" is a copyright holder who authorizes use under this
License of the Program or a work on which the Program is based.  The
work thus licensed is called the contributor's "contributor version".

  A contributor's "essential patent claims" are all patent claims
owned or controlled by the contributor, whether already acquired or
hereafter acquired, that would be infringed by some manner, permitted
by this License, of making, using, or selling its contributor version,
but do not include claims that would be infringed only as a
consequence of further modification of the contributor version.  For
purposes of this definition, "control" includes the right to grant
patent sublicenses in a manner consistent with the requirements of
this License.

  Each contributor grants you a non-exclusive, worldwide, royalty-free
patent license under the contributor's essential patent claims, to
make, use, sell, offer for sale, import and otherwise run, modify and
propagate the contents of its contributor version.

  In the following three paragraphs, a "patent license" is any express
agreement or commitment, however denominated, not to enforce a patent
(such as an express permission to practice a patent or covenant not to
sue for patent infringement).  To "grant" such a patent license to a
party means to make such an agreement or commitment not to enforce a
patent against the party.

  If you convey a covered work, knowingly relying on a patent license,
and the Corresponding Source of the work is not available for anyone
to copy, free of charge and under the terms of this License, through a
publicly available network server or other readily accessible means,
then you must either (1) cause the Corresponding Source to be so
available, or (2) arrange to deprive yourself of the benefit of the
patent license for this particular work, or (3) arrange, in a manner
```

```
consistent with the requirements of this License, to extend the patent
license to downstream recipients.  "Knowingly relying" means you have
actual knowledge that, but for the patent license, your conveying the
covered work in a country, or your recipient's use of the covered work
in a country, would infringe one or more identifiable patents in that
country that you have reason to believe are valid.

  If, pursuant to or in connection with a single transaction or
arrangement, you convey, or propagate by procuring conveyance of, a
covered work, and grant a patent license to some of the parties
receiving the covered work authorizing them to use, propagate, modify
or convey a specific copy of the covered work, then the patent license
you grant is automatically extended to all recipients of the covered
work and works based on it.

  A patent license is "discriminatory" if it does not include within
the scope of its coverage, prohibits the exercise of, or is
conditioned on the non-exercise of one or more of the rights that are
specifically granted under this License.  You may not convey a covered
work if you are a party to an arrangement with a third party that is
in the business of distributing software, under which you make payment
to the third party based on the extent of your activity of conveying
the work, and under which the third party grants, to any of the
parties who would receive the covered work from you, a discriminatory
patent license (a) in connection with copies of the covered work
conveyed by you (or copies made from those copies), or (b) primarily
for and in connection with specific products or compilations that
contain the covered work, unless you entered into that arrangement,
or that patent license was granted, prior to 28 March 2007.

  Nothing in this License shall be construed as excluding or limiting
any implied license or other defenses to infringement that may
otherwise be available to you under applicable patent law.

  12. No Surrender of Others' Freedom.

  If conditions are imposed on you (whether by court order, agreement or
otherwise) that contradict the conditions of this License, they do not
excuse you from the conditions of this License.  If you cannot convey a
```

```
covered work so as to satisfy simultaneously your obligations under this
License and any other pertinent obligations, then as a consequence you may
not convey it at all.  For example, if you agree to terms that obligate you
to collect a royalty for further conveying from those to whom you convey
the Program, the only way you could satisfy both those terms and this
License would be to refrain entirely from conveying the Program.

  13. Use with the GNU Affero General Public License.

  Notwithstanding any other provision of this License, you have
permission to link or combine any covered work with a work licensed
under version 3 of the GNU Affero General Public License into a single
combined work, and to convey the resulting work.  The terms of this
License will continue to apply to the part which is the covered work,
but the special requirements of the GNU Affero General Public License,
section 13, concerning interaction through a network will apply to the
combination as such.

  14. Revised Versions of this License.

  The Free Software Foundation may publish revised and/or new versions of
the GNU General Public License from time to time.  Such new versions will
be similar in spirit to the present version, but may differ in detail to
address new problems or concerns.

  Each version is given a distinguishing version number.  If the
Program specifies that a certain numbered version of the GNU General
Public License "or any later version" applies to it, you have the
option of following the terms and conditions either of that numbered
version or of any later version published by the Free Software
Foundation.  If the Program does not specify a version number of the
GNU General Public License, you may choose any version ever published
by the Free Software Foundation.

  If the Program specifies that a proxy can decide which future
versions of the GNU General Public License can be used, that proxy's
public statement of acceptance of a version permanently authorizes you
to choose that version for the Program.

```

```
  Later license versions may give you additional or different
permissions.  However, no additional obligations are imposed on any
author or copyright holder as a result of your choosing to follow a
later version.

  15. Disclaimer of Warranty.

  THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY
APPLICABLE LAW.  EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT
HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY
OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO,
THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
PURPOSE.  THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM
IS WITH YOU.  SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF
ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

  16. Limitation of Liability.

  IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING
WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS
THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY
GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE
USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF
DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD
PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS),
EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF
SUCH DAMAGES.

  17. Interpretation of Sections 15 and 16.

  If the disclaimer of warranty and limitation of liability provided
above cannot be given local legal effect according to their terms,
reviewing courts shall apply local law that most closely approximates
an absolute waiver of all civil liability in connection with the
Program, unless a warranty or assumption of liability accompanies a
copy of the Program in return for a fee.

                     END OF TERMS AND CONDITIONS

```

```
623             How to Apply These Terms to Your New Programs
624
625   If you develop a new program, and you want it to be of the greatest
626 possible use to the public, the best way to achieve this is to make it
627 free software which everyone can redistribute and change under these terms.
628
629   To do so, attach the following notices to the program.  It is safest
630 to attach them to the start of each source file to most effectively
631 state the exclusion of warranty; and each file should have at least
632 the "copyright" line and a pointer to where the full notice is found.
633
634     <one line to give the program's name and a brief idea of what it does.>
635     Copyright (C) <year>  <name of author>
636
637     This program is free software: you can redistribute it and/or modify
638     it under the terms of the GNU General Public License as published by
639     the Free Software Foundation, either version 3 of the License, or
640     (at your option) any later version.
641
642     This program is distributed in the hope that it will be useful,
643     but WITHOUT ANY WARRANTY; without even the implied warranty of
644     MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
645     GNU General Public License for more details.
646
647     You should have received a copy of the GNU General Public License
648     along with this program.  If not, see <https://www.gnu.org/licenses/>.
649
650 Also add information on how to contact you by electronic and paper mail.
651
652   If the program does terminal interaction, make it output a short
653 notice like this when it starts in an interactive mode:
654
655     <program>  Copyright (C) <year>  <name of author>
656     This program comes with ABSOLUTELY NO WARRANTY; for details type `show w'.
657     This is free software, and you are welcome to redistribute it
658     under certain conditions; type `show c' for details.
659
660 The hypothetical commands `show w' and `show c' should show the appropriate
661 parts of the General Public License.  Of course, your program's commands
```

```
might be different; for a GUI interface, you would use an "about box".

  You should also get your employer (if you work as a programmer) or school,
if any, to sign a "copyright disclaimer" for the program, if necessary.
For more information on this, and how to apply and follow the GNU GPL, see
<https://www.gnu.org/licenses/>.

  The GNU General Public License does not permit incorporating your program
into proprietary programs.  If your program is a subroutine library, you
may consider it more useful to permit linking proprietary applications with
the library.  If this is what you want to do, use the GNU Lesser General
Public License instead of this License.  But first, please read
<https://www.gnu.org/licenses/why-not-lgpl.html>.
```

C.2 models

```
FallArmyworm
└── models
    ├── development.py
    ├── dominance.py
    ├── forage.py
    ├── graph.py
    ├── graph_data_generator.py
    ├── growth.py
    ├── init_biomass.py
    ├── migration.py
    ├── movement.py
    ├── reproduction.py
    ├── simulator.py
    └── survival.py
```

C.2.1 development.py

```
import source.development.models as models

import models.dominance as dom


def egg_dev(mu_egg_dev:  float,
            sig_egg_dev: float):
    """
    Create an egg development model

    Args:
        mu_egg_dev:  mean development time for eggs
        sig_egg_dev: std in development time for eggs

    Returns:
        a functional mathematical model for the simulation
    """

    return models.Egg(mu_egg_dev,
                      sig_egg_dev)


def pupa_dev(mu_pupa_dev:  float,
             sig_pupa_dev: float):
    """
    Create a pupa development model

    Args:
        mu_pupa_dev:  mean development time for pupae
        sig_pupa_dev: std in development time for pupae

    Returns:
    """

    return models.Pupa(mu_pupa_dev,
                       sig_pupa_dev)

```

```
38
39 def larva_dev(mu_larva_dev_ss:  float,
40              mu_larva_dev_rr:  float,
41              sig_larva_dev_ss: float,
42              sig_larva_dev_rr: float,
43              dominance:        float):
44     """
45     Create a larva development model
46
47     Args:
48         mu_larva_dev_ss:  mean development mass for larvae of genotype ss
49         mu_larva_dev_rr:  mean development mass for larvae of genotype rr
50         sig_larva_dev_ss: std in development mass for larvae of genotype ss
51         sig_larva_dev_rr: std in development mass for larvae of genotype rr
52         dominance:        degree of dominance
53
54     Returns:
55         a functional mathematical model for the simulation
56     """
57
58     mu  = dom.dom(mu_larva_dev_ss,
59                   mu_larva_dev_rr,
60                   dominance)
61
62     print('SS Mu: {}'.format(mu['susceptible']))
63     print('RR Mu: {}'.format(mu['resistant']))
64
65     sig = dom.dom(sig_larva_dev_ss,
66                   sig_larva_dev_rr,
67                   dominance)
68
69     return models.Larva(mu, sig)
```

C.2.2 dominance.py

```
import source.hint    as hint
import source.keyword as keyword


def dom(homo_s:    float,
        homo_r:    float,
        dominance: float) -> hint.variable:
    """
    Calculate the heterozyous parameter using degree of dominance
        SR = SS + D*(RR - SS)

        SR = heterozyous
        SS = susceptible
        RR = resistant
        D  = degree of dominance

    Args:
        homo_s:    susceptible parameter
        homo_r:    resistant   parameter
        dominance: the dominance

    Returns:
        heterozygous parameter
    """

    hetero = homo_s + dominance*(homo_r - homo_s)

    return {
        keyword.homo_s: homo_s,
        keyword.hetero: hetero,
        keyword.homo_r: homo_r
    }
```

C.2.3 forage.py

```
import source.hint as hint

import source.biomass.models as bio
import source.forage.models  as models


def adlibitum(forage_steps: int):
    """
    Create a true adlibitum plant consume model

    Args:
        forage_steps: the number of steps that forage takes place over

    Returns:
        a functional mathematical model for the simulation
    """

    max_gut = bio.MaxGut()

    return models.PlantAdLibitum(forage_steps,
                                 max_gut)


def starvation(forage_steps: int,
               theta:        float,
               sig_starve:   float):
    """
    Create a starvation plat consume model

    Args:
        forage_steps: the number of steps that forage takes place over
        theta:        starvation factor
        sig_starve:   std in food amounts

    Returns:
        a functional mathematical model for the simulation
    """
```

```
38
39      max_gut = bio.MaxGut()
40
41      return models.PlantStarve(forage_steps,
42                                theta,
43                                sig_starve,
44                                max_gut)
45
46
47  def egg(egg_factor: float):
48      """
49      Create an egg consume model
50
51      Args:
52          egg_factor: the factor for how much of egg can be eaten
53
54      Returns:
55          a functional mathematical model for the simulation
56      """
57
58      return models.Egg(egg_factor)
59
60
61  def larva(larva_factor: float):
62      """
63      Create an larva consume model
64
65      Args:
66          larva_factor: the factor for how much of larva can be eaten
67
68      Returns:
69          a functional mathematical model for the simulation
70      """
71
72      return models.Larva(larva_factor)
73
74
75  def fight(fight_slope: float):
76      """
```

```
77      Create a larva fight model
78
79      Args:
80          fight_slope: slope of transition
81
82      Returns:
83          a functional mathematical model for the simulation
84      """
85
86      return models.Fight(fight_slope)
87
88
89  def radius(cannibalism_radius: float):
90      """
91      Create a mathematical model for radius
92
93      Args:
94          cannibalism_radius: the radius for cannibalism
95
96      Returns:
97          a functional mathematical model for the simulation
98      """
99
100     return models.Radius(cannibalism_radius)
101
102
103 def encounter(cannibalism_encounter: float):
104     """
105     Create a mathematical model for encounters
106
107     Args:
108         cannibalism_encounter: the encounter rate constant
109
110     Returns:
111         a functional mathematical model for the simulation
112     """
113
114     return models.Encounter(cannibalism_encounter)
115
```

```
116
117 def loss(loss_slope:   float,
118          mid:          hint.variable,
119          egg_factor:   float,
120          larva_factor: float):
121     """
122     Create a mathematical model for loss
123
124     Args:
125         loss_slope:   slope of transition
126         mid:          equality probability factors
127         egg_factor:   the factor for how much of egg can be eaten
128         larva_factor: the factor for how much of larva can be eaten
129
130     Returns:
131         a functional mathematical model for the simulation
132     """
133
134     max_gut      = bio.MaxGut()
135     forage_egg   = egg(  egg_factor)
136     forage_larva = larva(larva_factor)
137
138     return models.Loss(loss_slope,
139                        mid,
140                        max_gut,
141                        forage_egg,
142                        forage_larva)
```

C.2.4 graph.py

```
import os

import pickle   as pickle

import source.hint as hint

graph_path = os.path.dirname(os.path.abspath(__file__))
print('Graph path is: {}'.format(graph_path))


# MUST be Adjusted for corrected paths
hex_1x1   = '{}/hex_1x1.graph'.format(graph_path)
hex_2x2   = '{}/hex_2x2.graph'.format(graph_path)
hex_4x4   = '{}/hex_4x4.graph'.format(graph_path)
hex_8x8   = '{}/hex_8x8.graph'.format(graph_path)
hex_10x10 = '{}/hex_10x10.graph'.format(graph_path)
hex_25x25 = '{}/hex_25x25.graph'.format(graph_path)
hex_50x50 = '{}/hex_50x50.graph'.format(graph_path)


def graph(side: int) -> hint.graph:
    """
    Read a graph from data

    Args:
        side: the side count

    Returns:
        a prebuilt graph read from a file
    """

    if side == 1:
        file_name = hex_1x1
    elif side == 2:
        file_name = hex_2x2
    elif side == 4:
        file_name = hex_4x4
```

```
38      elif side == 8:
39          file_name = hex_8x8
40      elif side == 10:
41          file_name = hex_10x10
42      elif side == 25:
43          file_name = hex_25x25
44      elif side == 50:
45          file_name = hex_50x50
46      else:
47          raise FileExistsError('No file for graph of that side')
48
49      with open(file_name, 'rb') as graph_file:
50          return pickle.load(graph_file)
```

C.2.5 graph_data_generator.py

```
import sys
import datetime

import source.space.grid as grid


# print('Creating 1x1 at: {}'.format(datetime.datetime.now()))
# graph = grid.Hexagon.grid(1, 1, True, True)
# print('Size of graph: {}'.format(sys.getsizeof(graph)))
# print('Saving 1x1 at: {}'.format(datetime.datetime.now()))
# graph.save('hex_1x1.graph')

# print('Creating 2x2 at: {}'.format(datetime.datetime.now()))
# graph = grid.Hexagon.grid(2, 2, True, True)
# print('Size of graph: {}'.format(sys.getsizeof(graph)))
# print('Saving 2x2 at: {}'.format(datetime.datetime.now()))
# graph.save('hex_2x2.graph')
#
# print('Creating 4x4 at: {}'.format(datetime.datetime.now()))
# graph = grid.Hexagon.grid(4, 4, True, True)
# print('Size of graph: {}'.format(sys.getsizeof(graph)))
# print('Saving 4x4 at: {}'.format(datetime.datetime.now()))
# graph.save('hex_4x4.graph')

print('Creating 8x8 at: {}'.format(datetime.datetime.now()))
graph = grid.Hexagon.grid(8, 8, True, True)
print('Size of graph: {}'.format(sys.getsizeof(graph)))
print('Saving 8x8 at: {}'.format(datetime.datetime.now()))
graph.save('hex_8x8.graph')

# print('Creating 10x10 at: {}'.format(datetime.datetime.now()))
# graph = grid.Hexagon.grid(10, 10, True, True)
# print('Size of graph: {}'.format(sys.getsizeof(graph)))
# print('Saving 10x10 at: {}'.format(datetime.datetime.now()))
# graph.save('hex_10x10.graph')
#
# print('Creating 25x25 at: {}'.format(datetime.datetime.now()))
```

```
38 # graph = grid.Hexagon.grid(25, 25, True, True)
39 # print('Size of graph: {}'.format(sys.getsizeof(graph)))
40 # print('Saving 25x25 at: {}'.format(datetime.datetime.now()))
41 # graph.save('hex_25x25.graph')
42
43 # print('Creating 50x50 at: {}'.format(datetime.datetime.now()))
44 # graph = grid.Hexagon.grid(50, 50, True, True)
45 # print('Size of graph: {}'.format(sys.getsizeof(graph)))
46 # print('Saving 50x50 at: {}'.format(datetime.datetime.now()))
47 # graph.save('hex_50x50.graph')
48
49 # print('Creating 100x100 at: {}'.format(datetime.datetime.now()))
50 # graph = space_grid.Hexagon.grid(100, 100, True, True)
51 # print('Size of graph: {}'.format(sys.getsizeof(graph)))
52 # print('Saving 100x100 at: {}'.format(datetime.datetime.now()))
53 # graph.save('hex_100x100.graph')
54 # print('End of graph save: {}'.format(datetime.datetime.now()))
```

C.2.6 growth.py

```
import source.biomass.models as models

import models.dominance as dom


def max_gut():
    """
    Create the max_gut model

    Returns:
        a functional mathematical model for the simulation
    """

    return models.MaxGut()


def growth(alpha_ss:  float,
           alpha_rr:  float,
           beta_ss:   float,
           beta_rr:   float,
           dominance: float):
    """
    Create a growth model for larvae

    Args:
        alpha_ss:  growth rate for ss genotype
        alpha_rr:  growth rate for rr genotype
        beta_ss:   maintenance cost for ss genotype
        beta_rr:   maintenance cost for rr genotype
        dominance: degree of dominance

    Returns:
        a functional mathematical model for the simulation
    """

    alpha = dom.dom(alpha_ss,
                    alpha_rr,
```

```
38                      dominance)
39      beta  = dom.dom(beta_ss,
40                      beta_rr,
41                      dominance)
42
43      return models.Growth(alpha, beta)
```

C.2.7 init_biomass.py

```
import source.biomass.models as models

import models.dominance as dom


def init_num(lam_0_egg: float):
    """
    Create an init_num model for egg_mass

    Args:
        lam_0_egg: the mean number of eggs in egg_mass

    Returns:
        a functional mathematical model for the simulation
    """

    return models.InitNum(lam_0_egg)


def init_mass(mu_0_egg_ss:  float,
              mu_0_egg_rr:  float,
              sig_0_egg_ss: float,
              sig_0_egg_rr: float,
              dominance:    float):
    """
    Create an init_mass model for egg_mass

    Args:
        mu_0_egg_ss:  the mean mass of egg_mass for ss genotype
        mu_0_egg_rr:  the mean mass of egg_mass for rr genotype
        sig_0_egg_ss: the std in mass of egg_mass for ss genotype
        sig_0_egg_rr: the std in mass of egg_mass for rr genotype
        dominance:    the degree of dominance

    Returns:
        a functional mathematical model for the simulation
    """
```

```
38
39      mu_0_egg  = dom.dom(mu_0_egg_ss,
40                          mu_0_egg_rr,
41                          dominance)
42      sig_0_egg = dom.dom(sig_0_egg_ss,
43                          sig_0_egg_rr,
44                          dominance)
45
46      return models.InitMass(mu_0_egg,
47                             sig_0_egg)
48
49
50  def init_juvenile(mu_0_larva_ss:  float,
51                    mu_0_larva_rr:  float,
52                    sig_0_larva_ss: float,
53                    sig_0_larva_rr: float,
54                    dominance:      float):
55      """
56      Create an init_mass model for larva
57
58      Args:
59          mu_0_larva_ss:  the mean mass of larva for ss genotype
60          mu_0_larva_rr:  the mean mass of larva for rr genotype
61          sig_0_larva_ss: the std in mass of larva for ss genotype
62          sig_0_larva_rr: the std in mass of larva for rr genotype
63          dominance:      the degree of dominance
64
65      Returns:
66          a functional mathematical model for the simulation
67      """
68
69      mu_0_larva  = dom.dom(mu_0_larva_ss,
70                            mu_0_larva_rr,
71                            dominance)
72      sig_0_larva = dom.dom(sig_0_larva_ss,
73                            sig_0_larva_rr,
74                            dominance)
75
76      return models.InitJuvenile(mu_0_larva,
```

```
77                                sig_0_larva)
78
79
80  def init_mature(mu_0_mature_ss:  float,
81                  mu_0_mature_rr:  float,
82                  sig_0_mature_ss: float,
83                  sig_0_mature_rr: float,
84                  dominance:       float):
85      """
86      Create an init_mass model for mature agents
87
88      Args:
89          mu_0_mature_ss:  the mean mass of mature for ss genotype
90          mu_0_mature_rr:  the mean mass of mature for rr genotype
91          sig_0_mature_ss: the std in mass of mature for ss genotype
92          sig_0_mature_rr: the std in mass of mature for rr genotype
93          dominance:       the degree of dominance
94
95      Returns:
96          a functional mathematical model for the simulation
97      """
98
99      mu_0_mature  = dom.dom(mu_0_mature_ss,
100                             mu_0_mature_rr,
101                             dominance)
102     sig_0_mature = dom.dom(sig_0_mature_ss,
103                             sig_0_mature_rr,
104                             dominance)
105
106     return models.InitMature(mu_0_mature,
107                              sig_0_mature)
108
109
110 def init_plant(mu_leaf:  float,
111                sig_leaf: float):
112     """
113     Create an init_plant model for plant biomass
114     Args:
115         mu_leaf:  the mean mass of a leaf
```

```
116         sig_leaf: the std in mass of a leaf
117
118     Returns:
119         a functional mathematical model for the simulation
120     """
121
122     return models.InitPlant(mu_leaf,
123                             sig_leaf)
```

C.2.8 migration.py

```
import source.keyword as keyword


def emigration_adult(mean, sigma):
    """
    Create an emigration for adults
    Args:
        mean:  mean number of adults
        sigma: standard deviation

    Returns:
        adult immigration
    """

    emigration = (mean, sigma, [keyword.female, keyword.male, keyword.mated])

    return emigration
```

C.2.9 movement.py

```
import source.movement.models as models


def larva(scale: float,
          shape: float):
    """
    Create a larva movement model

    Args:
        scale: scale parameter
        shape: shape parameter

    Returns:
        a functional mathematical model for the simulation
    """

    return models.Larva(scale, shape)


def adult(scale: float,
          shape: float):
    """
    Create a adult movement model

    Args:
        scale: scale parameter
        shape: shape parameter

    Returns:
        a functional mathematical model for the simulation
    """

    return models.Adult(scale, shape)
```

C.2.10 reproduction.py

```
import source.reproduction.models as models


def init_sex(female_prob: float):
    """
    Create an initial sex model

    Args:
        female_prob: probability of being female

    Returns:
        a functional mathematical model for the simulation
    """

    return models.InitSex(female_prob)


def mating(mate_encounter: float):
    """
    Create a mate encounter model

    Args:
        mate_encounter: encounter constant for mating

    Returns:
        a functional mathematical model for the simulation
    """

    return models.Mating(mate_encounter)


def radius(mate_radius: float):
    """
    Create a mate radius model

    Args:
        mate_radius: radius of mating
```

```
38
39      Returns:
40          a functional mathematical model for the simulation
41      """
42
43      return models.Radius(mate_radius)
44
45
46  def fecundity(fecundity_maximum: float,
47                fecundity_decay:   float):
48      """
49      Create a fecundity model
50
51      Args:
52          fecundity_maximum: maximum fecundity
53          fecundity_decay:   decay rate of fecundity
54
55      Returns:
56          a functional mathematical model for the simulation
57      """
58
59      return models.Fecundity(fecundity_maximum,
60                              fecundity_decay)
61
62
63  def density(eta:   float,
64              gamma: float):
65      """
66      Create a density model
67
68      Args:
69          eta:   constant of proportionality
70          gamma: shape
71
72      Returns:
73          a functional mathematical model for the simulation
74      """
75
76      return models.Density(eta,
```

```
77                              gamma)
```

C.2.11 simulator.py

```
import datetime
import dataclasses as dclass

import parameters.data_tracking    as tracking
import parameters.model_parameters as param

import models.graph        as graph
import models.migration    as migration

import source.hint    as hint
import source.keyword as keyword

import source.simulation.simulation as main_simulation


@dclass.dataclass
class Simulator(object):
    """
    Class to setup and run a simulation of only biomass growth:

    Variables:
        nums:    initial population numbers
        bt_prop: bt proportion
    """

    grid        = [graph.graph(param.field_grid),
                   graph.graph(param.plant_grid)]
    attrs       = {0: tracking.genotype_attrs}

    steps = [({keyword.female: [keyword.reproduce,
                                keyword.move],
               keyword.male:   [keyword.move],
               keyword.mated:  [keyword.move],
               keyword.larva:  [keyword.consume,
                                keyword.move]}, param.forage_steps, True),
             ({keyword.female: [keyword.survive,
                                keyword.advance_age,
```

```
                                keyword.reset],
            keyword.male: [keyword.survive,
                           keyword.advance_age,
                           keyword.reset],
            keyword.mated: [keyword.survive,
                            keyword.advance_age,
                            keyword.reset],
            keyword.larva: [keyword.grow,
                            keyword.survive,
                            keyword.develop,
                            keyword.advance_age,
                            keyword.reset],
            keyword.pupa: [keyword.survive,
                           keyword.develop,
                           keyword.advance_age],
            keyword.egg: [keyword.survive,
                          keyword.develop,
                          keyword.advance_age],
            keyword.egg_mass: [keyword.reset]},)]

emigration  = [migration.emigration_adult(param.mean_adult,
                                          param.sigma_adult)]
immigration = []
input_variables = param.repro_values
timesteps       = param.timesteps

run_number:   int
nums:         hint.init_pops
bt_prop:      float
input_models: list

path_save: str
base_save: str

data:       tuple           = ()
simulation: hint.simulation = None
step:       int             = 1

def __post_init__(self):
```

```
77          save_name = '{}_{}{}'.format(self.base_save,
78                                       self.run_number,
79                                       '.sqlite')
80          self.data = (100, save_name, self.path_save)
81
82          self.simulation = main_simulation.Simulation. \
83              setup(self.nums,
84                    self.grid,
85                    self.attrs,
86                    self.data,
87                    self.bt_prop,
88                    self.steps,
89                    self.emigration,
90                    self.immigration,
91                    *self.input_models,
92                    **self.input_variables)
93
94      def execute(self) -> None:
95          """
96          Run the simulation
97
98          """
99
100         start_time = datetime.datetime.now()
101         times      = list(range(self.timesteps))
102         run_times = times[self.step:].copy()
103
104         for time in run_times:
105             self.step = time
106             print('     {} Simulation {}, Running step: {}'.
107                   format(datetime.datetime.now(), self.run_number, time))
108             self.simulation.step()
109
110         print('     {} Simulation {}, Complete Starting save'.
111               format(datetime.datetime.now(), self.run_number))
112         self.simulation.database.dump(self.simulation)
113         end_time = datetime.datetime.now()
114         print('Total elapsed time: {}'.format(end_time - start_time))
115
```

```
116     @classmethod
117     def run(cls, run_number:   int,
118                  nums:         hint.init_pops,
119                  bt_prop:      float,
120                  input_models: list,
121                  path_save:    str,
122                  base_save:    str) -> None:
123         """
124         Create and tun the simulation
125 
126         Args:
127             run_number:    the simulation run id
128             nums:          initial population
129             bt_prop:       proportion of bt
130             input_models:  list of input data
131             path_save:     path to save the data
132             base_save:     base for saving the data
133 
134         Effects:
135             runs simulation creating data files
136         """
137 
138         print('{} Run {} Setting Up Long Time Simulations'.
139               format(datetime.datetime.now(), run_number))
140         sim = cls(run_number, nums, bt_prop, input_models, path_save, base_save)
141         print('{} Run {} Running Long Time Simulations'.
142               format(datetime.datetime.now(), run_number))
143         sim.execute()
```

C.2.12 survival.py

```
import source.keyword as keyword

import source.survival.models as models

import models.dominance as dom


def egg_sur(egg_prob: float):
    """
    Create an egg survival model

    Args:
        egg_prob: daily survival probability

    Returns:
        a functional mathematical model for the simulation
    """

    return models.Egg(egg_prob)


def pupa_sur(pupa_prob: float):
    """
    Create a pupa survival model

    Args:
        pupa_prob: daily survival probability

    Returns:
        a functional mathematical model for the simulation
    """

    return models.Pupa(pupa_prob)


def adult_sur(adult_prob: float):
    """
```

```
38      Create an adult survival model
39
40      Args:
41          adult_prob: daily survival probability
42
43      Returns:
44          a functional mathematical model for the simulation
45      """
46
47      return models.Adult(adult_prob)
48
49
50  def larva_sur(larva_prob_non_bt_rr: float,
51                larva_prob_non_bt_ss: float,
52                larva_prob_bt_rr:     float,
53                larva_prob_bt_ss:     float,
54                dominance:            float):
55      """
56      Create a larva survival model
57
58      Args:
59          larva_prob_non_bt_rr: non-Bt RR prob
60          larva_prob_non_bt_ss: non-Bt SS prob
61          larva_prob_bt_rr:     Bt     RR prob
62          larva_prob_bt_ss:     Bt     SS prob
63          dominance:            degree of dominance
64
65      Returns:
66          a functional mathematical model for the simulation
67      """
68
69      prob_non_bt = dom.dom(larva_prob_non_bt_ss,
70                            larva_prob_non_bt_rr,
71                            dominance)
72      prob_bt     = dom.dom(larva_prob_bt_ss,
73                            larva_prob_bt_rr,
74                            dominance)
75
76      prob = {
```

```
77          keyword.bt:     prob_bt,
78          keyword.not_bt: prob_non_bt
79      }
80
81      return models.LarvaFixed(prob)
```

C.3 parameters

```
FallArmyworm
└── parameters
    ├── basic_data.py
    ├── data_tracking.py
    ├── development.py
    ├── forage.py
    ├── growth.py
    ├── init_biomass.py
    ├── model_parameters.py
    ├── movement.py
    ├── reproduction.py
    └── survival.py
```

C.3.1 base_data.py

```
# Included here are data points from a wide array of studies


hourly_steps = 12

mu_leaf  = 10000
sig_leaf = 0.001

# Reproduction mating settings
trials          = 5
lifetime_female = True
lifetime_male   = False
limited         = True

female_prob = 0.5

# Reproductive encounter
mate_encounter = 0.347
mate_radius    = 2

# Reproductive fecundity
fecundity_maximum = 2
fecundity_decay   = 0.5

# Reproductive density
eta   = 0.45
gamma = 4


# Whiteford et.al. 1988 (egg_mass data)
egg_mass_mean_number_eggs = [
    164.4,
    112.8,
    269.6,
    222.1,
    129.2,
    143.2,
```

```
    205.1,
    197.2,
    156.0,
    276.4,
    154.7,
    200.2
]
egg_mass_mean_mass = [
    10.4,
    14.3,
    12.8,
    14.9,
    14.7,
    10.8,
    12.0,
    11.3,
    11.2,
    14.4,
    10.3,
    13.7
]
ss_to_rr_mass_scale = 0.8

# Empirical Mature data
mu_0_mature_ss  = 140.083
sig_0_mature_ss = 17.834
mu_0_mature_rr  = 113.857
sig_0_mature_rr = 22.076


# Veenestra et.al. 1995 (Growth_ss data points)
growth_times_ss = [
    8.0,
    15.6
]
growth_times_sem_ss = [
    0.0,
    0.2
]
```

```
77 growth_mass_ss = [
78     54.4,
79     156.0
80 ]
81 growth_mass_sem_ss = [
82     3.7,
83     5.1
84 ]
85 growth_samples_ss = 40
86
87 # Prasifka et.al. 2009 (Growth_rr data points)
88 growth_times_rr = [
89     10.0,
90     24.3
91 ]
92 growth_times_sem_rr = [
93     0.0,
94     0.4
95 ]
96 growth_mass_rr = [
97     37.4,
98     145.4
99 ]
100 growth_mass_sem_rr = [
101     3.1,
102     5.9
103 ]
104 growth_samples_rr = 40
105
106
107 # Empirical scarcity factors
108 theta_adlibitum = 0.95
109 theta_scarce    = 0.8
110
111 sig_scarce      = 3.0
112
113
114 # Empirical cannibalism factors
115 fight_slope           = 0.5
```

```
cannibalism_radius    = 0
cannibalism_encounter = 0.07

loss_slope = 4
egg_loss   = 0.9
larva_loss = 0.25


# Empirical consumption factors
egg_factor   = 1
larva_factor = 1


# Empirical movement factors
larva_scale = 0.5
larva_shape = 10

adult_scale = 1
adult_shape = 1

# Egg development Sparks 1979 (duration)
mu_egg_dev  = 3.0
sig_egg_dev = 1.0


# Larva Development
larva_mu_dev_offset_rr = 31
larva_mu_dev_offset_ss = 16

# Pitre and Hogg 1983 (Pupation duration)
pupa_duration_pitre_1983 = [
    9.0,
    10.8,
    9.6
]
# Adamczyk .et.al. 2001 (Pupation duration)
pupa_duration_adamczyk_2001 = [
    8.43,
    7.88,
```

```
    8.25,
    7.29,
    8.05,
    7.20
]
pupa_duration = pupa_duration_pitre_1983 + pupa_duration_adamczyk_2001


# Pencoe and Martin (Adult duration)
adult_duration = [
    10.8,
    11.9,
    9.9,
    9.9,
    9.3,
    9.3
]


# Whiteford et.al. (Egg survival)
egg_survivals = [
    0.91,
    0.81,
    0.95,
    0.81,
    0.74,
    0.96,
    0.97,
    0.99,
    0.87,
    0.88,
    0.98,
    0.90
]

# Hardke et.al (Pupa survival/ecolsion)
pupa_survivals = [
    0.567,
    0.787,
```

```
    0.850
]

# Bernardi et.al. (Larva non-Bt survival)
larva_bernardi_non_bt_survivals = [
    0.8,
    0.8,
    0.85,
    0.75
]

# Hardkie et.al. (Larva non-Bt survival)
larva_hardke_non_bt_survivals = [
    0.66,
    0.88,
    0.917
]
larva_non_bt_survivals = larva_bernardi_non_bt_survivals + \
                         larva_hardke_non_bt_survivals

# Bernardi et.al (Larva Bt survival)
larva_bernardi_bt_survivals_rr = [
    0.75
]
larva_bernardi_bt_survivals_ss = [
    0.0
]

# Storer et.al. (Larva Bt survival)
larva_storer_bt_survivals_rr = [
    0.978
]
larva_storer_bt_survivals_ss = [
    0.005
]

# Hardke et.al. (Larva Bt survival)
larva_hardke_bt_survivals_ss = [
    0.367,
```

```
    0.313,
]

# Adamczyk et.al. (Larva Bt survival)
larva_adamczyk_bt_survivals_ss = [
    0.644,
    0.695,
    0.678
]

larva_bt_survivals_rr = larva_bernardi_bt_survivals_rr + \
                        larva_storer_bt_survivals_rr
larva_bt_low_survivals_ss  = larva_bernardi_bt_survivals_ss + \
                             larva_storer_bt_survivals_ss
larva_bt_mid_survivals_ss  = larva_hardke_bt_survivals_ss
larva_bt_high_survivals_ss = larva_adamczyk_bt_survivals_ss
```

C.3.2 data_tracking.py

```
import source.keyword as keyword


genotype_attr = (keyword.genotype,
                 keyword.genotype_keys,
                 False)
genotype      = {keyword.genotype: genotype_attr}

genotype_attrs = {agent_key: genotype
                    for agent_key in keyword.insect_keys}

death_attr = (keyword.death,
              keyword.death_keys,
              True)
death      = {keyword.death: death_attr}

death_attrs = {agent_key: death
                  for agent_key in keyword.insect_keys}

death_filter_attr = (keyword.death,
                     keyword.death_keys,
                     {keyword.genotype:
                          (keyword.genotype_keys,
                           True)
                      })
death_filter = {keyword.death_track: death_filter_attr}

death_filter_attrs = {agent_key: death_filter
                        for agent_key in keyword.insect_keys}

genotype_death       = {**genotype, **death}
genotype_death_attrs = {agent_key: genotype_death
                             for agent_key in keyword.insect_keys}

genotype_death_filters = {**genotype, **death_filter}
genotype_death_filters_attrs = {agent_key: genotype_death_filters
                                    for agent_key in keyword.insect_keys}
```

C.3.3 development.py

```python
import numpy as np

import parameters.basic_data as data


def normal_pupa(duration_data: list) -> tuple:
    """
    Calculate the normal distribution pupal stage duration

    Args:
        duration_data: data points on pupa durations

    Returns:
        mu, sigma for duration
    """

    mu    = float(np.mean(duration_data))
    sigma = float(np.std(duration_data))

    return mu, sigma


def normal_larva(mass:     list,
                 mass_sem: list,
                 offset:   float,
                 samples:  int) -> tuple:
    """
    Get a normal distribution for the mass of larva pupating
    Args:
        mass:     mass sample point data
        mass_sem: mass sem data
        offset:   discretization offset
        samples:  number of total samples for each point

    Returns:
        mu, sigma for pupation mass
    """
```

```
38
39      mu = mass[-1]
40
41      sem   = mass_sem[-1]
42      sigma = sem*np.sqrt(samples)
43
44      return mu + offset, sigma
45
46
47  # Pass through the egg development parameters
48  mu_egg  = data.mu_egg_dev
49  sig_egg = data.sig_egg_dev
50
51  # Generate the pupa development parameters
52  mu_pupa, sig_pupa = normal_pupa(data.pupa_duration)
53
54  # Generate the larva development parameters
55  mu_larva_ss, sig_larva_ss = normal_larva(data.growth_mass_ss,
56                                           data.growth_mass_sem_ss,
57                                           data.larva_mu_dev_offset_ss,
58                                           data.growth_samples_ss)
59  mu_larva_rr, sig_larva_rr = normal_larva(data.growth_mass_rr,
60                                           data.growth_mass_sem_rr,
61                                           data.larva_mu_dev_offset_rr,
62                                           data.growth_samples_rr)
```

C.3.4 forage.py

```
import source.keyword as keyword

import parameters.basic_data as data


# Get the consumption steps
forage_steps = data.hourly_steps

# Pass through the theta values
theta_adlibitum = data.theta_adlibitum
theta_scarce    = data.theta_scarce
sig_scarce      = data.sig_scarce

# Pass through the cannibalism factors
fight_slope           = data.fight_slope
cannibalism_radius    = data.cannibalism_radius
cannibalism_encounter = data.cannibalism_encounter


def mid(prob: float) -> float:
    """
    Calculate the actual value of mid point constant

    Args:
        prob: loss probability

    Returns:
        the mid factor
    """

    return (1 - prob) / prob


loss_slope = data.loss_slope
mid = {
    keyword.egg_mass: mid(data.egg_loss),
    keyword.larva:    mid(data.larva_loss)
```

```
}

# Pass through the forage factors

egg_factor   = data.egg_factor
larva_factor = data.larva_factor
```

C.3.5 growth.py

```
import dataclasses    as dclass
import numpy          as np
import scipy.optimize as opt

import parameters.basic_data   as data
import parameters.init_biomass as init_bio


@dclass.dataclass
class Growth(object):
    """
    Class to fit the biomass parameters to known data:
        Biomass is assumed to be modeled via the West et.al. Mass model
            m^0.25/M = 1 - [1 - m_0^0.25/M]exp(-at/4M)

    Variables:
        m_0: initial mass

        m_m: middle mass
        t_m: middle mass time

        m_f: final mass
        t_f: final mass time

    Methods:
        parameters: return a tuple of parameters
    """

    m_0: float

    m_m_mean: float
    m_m_sem:  float
    t_m_mean: float
    t_m_sem:  float

    m_f_mean: float
    m_f_sem:  float
```

```
38      t_f_mean: float
39      t_f_sem:  float
40
41      m_m: float = None
42      t_m: float = None
43      m_f: float = None
44      t_f: float = None
45
46      def __post_init__(self):
47          """Sets the actual values from the mean and sem values"""
48
49          self.m_m = self.m_m_mean + self.m_m_sem
50          self.t_m = self.t_m_mean - self.t_m_sem
51          self.m_f = self.m_f_mean + self.m_f_sem
52          self.t_f = self.t_f_mean - self.t_f_sem
53
54      def _root_fun(self, mass_const: float) -> float:
55          """
56          Mass constant root finding function
57              fun(M) = [(M-m_m^0.25)/(M - m_0^0.25)]^t_f -
58                  [(M-m_f^0.25)/(M-m_0^0.25)]^t_m
59
60          Args:
61              mass_const: mass (M in equation)
62
63          Returns:
64              evaluation of above equation fun(M)
65          """
66
67          bottom = mass_const - self.m_0**0.25
68
69          top_m = mass_const - self.m_m**0.25
70          top_f = mass_const - self.m_f**0.25
71
72          exp_m = top_m/bottom
73          exp_f = top_f/bottom
74
75          return exp_m**self.t_f - exp_f**self.t_m
76
```

```
77      def _mass_constant(self) -> float:
78          """
79          Calculate the mass constant M from West et.al. biomass model:
80              m^0.25/M = 1 - [1 - m_0^0.25/M]exp(-at/4M)
81
82          Returns:
83              the mass constant M
84          """
85
86          sol = opt.root(self._root_fun, np.array(4))
87
88          return float(sol.x[0])
89
90      def _alpha(self, mass_const: float) -> float:
91          """
92          Find the alpha constant in the West et.al. model
93              m^0.25/M = 1 - [1 - m_0^0.25/M]exp(-alpha*t/4M)
94          where M=m from input
95
96          That is:
97              alpha = (-4M/t_m)ln[(M-m_m^0.25)/(M-m_0^0.25)]
98
99          Args:
100             mass_const: the mass constant
101
102         Returns:
103             the value for alpha
104         """
105
106         bottom = mass_const - self.m_0**0.25
107         top    = mass_const - self.m_m**0.25
108
109         coeff = -4*mass_const/self.t_m
110         ln    = np.log(top/bottom)
111
112         return coeff*ln
113
114     @staticmethod
115     def _max_mass(mass_const: float) -> float:
```

```
116         """
117         Find the maximum mass:
118             m = max_mass^0.25
119 
120         Args:
121             mass_const: the mass constant
122 
123         Returns:
124             the maximum mass
125         """
126 
127         return mass_const**4
128 
129     @staticmethod
130     def _beta(alpha:      float,
131               mass_const: float) -> float:
132         """
133         Find the beta West et.al. constant which is given by:
134             beta = alpha/mass_const
135 
136         Args:
137             alpha:      alpha constant
138             mass_const: mass constant
139 
140         Returns:
141             value for beta
142         """
143 
144         return alpha/mass_const
145 
146     def params(self) -> tuple:
147         """
148         Calculate the parameters from the data
149 
150         Returns:
151             (alpha, beta, max_mass, mass_const)
152         """
153 
154         mass_const = self._mass_constant()
```

```
155         alpha      = self._alpha(mass_const)
156         max_mass   = self._max_mass(mass_const)
157         beta       = self._beta(alpha, mass_const)
158 
159         return alpha, beta, max_mass, mass_const
160 
161     @classmethod
162     def parameters(cls, mass_0:   float,
163                         time:     list,
164                         time_sem: list,
165                         mass:     list,
166                         mass_sem: list) -> tuple:
167         """
168         Use the growth class to calculate the parameters needed for the growth
169             model
170 
171         Args:
172             mass_0:   initial mass
173             time:     time points
174             time_sem: sem in each point
175             mass:     mass points
176             mass_sem: sem in each mass point
177 
178         Returns:
179             (alpha, beta, max_mass, mass_const)
180         """
181 
182         t_m_mean = time[0]
183         t_f_mean = time[1]
184         t_m_sem  = time_sem[0]
185         t_f_sem  = time_sem[1]
186 
187         m_m_mean = mass[0]
188         m_f_mean = mass[1]
189         m_m_sem  = mass_sem[0]
190         m_f_sem  = mass_sem[1]
191 
192         growth = cls(mass_0,
193                      m_m_mean, m_m_sem,
```

```
                    t_m_mean, t_m_sem,
                    m_f_mean, m_f_sem,
                    t_f_mean, t_f_sem)

        return growth.params()


# Generate the growth parameters
growth_ss = Growth.parameters(init_bio.mu_0_larva_ss,
                              data.growth_times_ss,
                              data.growth_times_sem_ss,
                              data.growth_mass_ss,
                              data.growth_mass_sem_ss)
growth_rr = Growth.parameters(init_bio.mu_0_larva_rr,
                              data.growth_times_rr,
                              data.growth_times_sem_rr,
                              data.growth_mass_rr,
                              data.growth_mass_sem_rr)

alpha_ss      = growth_ss[0]
beta_ss       = growth_ss[1]
max_mass_ss   = growth_ss[2]
mass_const_ss = growth_ss[3]

alpha_rr      = growth_rr[0]
beta_rr       = growth_rr[1]
max_mass_rr   = growth_rr[2]
mass_const_rr = growth_rr[3]
```

C.3.6 init_biomass.py

```python
import numpy as np

import parameters.basic_data as data


def poisson_egg_mass(number_data: list) -> float:
    """
    Find the value of the mean for the Poisson distribution of number of eggs
        in egg_mass from the data

    Args:
        number_data: list of number of eggs in egg_mass data points

    Returns:
        lambda
    """

    return float(np.mean(number_data))


def normal_egg_mass(mass_data: list) -> tuple:
    """
    Find the values for the normal distribution of egg_mass masses from the
        given data

    Args:
        mass_data: list of egg_mass mass data points

    Returns:
        mean, standard deviation
    """

    mu    = float(np.mean(mass_data))
    sigma = float(np.std(mass_data))

    return mu, sigma

```

```
38
39 def normal_scaling(mu:    float,
40                     sigma: float,
41                     scale: float) -> tuple:
42     """
43     Scale the parameters of a normal distribution correctly
44
45     Args:
46         mu:    mean of distribution
47         sigma: std of distribution
48         scale: scale factor
49
50     Returns:
51         scaled mu, scaled sigma
52     """
53
54     s_mu    = mu*scale
55     s_sigma = sigma*(scale**2)
56
57     return s_mu, s_sigma
58
59
60 def normal_larva(mu:    float,
61                  sigma: float,
62                  lam:   float) -> tuple:
63     """
64     Get the parameters of a normal distribution for the mass of a new larva
65
66     Args:
67         mu:    mean egg_mass mass
68         sigma: std  egg_mass mass
69         lam:   mean egg_mass num eggs
70
71     Returns:
72         larva mu, larva_sigma
73     """
74
75     scale = 1/lam
76
```

```
77      return normal_scaling(mu, sigma, scale)
78
79
80  # Generate the init_egg parameters
81  lam_0_egg                 = poisson_egg_mass(data.egg_mass_mean_number_eggs)
82  mu_0_egg_ss, sig_0_egg_ss = normal_egg_mass(data.egg_mass_mean_mass)
83  mu_0_egg_rr, sig_0_egg_rr = normal_scaling(mu_0_egg_ss, sig_0_egg_ss,
84                                             data.ss_to_rr_mass_scale)
85
86  # Generate the init_larva parameters
87  mu_0_larva_ss, sig_0_larva_ss = normal_larva(mu_0_egg_ss, sig_0_egg_ss,
88                                               lam_0_egg)
89  mu_0_larva_rr, sig_0_larva_rr = normal_larva(mu_0_egg_rr, sig_0_egg_rr,
90                                               lam_0_egg)
91
92  # Pass through the init_mature parameters
93  mu_0_mature_ss  = data.mu_0_mature_ss
94  sig_0_mature_ss = data.sig_0_mature_ss
95  mu_0_mature_rr  = data.mu_0_mature_rr
96  sig_0_mature_rr = data.sig_0_mature_rr
97
98
99  # Pass through the init_leaf parameters
100 mu_leaf  = data.mu_leaf
101 sig_leaf = data.sig_leaf
```

C.3.7 model_parameters.py

```
import parameters.development  as dev
import parameters.forage       as forage
import parameters.growth       as growth
import parameters.init_biomass as init_bio
import parameters.movement     as move
import parameters.reproduction as repro
import parameters.survival     as sur


print_data = True

field_grid = 50
plant_grid = 10

timesteps = 4200

mean_adult  = 300
sigma_adult = 10

dominance_0 = 0
dominance_1 = 0.11
dominance_2 = 0.29

start_preg_ss = 200
start_preg_rr = 200

mix_preg_rr = 50
mix_preg_ss = 200

bt_prop_0 = 0.0
bt_prop_7 = 0.7
bt_prop_8 = 0.8
bt_prop_9 = 0.9
bt_prop_1 = 1.0

# List out init biomass parameters
lam_0_egg    = init_bio.lam_0_egg
```

```
mu_0_egg_ss  = init_bio.mu_0_egg_ss
sig_0_egg_ss = init_bio.sig_0_egg_ss
mu_0_egg_rr  = init_bio.mu_0_egg_rr
sig_0_egg_rr = init_bio.sig_0_egg_rr

mu_0_larva_ss  = init_bio.mu_0_larva_ss
sig_0_larva_ss = init_bio.sig_0_larva_ss
mu_0_larva_rr  = init_bio.mu_0_larva_rr
sig_0_larva_rr = init_bio.sig_0_larva_rr

mu_0_mature_ss  = init_bio.mu_0_mature_ss
sig_0_mature_ss = init_bio.sig_0_mature_ss
mu_0_mature_rr  = init_bio.mu_0_mature_rr
sig_0_mature_rr = init_bio.sig_0_mature_rr

mu_leaf  = init_bio.mu_leaf
sig_leaf = init_bio.sig_leaf

# List out the growth parameters
alpha_ss      = growth.alpha_ss
beta_ss       = growth.beta_ss
max_mass_ss   = growth.max_mass_ss
mass_const_ss = growth.mass_const_ss

alpha_rr      = growth.alpha_rr
beta_rr       = growth.beta_rr
max_mass_rr   = growth.max_mass_rr
mass_const_rr = growth.mass_const_rr

# List out the forage parameters
forage_steps    = forage.forage_steps
theta_adlibitum = forage.theta_adlibitum
theta_scarce    = forage.theta_scarce
sig_scarce      = forage.sig_scarce

fight_slope           = forage.fight_slope
cannibalism_radius    = forage.cannibalism_radius
cannibalism_encounter = forage.cannibalism_encounter

```

```
77 cannib_0 = cannibalism_encounter
78 cannib_1 = 0.15
79 cannib_2 = 0.38
80 cannib_3 = 0.5
81 cannib_4 = 1.0
82
83 egg_factor   = forage.egg_factor
84 larva_factor = forage.larva_factor
85
86 loss_slope = forage.loss_slope
87 mid        = forage.mid
88
89 # List out development parameters
90 mu_egg_dev  = dev.mu_egg
91 sig_egg_dev = dev.sig_egg
92
93 mu_pupa_dev  = dev.mu_pupa
94 sig_pupa_dev = dev.sig_pupa
95
96 mu_larva_dev_ss  = dev.mu_larva_ss
97 sig_larva_dev_ss = dev.sig_larva_ss
98 mu_larva_dev_rr  = dev.mu_larva_rr
99 sig_larva_dev_rr = dev.sig_larva_rr
100
101 # List out reproduction parameters
102 female_prob = repro.female_prob
103
104 mate_encounter = repro.encounter
105 mate_radius    = repro.radius
106
107 fecundity_maximum = repro.maximum
108 fecundity_decay   = repro.decay
109
110 eta   = repro.eta
111 gamma = repro.gamma
112
113 repro_values = repro.values
114
115 # List out movement parameters
```

```
116 larva_scale = move.larva_scale
117 larva_shape = move.larva_shape
118 adult_scale = move.adult_scale
119 adult_shape = move.adult_shape
120
121 # List out survival parameters
122 egg_prob   = sur.egg_prob
123 pupa_prob  = sur.pupa_prob
124 adult_prob = sur.adult_prob
125
126 larva_prob_non_bt_rr  = sur.larva_non_bt_rr
127 larva_prob_non_bt_ss  = sur.larva_non_bt_ss
128 larva_prob_bt_rr      = sur.larva_bt_rr
129 larva_prob_bt_low_ss  = sur.larva_bt_low_ss
130 larva_prob_bt_mid_ss  = sur.larva_bt_mid_ss
131 larva_prob_bt_high_ss = sur.larva_bt_high_ss
132
133
134 # Print data to console
135 if print_data:
136     # Print the init_biomass
137     print('Init_egg_mass parameters')
138     print('    lam_0_egg: {}'.format(lam_0_egg))
139     print('    mu_0_egg_ss: {}, sig_0_egg_ss: {}'.
140           format(mu_0_egg_ss, sig_0_egg_ss))
141     print('    mu_0_egg_rr: {}, sig_0_egg_rr: {}'.
142           format(mu_0_egg_rr, sig_0_egg_rr))
143     print('Init_larva parameters')
144     print('    mu_0_larva_ss: {}, sig_0_larva_ss: {}'.
145           format(mu_0_larva_ss, sig_0_larva_ss))
146     print('    mu_0_larva_rr: {}, sig_0_larva_rr: {}'.
147           format(mu_0_larva_rr, sig_0_larva_rr))
148     print('Init_mature parameters')
149     print('    mu_0_mature_ss: {}, sig_0_mature_ss: {}'.
150           format(mu_0_mature_ss, sig_0_mature_ss))
151     print('    mu_0_mature_rr: {}, sig_0_mature_rr: {}'.
152           format(mu_0_mature_rr, sig_0_mature_rr))
153     print('    mu_leaf: {}, sig_leaf: {}'.
154           format(mu_leaf, sig_leaf))
```

```
155
156     # Print the growth
157     print('Growth parameters')
158     print('    alpha_ss: {}, beta_ss: {}, max_mass_ss: {}'.
159           format(alpha_ss, beta_ss, max_mass_ss))
160     print('    alpha_rr: {}, beta_rr: {}, max_mass_rr: {}'.
161           format(alpha_rr, beta_rr, max_mass_rr))
162
163     # Print the development
164     print('Development parameters')
165     print('    mu_egg_dev: {}, sig_egg_dev: {}'.
166           format(mu_egg_dev, sig_egg_dev))
167     print('    mu_pupa_dev: {}, sig_pupa_dev: {}'.
168           format(mu_pupa_dev, sig_pupa_dev))
169     print('    mu_larva_dev_ss: {}, sig_larva_dev_ss: {}'.
170           format(mu_larva_dev_ss, sig_larva_dev_ss))
171     print('    mu_larva_dev_rr: {}, sig_larva_dev_rr: {}'.
172           format(mu_larva_dev_rr, sig_larva_dev_rr))
173
174     # Print the movement
175     print('Movement parameters')
176     print('    larva_scale: {}, larva_shape: {}'.
177           format(larva_scale, larva_shape))
178     print('    adult_scale: {}, adult_shape: {}'.
179           format(adult_scale, adult_shape))
```

C.3.8 movement.py

```
import parameters.basic_data as data


# Pass through the larva data
larva_scale = data.larva_scale
larva_shape = data.larva_shape

# Pass through the adult data
adult_scale = data.adult_scale
adult_shape = data.adult_shape
```

C.3.9 reproduction.py

```
import source.keyword as keyword

import parameters.basic_data as data


# Pass through the sex probability
female_prob = data.female_prob

# Pass through the reproductive encounter factors
encounter = data.mate_encounter
radius    = data.mate_radius

# Pass through the fecundity factors
maximum = data.fecundity_maximum
decay   = data.fecundity_decay

# Pass through the density factors
eta   = data.eta
gamma = data.gamma

# Pass through fixed values
values = {keyword.trials:          data.trials,
          keyword.limited:         data.limited,
          keyword.lifetime_male:   data.lifetime_male,
          keyword.lifetime_female: data.lifetime_female}
```

C.3.10 survival.py

```
import numpy as np

import parameters.basic_data as data


def probability_egg(duration:       float,
                    total_survival: list) -> float:
    """
    Calculate the daily survival probability for eggs:

    Args:
        duration:       average duration of stage
        total_survival: data points on total survival

    Returns:
        the average daily survival probability for eggs
    """

    total = np.mean(total_survival)

    power = 1 / (duration - 1)

    daily = total**power

    print('Egg Survival')
    print('   Egg total survival: {}'. format(total))
    print('   Egg total duration: {}'. format(duration))
    print('   Egg daily survival: {}'. format(daily))

    return daily


def probability_pupa(duration:       list,
                     total_survival: list) -> float:
    """
    Calculate the daily survival probability for pupae

```

```
38      Args:
39          duration:       average duration data
40          total_survival: data points on total survival
41
42      Returns:
43          the average daily survival probability for pupae
44      """
45
46      t     = np.mean(duration)
47      total = np.mean(total_survival)
48
49      power = 1 / (t - 1)
50
51      daily = total**power
52
53      print('Pupa Survival')
54      print('   Pupa total survival: {}'. format(total))
55      print('   Pupa total duration: {}'. format(t))
56      print('   Pupa daily survival: {}'. format(daily))
57
58      return daily
59
60
61  def probability_adult(duration: list) -> float:
62      """
63      Calculate the daily survival probability for adults
64
65      Args:
66          duration: average duration for an adult
67
68      Returns:
69          the average daily survival probability for adults
70      """
71
72      t = np.mean(duration)
73
74      daily = (t - 1) / t
75
76      print('Adult Survival')
```

```
77      print('   Adult total duration: {}'. format(t))
78      print('   Adult daily survival: {}'. format(daily))
79
80      return daily
81
82
83  def probability_larva(duration_rr: float,
84                        duration_ss: float,
85                        total_survival_non_bt: list,
86                        total_survival_bt_rr:  list) -> tuple:
87      """
88      Calculate the daily survival probabilities for larvae
89      Args:
90          duration_rr:           duration of rr
91          duration_ss:           duration of ss
92          total_survival_non_bt: data on total survival on non Bt
93          total_survival_bt_rr:  data on total survival on Bt for rr
94
95      Returns:
96          the daily survival for non Bt rr,
97          the daily survival for non Bt ss,
98          the daily survival for Bt rr,
99          the daily survival for Bt ss
100     """
101
102     t = np.mean([duration_rr, duration_ss])
103
104     power     = 1 / (t - 1)
105     # power_rr  = 1 / (duration_rr - 1)
106
107     total_survival = total_survival_non_bt + total_survival_bt_rr
108     total          = np.mean(total_survival)
109     # total_non_bt = np.mean(total_survival_non_bt)
110     # total_bt_rr  = np.mean(total_survival_bt_rr)
111
112     daily = total**power
113     # daily_non_bt_rr = total_non_bt**power
114     # daily_non_bt_ss = total_non_bt**power
115     # daily_bt_rr     = total_bt_rr**power_rr
```

```
116
117     print('Larva non-Bt RR')
118     print('   Larva non-Bt RR total survival: {}'. format(total))
119     print('   Larva non-Bt RR total duration: {}'. format(t))
120     print('   Larva non-Bt RR daily survival: {}'. format(daily))
121
122     print('Larva non-Bt SS')
123     print('   Larva non-Bt SS total survival: {}'. format(total))
124     print('   Larva non-Bt SS total duration: {}'. format(t))
125     print('   Larva non-Bt SS daily survival: {}'. format(daily))
126
127     print('Larva Bt RR')
128     print('   Larva RR total survival: {}'. format(total))
129     print('   Larva RR total duration: {}'. format(t))
130     print('   Larva RR daily survival: {}'. format(daily))
131
132     return daily, daily, daily
133
134
135 def probability_larva_low(duration: float,
136                           total_survival: list) -> float:
137     """
138     Calculate the low ss survival probability
139     Args:
140         duration:       duration ss
141         total_survival: low total survival ss
142
143     Returns:
144         daily survival for ss (low)
145     """
146
147     power = 1 / (duration - 1)
148     total = np.mean(total_survival)
149     daily = total**power
150
151     print('Larva Bt SS')
152     print('   Larva SS low total survival: {}'. format(total))
153     print('   Larva SS low total duration: {}'. format(duration))
154     print('   Larva SS low daily survival: {}'. format(daily))
```

```
155
156     return daily
157
158
159 def probability_larva_mid(duration: float,
160                            total_survival: list) -> float:
161     """
162     Calculate the mid ss survival probability
163     Args:
164         duration:       duration ss
165         total_survival: mid total survival ss
166
167     Returns:
168         daily survival for ss (low)
169     """
170
171     power = 1 / (duration - 1)
172     total = np.mean(total_survival)
173     daily = total ** power
174
175     print('Larva Bt SS')
176     print('   Larva SS mid total survival: {}'.format(total))
177     print('   Larva SS mid total duration: {}'.format(duration))
178     print('   Larva SS mid daily survival: {}'.format(daily))
179
180     return daily
181
182
183 def probability_larva_high(duration: float,
184                            total_survival: list) -> float:
185     """
186     Calculate the high ss survival probability
187     Args:
188         duration:       duration ss
189         total_survival: high total survival ss
190
191     Returns:
192         daily survival for ss (low)
193     """
```

```
194
195     power = 1 / (duration - 1)
196     total = np.mean(total_survival)
197     daily = total ** power
198
199     print('Larva Bt SS')
200     print('   Larva SS high total survival: {}'.format(total))
201     print('   Larva SS high total duration: {}'.format(duration))
202     print('   Larva SS high daily survival: {}'.format(daily))
203
204     return daily
205
206
207 # Generate the egg survival probability
208 egg_prob = probability_egg(data.mu_egg_dev,
209                            data.egg_survivals)
210
211 # Generate the pupa survival probability
212 pupa_prob = probability_pupa(data.pupa_duration,
213                              data.pupa_survivals)
214
215 # Generate the adult survival probability
216 adult_prob = probability_adult(data.adult_duration)
217
218 # Generate the larva survival probabilities
219 larva_non_bt_rr, larva_non_bt_ss, larva_bt_rr = \
220     probability_larva(data.growth_times_rr[-1],
221                       data.growth_times_ss[-1],
222                       data.larva_non_bt_survivals,
223                       data.larva_bt_survivals_rr)
224 larva_bt_low_ss  = probability_larva_low(data.growth_times_ss[-1],
225                                          data.larva_bt_low_survivals_ss)
226 larva_bt_mid_ss  = probability_larva_mid(data.growth_times_ss[-1],
227                                          data.larva_bt_mid_survivals_ss)
228 larva_bt_high_ss = probability_larva_high(data.growth_times_ss[-1],
229                                           data.larva_bt_high_survivals_ss)
```

C.4 source

```
FallArmyworm
└── source
    ├── agents
    ├── biomass
    ├── data
    ├── development
    ├── forage
    ├── migration
    ├── movement
    ├── reproduction
    ├── schedule
    ├── simulation
    ├── space
    ├── survival
    ├── hint.py
    └── keyword.py
```

C.4.1 agents

```
FallArmyworm
└── source
    └── agents
        ├── adult.py
        ├── agent.py
        ├── egg.py
        ├── egg_mass.py
        ├── insect.py
        ├── larva.py
        └── pupa.py
```

C.4.1.1 adult.py

```
import dataclasses as dclass
import itertools   as i_tools

import source.hint    as hint
import source.keyword as keyword

import source.agents.insect as insect


@dclass.dataclass
class Adult(insect.Insect):
    """
    Class to contain a adult agent
        - inherits from base insect class

    Variables:
        female:   if female or not
        num_eggs: number of egg_masses that can still be laid
        survival: survival system
        movement: movement system
        lay:      egg_laying system
        mate:     adult mating system

    Methods:
        survive: have the adult survive
        move:    have the larva move
    """

    num_eggs: int
    mate:     hint.adult_mate
    survival: hint.adult_survival
    movement: hint.adult_movement
    lay:      hint.lay
    mating:   hint.mate

    def __post_init__(self):
        """Setup some helper systems"""
```

```
38
39          super().__post_init__()
40
41          self._id_count = i_tools.count()
42
43      def survive(self) -> hint.agent_list:
44          """
45          Run the survive behavior
46
47          Effects:
48              run behavior to determine if this survives
49
50          Returns:
51              empty list
52          """
53
54          self.survival.survive(self)
55
56          return []
57
58      def move(self) -> hint.agent_list:
59          """
60          Run the move behavior
61
62          Effects:
63              run behavior to move
64
65          Returns:
66              empty list
67          """
68
69          if self.alive:
70              self.movement.move(self)
71
72          return []
73
74      def new_unique_id(self) -> str:
75          """
76          Create a new unique_id
```

```
77
78          Returns:
79              a unique_id
80          """
81
82          return '{}_{}{}'.format(self.unique_id,
83                                  next(self._id_count),
84                                  keyword.egg_mass)
85
86      def new_egg_location(self) -> hint.location:
87          """
88          Create a new location for egg_mass
89
90          Returns:
91              a location for an egg_mass
92          """
93
94          location = self.location
95
96          for _ in range(keyword.egg_depth - keyword.adult_depth):
97              location = self.simulation.space.extend_location(location)
98
99          return location
100
101     def population(self) -> int:
102         """
103         Get the population of the plant the adult is on
104
105         Returns:
106             number of egg_masses and number of larvae
107         """
108
109         location_key = self.location.location_key
110         agent_bin    = self.simulation.agents[location_key]
111
112         num_eggs   = len(agent_bin[keyword.egg_mass].agents)
113         num_larvae = len(agent_bin[keyword.larva].   agents)
114
115         return num_eggs + num_larvae
```

```
116
117     def set_mate(self, mate: hint.adult) -> None:
118         """
119         Set the mate for adult
120
121         Args:
122             mate: the mate for the adult
123
124         Effects:
125             set the mate of the adult
126         """
127
128         if self.agent_key == keyword.male:
129             if self.simulation.models[keyword.lifetime_male]:
130                 self.deactivate()
131             elif self.simulation.models[keyword.limited]:
132                 self.transition(keyword.mated)
133         else:
134             self.mate = mate.genotype
135
136     def _location_keys(self, **kwargs) -> hint.location_keys:
137         """
138         Get the location_keys for vertices in range
139
140         Args:
141             **kwargs: bounds for the range
142
143         Returns:
144             list of location keys
145         """
146
147         vertices = self.vertices(**kwargs)
148
149         location_keys = []
150         for vertex in vertices:
151             loc                         = self.location.copy()
152             loc[keyword.adult_level] = vertex
153             location_keys.append(loc.location_key)
154
```

```
155         return location_keys
156
157     def mates(self, **kwargs) -> hint.mates:
158         """
159         Get a list of egg_masses within the space bounds given
160
161         Args:
162             **kwargs: bounds for the range
163
164         Returns:
165             list of target larvae and eggs
166         """
167
168         location_keys = self._location_keys(**kwargs)
169
170         mates = []
171         for location_key in location_keys:
172             agent_bin = self.simulation.agents[location_key]
173             mates    += agent_bin[keyword.male].agents
174
175         return mates
176
177     def reproduce(self) -> hint.agent_list:
178         """
179         Runs reproduction system:
180
181         Returns:
182             list of egg_masses which have been laid
183         """
184
185         if self.alive:
186             if self.mate is not None:
187                 new_eggs = self.lay.lay(self)
188             else:
189                 new_eggs = []
190
191                 self.mating.mate(self)
192
193             for new_egg in new_eggs:
```

```
194                 new_egg.activate()
195
196
197         return []
198
199     def reset(self) -> hint.agent_list:
200         """
201         Reset the agent
202
203         Effects:
204             reset the agent
205         """
206
207         if self.agent_key == keyword.female:
208             self.num_eggs = self.lay.reset(self)
209
210             if not self.simulation.models[keyword.lifetime_female]:
211                 self.mate = None
212
213         elif (self.agent_key == keyword.mated) and self.alive:
214             self.transition(keyword.male)
215
216         return []
217
218     def _set_sex(self) -> None:
219         """
220         Get the agent_key for the adult:
221
222         Returns:
223             the proper agent key for the sex
224         """
225
226         if self.mate is None:
227             if self.simulation.models[keyword.init_sex](self.genotype):
228                 self.agent_key = keyword.female
229             else:
230                 self.agent_key = keyword.male
231         else:
232             self.agent_key = keyword.female
```

```
233             self.num_eggs  = self.lay.reset(self)
234
235     @classmethod
236     def initialize(cls, unique_id:  str,
237                         simulation: hint.simulation,
238                         location:   hint.location,
239                         mass:       float,
240                         genotype:   str,
241                         mate:       hint.adult_mate = None) -> 'Adult':
242         """
243         Initialize a new adult agent
244
245         Args:
246             unique_id:  the agent's unique_id
247             simulation: the master simulation
248             location:   the agent's location
249             mass:       the agent's mass
250             genotype:   the agent's genotype
251             mate:       the agent's mate
252
253         Returns:
254             A fully initialized agent
255         """
256
257         alive    = True
258         age      = 0
259         death    = keyword.alive
260         num_eggs = 0
261
262         survival = simulation.behaviors.survive_adult
263         movement = simulation.behaviors.move_adult
264         lay      = simulation.behaviors.lay
265         mating   = simulation.behaviors.mate
266
267         new = cls('', unique_id, simulation, location,
268                   alive, mass, genotype, age, death, num_eggs, mate,
269                   survival, movement, lay, mating)
270         new._set_sex()
271
```

```
272         return new
273
274     @classmethod
275     def setup(cls, unique_id_num: int,
276                    initial_key:   str,
277                    simulation:    hint.simulation,
278                    genotype:      str,
279                    mate:          hint.adult_mate = None) -> 'Adult':
280         """
281         Setup an initial population adult
282
283         Args:
284             unique_id_num: unique_id number
285             initial_key:   key for where agent was initialized
286             simulation:    the master simulation
287             genotype:      the agent's genotype
288             mate:       the agent's mate
289
290         Returns:
291             a adult initialized by a population
292         """
293
294         unique_id = '{}{}{}'.format(initial_key,
295                                     unique_id_num,
296                                     keyword.adult)
297         location  = simulation.space.new_location(keyword.adult_depth)
298         mass      = simulation.models[keyword.init_mature](genotype)
299
300         return cls.initialize(unique_id, simulation, location,
301                               mass, genotype, mate)
302
303     @classmethod
304     def advance(cls, pupa: hint.pupa) -> 'Adult':
305         """
306         Create a adult agent for this pupa
307
308         Args:
309             pupa: the pupa in question
310
```

```
311         Returns:
312             A adult version of this pupa
313         """
314 
315         return cls.initialize(pupa.unique_id,
316                               pupa.simulation,
317                               pupa.location,
318                               pupa.mass,
319                               pupa.genotype)
```

C.4.1.2 agent.py

```
import dataclasses as dclass

import source.hint as hint


@dclass.dataclass
class Agent(object):
    """
    Base class to for an agent

    Variables:
        agent_key:  the agent type identifier
        unique_id:  the unique identifier
        simulation: the simulation
        location:   location tuple
        alive:      determine of the agent is alive

    Methods:
        activate:   activate   the agent
        deactivate: deactivate the agent
        transfer:   transfer   the agent
        die:        have agent die
    """

    agent_key:  str
    unique_id:  str
    simulation: hint.simulation
    location:   hint.location
    alive:      bool

    def activate(self) -> None:
        """
        Activate the agent

        Effects:
            activates the agent in the system
        """
```

```
38
39          self.simulation.agents.activate(self)
40
41      def deactivate(self) -> None:
42          """
43          Activate the agent
44
45          Effects:
46              deactivates the agent in the system
47          """
48
49          self.simulation.agents.deactivate(self)
50
51      def transfer(self, vertex: int,
52                           level:  int) -> None:
53          """
54          Transfer the agent to a new vertex
55
56          Args:
57              vertex: vertex
58              level:  level of vertex
59
60          Effects:
61              removes from current location
62              updates location
63              adds to new location
64          """
65
66          # print('{} agent currently at {} moving to vertex: {} at {}'.
67          #       format(self.unique_id,
68          #              self.location,
69          #              vertex,
70          #              level))
71
72          self.simulation.agents.deactivate(self)
73          self.location[level] = vertex
74          self.simulation.agents.activate(self)
75
76          # print('{} agent now at {}'.format(self.unique_id,
```

```
77          #                                   self.location))
78
79      def transition(self, agent_key: str) -> None:
80          """
81          Transition the agent_key of the agent
82
83          Args:
84              agent_key: new agent key
85
86          Effects:
87              removes agent
88              updates agent_key
89              adds agent back
90          """
91
92          self.simulation.agents.deactivate(self)
93          self.agent_key = agent_key
94          self.simulation.agents.activate(self)
95
96      def vertices(self, **kwargs) -> hint.vertices:
97          """
98          Get the vertices in the distance neighborhood of this agent's location
99
100         Args:
101             **kwargs: the bounds to use
102
103         Returns:
104             set of vertices at that distance
105         """
106
107         return self.simulation.space.neighborhood(self.location, **kwargs)
108
109     def die(self, death: str = '') -> None:
110         """
111
112         Have agent die
113
114         Args:
115             death: the type of death
```

```
116
117         Effects:
118             sets alive to false
119             deactivates agent
120         """
121
122         self.alive = False
123         self.deactivate()
124
125     def reset(self) -> hint.agent_list:
126         """
127         Reset the agent
128
129         Effects:
130             perform a reset
131         """
132
133         pass
```

C.4.1.3 egg.py

```
import dataclasses as dclass

import source.hint    as hint
import source.keyword as keyword

import source.agents.insect as insect


@dclass.dataclass
class Egg(insect.Insect):
    """
    Class to contain an egg agent
        - inherits from base insect class

    Variables:
        egg_mass:    the egg_mass this egg belongs to
        survival:    survival system
        development: development system

    Methods:
        survive: have the egg survive
        develop: have the egg develop
    """

    egg_mass:    hint.egg_mass
    survival:    hint.egg_survival
    development: hint.egg_development

    def deactivate(self) -> None:
        """
        Deactivate the agent

        Effects:
            adds    agent to   inactive
            removes agent from active   (if there)
        """

```

```
38          super().deactivate()
39          self.egg_mass.remove(self)
40
41      def survive(self) -> hint.agent_list:
42          """
43          Run the survive behavior
44
45          Effects:
46              run behavior to determine if this survives
47
48          Returns:
49              empty list
50          """
51
52          if self.alive:
53              self.survival.survive(self)
54
55          return []
56
57      def develop(self) -> hint.agent_list:
58          """
59          Run the develop behavior
60
61          Effects:
62              run behavior to determine if this develops
63
64          Returns:
65              empty list
66          """
67
68          if self.alive:
69              self.development.develop(self)
70
71          return []
72
73      @classmethod
74      def initialize(cls, unique_id:  str,
75                          simulation: hint.simulation,
76                          location:   hint.location,
```

```
77                      mass:       float,
78                      genotype:   str,
79                      egg_mass:   hint.egg_mass) -> 'Egg':
80          """
81          Initialize a new egg agent
82
83          Args:
84              unique_id:  the agent's unique_id
85              simulation: the master simulation
86              location:   the agent's location
87              mass:       the agent's mass
88              genotype:   the agent's genotype
89              egg_mass:   the egg_mass for egg
90
91          Returns:
92              A fully initialized agent
93          """
94
95          agent_key = keyword.egg
96          alive     = True
97          age       = 0
98          death     = keyword.alive
99
100         survival    = simulation.behaviors.survive_egg
101         development = simulation.behaviors.develop_egg
102
103         return cls(agent_key, unique_id, simulation, location,
104                    alive, mass, genotype, age, death, egg_mass,
105                    survival, development)
```

C.4.1.4 egg_mass.py

```
import dataclasses  as dclass
import collections  as collect
import itertools    as i_tools
import numpy        as np
import numpy.random as rnd

import source.hint    as hint
import source.keyword as keyword

import source.agents.agent as agent
import source.agents.egg   as agent_egg


class Eggs(collect.UserDict):
    """
    Class to handle the eggs for an egg_mass

    Variables:
        - list:
            list of eggs in class

        mass: mass of single egg
    """

    def __init__(self, eggs: hint.egg_dict,
                       mass: float):
        super().__init__(eggs)

        self.mass = mass

    def activate(self) -> None:
        """
        Activate the eggs in list in simulation

        Effects:
            activates all the eggs
        """
```

```
38
39          for egg in self.values():
40              egg.activate()
41
42      def deactivate(self) -> None:
43          """
44          Deactivate the eggs in list in simulation
45
46          Effects:
47              deactivates all the eggs
48          """
49
50          for egg in self.values():
51              egg.deactivate()
52
53      def cannibalize(self, number: int) -> None:
54          """
55          Cannibalize a number of eggs
56
57          Args:
58              number: the number of eggs to cannibalize
59
60          Effects:
61              randomly remove that number of eggs
62          """
63
64          unique_ids = list(self.keys())
65
66          rnd.shuffle(unique_ids)
67
68          for unique_id in unique_ids[:number]:
69              self[unique_id].die(keyword.cannibalism)
70
71      @classmethod
72      def initialize(cls, egg_mass:  hint.egg_mass,
73                          genotypes: hint.genotypes,
74                          mass:      float) -> 'Eggs':
75          """
76          Initialize a collection of eggs for egg_mass
```

```
77
78          Args:
79              egg_mass:   the egg_mass
80              mass:       the mass for each egg
81              genotypes:  the list of egg genotypes
82
83          Returns:
84              a setup collection of eggs
85          """
86
87          eggs = {}
88          for genotype in genotypes:
89              unique_id  = egg_mass.new_unique_id()
90              simulation = egg_mass.simulation
91              location   = egg_mass.location.copy()
92
93              new = agent_egg.Egg.initialize(unique_id,
94                                             simulation,
95                                             location,
96                                             mass,
97                                             genotype,
98                                             egg_mass)
99              eggs[new.unique_id] = new
100
101         return cls(eggs, mass)
102
103
104 @dclass.dataclass
105 class EggMass(agent.Agent):
106     """
107     Base class to for insect agents
108         - inherits from base agent class
109
110     Variables:
111         eggs: list of eggs in egg_mass
112         active: determine if this is active
113     """
114
115     eggs: hint.egg_mass_eggs
```

```
116
117     def __post_init__(self):
118         """Setup some helper systems"""
119
120         self._id_count = i_tools.count()
121
122     @property
123     def active(self) -> bool:
124         """Determine if this egg mass is still active"""
125
126         return (len(self.eggs) > 0) and self.alive
127
128     @property
129     def inactive(self) -> bool:
130         """Determine if this egg mass is not active"""
131
132         return (len(self.eggs) <= 0) and self.alive
133
134     @property
135     def mass(self) -> float:
136         """Get the total mass of egg_mass"""
137
138         return len(self.eggs) * self.eggs.mass
139
140     def activate(self) -> None:
141         """
142         Activate this egg_mass and all its eggs in the simulation
143
144         Effects:
145             activates this agent
146         """
147
148         super().activate()
149         self.eggs.activate()
150         self.alive = True
151
152     def deactivate(self) -> None:
153         """
154         Deactivate this egg_mass and all its eggs in the simulation
```

```
155
156         Effects:
157             deactivates this agent
158         """
159
160         super().deactivate()
161         self.eggs.deactivate()
162         self.alive = False
163
164     def _feed_number(self, amount: float) -> int:
165         """
166         Get the number of eggs to feed on
167
168         Args:
169             amount: the amount of mass to remove
170
171         Returns:
172             the number of eggs to remove
173         """
174
175         raw   = amount/self.eggs.mass
176         ceil  = int(np.ceil( raw))
177         floor = int(np.floor(raw))
178
179         number = len(self.eggs)
180
181         if ceil <= number:
182             return ceil
183         elif floor <= number:
184             return floor
185         else:
186             raise RuntimeError('Tried to consume too much egg')
187
188     def feed(self, amount: float) -> None:
189         """
190         Feed on amount of eggs
191
192         Args:
193             amount: amount to eat
```

```
194
195         Effects:
196             Cannibalizes amount of eggs
197         """
198
199         number = self._feed_number(amount)
200         self.eggs.cannibalize(number)
201
202         if self.inactive:
203             self.deactivate()
204
205     def remove(self, egg: hint.egg) -> None:
206         """
207         Remove the egg from the egg_mass
208
209         Args:
210             egg: the egg to remove
211
212         Effects:
213             removes the egg from the class
214         """
215
216         del self.eggs[egg.unique_id]
217
218     def reset(self) -> hint.agent_list:
219         """
220         Reset the agent
221
222         Effects:
223             reset the agent if needed
224         """
225
226         if self.inactive:
227             self.deactivate()
228
229         return []
230
231     def new_unique_id(self) -> str:
232         """
```

```
233         Create a new unique_id
234
235         Returns:
236             a unique_id
237         """
238
239         return '{}{}'.format(self.unique_id, next(self._id_count))
240
241     @staticmethod
242     def _alleles(genotype: str) -> hint.alleles:
243         """
244         Get the alleles for the given genotype
245
246         Args:
247             genotype: the genotype_key
248
249         Returns:
250             tuple of alleles
251         """
252
253         if genotype == keyword.homo_r:
254             return keyword.homo_r_alleles
255         elif genotype == keyword.hetero:
256             return keyword.hetero_alleles
257         elif genotype == keyword.homo_s:
258             return keyword.homo_s_alleles
259         else:
260             raise RuntimeError('Invalid genotype_key: {}'.format(genotype))
261
262     @staticmethod
263     def _generate_genotype(mother_alleles: hint.alleles,
264                            father_alleles: hint.alleles) -> str:
265         """
266         Generate the genotype
267
268         Args:
269             mother_alleles: the mother's alleles
270             father_alleles: the father's alleles
271
```

```
272         Returns:
273             a new genotype
274         """
275
276         mother = rnd.choice(mother_alleles)
277         father = rnd.choice(father_alleles)
278
279         allele_key = mother + father
280
281         if allele_key == keyword.homo_r_value:
282             return keyword.homo_r
283         elif allele_key == keyword.hetero_value:
284             return keyword.hetero
285         else:
286             return keyword.homo_s
287
288     def genotypes(self, number: int,
289                         mother: str,
290                         father: str) -> hint.genotypes:
291         """
292         Generate a list of genotypes from the mother and father
293
294         Args:
295             number: number to make
296             mother: mother's genotype_key
297             father: father's genotype_key
298
299         Returns:
300             list of genotypes
301         """
302
303         mother_alleles = self._alleles(mother)
304         father_alleles = self._alleles(father)
305
306         genotypes = []
307         for _ in range(number):
308             genotypes.append(self._generate_genotype(mother_alleles,
309                                                      father_alleles))
310
```

```
311         return genotypes
312
313     @classmethod
314     def empty(cls, unique_id:  str,
315                    simulation: hint.simulation,
316                    location:   hint.location) -> 'EggMass':
317         """
318         Initialize a new egg_mass without realized eggs_system
319
320         Args:
321             unique_id:  the agent's unique_id
322             simulation: the master simulation
323             location:   the agent's location
324
325         Returns:
326             An empty egg_mass system
327         """
328
329         agent_key = keyword.egg_mass
330         alive     = True
331
332         eggs = Eggs({}, -1)
333
334         return cls(agent_key, unique_id, simulation, location, alive, eggs)
335
336     @classmethod
337     def initialize(cls, unique_id:  str,
338                         simulation: hint.simulation,
339                         location:   hint.location,
340                         mother:     str,
341                         father:     str) -> 'EggMass':
342         """
343         Initialize a new egg_mass agent
344
345         Args:
346             unique_id:  the agent's unique_id
347             simulation: the master simulation
348             location:   the agent's location
349             mother:     mother's genotype_key
```

```
350             father:     father's genotype_key
351
352         Returns:
353             a fully initialized egg_mass
354         """
355
356         new = cls.empty(unique_id, simulation, location)
357
358         number    = simulation.models[keyword.init_num](mother)
359         mass      = simulation.models[keyword.init_mass](mother)
360         genotypes = new.genotypes(number, mother, father)
361         new.eggs  = Eggs.initialize(new, genotypes, mass)
362
363         return new
364
365     @classmethod
366     def setup(cls, unique_id_num: int,
367                    initial_key:   str,
368                    simulation:    hint.simulation,
369                    genotype:      str) -> 'EggMass':
370         """
371         Setup an initial population egg_mass
372
373         Args:
374             unique_id_num: unique_id number
375             initial_key:   key for where agent was initialized
376             simulation:    the master simulation
377             genotype:      the effective genotypes of eggs
378
379         Returns:
380             a egg_mass initialized by a population
381         """
382
383         if genotype == keyword.hetero:
384             parents = [keyword.homo_r, keyword.homo_s]
385         else:
386             parents = [genotype, genotype]
387         rnd.shuffle(parents)
388
```

```
389         unique_id = '{}{}{}'.format(initial_key,
390                                     unique_id_num,
391                                     keyword.egg_mass)
392         location  = simulation.space.new_location(keyword.egg_depth)
393 
394         return cls.initialize(unique_id, simulation, location,
395                               parents[0], parents[1])
396 
397     @classmethod
398     def birth(cls, adult: hint.adult) -> 'EggMass':
399         """
400         Create an egg_mass from this adult
401 
402         Args:
403             adult: the adult in question
404 
405         Returns:
406             a brand new egg_mass
407         """
408 
409         unique_id  = adult.new_unique_id()
410         simulation = adult.simulation
411         location   = adult.new_egg_location()
412         mother     = adult.genotype
413         father     = adult.mate
414 
415         return cls.initialize(unique_id, simulation, location, mother, father)
```

C.4.1.5 insect.py

```
import dataclasses as dclass
import itertools   as i_tools

import source.hint    as hint
import source.keyword as keyword

import source.agents.agent as agent


@dclass.dataclass
class Insect(agent.Agent):
    """
    Base class to for insect agents
        - inherits from base agent class

    Variables:
        mass:     the agent's mass
        genotype: the agent's genotype
        age:      agent's age
        death:    agent's death state

    Methods:
        advance_age: have the insect advance it's age
    """

    mass:     float
    genotype: str
    age:      int
    death:    str

    def __post_init__(self):
        """Setup some helper systems"""

        self._age_count = i_tools.count(self.age + 1)

    @property
    def bt(self) -> str:
```

```
38          """Get the bt state of the plant"""
39
40          loc = self.location[:keyword.bt_depth]
41
42          return self.simulation.agents[loc.location_key].environment.bt
43
44      @property
45      def plant(self) -> float:
46          """Get the mass of the plant"""
47
48          loc = self.location[:keyword.plant_depth]
49
50          return self.simulation.agents[loc.location_key].environment.plant
51
52      def advance_age(self) -> hint.agent_list:
53          """
54          Advance the age of the agent as a behavior
55
56          Effects:
57              advances age by 1
58
59          Returns:
60              empty list
61          """
62
63          self.age = next(self._age_count)
64
65          return []
66
67      def die(self, death: str = '') -> None:
68          """
69          Have agent die
70
71          Args:
72              death: the type of death
73
74          Effects:
75              sets alive to false
76              sets death to the death
```

```
77              deactivates agent
78          """
79
80          self.death = death
81          super().die()
```

C.4.1.6 larva.py

```
import dataclasses as dclass

import source.hint    as hint
import source.keyword as keyword

import source.agents.insect as insect


@dclass.dataclass
class Larva(insect.Insect):
    """
    Class to contain a larva agent
        - inherits from base insect class

    Variables:
        plant_gut:    amount of plant that the larva has eaten
        egg_gut:      amount of egg   that the larva has eaten
        larva_gut:    amount of larva that the larva has eaten
        full:         boolean if larva is full
        gut:          gut system
        biomass:      biomass system
        survival:     survival system
        development:  development system
        movement:     movement system
        forage_plant: forage a plant system
        forage_egg:   forage an egg  system
        forage_larva: forage a larva system
        loss:         target loss/consume system
        cannibalism:  perform cannibalism system

    Methods:
        add_plant:     add plant to the gut storage
        add_egg:       add egg   to the gut storage
        add_larva:     add larva to the gut storage
        grow:          have larva grow in mass
        survive:       have the larva survive
        develop:       have the larva develop
```

```
        move:           have the larva move
        consume_egg:    consume egg material
        consume_larva:  consume larva material
        targets:        get cannibalism targets
        consume:        have larva consume mass
    """

    plant_gut:    float
    egg_gut:      float
    larva_gut:    float
    full:         bool
    starve:       bool
    gut:          hint.gut
    biomass:      hint.mass
    survival:     hint.larva_survival
    development:  hint.larva_development
    movement:     hint.larva_movement
    forage_plant: hint.plant_forage
    forage_egg:   hint.egg_forage
    forage_larva: hint.larva_forage
    loss:         hint.target_loss
    cannibalism:  hint.cannibalism

    target: hint.target = None

    @property
    def active(self) -> bool:
        """Determine if this larva is still active"""

        return self.mass > 0

    @property
    def _can_consume(self) -> bool:
        """Determine if this larva can consume"""

        return self.alive

    @property
    def _has_target(self) -> bool:
```

```
77          """Determine if this larva has a valid target"""
78
79          return (self.target is not None) and self.target.active
80
81      def add_plant(self, available: float) -> float:
82          """
83          Consume plant from the available amount
84
85          Args:
86              available: the amount available
87
88          Effects:
89              adds amount consumed to plant_gut
90              checks if agent is full
91
92          Returns:
93              the amount consumed
94          """
95
96          amount, full    = self.gut.amount(self, available)
97          self.plant_gut += amount
98
99          if full:
100             self.full = True
101
102         return amount
103
104     def add_egg(self, available: float) -> float:
105         """
106         Consume egg from the available amount
107
108         Args:
109             available: the amount available
110
111         Effects:
112             adds amount consumed to egg_gut
113             checks if agent is full
114
115         Returns:
```

```
116             the amount consumed
117         """
118 
119         amount, full    = self.gut.amount(self, available)
120         self.egg_gut += amount
121 
122         if full:
123             self.full = True
124 
125         return amount
126 
127     def add_larva(self, available: float) -> float:
128         """
129         Consume larva from the available amount
130 
131         Args:
132             available: the amount available
133 
134         Effects:
135             adds amount consumed to larva_gut
136             checks if agent is full
137 
138         Returns:
139             the amount consumed
140         """
141 
142         amount, full    = self.gut.amount(self, available)
143         self.larva_gut += amount
144 
145         if full:
146             self.full = True
147 
148         return amount
149 
150     def grow(self) -> hint.agent_list:
151         """
152         Grow the larva as behavior
153 
154         Effects:
```

```
155             increase the mass of the larva
156
157         Returns:
158             empty list
159         """
160
161         if self.alive:
162             growth = self.biomass.grow(self)
163             if growth < 0:
164                 self.starve = True
165             else:
166                 self.mass += growth
167
168         return []
169
170     def survive(self) -> hint.agent_list:
171         """
172         Run the survive behavior
173
174         Effects:
175             run behavior to determine if this survives
176
177         Returns:
178             empty list
179         """
180
181         if self.alive:
182             self.survival.survive(self)
183
184         return []
185
186     def develop(self) -> hint.agent_list:
187         """
188         Run the develop behavior
189
190         Effects:
191             run behavior to determine if this develops
192
193         Returns:
```

```
194             empty list
195         """
196 
197         if self.alive:
198             self.development.develop(self)
199 
200         return []
201 
202     def move(self) -> hint.agent_list:
203         """
204         Run the move behavior
205 
206         Effects:
207             run behavior to move
208 
209         Returns:
210             empty list
211         """
212 
213         if self._can_consume and (not self._has_target):
214             self.movement.move(self)
215 
216         return []
217 
218     def _consume_plant(self) -> None:
219         """
220         Run consume plant behavior
221 
222         Effects:
223             consume the plant
224         """
225 
226         if self._can_consume:
227             self.forage_plant.consume(self)
228 
229     def consume_egg(self, egg_mass: hint.egg_mass) -> None:
230         """
231         Run consume egg behavior
232 
```

```
        Args:
            egg_mass: the egg_mass to consume

        Effects:
            consume the egg_mass
        """

        self.target = egg_mass
        self.forage_egg.consume(self, egg_mass)

    def consume_larva(self, target: hint.larva) -> None:
        """
        Run consume larva behavior

        Args:
            target: the target to consume

        Effects:
            consume the target
        """

        self.target = target
        self.forage_larva.consume(self, target)

    def _consume_target(self) -> None:
        """
        Run consume behavior on the target

        Effects:
            consumes the target
        """

        if self._has_target and self.alive:
            self.loss.consume(self)

    def _location_keys(self, **kwargs) -> hint.location_keys:
        """
        Get the location_keys for vertices in range

```

```
272         Args:
273             **kwargs: bounds for the range
274 
275         Returns:
276             list of location keys
277         """
278 
279         vertices = self.vertices(**kwargs)
280 
281         location_keys = []
282         for vertex in vertices:
283             loc                         = self.location.copy()
284             loc[keyword.larva_level] = vertex
285             location_keys.append(loc.location_key)
286 
287         return location_keys
288 
289     def targets(self, **kwargs) -> hint.targets:
290         """
291         Get a list of egg_masses within the space bounds given
292 
293         Args:
294             **kwargs: bounds for the range
295 
296         Returns:
297             list of target larvae and eggs
298         """
299 
300         location_keys = self._location_keys(**kwargs)
301 
302         targets = []
303         for location_key in location_keys:
304             agent_bin = self.simulation.agents[location_key]
305             targets  += agent_bin[keyword.egg_mass].agents
306             targets  += agent_bin[keyword.larva].agents
307 
308         targets.remove(self)
309 
310         return targets
```

```
311
312     def consume(self) -> hint.agent_list:
313         """
314         Run the full consume behavior
315
316         Effects:
317             consumes other agents and mass
318
319         Returns:
320             empty list
321         """
322
323         self._consume_target()
324         self.cannibalism.cannibalism(self)
325         self._consume_plant()
326
327         return []
328
329     def reset(self) -> hint.agent_list:
330         """
331         Reset the agent
332
333         Effects:
334             sets gut volumes to zero
335             sets full to false
336         """
337
338         self.plant_gut = 0
339         self.egg_gut   = 0
340         self.larva_gut = 0
341
342         self.full = False
343
344         return []
345
346     @classmethod
347     def initialize(cls, unique_id:  str,
348                         simulation: hint.simulation,
349                         location:   hint.location,
```

```
                    mass:       float,
                    genotype:   str) -> 'Larva':
        """
        Initialize a new larva agent

        Args:
            unique_id:  the agent's unique_id
            simulation: the master simulation
            location:   the agent's location
            mass:       the agent's mass
            genotype:   the agent's genotype

        Returns:
            A fully initialized agent
        """

        agent_key = keyword.larva
        alive     = True
        age       = 0
        death     = keyword.alive

        plant_gut = 0
        egg_gut   = 0
        larva_gut = 0
        full      = False
        starve    = False

        gut          = simulation.behaviors.gut
        biomass      = simulation.behaviors.mass
        survival     = simulation.behaviors.survive_larva
        development  = simulation.behaviors.develop_larva
        movement     = simulation.behaviors.move_larva
        cannibalism  = simulation.behaviors.cannibalism
        forage_plant = simulation.behaviors.forage_plant
        forage_egg   = simulation.behaviors.forage_egg
        forage_larva = simulation.behaviors.forage_larva
        loss         = simulation.behaviors.target

        return cls(agent_key, unique_id, simulation, location, alive,
```

```
                mass, genotype, age, death,
                plant_gut, egg_gut, larva_gut, full, starve,
                gut, biomass, survival, development, movement,
                forage_plant, forage_egg, forage_larva, loss, cannibalism)

    @classmethod
    def setup(cls, unique_id_num: int,
                initial_key:   str,
                simulation:    hint.simulation,
                genotype:      str) -> 'Larva':
        """
        Setup an initial population larva

        Args:
            unique_id_num: unique_id number
            initial_key:   key for where agent was initialized
            simulation:    the master simulation
            genotype:      the agent's genotype

        Returns:
            a larva initialized by a population
        """

        unique_id = '{}{}{}'.format(initial_key,
                                    unique_id_num,
                                    keyword.larva)
        location  = simulation.space.new_location(keyword.larva_depth)
        mass      = simulation.models[keyword.init_juvenile](genotype)

        return cls.initialize(unique_id, simulation, location, mass, genotype)

    @classmethod
    def advance(cls, egg: hint.egg) -> 'Larva':
        """
        Create a larva agent for this egg

        Args:
            egg: the egg in question

```

```
428         Returns:
429             A larva version of this egg
430         """
431 
432         return cls.initialize(egg.unique_id,
433                               egg.simulation,
434                               egg.location,
435                               egg.mass,
436                               egg.genotype)
```

C.4.1.7 pupa.py

```
import dataclasses as dclass

import source.hint    as hint
import source.keyword as keyword

import source.agents.insect as insect


@dclass.dataclass
class Pupa(insect.Insect):
    """
    Class to contain a pupa agent
        - inherits from base insect class

    Variables:
        survival:    survival system
        development: development system

    Methods:
        survive:   have the pupa survive
        develop:   have the pupa develop
    """

    survival:    hint.pupa_survival
    development: hint.pupa_development

    def survive(self) -> hint.agent_list:
        """
        Run the survive behavior

        Effects:
            run behavior to determine if this survives

        Returns:
            empty list
        """

```

```
38          if self.alive:
39              self.survival.survive(self)
40
41          return []
42
43      def develop(self) -> hint.agent_list:
44          """
45          Run the develop behavior
46
47          Effects:
48              run behavior to determine if this develops
49
50          Returns:
51              empty list
52          """
53
54          if self.alive:
55              self.development.develop(self)
56
57          return []
58
59      @classmethod
60      def initialize(cls, unique_id:  str,
61                          simulation: hint.simulation,
62                          location:   hint.location,
63                          mass:       float,
64                          genotype:   str) -> 'Pupa':
65          """
66          Initialize a new pupa agent
67
68          Args:
69              unique_id:  the agent's unique_id
70              simulation: the master simulation
71              location:   the agent's location
72              mass:       the agent's mass
73              genotype:   the agent's genotype
74
75          Returns:
76              A fully initialized agent
```

```
77          """
78
79          agent_key = keyword.pupa
80          alive     = True
81          age       = 0
82          death     = keyword.alive
83
84          survival    = simulation.behaviors.survive_pupa
85          development = simulation.behaviors.develop_pupa
86
87          return cls(agent_key, unique_id, simulation, location,
88                     alive, mass, genotype, age, death,
89                     survival, development)
90
91      @classmethod
92      def setup(cls, unique_id_num: int,
93                     initial_key:   str,
94                     simulation:    hint.simulation,
95                     genotype:      str) -> 'Pupa':
96          """
97          Setup an initial population pupa
98
99          Args:
100             unique_id_num: unique_id number
101             initial_key:   key for where agent was initialized
102             simulation:    the master simulation
103             genotype:      the agent's genotype
104
105         Returns:
106             a pupa initialized by a population
107         """
108
109         unique_id = '{}{}{}'.format(initial_key,
110                                     unique_id_num,
111                                     keyword.pupa)
112         location  = simulation.space.new_location(keyword.pupa_depth)
113         mass      = simulation.models[keyword.init_mature](genotype)
114
115         return cls.initialize(unique_id, simulation, location, mass, genotype)
```

```
116
117     @classmethod
118     def advance(cls, larva: hint.larva) -> 'Pupa':
119         """
120         Create a pupa agent for this larva
121
122         Args:
123             larva: the larva in question
124
125         Returns:
126             A pupa version of this larva
127         """
128
129         location = larva.location[:keyword.pupa_depth]
130
131         return cls.initialize(larva.unique_id,
132                               larva.simulation,
133                               location,
134                               larva.mass,
135                               larva.genotype)
```

C.4.2 biomass

```
FallArmyworm
└── source
    └── biomass
        ├── gut.py
        ├── mass.py
        └── models.py
```

C.4.2.1 gut.py

```
import dataclasses as dclass

import source.hint    as hint
import source.keyword as keyword


@dclass.dataclass
class Gut(object):
    """
    Class handle the gut behavior for larvae:

    Variables:
        max_gut: mathematical function for maximum gut volume

    Methods:
        amount: get the amount of food to eat

    Constructors:
        setup: setup the system from input arguments
    """

    max_gut: hint.max_gut

    @staticmethod
    def _volume(larva: hint.larva) -> float:
        """
        Get the volume of food eaten by a larva

        Args:
            larva: the larva in question

        Returns:
            volume of food larva has eaten
        """

        return larva.plant_gut + larva.egg_gut + larva.larva_gut

```

```
38      def _remaining(self, larva: hint.larva) -> float:
39          """
40          Get the remaining volume of gut for larva
41
42          Args:
43              larva: the larva in question
44
45          Returns:
46              volume of food larva can eat
47          """
48
49          return self.max_gut(larva.mass) - self._volume(larva)
50
51      def amount(self, larva:     hint.larva,
52                     available: float) -> hint.amount_tuple:
53          """
54          Get the amount of food that can be consumed from the available
55
56          Args:
57              larva:     the larva in question
58              available: the available food
59
60          Returns:
61              the amount that can be consumed
62          """
63
64          remaining = self._remaining(larva)
65
66          if available >= remaining:
67              return remaining, True
68          else:
69              return available, False
70
71      @classmethod
72      def setup(cls, **kwargs) -> 'Gut':
73          """
74          Setup the class
75
76          Args:
```

```
            **kwargs: simulation input models

        Returns:
            setup class
        """

        return cls(kwargs[keyword.max_gut])
```

C.4.2.2 mass.py

```
import dataclasses as dclass

import source.hint    as hint
import source.keyword as keyword


@dclass.dataclass
class Mass(object):
    """
    Class to handle the mass increase behavior for larvae:

    Variables:
        max_gut: mathematical function for maximum gut volume
        growth:  mathematical function for growth  of  mass

    Methods:
        grow: get the amount to grow

    Constructors:
        setup: setup the system from input arguments
    """

    max_gut: hint.max_gut
    growth:  hint.growth

    @staticmethod
    def _volume(larva: hint.larva) -> float:
        """
        Get the volume of food eaten by a larva

        Args:
            larva: the larva in question

        Returns:
            volume of food larva has eaten
        """

```

```
38          return larva.plant_gut + larva.egg_gut + larva.larva_gut
39
40      def _ratio(self, larva: hint.larva) -> float:
41          """
42          Get the ratio of gut volume to max_gut
43
44          Args:
45              larva: the larva in question
46
47          Returns:
48              volume/max_gut
49          """
50
51          return self._volume(larva) / self.max_gut(larva.mass)
52
53      def _energy(self, mass:  float,
54                          ratio: float) -> float:
55          """
56          Get the energy adjusted for RK4 shift
57
58          Args:
59              mass:  mass for the energy
60              ratio: gut/max_gut ratio
61
62          Returns:
63              ratio*max_gut(shift mass)
64          """
65
66          return ratio*self.max_gut(mass)
67
68      def _rhs(self, larva: hint.larva,
69                       ratio: float,
70                       shift: float) -> float:
71          """
72          The right hand side for an RK4 approx of the growth equation
73
74          Args:
75              larva: the larva in question
76              ratio: gut/max_gut ratio
```

```
77              shift: amount to shift
78
79          Returns:
80              rhs for RK4
81          """
82
83          mass   = larva.mass + shift
84          energy = self._energy(mass, ratio)
85
86          return self.growth(mass, energy, larva.genotype)
87
88      def grow(self, larva: hint.larva) -> float:
89          """
90          Get the amount the larva grows
91
92          Args:
93              larva: the larva in question
94
95          Returns:
96              the amount the larva grows this step
97          """
98
99          ratio = self._ratio(larva)
100
101         k1 = self._rhs(larva, ratio, 0)
102         k2 = self._rhs(larva, ratio, k1/2)
103         k3 = self._rhs(larva, ratio, k2/2)
104         k4 = self._rhs(larva, ratio, k3)
105
106         return (k1 + k2*2 + k3*2 + k4)/6
107
108     @classmethod
109     def setup(cls, **kwargs) -> 'Mass':
110         """
111         Setup the class
112
113         Args:
114             **kwargs: simulation input models
115
```

```
116         Returns:
117             setup class
118         """
119 
120         return cls(kwargs[keyword.max_gut],
121                    kwargs[keyword.growth])
```

C.4.2.3 models.py

```
import dataclasses as dclass
import numpy       as np
import scipy.stats as stats

import source.hint    as hint
import source.keyword as keyword

import source.simulation.models as models


@dclass.dataclass
class MaxGut(models.Model):
    """
    Class to contain a max_gut model:
        max_gut = mass^(3/4)

    Methods:
        __call__: call the model
    """

    model_key = keyword.max_gut

    def __call__(self, mass: float) -> float:
        """
        Run the mathematical model:
            max_gut = mass^(3/4)

        Args:
            mass: insect mass

        Returns:
            the result of the above equation (max_gut)
        """

        return mass**0.75


```

```
@dclass.dataclass
class Growth(models.Model):
    """
    Class to contain a growth model:
        growth = alpha*energy - beta*mass

    Variables:
        alpha: alpha constant
            dict:
                key:   genotype_key
                value: alpha value
        beta: beta constant
            dict:
                key:   genotype_key
                value: beta value

    Methods:
        __call__: call the model
    """

    model_key = keyword.growth

    alpha: hint.variable
    beta:  hint.variable

    def __call__(self, mass:     float,
                       energy:   float,
                       genotype: str) -> float:
        """
        Run the mathematical model
            growth = alpha*energy - beta*mass

        Args:
            mass:     insect mass
            energy:   insect energy
            genotype: insect genotype

        Returns:
            the result of the above equation (growth)
```

```
77          """
78
79          alpha = self.alpha[genotype]
80          beta  = self.beta[ genotype]
81
82          return alpha*energy - beta*mass
83
84
85  @dclass.dataclass()
86  class InitNum(models.Model):
87      """
88      Class to model initial number of eggs in egg mass:
89          draw from Poisson distribution
90
91      Variables:
92          lam: average number of eggs in egg_mass
93
94      Methods:
95          __call__: call the model
96      """
97
98      model_key = keyword.init_num
99
100     lam: float
101
102     def __call__(self, genotype: str) -> int:
103         """
104         Args:
105             genotype: the genotype of the mother
106
107         Returns:
108             number of eggs
109         """
110
111         return int(stats.poisson.rvs(self.lam))
112
113
114 @dclass.dataclass
115 class InitMass(models.Model):
```

```
    """
    Class to model an egg_mass's total mass
        draw from a normal distribution

    Variables:
        mu:    average mass of an egg_mass
            dict:
                key: genotype key
                value: mu value

        sigma: standard deviation in mass of egg_mass
            dict:
                key: genotype key
                value: sig value

    Methods:
        __call__: call the model
    """

    model_key = keyword.init_mass

    mu:      hint.variable
    sigma:   hint.variable

    def __call__(self, genotype: str) -> float:
        """
        Get the total mass of the egg_mass

        Args:
            genotype: insect genotype

        Returns:
            mass of egg_mass
        """

        mu    = self.mu[genotype]
        sigma = self.sigma[genotype]

        return float(stats.truncnorm.rvs(0, np.inf,
```

```
                                        loc=mu, scale=sigma))


@dclass.dataclass
class InitJuvenile(models.Model):
    """
    Class to contain a initial mass model for larvae:

    Variables:
        mu:    average mass of a larva
            dict:
                key: genotype key
                value: mu value

        sigma: standard deviation in mass of larva
            dict:
                key: genotype key
                value: sig value

    Methods:
        __call__: call the model
    """

    model_key = keyword.init_juvenile

    mu:    hint.variable
    sigma: hint.variable

    def __call__(self, genotype: str) -> float:
        """
        Get the mass of a new larva

        Args:
            genotype: insect genotype

        Returns:
            mass of egg_mass
        """

```

```
194         mu    = self.mu[genotype]
195         sigma = self.sigma[genotype]
196 
197         return float(stats.truncnorm.rvs(0, np.inf,
198                                          loc=mu, scale=sigma))
199 
200 
201 @dclass.dataclass
202 class InitMature(models.Model):
203     """
204     Class to model the mass of a mature insect:
205         - truncated normal between 0 and max
206 
207     Variables:
208         mu:    average mass of a mature insect
209             dict:
210                 key: genotype key
211                 value: mu value
212 
213         sigma: standard deviation in mass of larva
214             dict:
215                 key: genotype key
216                 value: sig value
217 
218     Methods:
219         __call__: call the model
220 
221     Constructors:
222         setup: setup the mathematical model
223     """
224 
225     model_key = keyword.init_mature
226 
227     mu:      hint.variable
228     sigma:   hint.variable
229 
230     def __call__(self, genotype: str) -> float:
231         """
232         Get the mass of a new larva
```

```
233
234         Args:
235             genotype: insect genotype
236
237         Returns:
238             mass of egg_mass
239         """
240
241         mu    = self.mu[genotype]
242         sigma = self.sigma[genotype]
243
244         return float(stats.truncnorm.rvs(0, np.inf,
245                                          loc=mu, scale=sigma))
246
247
248 @dclass.dataclass
249 class InitPlant(models.Model):
250     """
251     Class to model the mass of food in plant:
252         - truncated normal between 0 and max
253
254     Variables:
255         mu: average mass food in plant
256
257         sigma: standard deviation of food in plant
258
259     Methods:
260         __call__: call the model
261     """
262
263     model_key = keyword.init_plant
264
265     mu:      float
266     sigma:   float
267
268     def __call__(self, bt: str) -> float:
269         """
270         Get the mass of food on the plant
271
```

```
272         Args:
273             bt: the bt state of the plant
274 
275         Returns:
276             mass of food in plant
277         """
278 
279         return float(stats.truncnorm.rvs(0, np.inf,
280                                          loc=self.mu, scale=self.sigma))
```

C.4.3 data

```
FallArmyworm
└── source
    └── data
        ├── counter.py
        └── database.py
```

C.4.3.1 counter.py

```
import collections as collect

import pandas as pd

import source.hint as hint


class DataColumn(collect.UserList):
    """
    Class to keep up with data counts over time:

    Variables:
        list:
            index: time-step
            value: count at end of time-step

        attr_value: value of attribute to count

    Methods:
        record: record the value

    Constructors:
        empty: setup the list
    """

    def __init__(self, data: hint.data_list,
                       attr_value: str):
        super().__init__(data)

        self.attr_value = attr_value

    def record(self, count: hint.counter) -> None:
        """
        Record the attribute from count

        Args:
            count: count system
```

```
38
39          Effects:
40              records the current count
41          """
42
43          self.append(count[self.attr_value])
44
45      @classmethod
46      def empty(cls, attr_value: str) -> 'DataColumn':
47          """
48          Create an empty data column
49
50          Args:
51              attr_value: attribute to record
52
53          Returns:
54              an empty data column
55          """
56
57          return cls([], attr_value)
58
59
60  class DataColumns(collect.UserDict):
61      """
62      Class to keep track of all the attribute values overtime for a single count:
63
64      Variables:
65          dict:
66              key:   attr_value
67              value: a data column
68
69          attr: the attribute to be tracked
70
71      Methods:
72          record:   record the value
73          refresh:  reset the columns
74          columns:  create a dict
75              key:   attr_attr_value
76              value: DataColumn
```

```
77
78      Constructors:
79          empty: setup the list
80      """
81
82      def __init__(self, data: hint.data_column_dict,
83                           attr: str):
84          super().__init__(data)
85
86          self.attr = attr
87
88      def record(self, count: hint.counter) -> None:
89          """
90          Record the attribute from count
91
92          Args:
93              count: count system
94
95          Effects:
96              records the current count
97          """
98
99          for column in self.values():
100             column.record(count)
101
102     def refresh(self) -> None:
103         """
104         Reset all the data in columns to zero
105             - part of a data refresh system
106
107         Effects:
108             clear all the data to empty
109         """
110
111         for column in self.values():
112             column.clear()
113
114     def columns(self) -> hint.data_column_dict:
115         """
```

```
116         Create a dictionary of data_columns
117
118         Returns:
119             dictionary of data columns
120         """
121
122         data = {}
123
124         for attr_value, data_column in self.items():
125             key = '{}_{}'.format(self.attr, attr_value)
126             data[key] = data_column
127
128         return data
129
130     @classmethod
131     def empty(cls, attr:       str,
132                   attr_values: hint.attr_values) -> 'DataColumns':
133         """
134         Create an empty data column
135
136         Args:
137             attr:        attribute
138             attr_values: attribute values
139
140         Returns:
141             an empty set of data columns
142         """
143
144         data = {}
145         for attr_value in attr_values:
146             data[attr_value] = DataColumn.empty(attr_value)
147
148         return cls(data, attr)
149
150
151 class BaseCount(collect.UserDict):
152     """
153     Base class for attribute counting
154
```

```
155     Variables:
156         dict:
157             key:   attribute value
158             value: count tracker
159 
160         attr: attribute we are counting
161 
162     Methods:
163         add:     add agent to count
164         sub:     subtract agent from count
165         record:  record the count in the data columns
166         refresh: refresh the stored values in data columns
167 
168         get_data_columns: get the data columns to output
169     """
170 
171     def __init__(self, counts: hint.counts_dict,
172                          attr:   str):
173         super().__init__(counts)
174 
175         self.attr = attr
176 
177     def add(self, agent: hint.agent) -> None:
178         """
179         Adds agent to counter
180 
181         Args:
182             agent: agent to count
183 
184         Effects:
185             add count of attribute
186         """
187 
188         pass
189 
190     def sub(self, agent: hint.agent) -> None:
191         """
192         Subtracts agent from counter
193 
```

```
194         Args:
195             agent: agent to count
196 
197         Effects:
198             remove count of attribute
199         """
200 
201         pass
202 
203     def record(self) -> None:
204         """
205         Record the current count
206 
207         Effect:
208             Records all of the data
209         """
210 
211         pass
212 
213     def refresh(self) -> None:
214         """
215         Refresh the data in the columns
216 
217         Effects:
218             clears the columns
219             adds current count
220         """
221 
222         pass
223 
224     def get_data_columns(self) -> hint.data_column_dict:
225         """
226         Get the data_columns for class
227 
228         Returns:
229             the data columns
230         """
231 
232         pass
```

```
233
234
235 class Count(BaseCount):
236     """
237     Class to keep track of attribute counts
238
239     Variables:
240         dict:
241             key:   attribute value
242             value: count
243
244         attr:    attribute to keep track of
245         removal: boolean to keep count of removal only
246
247         data_columns: data columns for system
248
249     Methods:
250         add:     add agent to count
251         sub:     subtract agent from count
252         record:  record the count in the data columns
253         refresh: refresh the stored values in data columns
254
255     Constructors:
256         setup: create a counter
257     """
258
259     def __init__(self, counts:       hint.count_dict,
260                        attr:         str,
261                        removal:      bool,
262                        data_columns: hint.data_columns):
263         super().__init__(counts, attr)
264
265         self.removal = removal
266
267         self.data_columns = data_columns
268
269     def add(self, agent: hint.agent) -> None:
270         """
271         Adds agent to counter
```

```
272
273         Args:
274             agent: agent to count
275
276         Effects:
277             add count of attribute
278         """
279
280         if not self.removal:
281             value = getattr(agent, self.attr)
282             self[value] += 1
283
284     def sub(self, agent: hint.agent) -> None:
285         """
286         Subtracts agent from counter
287
288         Args:
289             agent: agent to count
290
291         Effects:
292             remove count of attribute
293         """
294
295         value = getattr(agent, self.attr)
296
297         if self.removal:
298             if value in self:
299                 self[value] += 1
300         else:
301             self[value] -= 1
302
303     def _reset(self) -> None:
304         """
305         Reset the counts of the system
306
307         Effects:
308             set counts to zero
309         """
310
```

```
311         for key in self:
312             self[key] = 0
313 
314     def record(self) -> None:
315         """
316         Record the current count
317 
318         Effect:
319             Records all of the data
320         """
321 
322         self.data_columns.record(self)
323 
324         if self.removal:
325             self._reset()
326 
327     def refresh(self) -> None:
328         """
329         Refresh the data in the columns
330 
331         Effects:
332             clears the columns
333             adds current count
334         """
335 
336         self.data_columns.refresh()
337         self.data_columns.record(self)
338 
339     def get_data_columns(self) -> hint.data_column_dict:
340         """
341         Get the data_columns for class
342 
343         Returns:
344             the data columns
345         """
346 
347         return self.data_columns.columns()
348 
349     @classmethod
```

```
350     def empty(cls, attr:    str,
351                    values:  hint.attr_values,
352                    removal: bool) -> 'Count':
353         """
354         Setup an empty counter
355 
356         Args:
357             attr:    attribute to count
358             values:  values for attribute
359             removal: determine if we only count removals
360 
361         Returns:
362             a setup class
363         """
364 
365         counts       = {value: 0 for value in values}
366         data_columns = DataColumns.empty(attr, values)
367 
368         return cls(counts, attr, removal, data_columns)
369 
370 
371 class CountFilter(BaseCount):
372     """
373     Class to allow for filtering of counts
374 
375     Variables:
376         dict:
377             key:   attribute value
378             value: count system
379 
380         attr: attribute we are counting
381 
382     Methods:
383         add:     add agent to count
384         sub:     subtract agent from count
385         record:  record the count in the data columns
386         refresh: refresh the stored values in data columns
387 
388         get_data_columns: get the data columns to output
```

```
389     """
390
391     def add(self, agent: hint.agent) -> None:
392         """
393         Adds agent to counter
394
395         Args:
396             agent: agent to count
397
398         Effects:
399             add count of attribute
400         """
401
402         value = getattr(agent, self.attr)
403         self[value].add(agent)
404
405     def sub(self, agent: hint.agent) -> None:
406         """
407         Subtracts agent from counter
408
409         Args:
410             agent: agent to count
411
412         Effects:
413             remove count of attribute
414         """
415
416         value = getattr(agent, self.attr)
417         self[value].sub(agent)
418
419     def record(self) -> None:
420         """
421         Record the current count
422
423         Effect:
424             Records all of the data
425         """
426
427         for count in self.values():
```

```
428             count.record()
429
430     def refresh(self) -> None:
431         """
432         Refresh the data in the columns
433
434         Effects:
435             clears the columns
436             adds current count
437         """
438
439         for count in self.values():
440             count.refresh()
441
442     def get_data_columns(self) -> hint.data_column_dict:
443         """
444         Get the data_columns for class
445
446         Returns:
447             the data columns
448         """
449
450         data = {}
451
452         for attr_value, count in self.items():
453             key_prefix   = '{}_{}'.format(self.attr, attr_value)
454             data_columns = count.get_data_columns()
455
456             for key_suffix, column in data_columns.items():
457                 key = '{}_{}'.format(key_prefix, key_suffix)
458                 data[key] = column
459
460         return data
461
462     @classmethod
463     def empty(cls, attr:    str,
464                    values:  hint.attr_values,
465                    attrs:   hint.attrs) -> 'CountFilter':
466         """
```

```
467         Setup an empty counter
468
469         Args:
470             attr:    attribute to count
471             values:  values for attribute
472             attrs:   dictionary of the sub filters
473
474         Returns:
475             a setup class
476         """
477
478
479         attr_list = list(attrs.keys())
480         if len(attr_list) == 1:
481             attr_value = attr_list.pop(0)
482             counts = {value: Count.empty(attr_value,
483                                          *attrs[attr_value])
484                       for value in values}
485
486         elif len(attr_list) > 1:
487             attr_value  = attr_list.pop(0)
488             attr_values = attrs[attr_value]
489             new_attrs = {attr_key: attrs[attr_key]
490                          for attr_key in attr_list}
491
492             counts = {value: cls.empty(attr_value, attr_values[0], new_attrs)
493                       for value in values}
494
495         else:
496             raise TypeError('Needs data for sub filter')
497
498         return cls(counts, attr)
499
500
501 class Counts(collect.UserDict):
502     """
503     Class to contain all attribute counts
504
505     Variables:
```

```
506         - dict:
507             key:   attribute
508             value: attribute counter
509 
510     Methods:
511         add:     add agent to count
512         sub:     subtract agent from count
513         record:  record the count in the data columns
514         refresh: refresh the stored values in data columns
515         columns: create the data columns for this set of counts
516 
517         dataframe: create a dataframe for storage
518 
519     Constructors:
520         setup: create a counter
521     """
522 
523     def __init__(self, counts: hint.counter_dict):
524         super().__init__(counts)
525 
526     def add(self, agent: hint.agent) -> None:
527         """
528         Adds agent to counter
529 
530         Args:
531             agent: agent to count
532 
533         Effects:
534             add count of attribute
535         """
536 
537         for counter in self.values():
538             counter.add(agent)
539 
540     def sub(self, agent: hint.agent) -> None:
541         """
542         Subtracts agent from counter
543 
544         Args:
```

```
545             agent: agent to count
546
547         Effects:
548             remove count of attribute
549         """
550
551         for counter in self.values():
552             counter.sub(agent)
553
554     def record(self) -> None:
555         """
556         Record the current counts
557
558         Effect:
559             Records all of the data
560         """
561
562         for counter in self.values():
563             counter.record()
564
565     def refresh(self) -> None:
566         """
567         Refresh all the stored counts
568
569         Effects:
570             starts the stored data over
571         """
572
573         for counter in self.values():
574             counter.refresh()
575
576     def columns(self) -> hint.data_column_dict:
577         """
578         Create a dictionary of all data_columns
579
580         Returns:
581             dictionary of data columns
582         """
583
```

```
584         columns = {}
585         for count in self.values():
586             columns.update(count.get_data_columns())
587 
588         return columns
589 
590     def dataframe(self) -> hint.dataframe:
591         """
592         Create a dataframe of the recorded data
593 
594         Returns:
595             A dataframe of the data
596         """
597 
598         return pd.DataFrame.from_dict(self.columns())
599 
600     def count(self, attr_key: str,
601                     attr:     str,
602                     values:   hint.attr_values,
603                     other:    hint.attr_other) -> None:
604         """
605         Add count to system
606 
607         Args:
608             attr_key: key for the storage
609             attr:     attribute to count
610             values:   values for attribute
611             other:    pass in last bit of information
612 
613         Effects:
614             Adds count to system
615         """
616 
617         if isinstance(other, bool):
618             self[attr_key] = Count.empty(attr, values, other)
619         else:
620             if len(other) > 0:
621                 self[attr_key] = CountFilter.empty(attr, values, other)
622             else:
```

```
623                 self[attr_key] = Count.empty(attr, values, False)
624
625     @classmethod
626     def empty(cls, attrs: hint.attrs_dict) -> 'Counts':
627         """
628         Setup an empty counter
629
630         Args:
631             attrs: attributes to count
632
633         Returns:
634             a setup class
635         """
636
637         new = cls({})
638         for attr_key, attr_filter in attrs.items():
639             new.count(attr_key, *attr_filter)
640
641         return new
```

C.4.3.2 database.py

```
import dataclasses as dclass
import sqlalchemy  as sql

import source.hint as hint


@dclass.dataclass
class Database(object):
    """
    Class to handle saving data

    Variables:
        spacing:   number of steps between saves
        file_name: base name for the save file
        next_dump: next dump step
        prev_dump: last dump step
    """

    spacing:   int
    file_name: str = ':memory:'
    file_path: str = ''
    prev_dump: int = 0
    next_dump: int = 0

    def __post_init__(self):
        if self.next_dump == 0:
            self.next_dump = self.prev_dump + self.spacing

    def sql_file_name(self, simulation: hint.simulation) -> str:
        """
        Create the sql file name

        Args:
            simulation: the master simulation

        Returns:
            a sql filename for a connection
```

```
38          """
39
40          dialect = 'sqlite:///'
41          time    = '{}/{}_to_{}_'.format(self.file_path,
42                                          self.prev_dump,
43                                          simulation.timestep)
44
45          return '{}{}{}'.format(dialect, time, self.file_name)
46
47      def _save(self, simulation: hint.simulation) -> None:
48          """
49          Save the data to a file
50
51          Args:
52              simulation: the master simulation
53
54          Effects:
55              save current data to a file
56          """
57
58          dataframes = simulation.agents.dataframes()
59          file_name  = self.sql_file_name(simulation)
60          engine     = sql.create_engine(file_name)
61
62          for table_name, dataframe in dataframes.items():
63              dataframe.to_sql(table_name, engine)
64
65      def dump(self, simulation: hint.simulation) -> None:
66          """
67          Dump the data to a file
68
69          Args:
70              simulation: the master simulation
71
72          Effects:
73              save the current data and then refresh everything
74          """
75
76          self._save(simulation)
```

```
77          simulation.agents.refresh()
78
79          self.prev_dump = simulation.timestep
80          self.next_dump = self.prev_dump + self.spacing
81
82      def save(self, simulation: hint.simulation) -> None:
83          """
84          Save if correct time
85
86          Args:
87              simulation: the master simulation
88
89          Effects:
90              save the current data and then refresh everything if correct time
91          """
92
93          if simulation.timestep == self.next_dump:
94              self.dump(simulation)
95
96      @classmethod
97      def setup(cls, data_tuple: hint.data_tuple) -> 'Database':
98          """
99          Setup the class
100
101         Args:
102             data_tuple: database arguments
103
104         Returns:
105             a setup class
106         """
107
108         return cls(*data_tuple)
```

C.4.4 development

```
FallArmyworm
└── source
    └── development
        ├── egg.py
        ├── larva.py
        ├── models.py
        └── pupa.py
```

C.4.4.1 egg.py

```
import dataclasses as dclass

import source.hint    as hint
import source.keyword as keyword

import source.agents.larva as larva


@dclass.dataclass
class Egg(object):
    """
    Class to handle egg development behavior

    Variables:
        development: mathematical function for if egg develops

    Methods:
        develop: run the behavior

    Constructors:
        setup: setup class
    """

    development: hint.development_egg = None

    @property
    def _use_development(self) -> bool:
        """Determine if we use the development model"""

        return self.development is not None

    def _develop(self, egg: hint.egg) -> bool:
        """
        Determine if the egg develops

        Args:
            egg: the egg in question
```

```
38
39          Returns:
40              if the egg develops or not
41          """
42
43          if self._use_development:
44              return self.development(egg.mass, egg.age, egg.genotype)
45          else:
46              return False
47
48      @staticmethod
49      def _make_larva(egg: hint.egg) -> None:
50          """
51          Create a larva from the egg
52
53          Args:
54              egg: the egg in question
55
56          Returns:
57              the larva for egg to develop into
58          """
59
60          new = larva.Larva.advance(egg)
61          new.activate()
62
63      def develop(self, egg: hint.egg) -> None:
64          """
65          Run development on the egg
66
67          Args:
68              egg: the egg in question
69
70          Effects:
71              if egg develops, replace it with a larva
72              else do nothing
73          """
74
75          if self._develop(egg):
76              egg.deactivate()
```

```
            self._make_larva(egg)

    @classmethod
    def setup(cls, **kwargs) -> 'Egg':
        """
        Setup the class

        Args:
            **kwargs: simulation input models

        Returns:
            setup class
        """

        if keyword.egg_development in kwargs:
            return cls(kwargs[keyword.egg_development])
        else:
            return cls()
```

C.4.4.2 larva.py

```
import dataclasses as dclass

import source.hint    as hint
import source.keyword as keyword

import source.agents.pupa as pupa


@dclass.dataclass
class Larva(object):
    """
    Class to handle larva development behavior

    Variables:
        development: mathematical function for if larva develops

    Methods:
        develop: run the behavior

    Constructors:
        setup: setup class
    """

    development: hint.development_larva = None

    @property
    def _use_development(self) -> bool:
        """Determine if we use the development model"""

        return self.development is not None

    def _develop(self, larva: hint.larva) -> bool:
        """
        Determine if the larva develops

        Args:
            larva: the larva in question
```

```
38
39          Returns:
40              if the larva develops or not
41          """
42
43          if self._use_development:
44              return self.development(larva.mass, larva.age, larva.genotype)
45          else:
46              return False
47
48      @staticmethod
49      def _make_pupa(larva: hint.larva) -> None:
50          """
51          Create a pupa version of this larva
52
53          Args:
54              larva: the larva in question
55
56          Returns:
57              a created pupa
58          """
59
60          new = pupa.Pupa.advance(larva)
61          new.activate()
62
63      def develop(self, larva: hint.larva) -> None:
64          """
65          Run development on the larva
66
67          Args:
68              larva: the larva in question
69
70          Effects:
71              if larva develops, replace it with a pupa
72              else do nothing
73          """
74
75          if self._develop(larva):
76              larva.deactivate()
```

```
            self._make_pupa(larva)

    @classmethod
    def setup(cls, **kwargs) -> 'Larva':
        """
        Setup the class

        Args:
            **kwargs: simulation input models

        Returns:
            setup class
        """

        if keyword.larva_development in kwargs:
            return cls(kwargs[keyword.larva_development])
        else:
            return cls()
```

C.4.4.3 models.py

```
import dataclasses  as dclass
import numpy.random as rnd
import scipy.stats  as stats

import source.hint    as hint
import source.keyword as keyword

import source.simulation.models as models


@dclass.dataclass
class BaseTime(models.Model):
    """
    Class to contain development model based on time
        USES CDF for Normal Distribution for probability
        Checks if minimum time has been achieved

    Variables:
        mu:      mean time for development
        sigma:   standard deviation in mean time
        minimum: minimum time to wait
    """

    mu:      float
    sigma:   float

    def __call__(self, mass:     float,
                       age:      int,
                       genotype: str) -> bool:
        """
        Determine if an agent develops

        Args:
            mass:     mass of agent
            age:      time agent has existed
            genotype: genotype of the agent

```

```
38          Returns:
39              if egg develops or not
40          """
41
42          return rnd.random() <= stats.norm.cdf(age,
43                                                loc=self.mu, scale=self.sigma)
44
45
46  @dclass.dataclass
47  class Egg(BaseTime):
48      """
49      Class to contain development model for egg
50          USES CDF for Normal Distribution for probability
51          Checks if minimum time has been achieved
52
53      Variables:
54          mu:      mean time for development
55          sigma:   standard deviation in mean time
56          minimum: minimum time to wait
57      """
58
59      model_key = keyword.egg_development
60
61
62  @dclass.dataclass
63  class Pupa(BaseTime):
64      """
65      Class to contain development model for pupa
66          USES CDF for Normal Distribution for probability
67          Checks if minimum time has been achieved
68
69      Variables:
70          mu:      mean time for development
71          sigma:   standard deviation in mean time
72          minimum: minimum time to wait
73      """
74
75      model_key = keyword.pupa_development
76
```

```
77
78 @dclass.dataclass
79 class Larva(models.Model):
80     """
81     Class to contain development model for larva
82         USES CDF for Normal Distribution for probability
83         Checks if minimum time has been achieved
84
85     Variables:
86         mu:      mean mass for development
87         sigma:   standard deviation in mean mass
88         minimum: minimum time to wait
89     """
90
91     model_key = keyword.larva_development
92
93     mu:      hint.variable
94     sigma:   hint.variable
95
96     def __call__(self, mass:     float,
97                        age:      int,
98                        genotype: str) -> bool:
99         """
100        Determine if a larva develops
101
102        Args:
103            mass:     mass of larva
104            age:      time larva has existed
105            genotype: genotype of the larva
106
107        Returns:
108            if egg develops or not
109        """
110
111        mu = self.mu[genotype]
112        sigma = self.sigma[genotype]
113
114
115        return rnd.random() <= stats.norm.cdf(mass, loc=mu, scale=sigma)
```

C.4.4.4 pupa.py

```
import dataclasses as dclass

import source.hint    as hint
import source.keyword as keyword

import source.agents.adult as adult


@dclass.dataclass
class Pupa(object):
    """
    Class to handle pupa development behavior

    Variables:
        development: mathematical function for if pupa develops

    Methods:
        develop: run the behavior

    Constructors:
        setup: setup class
    """

    development: hint.development_pupa = None

    @property
    def _use_development(self) -> bool:
        """Determine if we use the development model"""

        return self.development is not None

    def _develop(self, pupa: hint.pupa) -> bool:
        """
        Determine if the pupa develops

        Args:
            pupa: the pupa in question
```

```
38
39          Returns:
40              if the pupa develops or not
41          """
42
43          if self._use_development:
44              return self.development(pupa.mass, pupa.age, pupa.genotype)
45          else:
46              return False
47
48      @staticmethod
49      def _make_adult(pupa: hint.pupa) -> None:
50          """
51          Create a adult from the pupa
52
53          Args:
54              pupa: the pupa in question
55
56          Returns:
57              the adult for pupa to develop into
58          """
59
60          new = adult.Adult.advance(pupa)
61          new.activate()
62
63      def develop(self, pupa: hint.pupa) -> None:
64          """
65          Run development on the pupa
66
67          Args:
68              pupa: the pupa in question
69
70          Effects:
71              if pupa develops, replace it with a adult
72              else do nothing
73          """
74
75          if self._develop(pupa):
76              pupa.deactivate()
```

```
            self._make_adult(pupa)

    @classmethod
    def setup(cls, **kwargs) -> 'Pupa':
        """
        Setup the class

        Args:
            **kwargs: simulation input models

        Returns:
            setup class
        """

        if keyword.pupa_development in kwargs:
            return cls(kwargs[keyword.pupa_development])
        else:
            return cls()
```

C.4.5 forage

```
FallArmyworm
└── source
    └── forage
        ├── cannibalism.py
        ├── egg.py
        ├── larva.py
        ├── models.py
        ├── plant.py
        └── target.py
```

C.4.5.1 cannibalism.py

```
import dataclasses  as dclass
import numpy.random as rnd

import source.hint    as hint
import source.keyword as keyword


@dclass.dataclass
class Cannibalism(object):
    """
    Class to handle cannibalism behavior:

    Variables:
        fight:     mathematical function for determining fight winner
        encounter: mathematical function for determining if cannibalism occurs
        radius:    mathematical function for determining how far encounters
                                                                        reach

    Methods:
        cannibalism: run the behavior

    Constructors:
        setup: setup class
    """

    fight:     hint.fight     = None
    encounter: hint.encounter = None
    radius:    hint.radius    = None

    @property
    def _use_fight(self) -> bool:
        """Determine if we use the fight model"""

        return self.fight is not None

    def _winner(self, larva:  hint.larva,
                      target: hint.larva) -> bool:
```

```
38          """
39          Determine if this larva is winner of fight with target
40
41          Args:
42              larva: the larva starting fight
43              target: the larva's target
44
45          Returns:
46              if the larva running system is winner
47          """
48
49          return self.fight(larva.mass, target.mass)
50
51      def _fight(self, larva:  hint.larva,
52                     target: hint.larva) -> None:
53          """
54          Run one fight
55
56          Args:
57              larva: the larva starting fight
58              target: the larva's target
59
60          Effects:
61              runs full fight
62          """
63
64          if self._winner(larva, target):
65              larva.consume_larva(target)
66          else:
67              target.consume_larva(larva)
68
69      def _contest(self, larva:  hint.larva,
70                       target: hint.larva) -> None:
71          """
72          Run fight/consume when possible on target larva
73
74          Args:
75              larva: the larva starting fight
76              target: the larva's target
```

```
77
78          Effects:
79              consumes part of the larva
80          """
81
82          if self._use_fight:
83              self._fight(larva, target)
84
85      @property
86      def _use_radius(self) -> bool:
87          """Determine if we use the radius model"""
88
89          return self.radius is not None
90
91      def _bounds(self, larva: hint.larva) -> dict:
92          """
93          Get the bounds on the radius for finding encounters
94
95          Args:
96              larva: the larva in question
97
98          Returns:
99              dict:
100                 {upper: radius
101                  lower: 0}
102         """
103
104         if self._use_radius:
105             radius = self.radius(larva.mass, larva.genotype)
106         else:
107             radius = 0
108
109         return {keyword.upper: radius,
110                 keyword.lower: 0}
111
112     def _targets(self, larva: hint.larva) -> hint.targets:
113         """
114         Get a list of targets for cannibalism
115
```

```
        Args:
            larva: the larva in question

        Returns:
            list of targets
        """

        return larva.targets(**self._bounds(larva))

    @staticmethod
    def _get_target(targets: hint.targets) -> hint.target:
        """
        Get insect to encounter

        Args:
            targets: get list of potential targets

        Returns:
            target to encounter

        Effects:
            remove target from list
        """

        target = rnd.choice(targets)
        targets.remove(target)

        return target

    def _can_encounter(self, larva: hint.larva) -> bool:
        """
        Determine if larva can encounter

        Args:
            larva: the larva in question

        Returns:
            if larva can perform an encounter
        """
```

```
155
156         return (self.encounter is not None) and larva.alive and (not larva.full)
157
158     def _encounter(self, larva: hint.larva,
159                            targets: hint.targets) -> bool:
160         """
161         Determine if an encounter occurs
162
163         Args:
164             larva:   the larva in question
165             targets: get list of potential targets
166
167         Returns:
168             if an encounter occurs
169         """
170
171         if self._can_encounter(larva):
172             return self.encounter(len(targets), larva.mass, larva.genotype)
173         else:
174             return False
175
176     def _cannibalize(self, larva:   hint.larva,
177                              targets: hint.targets) -> None:
178         """
179         Perform cannibalism on target
180
181         Args:
182             larva:   the larva in question
183             targets: get list of potential targets
184
185         Effects:
186             run cannibalism on target
187         """
188
189         target = self._get_target(targets)
190
191         if target.agent_key == keyword.egg_mass:
192             larva.consume_egg(target)
193         else:
```

```
            self._contest(larva, target)

    def _cannibalism(self, larva:    hint.larva,
                             targets: hint.targets) -> bool:
        """
        Run single cannibalism step:

        Args:
            larva:   the larva in question
            targets: get list of potential targets

        Returns:
            if their was an encounter
        """

        if self._encounter(larva, targets):
            self._cannibalize(larva, targets)

            return True
        else:
            return False

    def _run_cannibalism(self, larva: hint.larva) -> None:
        """
        Run cannibalism system

        Args:
            larva: the larva in question

        Effects:
            Run all the cannibalism a larva
        """

        targets = self._targets(larva)

        cannibalism = True
        while cannibalism:
            cannibalism = self._cannibalism(larva, targets)

```

```
    def cannibalism(self, larva: hint.larva) -> None:
        """
        Run full cannibalism encounter

        Args:
            larva: the larva in question

        Effects:
            Runs single cannibalism instance
        """

        if self._can_encounter(larva):
            self._run_cannibalism(larva)

    @classmethod
    def setup(cls, **kwargs) -> 'Cannibalism':
        """
        Setup the class

        Args:
            **kwargs: simulation input models

        Returns:
            setup class
        """

        if keyword.fight in kwargs:
            fight = kwargs[keyword.fight]
        else:
            fight = None

        if keyword.encounter in kwargs:
            encounter = kwargs[keyword.encounter]
        else:
            encounter = None

        if keyword.radius in kwargs:
            radius = kwargs[keyword.radius]
        else:
```

```
            radius = None

        return cls(fight, encounter, radius)
```

C.4.5.2 egg.py

```
import dataclasses as dclass

import source.hint    as hint
import source.keyword as keyword


@dclass.dataclass
class Egg(object):
    """
    Class to handle egg foraging (eating) behavior

    Variables:
        forage: mathematical function for how much can be eaten

    Methods:
        consume: run the behavior

    Constructors:
        setup: setup class
    """

    forage: hint.forage_egg = None

    @property
    def _use_forage(self) -> bool:
        """Determine if we use the forage model"""

        return self.forage is not None

    def _available(self, larva:    hint.larva,
                         egg_mass: hint.egg_mass) -> float:
        """
        Get the amount of mass which can be foraged from a target egg_mass

        Args:
            larva:    the larva foraging
            egg_mass: the egg_mass being eaten
```

```
38
39          Returns:
40              amount of food which can be eaten
41          """
42
43          return self.forage(egg_mass.mass, larva.mass, larva.genotype)
44
45      def _consume(self, larva:    hint.larva,
46                         egg_mass: hint.egg_mass) -> None:
47          """
48          Consume available food
49
50          Args:
51              larva:    the larva foraging
52              egg_mass: the egg_mass being eaten
53
54          Effects:
55              eats the available amount of food as larva
56          """
57
58          available = self._available(larva, egg_mass)
59          amount    = larva.add_egg(available)
60          egg_mass.feed(amount)
61
62      def consume(self, larva:    hint.larva,
63                        egg_mass: hint.egg_mass) -> None:
64          """
65          Run forage/consume when possible on target egg_mass
66
67          Args:
68              larva:    the larva foraging
69              egg_mass: the egg_mass being eaten
70
71          Effects:
72              consumes part of the egg_mass
73          """
74
75          if self._use_forage:
76              self._consume(larva, egg_mass)
```

```
77
78      @classmethod
79      def setup(cls, **kwargs) -> 'Egg':
80          """
81          Setup the class
82
83          Args:
84              **kwargs: simulation input models
85
86          Returns:
87              setup class
88          """
89
90          if keyword.egg_forage in kwargs:
91              return cls(kwargs[keyword.egg_forage])
92          else:
93              return cls()
```

C.4.5.3 larva.py

```
import dataclasses as dclass

import source.hint    as hint
import source.keyword as keyword


@dclass.dataclass
class Larva(object):
    """
    Class to handle larva foraging (eating) behavior

    Variables:
        forage: mathematical function for how much can be eaten

    Methods:
        consume: run the behavior

    Constructors:
        setup: setup class
    """

    forage: hint.forage_larva = None

    @property
    def _use_forage(self) -> bool:
        """Determine if we use the forage model"""

        return self.forage is not None

    def _available(self, larva:  hint.larva,
                         target: hint.larva) -> float:
        """
        Get the amount of mass which can be foraged from a target larva

        Args:
            larva:  the larva foraging
            target: the larva being eaten
```

```
38
39          Returns:
40              amount of food which can be eaten
41          """
42
43          return self.forage(target.mass, larva.mass, larva.genotype)
44
45      def _consume(self, larva:  hint.larva,
46                         target: hint.larva) -> None:
47          """
48          Consume available food
49
50          Args:
51              larva:  the larva foraging
52              target: the larva being eaten
53
54          Effects:
55              eats the available amount of food as larva
56          """
57
58          available    = self._available(larva, target)
59          amount       = larva.add_larva(available)
60          target.mass -= amount
61
62          if target.alive:
63              target.die(keyword.cannibalism)
64
65      def consume(self, larva:  hint.larva,
66                        target: hint.larva) -> None:
67          """
68          Run forage/consume when possible on target larva
69
70          Args:
71              larva:  the larva foraging
72              target: the larva being eaten
73
74          Effects:
75              eats the target larva if foraging
76          """
```

```
77
78          if self._use_forage:
79              self._consume(larva, target)
80
81      @classmethod
82      def setup(cls, **kwargs) -> 'Larva':
83          """
84          Setup the class
85
86          Args:
87              **kwargs: simulation input models
88
89          Returns:
90              setup class
91          """
92
93          if keyword.larva_forage in kwargs:
94              return cls(kwargs[keyword.larva_forage])
95          else:
96              return cls()
```

C.4.5.4 models.py

```
import dataclasses   as dclass
import scipy.stats   as stats
import scipy.special as spcl
import numpy         as np
import numpy.random  as rnd

import source.hint    as hint
import source.keyword as keyword

import source.simulation.models as models


@dclass.dataclass
class PlantBase(models.Model):
    """
    Base class for Plant forage models

    Methods:
        __call__: call the model
    """

    model_key = keyword.plant_forage

    steps: int

    def __call__(self, mass:     float,
                       plant:    float,
                       genotype: str,
                       bt:       str) -> float:
        """
        Call the model

        Args:
            mass:     mass of larva
            plant:    mass of plant
            genotype: larva genotype
            bt:       plant type
```

```
38
39          Returns:
40              biomass which can be foraged
41          """
42
43          pass
44
45
46  @dclass.dataclass
47  class PlantAdLibitum(PlantBase):
48      """
49      Class for larvae consuming leaf ad libitum:
50          - ignores leaf mass present
51          - assumes leaf mass does not change (no recovery model)
52
53          Outputs 5 time maximum amount of food which can be consumed
54
55      Variables:
56          max_gut: the maximum gut model
57
58      Methods:
59          __call__: call the model
60      """
61
62      max_gut: hint.max_gut
63
64      def __call__(self, mass:     float,
65                         plant:    float,
66                         genotype: str,
67                         bt:       str) -> float:
68          """
69          Call the model
70
71          Args:
72              mass:     mass of larva
73              plant:    mass of plant
74              genotype: larva genotype
75              bt:       plant type
76
```

```
        Returns:
            biomass which can be foraged
        """

        return self.max_gut(mass) / self.steps


@dclass.dataclass
class PlantStarve(PlantBase):
    """
    Class for larvae consuming leaf with a normal distribution describing
        starvation

        takes maximum amount of consumed food available and multiplies it by
            a factor
        drawn from a normal distribution

    Variables:
        mu:    mean factor in product
        sigma: standard deviation

    Methods:
        __call__: call the model
    """

    theta:   float
    sigma:   float
    max_gut: hint.max_gut

    def _mu(self, mass: float) -> float:
        """
        Get the mean amount of food that can be consumed

        Args:
            mass: mass of the larva

        Returns:
            mean amount of food
        """
```

```
116
117         factor = self.theta / self.steps
118
119         return factor * self.max_gut(mass)
120
121     def __call__(self, mass:     float,
122                        plant:    float,
123                        genotype: str,
124                        bt:       str) -> float:
125         """
126         Call the model
127
128         Args:
129             mass:     mass of larva
130             plant:    mass of plant
131             genotype: larva genotype
132             bt:       plant type
133
134         Returns:
135             biomass which can be foraged
136         """
137
138         return float(stats.truncnorm.rvs(0, np.inf,
139                                          loc=self._mu(mass), scale=self.sigma))
140
141
142 @dclass.dataclass
143 class Egg(models.Model):
144     """
145     Class for describing the amount of food a larva can eat from an egg_mass
146         amount = factor*mass
147
148     Variables:
149         factor: the scale factor
150
151     Methods:
152         __call__: call the model
153
154     Constructors:
```

```
155         setup: setup the mathematical model
156     """
157 
158     model_key = keyword.egg_forage
159 
160     factor: float
161 
162     def __call__(self, egg_mass: float,
163                        mass:     float,
164                        genotype: str) -> float:
165         """
166         Call the model
167 
168         Args:
169             egg_mass: mass of egg_mass
170             mass:     mass of larva
171             genotype: genotype of consumer
172 
173         Returns:
174             amount of larva to eat
175         """
176 
177         return self.factor*egg_mass
178 
179 
180 @dclass.dataclass
181 class Larva(models.Model):
182     """
183     Class for describing the amount of food a larva can eat from another larva
184 
185         amount = factor*mass
186 
187     Variables:
188         factor: the scale factor
189 
190     Methods:
191         __call__: call the model
192     """
193 
```

```
    model_key = keyword.larva_forage

    factor: float

    def __call__(self, target_mass: float,
                       mass:        float,
                       genotype:    str) -> float:
        """
        Call the model

        Args:
            target_mass: mass of target larva
            mass:        mass of larva
            genotype:    genotype of consumer

        Returns:
            amount of larva to eat
        """

        return self.factor*target_mass


@dclass.dataclass
class Loss(models.Model):
    """
    Class for describing the probability that a larva leaves a target food:

        Fixed probability

    Variables:
        prob: probability of leaving target
    """

    model_key = keyword.loss

    slope: float
    mid:   hint.variable

    max_gut:      hint.max_gut
```

```
    forage_egg:   hint.forage_egg
    forage_larva: hint.forage_larva

    def _diff(self, mass:        float,
                    target_mass: float,
                    genotype:    str,
                    target_key:  str) -> float:
        """
        Find the diff between amount and gut

        Args:
            mass:        mass of larva
            target_mass: mass of target
            genotype:    genotype of larva
            target_key:  type of target

        Returns:
            Food - gut
        """

        gut = self.max_gut(mass)

        if target_key == keyword.egg_mass:
            food = self.forage_egg(target_mass, mass, genotype)
        else:
            food = self.forage_larva(target_mass, mass, genotype)

        return food - gut

    def _prob(self, mass:        float,
                    target_mass: float,
                    genotype:    str,
                    target_key:  str) -> float:
        """
        Find the probability of staying

        Args:
            mass:        mass of larva
            target_mass: mass of target
```

```
            genotype:    genotype of larva
            target_key:  type of target

        Returns:
            probability of staying with food source
        """

        d = self._diff(mass, target_mass, genotype, target_key)
        r = self.slope
        q = self.mid[  target_key]

        return 1 / (q * np.exp(-r*d) + 1)

    def __call__(self, mass:        float,
                       target_mass: float,
                       genotype:    str,
                       target_key:  str) -> bool:
        """
        Call the model

        Args:
            mass:        mass of larva
            target_mass: mass of target
            genotype:    genotype of larva
            target_key:  type of target

        Returns:
            if we leave the target
        """

        return rnd.random() <= self._prob(mass, target_mass,
                                          genotype, target_key)


@dclass.dataclass
class Fight(models.Model):
    """
    Class for describing results of a larva cannibalistic fight

```

```
        Win probability given by the function
            P(d) = 1/(1 + exp(-k*d))
        where d is the mass difference and k is the slope:
            d = m0 - m1
        so if m0 >> m1, p(d)->1
              m0 << m1, p(d)->0

    Variables:
        slope: the steepness of the model's transition

    Methods:
        __call__: call the model
    """

    model_key = keyword.fight

    slope: float

    def prob(self, mass0: float,
                    mass1: float) -> float:
        """
        Evaluate the logistic model for probability

        Args:
            mass0: mass of larva running fight
            mass1: mass of target larva

        Returns:
            result of logistic evaluation
        """

        x = self.slope*(mass0 - mass1)

        # noinspection PyTypeChecker
        return spcl.expit(x)

    def __call__(self, mass0: float,
                        mass1: float) -> bool:
        """
```

```
        Call the mathematical model to make decision

        Args:
            mass0: mass of larva running fight
            mass1: mass of target larva

        Returns:
            if mass0 larva wins
        """

        return rnd.random() <= self.prob(mass0, mass1)


@dclass.dataclass
class Encounter(models.Model):
    """
    Class to contain encounter model for cannibalism

        Probability for an encounter is given via:
            p(n) = 1 - exp(-k*n)
        where n is the number of other individuals and k is the scale factor

    Variables:
        factor: scale factor for encounters

    Methods:
        __call__: call the model
    """

    model_key = keyword.encounter

    factor: float

    def _prob(self, number: int) -> float:
        """
        Get the probability for an encounter

        Args:
            number: number of other individuals
```

```
389
390         Returns:
391             probability of an encounter
392         """
393
394         exp = -self.factor*number
395
396         return 1 - np.exp(exp)
397
398     def __call__(self, number:   int,
399                        mass:     float,
400                        genotype: str) -> bool:
401         """
402         Make an encounter decision
403
404         Args:
405             number:   number of other individuals
406             mass:     mass of consumer
407             genotype: genotype of consumer
408
409         Returns:
410             if an encounter occurs
411         """
412
413         return rnd.random() <= self._prob(number)
414
415
416 @dclass.dataclass
417 class Radius(models.Model):
418     """
419     Class to contain encounter radius model for cannibalism
420
421     Variables:
422         radius: the radius for encounters
423
424     Methods:
425         __call__: call the model
426     """
427
```

```
    model_key = keyword.radius

    radius: float

    def __call__(self, mass:     float,
                       genotype: str) -> float:
        """
        Call the model to get the encounter radius

        Args:
            mass:     mass of larva
            genotype: genotype of larva

        Returns:
            the radius of encounters
        """

        return self.radius
```

C.4.5.5 plant.py

```
import dataclasses as dclass

import source.hint    as hint
import source.keyword as keyword


@dclass.dataclass
class Plant(object):
    """
    Class to handle plant foraging (eating) behavior

    Variables:
        forage: mathematical function for how much can be eaten

    Methods:
        consume: run the behavior

    Constructors:
        setup: setup class
    """

    forage: hint.forage_plant = None

    @property
    def _use_forage(self) -> bool:
        """Determine if we use the forage model"""

        return self.forage is not None

    def _available(self, larva: hint.larva) -> float:
        """
        Get the amount of mass that can be foraged

        Args:
            larva: the larva foraging

        Returns:
```

```
            amount of food available
        """

        return self.forage(larva.mass, larva.plant, larva.genotype, larva.bt)

    def _consume(self, larva: hint.larva) -> None:
        """
        Consume available food

        Args:
            larva: the larva foraging

        Effects:
            eats the available amount of food as plant
        """

        available = self._available(larva)
        larva.add_plant(available)

    def consume(self, larva: hint.larva) -> None:
        """
        Run forage/consume when possible on plant

        Args:
            larva: the larva foraging

        Effects:
            consumes part of leaf
        """

        if self._use_forage:
            self._consume(larva)

    @classmethod
    def setup(cls, **kwargs) -> 'Plant':
        """
        Setup the class

        Args:
```

```
            **kwargs: simulation input models

        Returns:
            setup class
        """

        if keyword.plant_forage in kwargs:
            return cls(kwargs[keyword.plant_forage])
        else:
            return cls()
```

C.4.5.6 target.py

```
import dataclasses as dclass

import source.hint    as hint
import source.keyword as keyword


@dclass.dataclass
class Target(object):
    """
    Class to handle target consumption

    Variables:
        loss: mathematical function for if we lose/keep the target

    Methods:
        consume: run the behavior

    Constructors:
        setup: setup class
    """

    loss: hint.loss = None

    @property
    def _use_loss(self) -> bool:
        """Determine if we use the loss model"""

        return self.loss is not None

    def _keep_target(self, larva:  hint.larva,
                           target: hint.target) -> bool:
        """
        Determine if we loose the target

        Args:
            larva:  the larva in question
            target: the larva's target
```

```
38
39          Returns:
40              if the larva moves away from the target
41          """
42
43          if self._use_loss:
44              return self.loss(larva.mass, target.mass,
45                               larva.genotype, target.agent_key)
46          else:
47              return False
48
49      @staticmethod
50      def _consume_target(larva:  hint.larva,
51                          target: hint.target) -> None:
52          """
53          Consume the target
54
55          Args:
56              larva:  the larva in question
57              target: the larva's target
58
59          Effects:
60              run consume behavior on target
61          """
62
63          if target.agent_key == keyword.egg_mass:
64              larva.consume_egg(target)
65          else:
66              larva.consume_larva(target)
67
68      def consume(self, larva: hint.larva) -> None:
69          """
70          Run a target consume behavior on the larva's target
71
72          Args:
73              larva: the larva in question
74
75          Effects:
76              run target consumption
```

```
        """

        target = larva.target

        if self._keep_target(larva, target):
            self._consume_target(larva, target)
        else:
            larva.target = None

    @classmethod
    def setup(cls, **kwargs) -> 'Target':
        """
        Setup the class

        Args:
            **kwargs: simulation input models

        Returns:
            setup class
        """

        if keyword.loss in kwargs:
            return cls(kwargs[keyword.loss])
        else:
            return cls()
```

C.4.6 migration

```
FallArmyworm
└── source
    └── migration
        ├── emigration.py
        └── immigration.py
```

C.4.6.1 emigration.py

```
import dataclasses  as dclass
import collections  as collect
import numpy.random as rnd
import scipy.stats  as stats

import source.hint    as hint
import source.keyword as keyword

import source.space.location as agent_location


@dclass.dataclass
class Emigration(object):
    """
    Class to handle emigration of agents out of model
        - this is to limit the reproductive complexities in the model

        - Assumes the population should be normally distributed

    Variables:
        mu:          average adult population that is allowed
        sigma:       standard deviation in that population
        agent_keys: agent_keys for the population

    Methods:
        emigration: run emigration
    """

    location = agent_location.Location([0]).location_key

    mu:          float
    sigma:       float
    agent_keys: hint.agent_keys

    def _remove(self, population: int) -> bool:
        """
        Determine if the agent emigrates
```

```
38
39          Args:
40              population: the current population of agents
41
42          Returns:
43              if the agent migrates
44          """
45
46          return rnd.random() <= stats.norm.cdf(population,
47                                                loc=self.mu, scale=self.sigma)
48
49      def _emigrate(self, agent:      hint.agent,
50                          population: int) -> int:
51          """
52          Determine if the adult agent emigrates
53
54          Args:
55              agent:      the agent in question
56              population: the current population
57
58          Effects:
59              emigrates the agent or not
60
61          Returns:
62              new population
63          """
64
65          if self._remove(population):
66              agent.die(keyword.emigrate)
67
68              return population - 1
69          else:
70              return population
71
72      def _agents(self, agents: hint.agents) -> hint.agent_list:
73          """
74          Get the current agents
75
76          Args:
```

```
77              agents: the agents system
78
79          Returns:
80              the list of agents
81          """
82
83          population = []
84          for agent_key in self.agent_keys:
85              population.extend(agents[self.location][agent_key].agents)
86
87          return population
88
89      def emigration(self, agents: hint.agents) -> None:
90          """
91          Emigrate agents out of system
92
93          Args:
94              agents: the space agents system
95
96          Effects:
97              removes a collection of agents from the system
98          """
99
100         population = self._agents(agents)
101         pop        = len(population)
102
103         for agent in population:
104             pop = self._emigrate(agent, pop)
105
106
107 class Emigrations(collect.UserList):
108     """
109     Class to handle all the different emigration systems:
110
111     Variables:
112         - list: of all emigrations
113
114     Methods:
115         emigration: call all of the Emigration classes
```

```
    """

    def __init__(self, emigration_list: hint.emigration_list):
        super().__init__(emigration_list)

    def emigration(self, agents:  hint.agents) -> None:
        """
        Run emigration systems on agents

        Args:
            agents: the agents system

        Returns:
            list of agents to emigrate out of system
        """

        for emigration in self:
            emigration.emigration(agents)

    @classmethod
    def setup(cls, setup_tuples: hint.emigration_tuples) -> 'Emigrations':
        """
        Setup the emigration system

        Args:
            setup_tuples: list of setup arguments

        Returns:
            setup class
        """

        emigrations = []
        for setup in setup_tuples:
            emigrations.append(Emigration(*setup))

        return cls(emigrations)
```

C.4.6.2 immigration.py

```
import dataclasses  as dclass
import collections  as collect
import numpy.random as rnd
import scipy.stats  as stats

import source.hint    as hint
import source.keyword as keyword

import source.agents.adult    as adult
import source.agents.egg_mass as egg_mass
import source.agents.larva    as larva
import source.agents.pupa     as pupa


@dclass.dataclass
class Immigration(object):
    """
    Class to handle immigration of agents into the model
        - this is to prevent extinction of a given genotype
        - this is a constant input
        - the number of immigrants is drawn from a Poisson Distribution

    Variables:
        lam:       the mean number of immigrants each day
        genotype:  the genotype key of the agent to immigrate
        agent_key: type of agent to immigrate

    Methods:
        immigration: run immigration
    """

    lam:      float
    genotype: str
    agent_key: str

    def _number(self) -> int:
        """
```

```
38          Get the number of immigrants
39
40          Returns:
41              the number of immigrants
42          """
43
44          return int(stats.poisson.rvs(self.lam))
45
46      def _immigrate_egg_masses(self, simulation: hint.simulation) -> None:
47          """
48          Create and add immigrant egg_masses
49
50          Args:
51              simulation: the simulation
52
53          Effects:
54              adds the egg_masses
55          """
56
57          number = self._number()
58          for _ in range(number):
59              unique_id = simulation.new_unique_id()
60              new       = egg_mass.EggMass.setup(unique_id,
61                                                 keyword.immigrant,
62                                                 simulation,
63                                                 self.genotype)
64              new.activate()
65
66      def _immigrate_larvae(self, simulation: hint.simulation) -> None:
67          """
68          Create and add immigrant larvae
69
70          Args:
71              simulation: the simulation
72
73          Effects:
74              adds the larvae
75          """
76
```

```
77          number = self._number()
78          for _ in range(number):
79              unique_id = simulation.new_unique_id()
80              new       = larva.Larva.setup(unique_id,
81                                            keyword.immigrant,
82                                            simulation,
83                                            self.genotype)
84              new.activate()
85
86      def _immigrate_pupae(self, simulation: hint.simulation) -> None:
87          """
88          Create and add immigrant pupae
89
90          Args:
91              simulation: the simulation
92
93          Effects:
94              adds the pupae
95          """
96
97          number = self._number()
98          for _ in range(number):
99              unique_id = simulation.new_unique_id()
100             new       = pupa.Pupa.setup(unique_id,
101                                         keyword.immigrant,
102                                         simulation,
103                                         self.genotype)
104             new.activate()
105
106     def _immigrate_adults(self, simulation: hint.simulation) -> None:
107         """
108         Create and add immigrant adults
109
110         Args:
111             simulation: the simulation
112
113         Effects:
114             adds the adults
115         """
```

```
116
117         number = self._number()
118         for _ in range(number):
119             unique_id = simulation.new_unique_id()
120             new       = adult.Adult.setup(unique_id,
121                                           keyword.immigrant,
122                                           simulation,
123                                           self.genotype)
124             new.activate()
125
126     def _immigrate_pregnant(self, simulation: hint.simulation) -> None:
127         """
128         Create and add immigrant pregnant
129
130         Args:
131             simulation: the simulation
132
133         Effects:
134             adds the pregnant
135         """
136
137         if self.genotype == keyword.hetero:
138             parents = [keyword.homo_r, keyword.homo_s]
139         else:
140             parents = [self.genotype, self.genotype]
141
142         number = self._number()
143         for _ in range(number):
144             rnd.shuffle(parents)
145             unique_id = simulation.new_unique_id()
146             new       = adult.Adult.setup(unique_id,
147                                           keyword.immigrant,
148                                           simulation,
149                                           parents[0], parents[1])
150             new.activate()
151
152     def immigration(self, simulation: hint.simulation) -> None:
153         """
154         Create and add the agents to immigrate
```

```
155
156         Args:
157             simulation: the simulation
158
159         Effects:
160             immigrates agents into the agent's feed
161         """
162
163         if self.agent_key == keyword.egg_mass:
164             self._immigrate_egg_masses(simulation)
165         elif self.agent_key == keyword.larva:
166             self._immigrate_larvae(simulation)
167         elif self.agent_key == keyword.pupa:
168             self._immigrate_pupae(simulation)
169         elif self.agent_key == keyword.adult:
170             self._immigrate_adults(simulation)
171         elif self.agent_key == keyword.pregnant:
172             self._immigrate_pregnant(simulation)
173
174
175 class Immigrations(collect.UserList):
176     """
177     Class to handle all the different immigration systems:
178
179     Variables:
180         - list: of all immigrations
181
182     Methods:
183         immigration: call all of the immigration classes
184
185     Constructors:
186         setup: setup the class
187     """
188
189     def __init__(self, immigration_list: hint.immigration_list):
190         super().__init__(immigration_list)
191
192     def immigration(self, simulation: hint.simulation) -> None:
193         """
```

```
        Run immigration systems on agents

        Args:
            simulation: the simulation

        Effects:
            run all forms of immigration
        """

        for immigration in self:
            immigration.immigration(simulation)

    @classmethod
    def setup(cls, setup_tuples: hint.immigration_tuples) -> 'Immigrations':
        """
        Setup the immigration system

        Args:
            setup_tuples: list of setup arguments

        Returns:
            setup class
        """

        immigrations = []
        for setup in setup_tuples:
            immigrations.append(Immigration(*setup))

        return cls(immigrations)
```

C.4.7 movement

```
FallArmyworm
└── source
    └── movement
        ├── adult.py
        ├── larva.py
        └── models.py
```

C.4.7.1 adult.py

```
import dataclasses  as dclass
import numpy.random as rnd

import source.hint    as hint
import source.keyword as keyword


@dclass.dataclass
class Adult(object):
    """
    Class to handle adult movement behavior

    Variables:
        movement: mathematical function for how far adult moves

    Methods:
        move: run the behavior

    Constructors:
        setup: setup class
    """

    movement: hint.movement_adult = None

    @property
    def _use_movement(self) -> bool:
        """Determine if we use the move model"""

        return self.movement is not None

    def _distance(self, adult: hint.adult) -> float:
        """
        Get the distance the adult will move

        Args:
            adult: the adult in question

```

```
        Returns:
            the distance the adult can move
        """

        return self.movement(adult.mass, adult.genotype)

    def _vertex(self, adult: hint.adult) -> int:
        """
        Get the vertex for adult to move to

        Args:
            adult: the adult in question

        Returns:
            the vertex to move to
        """

        distance = self._distance(adult)
        kwargs = {keyword.upper: distance,
                  keyword.lower: distance}
        vertices = adult.vertices(**kwargs)

        return rnd.choice(list(vertices))

    def move(self, adult: hint.adult) -> None:
        """
        Move the adult

        Args:
            adult: the adult in question

        Effects:
            moves the adult in space
        """

        if self._use_movement:
            vertex = self._vertex(adult)
            adult.transfer(vertex, keyword.adult_level)

```

```
    @classmethod
    def setup(cls, **kwargs) -> 'Adult':
        """
        Setup the class

        Args:
            **kwargs: simulation input models

        Returns:
            setup class
        """

        if keyword.adult_movement in kwargs:
            return cls(kwargs[keyword.adult_movement])
        else:
            return cls()
```

C.4.7.2 larva.py

```
import dataclasses  as dclass
import numpy.random as rnd

import source.hint    as hint
import source.keyword as keyword


@dclass.dataclass
class Larva(object):
    """
    Class to handle larva movement behavior

    Variables:
        movement: mathematical function for how far larva moves

    Methods:
        move: run the behavior

    Constructors:
        setup: setup class
    """

    movement: hint.movement_larva = None

    @property
    def _use_movement(self) -> bool:
        """Determine if we use the move model"""

        return self.movement is not None

    def _distance(self, larva: hint.larva) -> float:
        """
        Get the distance the larva will move

        Args:
            larva: the larva in question

```

```
        Returns:
            the distance the larva can move
        """

        return self.movement(larva.mass, larva.genotype)

    def _vertex(self, larva: hint.larva) -> int:
        """
        Get the vertex for larva to move to

        Args:
            larva: the larva in question

        Returns:
            the vertex to move to
        """

        distance = self._distance(larva)
        kwargs = {keyword.upper: distance,
                  keyword.lower: distance}
        vertices = larva.vertices(**kwargs)

        return rnd.choice(list(vertices))

    def move(self, larva: hint.larva) -> None:
        """
        Move the larva

        Args:
            larva: the larva in question

        Effects:
            moves the larva in space
        """

        if self._use_movement:
            vertex = self._vertex(larva)
            larva.transfer(vertex, keyword.larva_level)

```

```
    @classmethod
    def setup(cls, **kwargs) -> 'Larva':
        """
        Setup the class

        Args:
            **kwargs: simulation input models

        Returns:
            setup class
        """

        if keyword.larva_movement in kwargs:
            return cls(kwargs[keyword.larva_movement])
        else:
            return cls()
```

C.4.7.3 models.py

```
import dataclasses as dclass
import scipy.stats as stats

import source.keyword as keyword

import source.simulation.models as models


@dclass.dataclass
class Levy(models.Model):
    """
    Class to contain a model to select a travel distance for Levy flight

    Variables:
        loc:   location of mean
        scale: scale factor
        shape: distribution shape constant

    Methods:
        __call__: call the model

    Constructors:
        setup: setup the model
    """

    scale: float
    shape: float

    def __call__(self, mass:   float,
                       genotype: str) -> float:
        """
        Call the model to get the distance to travel

        Args:
            mass:     mass of agent
            genotype: genotype of agent

```

```
        Returns:
            the distance to travel
        """

        return float(stats.pareto.rvs(self.shape, self.scale))


@dclass.dataclass
class Larva(Levy):
    """
    Class containing distance model for larva movement

    Variables:
        loc:   location of mean
        scale: scale factor
        shape: distribution shape constant

    Methods:
        __call__: call the model
    """

    model_key = keyword.larva_movement


@dclass.dataclass
class Adult(Levy):
    """
    Class containing distance model for adult movement

    Variables:
        loc:   location of mean
        scale: scale factor
        shape: distribution shape constant

    Methods:
        __call__: call the model
    """

    model_key = keyword.adult_movement
```

C.4.8 reproduction

```
FallArmyworm
└── source
    └── reproduction
        ├── lay.py
        ├── mate.py
        └── models.py
```

C.4.8.1 lay.py

```
import dataclasses as dclass

import source.hint    as hint
import source.keyword as keyword

import source.agents.egg_mass as agent_egg_mass


@dclass.dataclass
class Lay(object):
    """
    Class to handle egg laying behavior:

    Variables:
        trials:    the number of times ot attempt to lay
        fecundity: mathematical function for number of egg_masses to lay
        density:   mathematical function for the density of agents in local
                                                                        area

    Methods:
        reset: reset the number of egg_masses to lay
        lay:   run the behavior

    Constructors:
        setup: setup class
    """

    fecundity: hint.fecundity = None
    density:   hint.density   = None

    @property
    def _use_fecundity(self) -> bool:
        """Determine if we have a fecundity model"""

        return self.fecundity is not None

    @property
```

```
    def _use_density(self) -> bool:
        """Determine if we have a density model"""

        return self.density is not None

    def reset(self, adult: hint.adult) -> int:
        """
        Reset the number of egg_masses to lay

        Args:
            adult: the adult in question

        Effects:
            Sets number to number of egg masses
        """

        if self._use_fecundity:
            return self.fecundity(adult.age, adult.mass, adult.genotype)
        else:
            return 0


    def _check_density(self, adult:  hint.adult,
                           number: int) -> bool:
        """
        Check if density is low enough

        Args:
            adult:  the adult in question
            number: the local number of insects

        Returns:
            the result of the model test
        """

        if self._use_density:
            return self.density(number, adult.mass, adult.genotype)
        else:
            return True
```

```
77
78      def _lay_egg_mass(self, adult:  hint.adult,
79                                number: int) -> hint.egg_lay:
80          """
81          Try to lay an egg mass
82
83          Args:
84              adult:  the adult in question
85              number: the local number of insects
86
87          Returns:
88              eggs to lay
89          """
90
91          if self._check_density(adult, number):
92              adult.num_eggs -= 1
93              return [agent_egg_mass.EggMass.birth(adult)], number + 1, False
94          else:
95              return [], number, True
96
97      def lay(self, adult: hint.adult) -> hint.egg_masses:
98          """
99          Run loop to create egg_masses
100
101         Args:
102             adult: the adult in question
103
104         Returns:
105             list of egg_masses
106         """
107
108         number   = adult.population()
109         num_eggs = adult.num_eggs
110         stop_lay = False
111
112         egg_masses = []
113         for num in range(num_eggs):
114             if stop_lay:
115                 break
```

```
            else:
                new, number, stop_lay = self._lay_egg_mass(adult, number)
                egg_masses.extend(new)

        return egg_masses

    @classmethod
    def setup(cls, **kwargs) -> 'Lay':
        """
        Setup the class

        Args:
            **kwargs: simulation input models

        Returns:
            setup class
        """

        if keyword.fecundity in kwargs:
            fecundity = kwargs[keyword.fecundity]
        else:
            fecundity = None

        if keyword.density in kwargs:
            density = kwargs[keyword.density]
        else:
            density = None

        return cls(fecundity, density)
```

C.4.8.2 mate.py

```
import dataclasses  as dclass
import numpy.random as rnd

import source.hint    as hint
import source.keyword as keyword


@dclass.dataclass
class Mate(object):
    """
    Class to handle adult mating:

    Variables:
        mating: mathematical function for the encounter of mates
        radius: mathematical function for the radius mates can be found

    Methods:
        mate: run the behavior

    Constructors:
        setup: setup class
    """

    mating: hint.mating      = None
    radius: hint.mate_radius = None

    @property
    def _use_mating(self) -> bool:
        """Determine if we can use mating"""

        return self.mating is not None

    @property
    def _use_radius(self) -> bool:
        """Determine if we can use radius"""

        return self.radius is not None
```

```
    @staticmethod
    def _mate_with(adult: hint.adult,
                   mate:  hint.adult) -> None:
        """
        Mate with the mate

        Args:
            adult: the adult in question
            mate: the target mate

        Effects:
            mate with the target
        """

        adult.set_mate(mate)
        mate.set_mate(adult)

    def _bounds(self, adult: hint.adult) -> dict:
        """
        Get the bounds on the radius for finding encounters

        Args:
            adult: the adult in question

        Returns:
            dict:
                {upper: radius
                 lower: 0}
        """

        if self._use_radius:
            radius = self.radius(adult.mass, adult.genotype)
        else:
            radius = 0

        return {keyword.upper: radius,
                keyword.lower: 0}

```

```
    def _mates(self, adult: hint.adult) -> hint.mates:
        """
        Get a list of mates for cannibalism

        Args:
            adult: the adult in question

        Returns:
            list of mates
        """

        return adult.mates(**self._bounds(adult))

    def _encounter(self, adult: hint.adult,
                           mates: hint.mates) -> bool:
        """
        Determine if this adult mates

        Args:
            adult: the adult in question
            mates: the list of possible mates

        Returns:
            if this adult mates
        """

        return self.mating(len(mates), adult.mass, adult.genotype)

    def _perform(self, adult: hint.adult) -> None:
        """
        Run a mating ritual

        Args:
            adult: the adult in question

        Effects:
            Perform a mating ritual
        """

```

```
        mates = self._mates(adult)

        if self._encounter(adult, mates):
            mate = rnd.choice(mates)
            self._mate_with(adult, mate)

    def mate(self, adult: hint.adult) -> None:
        """
        Performs a mating system ritual

        Args:
            adult: the adult in question

        Effects:
            Performs mating
        """

        if self._use_mating:
            self._perform(adult)

    @classmethod
    def setup(cls, **kwargs) -> 'Mate':
        """
        Setup the class

        Args:
            **kwargs: simulation input models

        Returns:
            setup class
        """

        if keyword.mating in kwargs:
            mating = kwargs[keyword.mating]
        else:
            mating = None

        if keyword.mate_radius in kwargs:
            radius = kwargs[keyword.mate_radius]
```

```
155         else:
156             radius = None
157 
158         return cls(mating, radius)
```

C.4.8.3 models.py

```
import dataclasses  as dclass
import numpy        as np
import numpy.random as rnd
import scipy.stats  as stats

import source.keyword as keyword

import source.simulation.models as models


@dclass.dataclass
class InitSex(models.Model):
    """
    Class to contain a model for the adult sex:
        P is the probability of being female

    Variables:
        prob: the probability of being female

    Methods:
        __call__: call the model

    Constructors:
        setup: setup the mathematical model
    """

    model_key = keyword.init_sex

    prob: float

    def __call__(self, genotype: str) -> bool:
        """
        Call model to determine if female

        Args:
            genotype: the adult's genotype

```

```
        Returns:
            if this is female
        """

        return rnd.random() <= self.prob


@dclass.dataclass
class Mating(models.Model):
    """
    Class to contain encounter model for mating

        Probability for an encounter is given via:
            p(n) = 1 - exp(-k*n)
        where n is the number of other individuals and k is the scale factor

    Variables:
        factor: scale factor for encounters

    Methods:
        __call__: call the model
    """

    model_key = keyword.mating

    factor: float

    def _prob(self, number: int) -> float:
        """
        Get the probability for an encounter

        Args:
            number: number of other individuals

        Returns:
            probability of an encounter
        """

        exp = -self.factor*number
```

```
77
78          return 1 - np.exp(exp)
79
80      def __call__(self, number:   int,
81                           mass:     float,
82                           genotype: str) -> bool:
83          """
84          Make an encounter decision
85
86          Args:
87              number    number of other individuals
88              mass:     mass of consumer
89              genotype: genotype of consumer
90
91          Returns:
92              if an encounter occurs
93          """
94
95          return rnd.random() <= self._prob(number)
96
97
98  @dclass.dataclass
99  class Radius(models.Model):
100     """
101     Class to contain mate radius model
102
103     Variables:
104         radius: the radius for mates
105
106     Methods:
107         __call__: call the model
108     """
109
110     model_key = keyword.mate_radius
111
112     radius: float
113
114     def __call__(self, mass:     float,
115                          genotype: str) -> float:
```

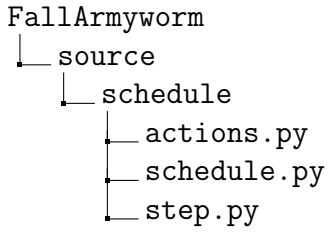
```
        """
        Call the model to get the encounter radius
        Args:
            mass:     mass of adult
            genotype: genotype of adult

        Returns:
            the radius of encounters
        """

        return self.radius


@dclass.dataclass
class Fecundity(models.Model):
    """
    Class to contain fecundity model for adults
        Mean (mu(t)) is given by
            mu(t) = 2*m/(1 + exp(r*t))
        where
            m = maximum
            r = decay rate

        Sample value from Poisson distribution with mu(t) as mean

    Variables:
        maximum: maximum probability
        decay:   decay rate after maximum

    Methods:
        __call__: call the model
    """

    model_key = keyword.fecundity

    maximum: float
    decay:   float

    def _lam(self, time: int) -> float:
```

```
        """
        Get the mean for the distribution

        Args:
            time: time as adult

        Returns:
            mean of Poisson distribution
        """

        return (self.maximum * 2)/(np.exp(self.decay * time) + 1)

    def __call__(self, age:      int,
                       mass:     float,
                       genotype: str) -> int:
        """
        Get the number of egg masses which can be laid

        Args:
            age:      time as adult
            mass:     mass of adult
            genotype: genotype of adult

        Returns:
            Number of egg_masses
        """

        lam = self._lam(age)

        return int(stats.poisson.rvs(lam))


@dclass.dataclass
class Density(models.Model):
    """
    Class to density model for laying eggs:
        Probability that density is low enough:
            P(n) = exp(-(n/a)^b)
        where
```

```
            n = number of eggs and larvae
            a = eta
            b = gamma

    Variables:
        eta:   density scale parameter
        gamma: density exponent

    Methods:
        __call__: call the model
    """

    model_key = keyword.density

    eta:   float
    gamma: float

    def _prob(self, number: int) -> float:
        """
        Get the probability

        Args:
            number: number of eggs/larvae

        Returns:
            the probability
        """

        return np.exp(-((number/self.eta)**self.gamma))

    def __call__(self, number:   int,
                       mass:     float,
                       genotype: str) -> bool:
        """
        Determine if density is low enough

        Args:
            number:   number of eggs and larvae
            mass:     mass of the adult
```

```
233             genotype: genotype of adult
234
235         Returns:
236             if the density is low enough
237         """
238
239         return rnd.random() <= self._prob(number)
```

C.4.9 schedule



C.4.9.1 actions.py

```
import collections as collect
import dataclasses as dclass

import source.hint as hint


@dclass.dataclass
class Action(object):
    """
    Class to handle having agent perform action

    Variables:
        action_key: string for method to perform

    Methods:
        perform: run the action
    """

    action: str

    def perform(self, agent: hint.agent) -> hint.agent_list:
        """
        Perform the action on the agent

        Args:
            agent: agent to perform action

        Returns:
            a list of agents to add to simulation
        """

        return getattr(agent, self.action)()


class Actions(collect.UserList):
    """
    Class to handle performing a list of actions with agent
```

```
38
39      Variables:
40          dict:
41              index: index of action
42              value: action to perform
43
44          agent_key: key for agent that will do the actions
45
46      Methods:
47          perform: run the actions
48
49      Constructors:
50          setup: setup the actions
51      """
52
53      def __init__(self, actions: hint.action_list,
54                           agent_key: str):
55          super().__init__(actions)
56
57          self.agent_key = agent_key
58
59      def perform(self, agent: hint.agent) -> hint.agent_list:
60          """
61          Perform the all the actions on the agent
62
63          Args:
64              agent: agent to perform the actions
65
66          Returns:
67              a list of agents to add to simulation
68          """
69
70          results = []
71          for action in self:
72              results += action.perform(agent)
73
74          return results
75
76      @classmethod
```

```
    def setup(cls, agent_key: str,
                  actions:   hint.action_keys) -> 'Actions':
        """
        Setup the agent actions

        Args:
            agent_key: key for type of agent
            actions:   list of action keys in order of performance

        Returns:
            A setup action class
        """

        perform = []
        for action in actions:
            perform.append(Action(action))

        return cls(perform, agent_key)
```

C.4.9.2 schedule.py

```
import collections as collect

import source.hint as hint

import source.schedule.step as agent_step


class Schedule(collect.UserList):
    """
    Class to contain a complete schedule of steps for single simulation step

    Variables:
        - list:
            list in order of the steps needed
    """

    def __init__(self, steps: hint.steps):
        super().__init__(steps)

    def _perform(self, space:  hint.space,
                        agents: hint.agents) -> hint.agent_list:
        """
        Perform complete schedule

        Args:
            space:  the space system
            agents: the agent storage system

        Returns:
            list of agents to add in
        """

        results = []
        for step in self:
            results += step.perform(space, agents)

        return results
```

```
38
39      @staticmethod
40      def _activate(results: hint.agent_list) -> None:
41          """
42          Activate all of the result agents
43
44          Args:
45              results: the agents to activate
46
47          Effects:
48              activates all of the agents
49          """
50
51          for agent in results:
52              agent.activate()
53
54      def perform(self, space: hint.space,
55                      agents: hint.agents) -> None:
56          """
57          Perform complete schedule
58
59          Args:
60              space:  the space system
61              agents: the agent storage system
62
63          Effects:
64              run a new step
65              add the new agents
66          """
67
68          results = self._perform(space, agents)
69          self._activate(results)
70
71      @classmethod
72      def setup(cls, step_tuples: hint.step_tuples) -> 'Schedule':
73          """
74          Create a schedule of steps
75
76          Args:
```

```
            step_tuples: list in order of the steps to schedule

        Returns:
            a setup schedule
        """

        steps = []
        for step_tuple in step_tuples:
            new_step = agent_step.Step.setup(*step_tuple)
            steps.append(new_step)

        return cls(steps)
```

C.4.9.3 step.py

```
import collections  as collect
import itertools    as i_tools
import numpy.random as rnd

import joblib          as para
import multiprocessing as multi

import source.hint as hint

import source.schedule.actions as agent_actions


num_cpu = multi.cpu_count()
# num_cpu = 16


class Step(collect.UserList):
    """
    Class to contain a single step on agents:

    Variables:
        - list: list of actions to perform

        number:          number of times to repeat the actions in list
        shuffle_agents:  if we shuffle the agents
        shuffle_actions: if we can shuffle the actions in list
        parallel_reg:    if we perform actions in parallel by agents
        parallel_loc:    if we perform actions in parallel via locations
        level:           level we group agents by
    """

    def __init__(self, actions:         hint.actions_list,
                       number:          int  = 1,
                       shuffle_agents:  bool = False,
                       shuffle_actions: bool = False,
                       parallel_reg:    bool = False,
                       parallel_loc:    bool = False,
```

```
                        level:           int  = 0):
        super().__init__(actions)

        self.number = number

        self.shuffle_agents  = shuffle_agents
        self.shuffle_actions = shuffle_actions

        self.parallel_reg = parallel_reg
        self.parallel_loc = parallel_loc

        self.level = level

    @staticmethod
    def _perform_agent_action_regular(action: hint.actions,
                                      agents: hint.agent_list) \
            -> hint.agent_list:
        """
        Perform the specific action on the agents

        Args:
            action: action to perform
            agents: list of agents to use

        Returns:
            list of agents to add in
        """

        results = []
        for agent in agents:
            results += action.perform(agent)

        return results

    def _perform_agent_action_parallel(self, action: hint.actions,
                                           agents: hint.agent_list) \
            -> hint.agent_list:
        """
        Perform the specific action on the agents in parallel by agent
```

```
77
78          Args:
79              action: action to perform
80              agents: list of agents to use
81
82          Returns:
83              list of agents to add in
84          """
85
86          def step(agent: hint.agent_list) -> hint.agent_list:
87              """
88              Create a loop function to parallelize actions
89
90              Args:
91                  agent: sub_list of agents
92
93              Returns:
94                  list of agents to add in
95              """
96
97              return self._perform_agent_action_regular(action, agent)
98
99          n      = num_cpu
100         splits = [agents[i::n] for i in range(n)]
101         # values = para.Parallel(n_jobs=n, require='sharedmem')(
102         values = para.Parallel(n_jobs=n, prefer='threads')(
103                 para.delayed(step)(ags) for ags in splits)
104
105         return list(i_tools.chain.from_iterable(values))
106
107     def _perform_agent_action(self, action:    hint.actions,
108                                   agent_bin: hint.agents_bin) \
109             -> hint.agent_list:
110         """
111         Perform the specific action on the agent_bin
112
113         Args:
114             action:    action to perform
115             agent_bin: agent bin to use
```

```
116
117         Returns:
118             list of agents to add in
119         """
120
121         agents: hint.agent_list = agent_bin[action.agent_key].agents.copy()
122
123         if self.shuffle_agents:
124             rnd.shuffle(agents)
125
126         if self.parallel_reg:
127             return self._perform_agent_action_parallel(action, agents)
128         else:
129             return self._perform_agent_action_regular( action, agents)
130
131     def _perform_actions_step(self, location_key: hint.location_key,
132                                     agents:       hint.agents) \
133             -> hint.agent_list:
134         """
135         Perform all the actions at the specific location
136
137         Args:
138             location_key: the location to do step
139             agents:       the agent storage system
140
141         Returns:
142             list of agents to add in
143         """
144
145         agent_bin = agents[location_key]
146
147         results = []
148         for action in self:
149             results += self._perform_agent_action(action, agent_bin)
150
151         return results
152
153     def _perform_regular_step(self, location_keys: hint.location_keys,
154                                     agents:        hint.agents) \
```

```
            -> hint.agent_list:
        """
        Perform a single step on the agents divided by each location_key

        Args:
            location_keys: the list of location keys
            agents:        the agent storage system

        Returns:
            list of agents to add in
        """

        results = []
        for location_key in location_keys:
            results += self._perform_actions_step(location_key, agents)

        return results

    def _perform_parallel_step(self, location_keys: hint.location_keys,
                                     agents:        hint.agents) \
            -> hint.agent_list:
        """
        Perform a single step on the agents if in parallel

        Args:
            location_keys: the list of location keys
            agents:        the agent storage system

        Returns:
            list of agents to add in
        """

        def step(keys: hint.location_keys) -> hint.agent_list:
            """
            Create a loop function to parallelize actions

            Args:
                keys: sub_list of location keys

```

```
            Returns:
                list of agents to add in
            """

            return self._perform_regular_step(keys, agents)

        n      = num_cpu
        splits = [location_keys[i::n].copy() for i in range(n)]
        values = para.Parallel(n_jobs=n, require='sharedmem')(
            para.delayed(step)(loc_keys) for loc_keys in splits)

        return list(i_tools.chain.from_iterable(values))

    def _perform_step(self, space:  hint.space,
                            agents: hint.agents) -> hint.agent_list:
        """
        Perform a single repeat of actions

        Args:
            space:  the space system
            agents: the agent storage system

        Returns:
            list of agents to add in
        """

        if self.shuffle_actions:
            rnd.shuffle(self)

        location_keys = space.location_keys[self.level]

        if self.parallel_loc:
            return self._perform_parallel_step(location_keys, agents)
        else:
            return self._perform_regular_step(location_keys, agents)

    def perform(self, space:  hint.space,
                      agents: hint.agents) -> hint.agent_list:
        """
```

```
        Perform all the steps on the agents

        Args:
            space:  the space system
            agents: the agent storage system

        Returns:
            list of agents to add in
        """

        results = []
        for _ in range(self.number):
            results += self._perform_step(space, agents)

        return results

    @classmethod
    def setup(cls, actions:         hint.actions_dict,
                   number:          int  = 1,
                   shuffle_agents:  bool = False,
                   shuffle_actions: bool = False,
                   parallel_reg:    bool = False,
                   parallel_loc:    bool = False,
                   level:           int  = 0) -> 'Step':
        """
        Setup the entire step

        Args:
            actions:         dict of actions for each agent type
            number:          number of steps to perform at once
            shuffle_agents:  if we shuffle agents
            shuffle_actions: if the actions can be shuffled
            parallel_reg:    if we parallelize on agents
            parallel_loc:    if we parallelize on locations
            level:           locations to split across

        Returns:
            A setup simulation step
        """
```

```
272
273
274         if parallel_loc and parallel_reg:
275             raise TypeError('Cannot have both location and regular parallel')
276
277         actions_list = []
278         for agent_key, action_keys in actions.items():
279             new_action = agent_actions.Actions.setup(agent_key, action_keys)
280             actions_list.append(new_action)
281
282         return cls(actions_list, number, shuffle_agents, shuffle_actions,
283                    parallel_reg, parallel_loc, level)
```

C.4.10 simulation

```
FallArmyworm
└── source
    └── simulation
        ├── behaviors.py
        ├── models.py
        └── simulation.py
```

C.4.10.1 behaviors.py

```
import dataclasses as dclass

import source.hint as hint

import source.biomass.gut  as agent_gut
import source.biomass.mass as agent_mass

import source.development.egg   as agent_develop_egg
import source.development.larva as agent_develop_larva
import source.development.pupa  as agent_develop_pupa

import source.forage.cannibalism as agent_cannibalism
import source.forage.egg         as agent_forage_egg
import source.forage.larva       as agent_forage_larva
import source.forage.plant       as agent_forage_plant
import source.forage.target      as agent_target

import source.movement.adult as agent_move_adult
import source.movement.larva as agent_move_larva

import source.reproduction.lay  as agent_lay
import source.reproduction.mate as agent_mate

import source.survival.adult as agent_survive_adult
import source.survival.egg   as agent_survive_egg
import source.survival.larva as agent_survive_larva
import source.survival.pupa  as agent_survive_pupa


@dclass.dataclass
class Behaviors(object):
    """
    Class to handle all of the behavior actions:

    Variables:
        gut:  gut  behavior
        mass: mass behavior
```

```
38
39      develop_egg:   egg   development behavior
40      develop_larva: larva development behavior
41      develop_pupa:   pupa development behavior
42
43      cannibalism:  cannibalism      behavior
44      forage_egg:   egg    consuming behavior
45      forage_larva: larva  consuming behavior
46      forage_plant: plant  consuming behavior
47      target:       target consuming behavior
48
49      move_adult: adult movement behavior
50      move_larva: larva movement behavior
51
52      lay:  egg   laying behavior
53      mate: adult mating behavior
54
55      survive_adult: adult survival behavior
56      survive_egg:   egg   survival behavior
57      survive_larva: larva survival behavior
58      survive_pupa:  pupa  survival behavior
59  """
60
61  gut:  hint.gut  = None
62  mass: hint.mass = None
63
64  develop_egg:   hint.egg_development   = None
65  develop_larva: hint.larva_development = None
66  develop_pupa:  hint.pupa_development  = None
67
68  cannibalism:  hint.cannibalism  = None
69  forage_egg:   hint.egg_forage   = None
70  forage_larva: hint.larva_forage = None
71  forage_plant: hint.plant_forage = None
72  target:       hint.target_loss  = None
73
74  move_adult: hint.adult_movement = None
75  move_larva: hint.larva_movement = None
76
```

```
    lay:  hint.lay  = None
    mate: hint.mate = None

    survive_adult: hint.adult_survival = None
    survive_egg:   hint.egg_survival   = None
    survive_larva: hint.larva_survival = None
    survive_pupa:  hint.pupa_survival  = None

    def make_biomass(self, **kwargs) -> None:
        """
        Create the biomass behaviors

        Args:
            **kwargs: input mathematical models

        Effects:
            create the biomass behaviors if needed
        """

        if self.gut is None:
            self.gut = agent_gut.Gut.setup(**kwargs)

        if self.mass is None:
            self.mass = agent_mass.Mass.setup(**kwargs)

    def make_development(self, **kwargs) -> None:
        """
        Create the development behaviors

        Args:
            **kwargs: input mathematical models

        Effects:
            create the development behaviors if needed
        """

        if self.develop_egg is None:
            self.develop_egg = agent_develop_egg.Egg.setup(**kwargs)

```

```
116         if self.develop_larva is None:
117             self.develop_larva = agent_develop_larva.Larva.setup(**kwargs)
118 
119         if self.develop_pupa is None:
120             self.develop_pupa = agent_develop_pupa.Pupa.setup(**kwargs)
121 
122     def make_forage(self, **kwargs) -> None:
123         """
124         Create the forage behaviors
125 
126         Args:
127             **kwargs: input mathematical models
128 
129         Effects:
130             create the forage behaviors if needed
131         """
132 
133         if self.cannibalism is None:
134             self.cannibalism = agent_cannibalism.Cannibalism.setup(**kwargs)
135 
136         if self.forage_egg is None:
137             self.forage_egg = agent_forage_egg.Egg.setup(**kwargs)
138 
139         if self.forage_larva is None:
140             self.forage_larva = agent_forage_larva.Larva.setup(**kwargs)
141 
142         if self.forage_plant is None:
143             self.forage_plant = agent_forage_plant.Plant.setup(**kwargs)
144 
145         if self.target is None:
146             self.target = agent_target.Target.setup(**kwargs)
147 
148     def make_movement(self, **kwargs) -> None:
149         """
150         Create the movement behaviors
151 
152         Args:
153             **kwargs: input mathematical models
154 
```

```
        Effects:
            create the movement behaviors if needed
        """

        if self.move_adult is None:
            self.move_adult = agent_move_adult.Adult.setup(**kwargs)

        if self.move_larva is None:
            self.move_larva = agent_move_larva.Larva.setup(**kwargs)

    def make_reproduction(self, **kwargs) -> None:
        """
        Create the reproduction behaviors

        Args:
            **kwargs: input mathematical models

        Effects:
            create the reproduction behaviors if needed
        """

        if self.lay is None:
            self.lay = agent_lay.Lay.setup(**kwargs)

        if self.mate is None:
            self.mate = agent_mate.Mate.setup(**kwargs)

    def make_survival(self, **kwargs) -> None:
        """
        Create the survival behaviors

        Args:
            **kwargs: input mathematical models

        Effects:
            create the survival behaviors if needed
        """

        if self.survive_adult is None:
```

```
            self.survive_adult = agent_survive_adult.Adult.setup(**kwargs)

        if self.survive_egg is None:
            self.survive_egg = agent_survive_egg.Egg.setup(**kwargs)

        if self.survive_larva is None:
            self.survive_larva = agent_survive_larva.Larva.setup(**kwargs)

        if self.survive_pupa is None:
            self.survive_pupa = agent_survive_pupa.Pupa.setup(**kwargs)

    @classmethod
    def setup(cls, **kwargs) -> 'Behaviors':
        """
        Setup all of the behaviors

        Args:
            **kwargs: input mathematical models

        Returns:
            setup class with all behaviors
        """

        new = cls()

        new.make_biomass(     **kwargs)
        new.make_development( **kwargs)
        new.make_forage(      **kwargs)
        new.make_movement(    **kwargs)
        new.make_reproduction(**kwargs)
        new.make_survival(    **kwargs)

        return new
```

C.4.10.2 models.py

```
import dataclasses as dclass
import collections as collect

import source.hint    as hint
import source.keyword as keyword


@dclass.dataclass
class Model(object):
    """
    Base class mathematical input models:

        model_key: is the keyword for the model to be stored under

    Methods:
        __call__: call the model
    """

    model_key = None

    def __call__(self, *args, **kwargs):
        """
        Call the model

        Args:
            *args:    input args
            **kwargs: input kwargs

        Returns:
            result of model
        """

        pass


class Models(collect.UserDict):
    """
```

```
38      Class to handle the input mathematical models
39
40      Variables:
41          - dict:
42              key: model_key
43              value: mathematical model
44
45      Methods:
46          add_model:    add model
47          add_variable: add a variable
48          check_inputs: check that all the requirements are in place
49
50      Constructors:
51          setup: setup the model from data
52      """
53
54      def add_model(self, model: hint.model) -> None:
55          """
56          Add the model to the system
57
58          Args:
59              model: the mathematical model
60
61          Effects:
62              Add model to system
63          """
64
65          if model.model_key not in self:
66              self[model.model_key] = model
67          else:
68              raise TypeError('Input data clash: {}'.format(model.model_key))
69
70      def add_variable(self, variable_key: str,
71                               variable) -> None:
72          """
73          Add the fixed variable under the keyword
74          Args:
75              variable_key: keyword for variable
76              variable:     variable value
```

```
77
78          Effects:
79              Add variable to system
80          """
81
82          if variable_key not in self:
83              self[variable_key] = variable
84          else:
85              raise TypeError('Input data clash: {}'.format(variable_key))
86
87      def check_inputs(self) -> None:
88          """
89          Run check on the inputs
90
91          Raises:
92              TypeError for any input not in system
93          """
94
95          for input_key in keyword.required_inputs:
96              if input_key not in self:
97                  raise TypeError('Required input, {}, not given'.
98                                  format(input_key))
99
100     @classmethod
101     def setup(cls, *args, **kwargs) -> 'Models':
102         """
103         Add all the input data to the system
104
105         Args:
106             *args:    the input mathematical models
107             **kwargs: the input variables
108
109         Returns:
110             setup class
111         """
112
113         new = cls()
114         for model in args:
115             new.add_model(model)
```

```
116
117         for variable_key, variable in kwargs.items():
118             new.add_variable(variable_key, variable)
119
120         new.check_inputs()
121
122         return new
```

C.4.10.3 simulation.py

```
import dataclasses  as dclass
import itertools    as i_tools
import numpy.random as rnd
import pickle       as pk

import source.hint    as hint
import source.keyword as keyword

import source.agents.adult    as adult
import source.agents.egg_mass as egg_mass
import source.agents.larva    as larva
import source.agents.pupa     as pupa

import source.data.database as main_database

import source.migration.emigration  as main_emigration
import source.migration.immigration as main_immigration

import source.schedule.schedule as main_schedule

import source.simulation.behaviors as main_behaviors
import source.simulation.models    as main_models

import source.space.agents as main_agents
import source.space.space  as main_space


@dclass.dataclass
class Simulation(object):
    """
    Class to contain the whole simulation:
    """

    space:       hint.space
    agents:      hint.agents
    schedule:    hint.schedule
    models:      hint.models
```

```
    behaviors:   hint.behaviors
    database:    hint.database
    emigration:  hint.emigrations
    immigration: hint.immigrations

    timestep: int = 0

    def __post_init__(self):
        """Setup some helper systems"""

        self._id_count   = i_tools.count()

    def count_step(self) -> int:
        """
        Count a step

        Returns:
            the step count
        """

        self.timestep += 1

        return self.timestep

    def new_unique_id(self) -> int:
        """
        Generate a new unique_id

        Returns:
            a new unique_id
        """

        return next(self._id_count)

    def populate_egg_masses(self, nums: hint.init_pop) -> None:
        """
        Create initial egg_masses

        Args:
```

```
77              nums: (num_homo_r, num_hetero, num_homo_s)
78
79          Effects:
80              adds egg_masses of the different amounts to the simulation
81          """
82
83          for index, genotype in enumerate(keyword.genotype_keys):
84              for _ in range(nums[index]):
85                  unique_id = self.new_unique_id()
86                  new       = egg_mass.EggMass.setup(unique_id,
87                                                     keyword.init,
88                                                     self,
89                                                     genotype)
90                  new.activate()
91
92      def populate_larvae(self, nums: hint.init_pop) -> None:
93          """
94          Create the initial larvae
95
96          Args:
97              nums: (num_homo_r, num_hetero, num_homo_s)
98
99          Effects:
100             adds larvae of the different amounts to the simulation
101         """
102
103         for index, genotype in enumerate(keyword.genotype_keys):
104             for _ in range(nums[index]):
105                 unique_id = self.new_unique_id()
106                 new       = larva.Larva.setup(unique_id,
107                                               keyword.init,
108                                               self,
109                                               genotype)
110                 new.activate()
111
112     def populate_pupae(self, nums: hint.init_pop) -> None:
113         """
114         Create the initial pupae
115
```

```
116         Args:
117             nums: (num_homo_r, num_hetero, num_homo_s)
118 
119         Effects:
120             adds pupae of the different amounts to the simulation
121         """
122 
123         for index, genotype in enumerate(keyword.genotype_keys):
124             for _ in range(nums[index]):
125                 unique_id = self.new_unique_id()
126                 new       = pupa.Pupa.setup(unique_id,
127                                             keyword.init,
128                                             self,
129                                             genotype)
130                 new.activate()
131 
132     def populate_adults(self, nums: hint.init_pop) -> None:
133         """
134         Create the initial adults
135 
136         Args:
137             nums: (num_homo_r, num_hetero, num_homo_s)
138 
139         Effects:
140             adds adults of the different amounts to the simulation
141         """
142 
143         for index, genotype in enumerate(keyword.genotype_keys):
144             for _ in range(nums[index]):
145                 unique_id = self.new_unique_id()
146                 new       = adult.Adult.setup(unique_id,
147                                               keyword.init,
148                                               self,
149                                               genotype)
150                 new.activate()
151 
152     def populate_pregnant(self, nums: hint.init_pop) -> None:
153         """
154         Create the initial pregnant adults
```

```
155
156         Args:
157             nums: (num_homo_r, num_hetero, num_homo_s)
158
159         Effects:
160             adds pregnant of the different amounts to the simulation
161         """
162
163         for index, genotype in enumerate(keyword.genotype_keys):
164             for _ in range(nums[index]):
165                 if genotype == keyword.hetero:
166                     parents = [keyword.homo_r, keyword.homo_s]
167                 else:
168                     parents = [genotype, genotype]
169                 rnd.shuffle(parents)
170
171                 unique_id = self.new_unique_id()
172                 new       = adult.Adult.setup(unique_id,
173                                               keyword.init,
174                                               self,
175                                               parents[0],
176                                               parents[1])
177                 new.activate()
178
179     def populate(self, nums: hint.init_pops) -> None:
180         """
181         Create the initial populations
182
183         Args:
184             nums: (egg_masses, larvae, pupae, adults, pregnant)
185
186         Effects:
187             adds the initial population to simulation
188         """
189
190         self.populate_egg_masses(nums[0])
191         self.populate_larvae(    nums[1])
192         self.populate_pupae(     nums[2])
193         self.populate_adults(    nums[3])
```

```
194         self.populate_pregnant(  nums[4])
195
196     def step(self) -> None:
197         """
198         Advance the simulation forward one step
199
200         Effect:
201             advance simulation forward by 1
202         """
203
204         self.count_step()
205
206         self.schedule.   perform(    self.space, self.agents)
207         self.immigration.immigration(self)
208         self.emigration. emigration( self.agents)
209         self.agents.     record()
210         self.database.   save(self)
211
212     def save(self, filename: str) -> None:
213         """
214         Pickle the simulation to a file for reuse
215
216         Args:
217             filename: name of pickle file
218
219         Effects:
220             write entire simulation to a file
221         """
222
223         with open(filename, 'wb') as sim_dump:
224             pk.dump(self, sim_dump, protocol=pk.HIGHEST_PROTOCOL)
225
226     @classmethod
227     def setup(cls, nums:               hint.init_pops,
228                    grid_generators:    hint.grid_generators,
229                    attrs:              hint.attrs_depth,
230                    data_tuple:         hint.data_tuple,
231                    bt_prop:            float,
232                    step_tuples:        hint.step_tuples,
```

```
                emigration_tuples:  hint.emigration_tuples,
                immigration_tuples: hint.immigration_tuples,
                *args, **kwargs) -> 'Simulation':
        """
        Setup the full model

        Args:
            nums:               initial populations
            grid_generators:    space generation arguments
            attrs:              data tracking arguments
            data_tuple:         save data arguments
            bt_prop:            proportion of bt in environment
            step_tuples:        schedule order arguments
            emigration_tuples:  emigration setup
            immigration_tuples: immigration setup
            *args:              input models
            **kwargs:           input values

        Returns:
            A fully initialized model
        """

        models      = main_models.Models.setup(*args, **kwargs)
        behaviors   = main_behaviors.Behaviors.setup(**models)
        schedule    = main_schedule.Schedule.setup(step_tuples)
        database    = main_database.Database.setup(data_tuple)
        emigration  = main_emigration.Emigrations.setup(emigration_tuples)
        immigration = main_immigration.Immigrations.setup(immigration_tuples)

        space   = main_space.Space.setup(grid_generators)
        cutoff  = bt_prop * space[keyword.bt_level].adjacency.num
        environ = (cutoff, models[keyword.init_plant])
        agents  = main_agents.Agents.empty(space,
                                           keyword.agent_keys,
                                           attrs, environ)

        new = cls(space, agents, schedule, models, behaviors, database,
                  emigration, immigration)
        new.populate(nums)
```

```
272         new.agents.record()
273 
274         return new
```

C.4.11 space

```
FallArmyworm
└── source
    └── space
        ├── agents.py
        ├── environment.py
        ├── graph.py
        ├── grid.py
        ├── location.py
        └── space.py
```

C.4.11.1 agents.py

```
import collections as collect

import source.hint    as hint
import source.keyword as keyword

import source.data.counter as count

import source.space.environment as agent_environment


class AgentBin(collect.UserDict):
    """
    Class to contain agents

    Variables:
        - dict:
            key:   unique_id
            value: agent

        counts:    counts of attributes of agents
        agent_key: key for type of agent

    Methods:
        activate:   add    agent to bin
        deactivate: remove agent from bin
        count:      add an attribute to count

    Constructors:
        empty: setup a class
    """

    def __init__(self, agents:    hint.agent_dict,
                       counts:    hint.counts,
                       agent_key: str):
        super().__init__(agents)

        self.counts    = counts
```

```
        self.agent_key = agent_key

    @property
    def agents(self) -> hint.agent_list:
        """Get the agents in bin in a list format"""

        return list(self.values())

    def activate(self, agent: hint.agent) -> None:
        """
        Activate the agent

        Args:
            agent: agent to activate

        Effects:
            add agent to bin
        """

        self[agent.unique_id] = agent
        self.counts.add(agent)

    def deactivate(self, agent: hint.agent) -> None:
        """
        Deactivate the agent

        Args:
            agent: agent to deactivate

        Effects:
            remove agent from bin
        """

        del self[agent.unique_id]
        self.counts.sub(agent)

    @classmethod
    def empty(cls, agent_key: str,
                   attrs:     hint.attrs_dict)  -> 'AgentBin':
```

```
77          """
78          Setup an empty agent bin
79
80          Args:
81              agent_key: key for the agent
82              attrs:     attributes to track
83
84          Returns:
85              a setup class
86          """
87
88          counts = count.Counts.empty(attrs)
89
90          return cls({}, counts, agent_key)
91
92
93  class AgentsBin(collect.UserDict):
94      """
95      Class to contain all of the agents in a particular place
96
97      Variables:
98          - dict:
99              key:   agent_key
100             value: bin of agents
101
102         location_key: location for this bin
103         environment:  environment at this location
104
105     Methods:
106         activate:   add    agent to bin
107         deactivate: remove agent from bin
108         count:      add an attribute to count
109         record:     record the current counts
110         refresh:    refresh the stored counts
111         dataframes: create dictionary of all the dataframes
112     """
113
114     def __init__(self, agents:       hint.agent_bins,
115                        location_key: hint.location_key,
```

```
                        environment:  hint.environment):
        super().__init__(agents)

        self.location_key = location_key
        self.environment  = environment

    def activate(self, agent: hint.agent) -> None:
        """
        Activate the agent

        Args:
            agent: agent to activate

        Effects:
            add agent to bin
        """

        self[agent.agent_key].activate(agent)

    def deactivate(self, agent: hint.agent) -> None:
        """
        Deactivate the agent

        Args:
            agent: agent to deactivate

        Effects:
            remove agent from bin
        """

        self[agent.agent_key].deactivate(agent)

    def record(self) -> None:
        """
        Record the current counts

        Effect:
            Records all of the data
        """
```

```
155
156         for agent_bin in self.values():
157             agent_bin.counts.record()
158
159     def refresh(self) -> None:
160         """
161         Refresh the stored counts
162
163         Effects:
164             Refresh all of the stored data
165         """
166
167         for agent_bin in self.values():
168             agent_bin.counts.refresh()
169
170     def dataframes(self) -> hint.dataframes:
171         """
172         Create a dictionary of all of dataframes for bin
173
174         Returns:
175             a dictionary of dataframes
176         """
177
178         dataframes = {}
179         for agent_key, agent_bin in self.items():
180             key = '{}_{}'.format(self.location_key, agent_key)
181             dataframe = agent_bin.counts.dataframe()
182             if not dataframe.empty:
183                 dataframes[key] = dataframe
184
185         return dataframes
186
187     @staticmethod
188     def make_environment(location:    hint.location,
189                          environment: hint.environment_tuple) \
190             -> hint.environment:
191         """
192         Make the bin's environment
193         Args:
```

```
194             location:     location represented by bin
195             environment:  environment inputs
196
197         Returns:
198             an environment
199         """
200
201         cutoff, init_plant = environment
202
203         if location.depth == keyword.bt_depth:
204             if location[-1] < cutoff:
205                 return agent_environment.Environment.setup(keyword.bt,
206                                                            init_plant)
207             else:
208                 return agent_environment.Environment.setup(keyword.not_bt,
209                                                            init_plant)
210         else:
211             return agent_environment.Environment()
212
213     @staticmethod
214     def make_bins(agent_keys: hint.agent_keys,
215                   attrs:      hint.attrs_dict) -> hint.agent_bins:
216         """
217         Create the bins for the system
218         Args:
219             agent_keys: keys for the agents
220             attrs:      tracking for the agents
221
222         Returns:
223             empty bins
224         """
225
226         agents = {}
227         for agent_key in agent_keys:
228             if agent_key in attrs:
229                 attr = attrs[agent_key]
230             else:
231                 attr = {}
232
```

```
233             agents[agent_key] = AgentBin.empty(agent_key, attr)
234
235         return agents
236
237     @staticmethod
238     def get_attrs(location: hint.location,
239                   attrs:    hint.attrs_depth) -> hint.attrs_dict:
240         """
241         Get the correct attrs system
242
243         Args:
244             location: location for the bin
245             attrs:    attrs input system
246
247         Returns:
248             attrs for this location
249         """
250
251         level = location.level
252         if level in attrs:
253             return attrs[level]
254         else:
255             return {}
256
257     @classmethod
258     def empty(cls, agent_keys:   hint.agent_keys,
259                    location:     hint.location,
260                    attrs:        hint.attrs_depth,
261                    environment:  hint.environment_tuple)  -> 'AgentsBin':
262         """
263         Setup an empty agent bin
264
265         Args:
266             agent_keys:   keys for the agents
267             location:     location represented by bin
268             attrs:        tracking attributes
269             environment:  environment inputs
270
271         Returns:
```

```
            a setup class
        """

        location_key = location.location_key
        attrs_dict   = cls.get_attrs(location, attrs)
        agents       = cls.make_bins(agent_keys, attrs_dict)
        environ      = cls.make_environment(location, environment)

        return cls(agents, location_key, environ)


class Agents(collect.UserDict):
    """
    Class to contain all of the agents

    Variables:
        - dict:
            key:   agent_key
            value: bin of agents

    Methods:
        activate:   add    agent to bin
        deactivate: remove agent from bin
        count:      add an attribute to count
    """

    def __init__(self, agents: hint.agents_dict):
        super().__init__(agents)

    def agents(self, agent_key: str) -> hint.agent_list:
        """
        Get a list of all the agents for the given key

        Args:
            agent_key: type of agent

        Returns:
            agents from master location
        """
```

```
311
312         return self[(0,)][agent_key].agents
313
314     def activate(self, agent: hint.agent) -> None:
315         """
316         Activate the agent
317
318         Args:
319             agent: agent to activate
320
321         Effects:
322             add agent to bin
323         """
324
325         location = agent.location
326
327         for index in range(1, location.depth + 1):
328             location_key = location[:index].location_key
329             self[location_key].activate(agent)
330
331     def deactivate(self, agent: hint.agent) -> None:
332         """
333         Deactivate the agent
334
335         Args:
336             agent: agent to deactivate
337
338         Effects:
339             remove agent from bin
340         """
341
342         location = agent.location
343
344         for index in range(1, location.depth + 1):
345             location_key = location[:index].location_key
346             self[location_key].deactivate(agent)
347
348     def record(self) -> None:
349         """
```

```
350         Record all the current counts
351
352         Effects:
353             records all of the current counts
354         """
355
356         for agents_bin in self.values():
357             agents_bin.record()
358
359     def refresh(self) -> None:
360         """
361         Refresh all of the stored counts
362
363         Effects:
364             refresh all the current counts
365         """
366
367         for agents_bin in self.values():
368             agents_bin.refresh()
369
370     def dataframes(self) -> hint.dataframes:
371         """
372         Create a dictionary of all of dataframes
373
374         Returns:
375             a dictionary of dataframes
376         """
377
378         dataframes = {}
379         for agents_bin in self.values():
380             dataframes.update(agents_bin.dataframes())
381
382         return dataframes
383
384     @classmethod
385     def empty(cls, space:       hint.space,
386                    agent_keys:  hint.agent_keys,
387                    attrs:       hint.attrs_depth,
388                    environment: hint.environment_tuple) -> 'Agents':
```

```
389         """
390         Setup an empty agent bin
391 
392         Args:
393             space:       the main space system
394             agent_keys:  keys for the agents
395             attrs:       tracking attributes
396             environment: the arguments to generate an environment
397 
398         Returns:
399             a setup class
400         """
401 
402         agents = {}
403         for location in space.locations:
404             location_key = location.location_key
405 
406             agents[location_key] = AgentsBin.empty(agent_keys, location,
407                                                    attrs, environment)
408 
409         return cls(agents)
```

C.4.11.2 environment.py

```
import dataclasses as dclass

import source.hint as hint


@dclass.dataclass
class Environment(object):
    """
    Class to handle local environmental conditions:

    Variables:
        bt:    bt state of local environment
        plant: mass available to each individual to consume

    Constructors:
        read the data
    """

    bt:    str   = None
    plant: float = None

    @classmethod
    def setup(cls, bt:         str,
                   init_plant: hint.init_plant) -> 'Environment':
        """
        Setup the environment

        Args:
            bt:         bt state of the local environment
            init_plant: plant mass model

        Returns:
            setup environment
        """

        plant = init_plant(bt)

```

```
        return cls(bt, plant)
```

C.4.11.3 graph.py

```
import collections as collect
import dataclasses as dclass
import numpy       as np
import pickle      as pickle

import joblib          as para
import multiprocessing as multi

import source.hint    as hint
import source.keyword as keyword


class VertexNeighborhood(collect.UserDict):
    """
    Class to find vertices a given distance from a vertex in a graph

    Variables:
        dictionary:
            key   -> distance
            value -> set of vertices at distance
        _vertex: vertex measuring from

    Properties:
        minimum: minimum distance from vertex
        maximum: maximum distance from vertex

    Methods:
        add:          vertex at distance
        neighborhood: get vertices in a distance range

    Constructors:
        empty: setup empty class
    """

    def __init__(self, vertex:       int,
                       neighborhood: hint.neighborhood_dict):
        super().__init__(neighborhood)
```

```
38
39          self.vertex = vertex
40
41      @property
42      def minimum(self) -> float:
43          """Get minimum distance from vertex"""
44
45          return min(self.keys())
46
47      @property
48      def maximum(self) -> float:
49          """Get maximum distance from vertex"""
50
51          return max(self.keys())
52
53      def add(self, distance: float,
54                    vertex:   int) -> None:
55          """
56          Add a vertex at distance
57
58          Args:
59              distance: distance for vertex
60              vertex:   vertex to add
61
62          Effects:
63              adds vertex at distance
64          """
65
66          if distance in self:
67              self[distance].add(vertex)
68          else:
69              self[distance] = {vertex}
70
71      def _convert(self, distance: float) -> float:
72          """
73          Convert a given distance to the closest value in dictionary
74
75          Args:
76              distance: distance to convert
```

```
77
78          Returns:
79              a distance within dictionary
80          """
81
82          return min(self.keys(), key=lambda x: abs(x - distance))
83
84      def _upper_lower(self, **kwargs) -> hint.upper_lower:
85          """
86          Get the upper and lower bounds on distance
87
88          Args:
89              **kwargs: pass in of upper and lower bounds
90
91          Returns:
92              the upper and lower bounds on the distance
93          """
94
95          if keyword.upper in kwargs:
96              distance: float = kwargs[keyword.upper]
97              upper = self._convert(distance)
98          else:
99              upper = np.inf
100
101         if keyword.lower in kwargs:
102             distance: float = kwargs[keyword.lower]
103             lower = self._convert(distance)
104         else:
105             lower = 0
106
107         return upper, lower
108
109     @staticmethod
110     def _append_distance(distance: float,
111                          upper:    float,
112                          lower:    float,
113                          union:    hint.vertices,
114                          vertices: hint.vertices) -> hint.vertices:
115         """
```

```
        Union to vertices if distance is okay

        Args:
            distance: distance in question
            upper:    upper bound on distance
            lower:    lower bound on distance
            union:    vertices which may be used
            vertices: vertices to union to

        Effects:
            union to vertices if correct
        """
        if lower <= distance <= upper:
            return vertices.union(union)
        else:
            return vertices

    def neighborhood(self, **kwargs) -> hint.vertices:
        """
        Get the vertices in distance range given

        Args:
            **kwargs: pass in of upper and lower bounds

        Returns:
            vertices in distance range
        """

        upper, lower = self._upper_lower(**kwargs)

        vertices = set()
        for distance, union in self.items():
            vertices = self._append_distance(distance, upper, lower,
                                             union, vertices)

        return vertices

    @classmethod
    def empty(cls, vertex: int) -> 'VertexNeighborhood':
```

```
        """
        Setup an empty vertex neighborhood

        Args:
            vertex: the origin vertex

        Returns:
            setup class
        """

        return cls(vertex, {})


class VertexDistance(collect.UserDict):
    """
    Class to measure between vertex distances:

    Variables:
        dictionary:
            key   -> vertex measuring to
            value -> distance between
        vertex: vertex measuring from

    Properties:
        minimum: get the vertex of minimum distance
        radius:  get the minimum distance

    Methods:
        add:       vertex at distance
        convert:   convert class to neighborhood
        distances: find subset of class

    Constructors:
        empty: create an empty class (all distances are inf)
    """

    def __init__(self, vertex:    int,
                       distances: hint.distance_dict):
        super().__init__(distances)
```

```
194
195         self.vertex = vertex
196
197     @property
198     def minimum(self) -> int:
199         """Get the vertex of minimum distance"""
200
201         return min(self, key=self.get)
202
203     @property
204     def radius(self) -> float:
205         """Get the minimum distance"""
206
207         return min(self.values())
208
209     def add(self, vertex:   int,
210                 distance: float) -> None:
211         """
212         Set the distance between vertices if distance is less than old distance
213
214         Args:
215             vertex:   vertex measuring to
216             distance: distance to vertex
217
218         Effects:
219             if distance is less than current distance set distance
220             else leave alone
221         """
222
223         if vertex not in self:
224             self[vertex] = np.inf
225
226         if distance < self[vertex]:
227             self[vertex] = distance
228
229     def convert(self) -> hint.vertex_neighborhood:
230         """
231         Convert class to a vertex neighborhood
232
```

```
        Returns:
            a vertex neighborhood representation of class
        """

        neighborhood = VertexNeighborhood.empty(self.vertex)
        for vertex, distance in self.items():
            neighborhood.add(distance, vertex)

        return neighborhood

    def distances(self, vertices: hint.vertices) -> 'VertexDistance':
        """
        Return the subset of distances for class

        Args:
            vertices: the subset of vertices measuring to

        Returns:
            a subset of the distances
        """

        distances = {}
        for vertex in vertices:
            distances[vertex] = self[vertex]

        return VertexDistance(self.vertex, distances)

    @classmethod
    def empty(cls, vertex: int,
                  vertices: hint.vertices) -> 'VertexDistance':
        """
        Setup an empty vertex distance class

        Args:
            vertex:   vertex measuring from
            vertices: vertices measuring to

        Returns:
            no distance set class
```

```
        """

        new = cls(vertex, {})
        for this in vertices:
            new.add(this, np.inf)

        return new


class GraphNeighborhood(collect.UserDict):
    """
    Class to contain all neighborhoods in graph

    Variables:
        dictionary:
            key   -> vertex
            value -> neighborhood

    Properties:
        minimum: minimum distance in graph
        maximum: maximum distance in graph

    Methods:
        add:          vertex at distance
        neighborhood: get neighborhood of a vertex
    """

    def __init__(self, neighborhoods: hint.neighborhoods):
        super().__init__(neighborhoods)

    @property
    def minimum(self) -> float:
        """Get the minimum distance in graph"""

        return min([neighborhood.minimum for neighborhood in self.values()])

    @property
    def maximum(self) -> float:
        """Get the maximum distance in graph"""
```

```
311
312         return max([neighborhood.maximum for neighborhood in self.values()])
313
314     def add(self, vertex: int,
315                   pair:   hint.distance_pair) -> None:
316         """
317         Add a vertex at distance from another vertex
318
319         Args:
320             vertex: start vertex
321             pair:   a distance pair (end vertex, distance from start)
322
323         Effects:
324             adds vertex at distance
325         """
326
327         end, distance = pair
328
329         if vertex not in self:
330             self[vertex] = VertexNeighborhood.empty(vertex)
331
332         self[vertex].add(distance, end)
333
334     def neighborhood(self, vertex: int, **kwargs) -> hint.vertices:
335         """
336         Get the neighborhood defined by the bounds for the vertex
337
338         Args:
339             vertex:   start vertex
340             **kwargs: pass in of upper and lower bounds
341
342         Returns:
343             neighborhood of the vertex
344         """
345
346         return self[vertex].neighborhood(**kwargs)
347
348
349 class GraphDistance(collect.UserDict):
```

```
350     """
351     Class to measure between vertex distances:
352 
353     Variables:
354         dictionary:
355             key   -> vertex measuring to
356             value -> distance between
357 
358     Methods:
359         add:     add vertex at distance
360         convert: convert class to neighborhood
361 
362     Constructors:
363         empty: create an empty class (all distances are inf)
364     """
365 
366     def __init__(self, distances: hint.distances):
367         super().__init__(distances)
368 
369     def add(self, vertex: int,
370                   pair:   hint.distance_pair) -> None:
371         """
372         Add a distance between vertices
373 
374         Args:
375             vertex: start vertex
376             pair:   a distance pair (end vertex, distance from start)
377 
378         Effects:
379             updates distance between vertices
380         """
381 
382         end, distance = pair
383 
384         if vertex not in self:
385             self[vertex] = VertexDistance.empty(vertex, set())
386 
387         self[vertex].add(end, distance)
388 
```

```
    def convert(self) -> hint.graph_neighborhood:
        """
        Convert class to a graph neighborhood

        Returns:
            a converted class
        """

        neighborhoods = {}
        for vertex, distances in self.items():
            neighborhoods[vertex] = distances.convert()

        return GraphNeighborhood(neighborhoods)

    @classmethod
    def empty(cls, vertices: hint.vertices) -> 'GraphDistance':
        """
        Initialize a graph distance class

        Args:
            vertices: set of vertices for class

        Returns:
            an initialized graph distance class
        """

        distances = {}
        for vertex in vertices:
            distances[vertex] = VertexDistance.empty(vertex, vertices)

        return cls(distances)


class Adjacency(np.ndarray):
    """
    Class to contain an adjacency matrix

    Variables:
        array -> square matrix array
```

```
    Properties:
        num:      number of vertices
        vertices: the vertex set

    Methods:
        dijkstra: generate a distance table
    """

    def __new__(cls, array):
        obj = np.asarray(array).view(cls)

        return obj

    @property
    def num(self) -> int:
        """Number of vertices"""

        return self.shape[0]

    @property
    def vertices(self) -> hint.vertices:
        """Get the set of all vertices"""

        return set(range(self.num))

    def _start_search(self, distance: hint.vertex_distance) -> hint.vertices:
        """
        Setup to start the search

        Args:
            distance: the distance class for a single root vertex

        Returns:
            the unvisited set of vertices
            and the current search neighborhood
        """

        vertex = distance.vertex
```

```
467         distance.add(vertex, 0)
468 
469         return self.vertices
470 
471     @staticmethod
472     def _minimum(unvisited: hint.vertices,
473                  distance:  hint.vertex_distance) -> float:
474         """
475         Find the minimum distance within the unvisited vertices
476 
477         Args:
478             unvisited: list of unvisited vertices
479             distance:  the distance class for a single root vertex
480 
481         Returns:
482             a minimum within the unvisited
483         """
484 
485         distances = distance.distances(unvisited)
486 
487         return distances.radius
488 
489     def _terminate(self, unvisited: hint.vertices,
490                          distance:  hint.vertex_distance) -> bool:
491         """
492         Determine if search is finished
493 
494         Args:
495             unvisited: list of unvisited vertices
496             distance:  the distance class for a single root vertex
497 
498         Returns:
499             True if no more unvisited
500             True if the radius of the unvisited vertices is inf
501             False otherwise
502         """
503 
504         if not unvisited:
505             return True
```

```
506
507         if self._minimum(unvisited, distance) == np.inf:
508             return True
509
510         return False
511
512     @staticmethod
513     def _current(unvisited: hint.vertices,
514                  distance:  hint.vertex_distance) -> int:
515         """
516         Get the current target of a dijkstra search
517
518         Args:
519             unvisited: list of unvisited vertices
520             distance:  the distance class for a single root vertex
521
522         Returns:
523             a search target
524         """
525
526         distances = distance.distances(unvisited)
527
528         return distances.minimum
529
530     def _adjacent(self, vertex: int) -> hint.vertices:
531         """
532         Get the vertices adjacent to the vertex
533
534         Args:
535             vertex: the vertex to search from
536
537         Returns:
538             set of vertices adjacent to vertex
539         """
540
541         return set(self[vertex].nonzero()[0])
542
543     def _visit(self, current:   int,
544                    unvisited: hint.vertices) -> hint.vertices:
```

```
545         """
546         Get the vertices to visit from current vertex
547 
548         Args:
549             current:   the current vertex we are searching from
550             unvisited: the unvisited vertices
551 
552         Returns:
553             a set of vertices to visit
554         """
555 
556         adjacent = self._adjacent(current)
557 
558         return unvisited.intersection(adjacent)
559 
560     def _distance(self, current:  int,
561                         vertex:   int,
562                         distance: hint.vertex_distance) -> float:
563         """
564         Get the distance via current to vertex
565 
566         Args:
567             current:  the current vertex we are searching from
568             vertex:   the vertex we are going to
569             distance: the distance class for a single root vertex
570 
571         Returns:
572             the distance via the current vertex from the root vertex
573         """
574 
575         length = distance[current]
576         extra  = self[current][vertex]
577 
578         return length + extra
579 
580     def _update(self, current: int,
581                       visit:   hint.vertices,
582                       distance: hint.vertex_distance) -> None:
583         """
```

```
584         Update the distances in distance based on visit and current
585
586         Args:
587             current:  the current vertex we are searching from
588             visit:    the vertices to visit
589             distance: the distance class for a single root vertex
590
591         Effects:
592             updates the distances in the distance class
593         """
594
595         for vertex in visit:
596             new = self._distance(current, vertex, distance)
597             distance.add(vertex, new)
598
599     def _search(self, unvisited: hint.vertices,
600                 distance:  hint.vertex_distance) -> None:
601         """
602         Run a search on the current distances for next step
603         Args:
604             unvisited: the set of unvisited vertices
605             distance:  the distance class for a single root vertex
606
607         Effects:
608             Runs a single search level on the distance
609         """
610
611         current = self._current(unvisited, distance)
612         visit   = self._visit(current, unvisited)
613
614         self._update(current, visit, distance)
615         unvisited.remove(current)
616
617     def _dijkstra(self, distance: hint.vertex_distance) -> None:
618         """
619         Run Dijkstra's to find min distance starting at root vertex
620
621         Args:
622             distance: the distance class for a single root vertex
```

```
623
624         Effects:
625             Fill in distances in distance
626         """
627
628         unvisited = self._start_search(distance)
629
630         for _ in range(self.num + 1):
631             if self._terminate(unvisited, distance):
632                 break
633             else:
634                 self._search(unvisited, distance)
635         else:
636             raise RuntimeError('Search of vertex {} has failed'.
637                                format(distance.vertex))
638
639     def _para_dijkstra(self, vertex: int) -> hint.vertex_distance:
640         """
641         Run Dijkstra's to find min distance starting at root vertex using
642             parallel library
643
644         Args:
645             vertex:   the vertex
646
647         Returns:
648             vertex distance system for vertex
649         """
650
651         print('Dijkstra for vertex: {}'.format(vertex))
652
653         distance = VertexDistance.empty(vertex, self.vertices)
654         self._dijkstra(distance)
655
656         return distance
657
658     def dijkstra(self, parallel: bool = False) -> hint.graph_distance:
659         """
660         Run Dijkstra's Algorithm to find a graph distance object
661
```

```
662         Returns:
663             a graph distance object
664         """
665 
666         if parallel:
667             inputs = range(self.num)
668             num    = multi.cpu_count()
669             print('Num Cores: {}'.format(num))
670 
671             values = para.Parallel(n_jobs=num)(para.delayed(self._para_dijkstra)(values)
672                                                for values in inputs)
673             dist_dict = {}
674             for index, value in enumerate(values):
675                 dist_dict[index] = value
676 
677             distances = GraphDistance(dist_dict)
678         else:
679             distances = GraphDistance.empty(self.vertices)
680             for distance in distances.values():
681                 self._dijkstra(distance)
682 
683         return distances
684 
685 
686 @dclass.dataclass
687 class Graph(object):
688     """
689     Class to handle a graph
690 
691     Variables:
692         _neighborhood: neighborhood table
693         _distance:     distance table
694         _adjacency:    adjacency matrix
695 
696     Properties:
697         minimum: minimum distance in graph
698         maximum: maximum distance in graph
699 
700         distance: distance table (pass through)
```

```
701
702         num:      number of vertices
703         vertices: the vertex set
704
705     Methods:
706         neighborhood: neighborhood finding
707
708     Constructors:
709         setup: setup the class from a matrix
710         empty: create a single vertex graph
711     """
712
713     neighborhood: hint.graph_neighborhood
714     distance:     hint.graph_distance
715     adjacency:    hint.graph_adjacency
716
717     def save(self, file_name: str) -> None:
718         """
719         Save graph to file_name for reuse
720
721         Args:
722             file_name: name of file to pickle to
723
724         Effects:
725             writes data to file
726         """
727
728         with open(file_name, 'wb') as graph_dump:
729             pickle.dump(self, graph_dump, protocol=pickle.HIGHEST_PROTOCOL)
730
731     @classmethod
732     def setup(cls, matrix,
733                   parallel: bool = False) -> 'Graph':
734         """
735         Setup a graph from a matrix
736
737         Args:
738             matrix: the matrix
739             parallel: determine if this is a parallel compute
```

```
740
741         Returns:
742             Fully setup class
743         """
744
745         adjacency    = Adjacency(matrix)
746         distance     = adjacency.dijkstra(parallel)
747         neighborhood = distance.convert()
748
749         return cls(neighborhood, distance, adjacency)
750
751     @classmethod
752     def empty(cls) -> 'Graph':
753         """
754         Generate an empty class
755
756         Returns:
757             a single vertex graph
758         """
759
760         return cls.setup([[0]])
```

C.4.11.4 grid.py

```
import datetime
import dataclasses as dclass

import source.hint as hint

import source.space.graph as graph


@dclass.dataclass
class Grid(graph.Graph):
    """
    Base class for grid graphs

    Constructor:
        grid: create the grid graph
    """

    @staticmethod
    def _boundary(n: int,
                  m: int) -> hint.boundary:
        """
        Get vertices on grid boundary

        Args:
            n:
            m:

        Returns:
            (bottom vertices,
             top    vertices,
             left   vertices,
             right  vertices)
            on grid boundary
        """

        vertices = list(range(n*m))

```

```
        bottom = vertices[:n]
        top    = vertices[-n:]

        left  = []
        right = []
        for vertex in vertices:
            if 0 == (vertex % n):
                left.append(vertex)

            if (n - 1) == (vertex % n):
                right.append(vertex)

        return bottom, top, left, right

    def _upper_indicator(self, i: int,
                               j: int,
                               n: int,
                               m: int) -> bool:
        """
        Determine if there will be a 1 or 0 in the jth position of the
            ith column of the adjacency matrix

        Args:
            i: column
            j: row
            n: number of rows    in grid
            m: number of columns in grid

        Returns:
            0 or 1
        """

        pass

    def _upper_generator(self, n: int,
                               m: int) -> hint.upper_grid:
        """
        Generate the upper triangle of the adjacency matrix

```

```
77          Args:
78              n: number of rows    in grid
79              m: number of columns in grid
80
81          Returns:
82              upper triangular matrix
83          """
84
85          num = n*m
86
87          column = []
88          for i in range(num):
89              row = []
90              for j in range(num):
91                  if self._upper_indicator(i, j, n, m):
92                      row.append(1)
93                  else:
94                      row.append(0)
95              column.append(row)
96
97          return column
98
99      def _upper_torus(self, upper: hint.upper_grid,
100                             n:     int,
101                             m:     int) -> None:
102         """
103         Adjust grid adjacency matrix to be for a torus
104
105         Args:
106             upper: the upper adjacency matrix
107             n:     number of rows    in grid
108             m:     number of columns in grid
109
110         Effects:
111             modify the upper adjacency matrix to be for a torus grid
112         """
113
114         pass
115
```

```
116     @staticmethod
117     def _full(upper: hint.upper_grid) -> hint.graph_adjacency:
118         """
119         Generate a full adjacency matrix from an upper grid
120 
121         Args:
122             upper: the upper adjacency matrix
123 
124         Returns:
125             a full adjacency matrix
126         """
127 
128         upper_matrix = graph.Adjacency(upper)
129         lower_matrix = upper_matrix.transpose()
130 
131         return upper_matrix + lower_matrix
132 
133     def _generator(self, n:     int,
134                          m:     int,
135                          torus: bool) -> hint.graph_adjacency:
136         """
137         Generate an adjacency matrix for grid
138 
139         Args:
140             n:     number of rows    in grid
141             m:     number of columns in grid
142             torus: if the grid is a torus
143 
144         Returns:
145             An adjacency matrix
146         """
147 
148         upper = self._upper_generator(n, m)
149         if torus:
150             self._upper_torus(upper, n, m)
151 
152         return self._full(upper)
153 
154     @classmethod
```

```
155     def grid(cls, n:        int,
156                   m:        int,
157                   torus:    bool,
158                   parallel: bool = False) -> 'Grid':
159         """
160         Create a grid graph
161 
162         Args:
163             n:     number of rows    in grid
164             m:     number of columns in grid
165             torus: if the grid is a torus
166             parallel: determine if this is a parallel compute
167 
168         Returns:
169             a setup class
170         """
171 
172         grid = cls.empty()
173 
174         if parallel:
175             print('Creating Adjacency {}'.format(datetime.datetime.now()))
176         adjacency    = grid._generator(n, m, torus)
177         if parallel:
178             print('Creating Distance {}'.format(datetime.datetime.now()))
179         distance     = adjacency.dijkstra(parallel)
180         if parallel:
181             print('Creating Neighborhood {}'.format(datetime.datetime.now()))
182         neighborhood = distance.convert()
183 
184         return cls(neighborhood, distance, adjacency)
185 
186 
187 @dclass.dataclass
188 class Hexagon(Grid):
189     """
190     Class for hexagon tile grid graphs
191     """
192 
193     def _upper_indicator(self, i: int,
```

```
                              j: int,
                              n: int,
                              m: int) -> bool:
        """
        Determine if there will be a 1 or 0 in the jth position of the
            ith column of the adjacency matrix

        Args:
            i: column
            j: row
            n: number of rows    in grid
            m: number of columns in grid

        Returns:
            0 or 1
        """

        # Above
        if j == (i + n):
            return True

        # To the right
        if ((n - 1) != (i % n)) and (j == (i + 1)):
            return True

        # Diagonal
        if (0 != (i % n)) and (j == (i + n - 1)):
            return True

        return False

    def _upper_torus(self, upper: hint.upper_grid,
                             n:     int,
                             m:     int) -> None:
        """
        Adjust grid adjacency matrix to be for a torus

        Args:
            upper: the upper adjacency matrix
```

```
            n:      number of rows    in grid
            m:      number of columns in grid

        Effects:
            modify the upper adjacency matrix to be for a torus grid
        """

        bottom, top, left, right = self._boundary(n, m)

        # Top-Bottom
        for index in range(n):
            # Vertical edges
            upper[bottom[index]][top[index]] = 1

            # Diagonals
            if index < (n - 1):
                upper[bottom[index]][top[index + 1]] = 1

        upper[bottom[n - 1]][top[0]] = 1

        # Left-Right
        for index in range(m):
            # Horizontal edges
            upper[left[index]][right[index]] = 1

            # Diagonals
            if index < (m - 1):
                upper[left[index]][right[index + 1]] = 1


@dclass.dataclass
class Square(Grid):
    """
    Class for square tile grid graphs
    """

    def _upper_indicator(self, i: int,
                                j: int,
                                n: int,
```

```
                              m: int) -> bool:
        """
        Determine if there will be a 1 or 0 in the jth position of the
            ith column of the adjacency matrix

        Args:
            i: column
            j: row
            n: number of rows    in grid
            m: number of columns in grid

        Returns:
            0 or 1
        """

        # Above
        if j == (i + n):
            return True

        # To the right
        if ((n - 1) != (i % n)) and (j == (i + 1)):
            return True

        return False

    def _upper_torus(self, upper: hint.upper_grid,
                           n:     int,
                           m:     int) -> None:
        """
        Adjust grid adjacency matrix to be for a torus

        Args:
            upper: the upper adjacency matrix
            n:     number of rows    in grid
            m:     number of columns in grid

        Effects:
            modify the upper adjacency matrix to be for a torus grid
        """
```

```
311
312         bottom, top, left, right = self._boundary(n, m)
313
314         # Top-Bottom
315         for index in range(n):
316             # Vertical edges
317             upper[bottom[index]][top[index]] = 1
318
319         # Left-Right
320         for index in range(m):
321             # Horizontal edges
322             upper[left[index]][right[index]] = 1
323
324
325 @dclass.dataclass
326 class Moore(Grid):
327     """
328     Class for Moore tile grid graphs
329         -square tiles that allow movements to the diagonals
330     """
331
332     def _upper_indicator(self, i: int,
333                                j: int,
334                                n: int,
335                                m: int) -> bool:
336         """
337         Determine if there will be a 1 or 0 in the jth position of the
338             ith column of the adjacency matrix
339
340         Args:
341             i: column
342             j: row
343             n: number of rows    in grid
344             m: number of columns in grid
345
346         Returns:
347             0 or 1
348         """
349
```

```
        # Above
        if j == (i + n):
            return True

        # To the right
        if ((n - 1) != (i % n)) and (j == (i + 1)):
            return True

        # Diagonal left
        if (0 != (i % n)) and (j == (i + n - 1)):
            return True

        # Diagonal right
        if ((n - 1) != (i % n)) and (j == i + n + 1):
            return True

        return False

    def _upper_torus(self, upper: hint.upper_grid,
                             n:     int,
                             m:     int) -> None:
        """
        Adjust grid adjacency matrix to be for a torus

        Args:
            upper: the upper adjacency matrix
            n:     number of rows    in grid
            m:     number of columns in grid

        Effects:
            modify the upper adjacency matrix to be for a torus grid
        """

        bottom, top, left, right = self._boundary(n, m)

        # Top-Bottom
        for index in range(n):
            # Vertical edges
            upper[bottom[index]][top[index]] = 1
```

```
389
390             # Diagonals left
391             if index < (n - 1):
392                 upper[bottom[index]][top[index + 1]] = 1
393
394             # Diagonals right
395             if index > 0:
396                 upper[bottom[index]][top[index - 1]] = 1
397
398         upper[bottom[n - 1]][top[0]] = 1
399         upper[bottom[0]][top[n - 1]] = 1
400
401         # Left-Right
402         for index in range(m):
403             # Horizontal edges
404             upper[left[index]][right[index]] = 1
405
406             # Diagonals left
407             if index < (m - 1):
408                 upper[left[index]][right[index + 1]] = 1
409
410             # Diagonals right
411             if index > 0:
412                 upper[left[index]][right[index - 1]] = 1
413
414
415 @dclass.dataclass
416 class Triangle(Grid):
417     """
418     Class for triangle tile grid graphs
419     """
420
421     def _upper_indicator(self, i: int,
422                                j: int,
423                                n: int,
424                                m: int) -> bool:
425         """
426         Determine if there will be a 1 or 0 in the jth position of the
427             ith column of the adjacency matrix
```

```
428
429         Args:
430             i: column
431             j: row
432             n: number of rows    in grid
433             m: number of columns in grid
434
435         Returns:
436             0 or 1
437         """
438
439         # To the right
440         if ((n - 1) != (i % n)) and (j == (i + 1)):
441             return True
442
443         # Above on even rows
444         if (0 == ((i // n) % 2)) and (0 == (i % 2)) and (j == (i + n)):
445             return True
446
447         # Above on odd rows
448         if (1 == ((i // n) % 2)) and (1 == (i % 2)) and (j == (i + n)):
449             return True
450
451         return False
452
453     def _upper_torus(self, upper: hint.upper_grid,
454                              n:     int,
455                              m:     int) -> None:
456         """
457         Adjust grid adjacency matrix to be for a torus
458
459         Args:
460             upper: the upper adjacency matrix
461             n:     number of rows    in grid
462             m:     number of columns in grid
463
464         Effects:
465             modify the upper adjacency matrix to be for a torus grid
466         """
```

```
467
468         bottom, top, left, right = self._boundary(n, m)
469
470         if 1 == (n % 2):
471             raise ValueError('n must be even')
472
473         # Top-Bottom
474         for index in range(n):
475             if (0 == (m % 2)) and (1 == (index % 2)):
476                 upper[bottom[index]][top[index]] = 1
477             elif (1 == (m % 2)) and (0 == (index % 2)):
478                 upper[bottom[index]][top[index]] = 1
479
480         # Left-Right
481         for index in range(m):
482             upper[left[index]][right[index]] = 1
```

C.4.11.5 location.py

```
import collections as collect

import source.hint as hint


class Location(collect.UserList):
    """
    Class to contain a location

    Variables:
        - list:
            index: level of location
            value: location vertex

    Properties:
        location_key: a tuple form of location
        depth:        length of location
    """

    def __init__(self, locs: hint.locs):
        super().__init__(locs)

    @property
    def location_key(self) -> hint.location_key:
        """Get the location key for the location"""

        return tuple(self)

    @property
    def depth(self) -> int:
        """Get the depth of this location"""

        return len(self)

    @property
    def level(self) -> int:
        """Get level this location operates at"""
```

```
38
39          return len(self) - 1
```

C.4.11.6 space.py

```
import collections  as collect
import numpy.random as rnd

import source.hint    as hint
import source.keyword as keyword

import source.space.graph    as main_graph
import source.space.grid     as grid
import source.space.location as agent_location


class Space(collect.UserList):
    """
    Class to contain space references for class

    Variables:
        - list:
            index: level of graph
            value: graph at level

        locations:     list of all possible locations
        location_keys: dict
            key:   level of interest
            value: list of location keys at that level
    """

    def __init__(self, graphs:        hint.graphs,
                       locations:     hint.locations,
                       location_keys: hint.locations_key):
        super().__init__(graphs)

        self.locations     = locations
        self.location_keys = location_keys

    @property
    def depth(self) -> int:
        """Get the depth of this location"""
```

```
38
39          return len(self)
40
41      def neighborhood(self, location: hint.location, **kwargs) -> hint.vertices:
42          """
43          Get the vertices in distance range of location
44
45          Args:
46              location: location to search from
47              **kwargs: upper/lower bounds of search
48
49          Returns:
50              vertices in range of location
51          """
52
53          level             = location.level
54          graph: hint.graph = self[level]
55          vertex            = location[level]
56
57          return graph.neighborhood.neighborhood(vertex, **kwargs)
58
59      def extend_location(self, location: hint.location) -> hint.location:
60          """
61          Extend a location by one level to new depth
62
63          Args:
64              location: location to extend
65
66          Returns:
67              a location randomly extend by 1 level
68          """
69
70          graph: hint.graph = self[location.depth]
71          vertices          = list(graph.adjacency.vertices)
72          vertex            = rnd.choice(vertices)
73
74          new = location.copy()
75          new.append(vertex)
76
```

```
77          return new
78
79      def new_location(self, depth: int) -> hint.location:
80          """
81          Create a new location at random
82
83          Args:
84              depth: depth of location desired
85
86          Returns:
87              a random location of the correct depth
88          """
89
90          locs = []
91          for level in range(depth):
92              graph: hint.graph = self[level]
93              vertices          = list(graph.adjacency.vertices)
94              locs.append(rnd.choice(vertices))
95
96          return agent_location.Location(locs)
97
98      def _make_locations(self, locations: hint.locations,
99                                level:     int) -> hint.location_pairs:
100         """
101         Extend the location keys out to level
102
103         Args:
104             locations: current list of location keys
105             level:     new level
106
107         Returns:
108             all of the new locations
109         """
110
111         graph: hint.graph = self[level]
112         vertices          = graph.adjacency.vertices
113
114         new  = []
115         keys = []
```

```
        for location in locations:
            for vertex in vertices:
                new_location = location.copy()
                new_location.append(vertex)
                new. append(new_location)
                keys.append(new_location.location_key)

        return new, keys

    def get_locations(self) -> hint.location_data:
        """
        Generate all of the locations possible

        Returns:
            list of all locations
        """

        locations     = [agent_location.Location([0])]
        location_keys = {0: [locations[0].location_key]}

        locs = locations.copy()
        for level in range(1, self.depth):
            locs, keys = self._make_locations(locs, level)
            locations += locs
            location_keys[level] = keys

        return locations, location_keys

    @staticmethod
    def create_grid(grid_generator: hint.grid_generator) -> hint.graph:
        """
        Create a grid for the space

        Args:
            grid_generator: generation arguments

        Returns:
            A setup grid
        """
```

```
155
156         grid_key  = grid_generator[0]
157         grid_args = grid_generator[1:]
158
159         if grid_key == keyword.hexagon:
160             return grid.Hexagon.grid(*grid_args)
161         elif grid_key == keyword.square:
162             return grid.Square.grid(*grid_args)
163         elif grid_key == keyword.moore:
164             return grid.Moore.grid(*grid_args)
165         elif grid_key == keyword.triangle:
166             return grid.Triangle.grid(*grid_args)
167         else:
168             raise TypeError('Invalid type of grid')
169
170     @classmethod
171     def setup(cls, grid_generators: hint.grid_generators) -> 'Space':
172         """
173         Setup space properly
174
175         Args:
176             grid_generators: the grid generators for the class
177                 (note master is not needed)
178
179         Returns:
180             fully setup class
181         """
182
183         graphs = [main_graph.Graph.setup([[0]])]
184
185         for grid_generator in grid_generators:
186             if isinstance(grid_generator, main_graph.Graph):
187                 graphs.append(grid_generator)
188             else:
189                 graphs.append(cls.create_grid(grid_generator))
190
191         new           = cls(graphs, [], {})
192         new.locations, new.location_keys = new.get_locations()
193
```

```
194         return new
```

C.4.12 survival

```
FallArmyworm
└── source
    └── survival
        ├── adult.py
        ├── egg.py
        ├── larva.py
        ├── models.py
        └── pupa.py
```

C.4.12.1 adult.py

```
import dataclasses as dclass

import source.hint    as hint
import source.keyword as keyword


@dclass.dataclass
class Adult(object):
    """
    Class to handle adult survival behavior

    Variables:
        survival: mathematical function for if adult dies of survival

    Methods:
        survive: run the behavior

    Constructors:
        setup: setup class
    """

    survival: hint.survival_adult = None

    @property
    def _use_survival(self) -> bool:
        """Determine if we use the survival model"""

        return self.survival is not None

    def _survive(self, adult: hint.adult) -> bool:
        """
        Determine if the adult survives

        Args:
            adult: the adult in question

        Returns:
```

```
            if the adult survives or not
        """

        if self._use_survival:
            return self.survival(adult.mass, adult.genotype)
        else:
            return True

    def survive(self, adult: hint.adult) -> None:
        """
        Run the survival model

        Args:
            adult: the adult in question

        Effects:
            kills adult if it fails to survive
        """

        if not self._survive(adult):
            adult.die(keyword.survival)

    @classmethod
    def setup(cls, **kwargs) -> 'Adult':
        """
        Setup the class

        Args:
            **kwargs: simulation input models

        Returns:
            setup class
        """

        if keyword.adult_survival in kwargs:
            return cls(kwargs[keyword.adult_survival])
        else:
            return cls()
```

C.4.12.2 egg.py

```
import dataclasses as dclass

import source.hint    as hint
import source.keyword as keyword


@dclass.dataclass
class Egg(object):
    """
    Class to handle egg survival behavior

    Variables:
        survival: mathematical function for if egg dies of survival

    Methods:
        survive: run the behavior

    Constructors:
        setup: setup class
    """

    survival: hint.survival_egg = None

    @property
    def _use_survival(self) -> bool:
        """Determine if we use the survival model"""

        return self.survival is not None

    def _survive(self, egg: hint.egg) -> bool:
        """
        Determine if the egg survives

        Args:
            egg: the egg in question

        Returns:
```

```
            if the egg survives or not
        """

        if self._use_survival:
            return self.survival(egg.mass, egg.genotype, egg.bt)
        else:
            return True

    def survive(self, egg: hint.egg) -> None:
        """
        Run the survival model

        Args:
            egg: the egg in question

        Effects:
            kills egg if it fails to survive
        """

        if not self._survive(egg):
            egg.die(keyword.survival)

    @classmethod
    def setup(cls, **kwargs) -> 'Egg':
        """
        Setup the class

        Args:
            **kwargs: simulation input models

        Returns:
            setup class
        """

        if keyword.egg_survival in kwargs:
            return cls(kwargs[keyword.egg_survival])
        else:
            return cls()
```

C.4.12.3 larva.py

```
import dataclasses as dclass

import source.hint    as hint
import source.keyword as keyword


@dclass.dataclass
class Larva(object):
    """
    Class to handle larva survival behavior

    Variables:
        survival: mathematical function for if egg dies of survival

    Methods:
        survive: run the behavior

    Constructors:
        setup: setup class
    """

    survival: hint.survival_larva = None

    @property
    def _use_survival(self) -> bool:
        """Determine if we use the survive model"""

        return self.survival is not None

    @staticmethod
    def _starve(larva: hint.larva) -> None:
        """
        Check if larva starves

        Args:
            larva: the larva in question

```

```
        Effects:
            if starves then death by starvation
        """

        if larva.starve:
            larva.die(keyword.starve)

    def _survive(self, larva: hint.larva) -> bool:
        """
        Determine if the larva survives

        Args:
            larva: the larva in question

        Returns:
            boolean indicating survival
        """

        if self._use_survival and larva.alive:
            return self.survival(larva.mass, larva.genotype, larva.bt)
        else:
            return True

    def survive(self, larva: hint.larva) -> None:
        """
        Run survival model

        Args:
            larva: the larva in question

        Effects:
            Run one complete survival step
        """

        self._starve(larva)
        if not self._survive(larva):
            larva.die(keyword.survival)

    @classmethod
```

```
    def setup(cls, **kwargs) -> 'Larva':
        """
        Setup the class

        Args:
            **kwargs: simulation input models

        Returns:
            setup class
        """

        if keyword.larva_survival in kwargs:
            return cls(kwargs[keyword.larva_survival])
        else:
            return cls()
```

C.4.12.4 models.py

```
import dataclasses   as dclass
import numpy         as np
import numpy.random  as rnd

import source.hint    as hint
import source.keyword as keyword

import source.simulation.models as models


@dclass.dataclass
class Larva(models.Model):
    """
    Class to describe survival model for larvae
        - assumes probability is a general logistic:
            P(m) = L + (U-L)/(1 + exp(-k(m-m0)))

            L  = minimum probability
            U  = maximum probability
            m0 = inflection point
            k  = steepness
            m  = mass


    Variables:
        minimum:    minimum probability
        maximum:    maximum probability
        inflection: inflection point
        steepness:  steepness of transition

    Methods:
        __call__: call the model
    """

    model_key = keyword.larva_survival

    minimum: hint.bt_variable
```

```
    maximum: hint.bt_variable

    inflection: hint.variable
    steepness:  hint.variable

    def _logistic(self, mass:     float,
                        genotype: str,
                        bt:       str) -> float:
        """
        Evaluate the generalized logistic function above

        Args:
            mass:     insect mass
            genotype: larva's genotype
            bt:       bt state of plant

        Returns:
            value of generalized logistic function described
        """

        lower = self.minimum[bt][genotype]
        upper = self.maximum[bt][genotype]

        m0 = self.inflection[genotype]
        k  = self.steepness[ genotype]

        top = upper - lower
        bot = np.exp(-k*(mass - m0)) + 1

        return lower + top/bot

    def __call__(self, mass:     float,
                       genotype: str,
                       bt:       str) -> bool:
        """
        Run the survival model

        Args:
            mass:     insect mass
```

```
77              genotype: larva's genotype
78              bt:       bt state of plant
79
80          Returns:
81              result of flipping a coin weighted by logistic function's resulting
82              probability
83          """
84
85          return rnd.random() <= self._logistic(mass, genotype, bt)
86
87
88  @dclass.dataclass
89  class LarvaFixed(models.Model):
90      """
91      Class to describe a survival model for larvae with fixed probabilities
92
93      Variables:
94          prob: probabilities
95      """
96
97      model_key = keyword.larva_survival
98
99      prob: hint.bt_variable
100
101     def __call__(self, mass:     float,
102                        genotype: str,
103                        bt:       str) -> bool:
104         """
105         Run the survival model
106
107         Args:
108             mass:     insect mass
109             genotype: larva's genotype
110             bt:       bt state of plant
111
112         Returns:
113             result of flipping a coin weighted by logistic function's resulting
114             probability
115         """
```

```
116
117         prob = self.prob[bt][genotype]
118
119         return rnd.random() <= prob
120
121
122 @dclass.dataclass
123 class Fixed(models.Model):
124     """
125     Class to describe a survival model with a fixed probability
126
127     Variables:
128         prob: probability of survival
129
130     Methods:
131         __call__: call the model
132     """
133
134     prob: float
135
136     def __call__(self, mass: float, *args) -> bool:
137         """
138         Call the model to determine if the agent survives
139
140         Args:
141             mass:  mass of agent
142             *args: genotype and bt (possibly)
143
144         Returns:
145             if egg survives
146         """
147
148         return rnd.random() <= self.prob
149
150
151 @dclass.dataclass
152 class Egg(Fixed):
153     """
154     Class to describe the survival model for single egg
```

```
155
156     Variables:
157         prob: probability of survival
158
159     Methods:
160         __call__: call the model
161     """
162
163     model_key = keyword.egg_survival
164
165
166 @dclass.dataclass
167 class Pupa(Fixed):
168     """
169     Class to describe the survival model for pupa
170
171     Variables:
172         prob: probability of survival
173
174     Methods:
175         __call__: call the model
176     """
177
178     model_key = keyword.pupa_survival
179
180
181 @dclass.dataclass
182 class Adult(Fixed):
183     """
184     Class to describe the survival model for adult
185
186     Variables:
187         prob: probability of survival
188
189     Methods:
190         __call__: call the model
191     """
192
193     model_key = keyword.adult_survival
```

C.4.12.5 pupa.py

```
import dataclasses as dclass

import source.hint    as hint
import source.keyword as keyword


@dclass.dataclass
class Pupa(object):
    """
    Class to handle pupa survival behavior

    Variables:
        survival: mathematical function for if pupa dies of survival

    Methods:
        survive: run the behavior

    Constructors:
        setup: setup class
    """

    survival: hint.survival_pupa = None

    @property
    def _use_survival(self) -> bool:
        """Determine if we use the survival model"""

        return self.survival is not None

    def _survive(self, pupa: hint.pupa) -> bool:
        """
        Determine if the pupa survives

        Args:
            pupa: the pupa in question

        Returns:
```

```
            if the pupa survives or not
        """

        if self._use_survival:
            return self.survival(pupa.mass, pupa.genotype)
        else:
            return True

    def survive(self, pupa: hint.pupa) -> None:
        """
        Run the survival model

        Args:
            pupa: the pupa in question

        Effects:
            kills pupa if it fails to survive
        """

        if not self._survive(pupa):
            pupa.die(keyword.survival)

    @classmethod
    def setup(cls, **kwargs) -> 'Pupa':
        """
        Setup the class

        Args:
            **kwargs: simulation input models

        Returns:
            setup class
        """

        if keyword.pupa_survival in kwargs:
            return cls(kwargs[keyword.pupa_survival])
        else:
            return cls()
```

C.4.13 hint.py

```
import typing

import pandas as pd

if typing.TYPE_CHECKING:
    # noinspection PyUnresolvedReferences
    import source.agents.agent    as main_agent
    # noinspection PyUnresolvedReferences
    import source.agents.adult    as main_adult
    # noinspection PyUnresolvedReferences
    import source.agents.egg      as main_egg
    # noinspection PyUnresolvedReferences
    import source.agents.egg_mass as main_egg_mass
    # noinspection PyUnresolvedReferences
    import source.agents.larva    as main_larva
    # noinspection PyUnresolvedReferences
    import source.agents.pupa     as main_pupa

    # noinspection PyUnresolvedReferences
    import source.biomass.gut  as main_gut
    # noinspection PyUnresolvedReferences
    import source.biomass.mass as main_mass

    # noinspection PyUnresolvedReferences
    import source.data.database as main_database
    # noinspection PyUnresolvedReferences
    import source.data.counter  as main_counter

    # noinspection PyUnresolvedReferences
    import source.development.egg   as main_egg_development
    # noinspection PyUnresolvedReferences
    import source.development.larva as main_larva_development
    # noinspection PyUnresolvedReferences
    import source.development.pupa  as main_pupa_development

    # noinspection PyUnresolvedReferences
    import source.forage.cannibalism as main_cannibalism
```

```
38    # noinspection PyUnresolvedReferences
39    import source.forage.egg         as main_egg_forage
40    # noinspection PyUnresolvedReferences
41    import source.forage.larva       as main_larva_forage
42    # noinspection PyUnresolvedReferences
43    import source.forage.plant       as main_plant_forage
44    # noinspection PyUnresolvedReferences
45    import source.forage.target      as main_target
46
47    # noinspection PyUnresolvedReferences
48    import source.migration.emigration  as main_emigration
49    # noinspection PyUnresolvedReferences
50    import source.migration.immigration as main_immigration
51
52    # noinspection PyUnresolvedReferences
53    import source.movement.adult as main_adult_movement
54    # noinspection PyUnresolvedReferences
55    import source.movement.larva as main_larva_movement
56
57    # noinspection PyUnresolvedReferences
58    import source.schedule.actions  as main_actions
59    # noinspection PyUnresolvedReferences
60    import source.schedule.schedule as main_schedule
61    # noinspection PyUnresolvedReferences
62    import source.schedule.step     as main_step
63
64    # noinspection PyUnresolvedReferences
65    import source.reproduction.lay  as main_lay
66    # noinspection PyUnresolvedReferences
67    import source.reproduction.mate as main_mate
68
69    # noinspection PyUnresolvedReferences
70    import source.simulation.behaviors  as main_behaviors
71    # noinspection PyUnresolvedReferences
72    import source.simulation.models     as main_models
73    # noinspection PyUnresolvedReferences
74    import source.simulation.simulation as main_simulation
75
76    # noinspection PyUnresolvedReferences
```

```
77      import source.space.agents      as main_agents
78      # noinspection PyUnresolvedReferences
79      import source.space.environment as main_environment
80      # noinspection PyUnresolvedReferences
81      import source.space.graph       as main_graph
82      # noinspection PyUnresolvedReferences
83      import source.space.location    as main_location
84      # noinspection PyUnresolvedReferences
85      import source.space.space       as main_space
86
87      # noinspection PyUnresolvedReferences
88      import source.survival.adult as main_adult_survival
89      # noinspection PyUnresolvedReferences
90      import source.survival.egg   as main_egg_survival
91      # noinspection PyUnresolvedReferences
92      import source.survival.larva as main_larva_survival
93      # noinspection PyUnresolvedReferences
94      import source.survival.pupa  as main_pupa_survival
95
96  # Agent hints
97  agent      = 'main_agent.Agent'
98  agent_list = typing.List[agent]
99  agent_dict = typing.Dict[str, agent]
100 #
101 # agent_dict = typing.Dict[int, agent]
102
103 egg      = 'main_egg.Egg'
104 egg_mass = 'main_egg_mass.EggMass'
105 larva    = 'main_larva.Larva'
106 pupa     = 'main_pupa.Pupa'
107 adult    = 'main_adult.Adult'
108
109 egg_mass_eggs = 'main_egg_mass.Eggs'
110 egg_masses    = typing.List[egg_mass]
111
112 genotypes = typing.List[str]
113 eggs      = typing.List[egg]
114
115 egg_dict = typing.Dict[str, egg]
```

```
116
117 alleles = typing.Tuple[int, int]
118
119
120 # Space hints
121 #       Location hints
122 sim_key       = typing.Tuple[int]
123 plant_key     = typing.Tuple[int, int]
124 leaf_key      = typing.Tuple[int, int, int]
125 location_key  = typing.Union[sim_key, plant_key, leaf_key]
126 location_keys = typing.List[location_key]
127 locations_key = typing.Dict[int, location_keys]
128
129 locs      = typing.List[int]
130 location  = 'main_location.Location'
131 locations = typing.List[location]
132
133 location_pairs = typing.Tuple[locations, location_keys]
134 location_data  = typing.Tuple[locations, locations_key]
135
136 #       Environment hints
137 init_plant  = typing.Callable[[str], float]
138 environment = 'main_environment.Environment'
139
140 environment_tuple = typing.Tuple[float, init_plant]
141
142 #       Agents hints
143 agent_keys  = typing.List[str]
144 agent_bin   = 'main_agents.AgentBin'
145 agent_bins  = typing.Dict[str, agent_bin]
146 agents_bin  = 'main_agents.AgentsBin'
147 agents_dict = typing.Dict[location_key, agents_bin]
148 agents      = 'main_agents.Agents'
149
150 #       Graph hints
151 vertex_neighborhood = 'main_graph.VertexNeighborhood'
152 vertex_distance     = 'main_graph.VertexDistance'
153
154 graph_neighborhood = 'main_graph.GraphNeighborhood'
```

```
graph_distance     = 'main_graph.GraphDistance'
graph_adjacency    = 'main_graph.Adjacency'
graph              = 'main_graph.Graph'

vertices      = typing.Set[int]
upper_lower   = typing.Tuple[float, float]
distance_pair = typing.Tuple[int, float]

neighborhood_dict = typing.Dict[float, vertices]
neighborhoods     = typing.Dict[int,   vertex_neighborhood]

distance_dict = typing.Dict[int, float]
distances     = typing.Dict[int, vertex_distance]

upper_grid = typing.List[typing.List[int]]
boundary   = typing.Tuple[typing.List[int],
                          typing.List[int],
                          typing.List[int],
                          typing.List[int]]

#       Space hints
graphs = typing.List[graph]

grid_regular_grid_generator  = typing.Tuple[str, int, int, bool]
grid_parallel_grid_generator = typing.Tuple[str, int, int, bool, bool]
grid_generator               = typing.Union[grid_regular_grid_generator,
                                            grid_parallel_grid_generator]
grid_gen                     = typing.Union[graph, grid_generator]
grid_generators              = typing.List[grid_gen]

space = 'main_space.Space'


# Data hints
#       Database Hints
database   = 'main_database.Database'
data_tuple_spacing = typing.Tuple[int]
data_tuple_name    = typing.Tuple[int, str, str]
data_tuple         = typing.Union[data_tuple_spacing, data_tuple_name]
```

```
#       Counter Hints
dataframe      = pd.DataFrame
dataframes     = typing.Dict[str, dataframe]
dataframe_list = typing.List[dataframe]

data_list        = typing.List[int]
data_column      = 'main_counter.DataColumn'
data_column_dict = typing.Dict[str, data_column]
data_columns     = 'main_counter.DataColumns'

counts_dict  = typing.Dict[str, any]
count_dict   = typing.Dict[str, int]
attr_values  = typing.List[str]
attrs        = typing.Dict[str, typing.Tuple[attr_values, bool]]
counter      = 'main_counter.Count'
count_filter = 'main_counter.CountFilter'
counting     = typing.Union[counter, count_filter]
counter_dict = typing.Dict[str, counting]
attr_other   = typing.Union[attrs, bool]
counts       = 'main_counter.Counts'

attr_filter = typing.Tuple[str, attr_values, attr_other]

attrs_dict  = typing.Dict[str, attr_filter]
attrs_depth = typing.Dict[int, attrs_dict]


# Schedule hints
#       Actions hints
action_keys = typing.List[str]
action      = 'main_actions.Action'
action_list = typing.List[action]

actions      = 'main_actions.Actions'
actions_list = typing.List[actions]
actions_dict = typing.Dict[str, action_keys]

#       Step hints
step  = 'main_step.Step'
```

```
steps = typing.List[step]

step_tuple_basic     = typing.Tuple[actions_dict]
step_tuple_repeat    = typing.Tuple[actions_dict, int]
step_tuple_shuffle_0 = typing.Tuple[actions_dict, int, bool]
step_tuple_shuffle_1 = typing.Tuple[actions_dict, int, bool, bool]
step_tuple_reg       = typing.Tuple[actions_dict, int, bool, bool, bool]
step_tuple_loc       = typing.Tuple[actions_dict, int, bool, bool, bool,
                                    bool, int]
step_tuple           = typing.Union[step_tuple_basic,
                                    step_tuple_repeat,
                                    step_tuple_shuffle_0,
                                    step_tuple_shuffle_1,
                                    step_tuple_reg,
                                    step_tuple_loc]
step_tuples           = typing.List[step_tuple]

#       Schedule hints
schedule = 'main_schedule.Schedule'


# Simulation Hints
simulation = 'main_simulation.Simulation'
model      = 'main_models.Model'
models     = 'main_models.Models'
behaviors  = 'main_behaviors.Behaviors'

init_pop  = typing.Tuple[int, int, int]
init_pops = typing.Tuple[init_pop, init_pop, init_pop, init_pop, init_pop]

variable    = typing.Dict[str, float]
bt_variable = typing.Dict[str, variable]


# Biomass Hints
#       Gut Hints
max_gut      = typing.Callable[[float], float]
amount_tuple = typing.Tuple[float, bool]
gut          = 'main_gut.Gut'
```

```
#       Mass Hints
growth = typing.Callable[[float, float, str], float]
mass   = 'main_mass.Mass'


# Survival Hints
survival_egg   = typing.Callable[[float, str, str], bool]
survival_larva = typing.Callable[[float, str, str], bool]
survival_pupa  = typing.Callable[[float, str], bool]
survival_adult = typing.Callable[[float, str], bool]

egg_survival   = 'main_egg_survival.Egg'
larva_survival = 'main_larva_survival.Larva'
pupa_survival  = 'main_pupa_survival.Pupa'
adult_survival = 'main_adult_survival.Adult'


# Development Hints
development_egg   = typing.Callable[[float, int, str], bool]
development_larva = typing.Callable[[float, int, str], bool]
development_pupa  = typing.Callable[[float, int, str], bool]

egg_development   = 'main_egg_development.Egg'
larva_development = 'main_larva_development.Larva'
pupa_development  = 'main_pupa_development.Pupa'


# Movement Hints
movement_larva = typing.Callable[[float, str], float]
movement_adult = typing.Callable[[float, str], float]

larva_movement = 'main_larva_movement.Larva'
adult_movement = 'main_adult_movement.Adult'


# Forage Hints
#       Foraging Hints
forage_plant = typing.Callable[[float, float, str, str], float]
forage_egg   = typing.Callable[[float, float, str],      float]
```

```
forage_larva = typing.Callable[[float, float, str],      float]
loss         = typing.Callable[[float, float, str, str], bool]

plant_forage = 'main_plant_forage.Plant'
egg_forage   = 'main_egg_forage.Egg'
larva_forage = 'main_larva_forage.Larva'
target_loss  = 'main_target.Target'

#       Cannibalism Hints
target    = typing.Union[egg_mass, larva]
targets   = typing.List[target]

fight     = typing.Callable[[float, float], bool]
encounter = typing.Callable[[int, float, str], bool]
radius    = typing.Callable[[float, str], bool]

cannibalism = 'main_cannibalism.Cannibalism'


# Reproduction Hints
#       Lay Hints
fecundity = typing.Callable[[float, int, str], int]
density   = typing.Callable[[int, float, str], bool]

egg_lay = typing.Tuple[egg_masses, int, bool]

lay = 'main_lay.Lay'

#       Mate Hints
mates = typing.List[adult]

mating      = typing.Callable[[int, float, str], bool]
mate_radius = typing.Callable[[float, str], bool]

adult_mate = typing.Union[str, None]

mate = 'main_mate.Mate'


```

```
# Migration Hints
emigration        = 'main_emigration.Emigration'
emigration_list   = typing.List[emigration]
emigrations       = 'main_emigration.Emigrations'
emigration_tuple  = typing.Tuple[float, float, agent_keys]
emigration_tuples = typing.List[emigration_tuple]

immigration        = 'main_immigration.Immigration'
immigration_list   = typing.List[immigration]
immigrations       = 'main_immigration.Immigrations'
immigration_tuple  = typing.Tuple[float, str, str]
immigration_tuples = typing.List[immigration_tuple]
```

C.4.14 keyword.py

```
# Agent_keys
egg       = 'egg'
egg_mass = 'eggMass'
larva     = 'larva'
pupa      = 'pupa'
female    = 'female'
male      = 'male'
mated     = 'mated'

agent_keys = [egg, egg_mass, larva, pupa, female, male, mated]
insect_keys = [egg, larva, pupa, female, male, mated]

adult     = 'adult'
pregnant = 'pregnant'

# Initialization keys
init       = 'init'
immigrant = 'immigrant'

# upper_lower keys
upper = 'upper'
lower = 'lower'


# attributes
genotype = 'genotype'
death     = 'death'


# action_keys
advance_age = 'advance_age'
reset        = 'reset'
grow         = 'grow'
survive      = 'survive'
develop      = 'develop'
move         = 'move'
consume      = 'consume'
```

```
reproduce   = 'reproduce'


# Genotype Keywords
homo_r = 'resistant'
hetero = 'heterozygous'
homo_s = 'susceptible'

genotype_keys = [homo_r, hetero, homo_s]

homo_r_value = 0
hetero_value = 1
homo_s_value = 2

homo_r_allele = 0
homo_s_allele = 1

homo_r_alleles = (homo_r_allele, homo_r_allele)
hetero_alleles = (homo_r_allele, homo_s_allele)
homo_s_alleles = (homo_s_allele, homo_s_allele)

# bt keys
bt     = 'Bt'
not_bt = 'not_Bt'
plant  = 'plant'

# level
egg_level      = 2
egg_mass_level = 2
larva_level    = 2
pupa_level     = 1
adult_level    = 1
bt_level       = 1
plant_level    = 1
leaf_level     = 2

# depth
egg_depth      = 3
egg_mass_depth = 3
```

```
77 larva_depth    = 3
78 pupa_depth     = 2
79 adult_depth    = 2
80 bt_depth       = 2
81 plant_depth    = 2
82
83 # grid_keys
84 hexagon  = 'hexagon'
85 square   = 'square'
86 moore    = 'moore'
87 triangle = 'triangle'
88
89 # death_keys
90 cannibalism = 'cannibalism'
91 emigrate    = 'emigrate'
92 survival    = 'survival'
93 starve      = 'starve'
94 alive       = 'alive'
95 death_keys  = [cannibalism, emigrate, survival, starve, alive]
96
97 death_track = 'death_track'
98
99 # mathematical model keys
100 max_gut = 'max_gut'
101 growth  = 'growth'
102
103 init_num      = 'init_num'
104 init_mass     = 'init_mass'
105 init_juvenile = 'init_juvenile'
106 init_mature   = 'init_mature'
107 init_plant    = 'init_plant'
108 init_sex      = 'init_sex'
109
110 egg_development   = 'egg_development'
111 larva_development = 'larva_development'
112 pupa_development  = 'pupa_development'
113
114 egg_forage   = 'egg_forage'
115 larva_forage = 'larva_forage'
```

```
116 plant_forage = 'plant_forage'
117 loss         = 'loss'
118
119 fight     = 'fight'
120 encounter = 'encounter'
121 radius    = 'radius'
122
123 larva_movement = 'larva_movement'
124 adult_movement = 'adult_movement'
125
126 egg_survival   = 'egg_survival'
127 larva_survival = 'larva_survival'
128 pupa_survival  = 'pupa_survival'
129 adult_survival = 'adult_survival'
130
131 trials    = 'trials'
132 fecundity = 'fecundity'
133 density   = 'density'
134
135 mating      = 'mating'
136 mate_radius = 'mate_radius'
137
138 lifetime_male   = 'lifetime_male'
139 lifetime_female = 'lifetime_female'
140 limited         = 'limited'
141
142 required_inputs = [max_gut, growth, init_num, init_mass, init_juvenile,
143                    init_mature, init_plant, init_sex,
144                    lifetime_female, lifetime_male, limited]
```

C.5 test

```
FallArmyworm
└── test
    ├── test_agents
    ├── test_biomass
    ├── test_data
    ├── test_development
    ├── test_forage
    ├── test_migration
    ├── test_movement
    ├── test_reproduction
    ├── test_schedule
    ├── test_simulation
    ├── test_space
    └── test_survival
```

C.5.1 test_agents

```
FallArmyworm
└── test
    └── test_agents
        ├── test_adult.py
        ├── test_agent.py
        ├── test_egg.py
        ├── test_egg_mass.py
        ├── test_insect.py
        ├── test_larva.py
        └── test_pupa.py
```

C.5.1.1 test_adult.py

```
import unittest      as ut
import unittest.mock as mk

import dataclasses as dclass
import itertools   as i_tools

import source.keyword as keyword

import source.agents.agent    as agent
import source.agents.adult    as adult
import source.agents.egg_mass as egg_mass
import source.agents.insect   as insect
import source.agents.larva    as larva
import source.agents.pupa     as agent_pupa

import source.simulation.behaviors  as behaviors
import source.simulation.models     as models
import source.simulation.simulation as simulation

import source.space.agents   as agents
import source.space.location as location
import source.space.space    as space

import source.movement.adult    as movement
import source.reproduction.lay  as lay
import source.reproduction.mate as mate
import source.survival.adult    as survival


class BehaviorsTest(behaviors.Behaviors):
    """Class to add dynamic values for tests"""

    survive_adult = mk.create_autospec(survival.Adult, spec_set=True)
    move_adult    = mk.create_autospec(movement.Adult, spec_set=True)
    lay           = mk.create_autospec(lay.Lay,        spec_set=True)
    mate          = mk.create_autospec(mate.Mate,      spec_set=True)

```

```
38
39 class SimulationTest(simulation.Simulation):
40     """Class to add dynamic values for tests"""
41
42     agents    = mk.create_autospec(agents.Agents, spec_set=True)
43     behaviors = mk.create_autospec(BehaviorsTest, spec_set=True)
44     space     = mk.create_autospec(space.Space,   spec_set=True)
45     models    = mk.create_autospec(models.Models, spec_set=True)
46
47
48 class PupaTest(agent_pupa.Pupa):
49     """Class to add dynamic values for tests"""
50
51     unique_id  = mk.MagicMock(spec=str)
52     simulation = mk.create_autospec(SimulationTest,    spec_set=True)
53     location   = mk.create_autospec(location.Location, spec_set=True)
54     mass       = mk.MagicMock(spec=float)
55     genotype   = mk.MagicMock(spec=str)
56
57
58 class AdultTest(adult.Adult):
59     """Class to add dynamic values for tests"""
60
61     genotype = mk.MagicMock(spec=str)
62
63
64 class TestAdult(ut.TestCase):
65     """test base Adult class"""
66
67     def setUp(self):
68         """Setup the tests"""
69
70         self.agent_key = mk.MagicMock(spec=str)
71         self.unique_id = mk.MagicMock(spec=str)
72         self.alive     = mk.MagicMock(spec=bool)
73         self.mass      = mk.MagicMock(spec=float)
74         self.genotype  = mk.MagicMock(spec=str)
75         self.age       = mk.MagicMock(spec=int)
76         self.death     = mk.MagicMock(spec=str)
```

```
77
78          self.simulation = mk.create_autospec(SimulationTest,    spec_set=True)
79          self.location   = mk.create_autospec(location.Location, spec_set=True)
80
81          self.num_eggs = mk.MagicMock(spec=int)
82          self.mate     = mk.MagicMock(spec=str)
83
84          self.survival   = mk.create_autospec(survival.Adult, spec_set=True)
85          self.movement = mk.create_autospec(movement.Adult,   spec_set=True)
86          self.lay      = mk.create_autospec(lay.Lay,          spec_set=True)
87          self.mating   = mk.create_autospec(mate.Mate,        spec_set=True)
88
89          self.Adult = adult.Adult(self.agent_key,
90                                   self.unique_id,
91                                   self.simulation,
92                                   self.location,
93                                   self.alive,
94                                   self.mass,
95                                   self.genotype,
96                                   self.age,
97                                   self.death,
98                                   self.num_eggs,
99                                   self.mate,
100                                  self.survival,
101                                  self.movement,
102                                  self.lay,
103                                  self.mating)
104
105     def test___init__(self):
106         """test __init__ for class"""
107
108         self.assertIsInstance(self.Adult, agent.Agent)
109         self.assertIsInstance(self.Adult, insect.Insect)
110         self.assertIsInstance(self.Adult, adult.Adult)
111
112         self.assertEqual(self.Adult.agent_key,  self.agent_key)
113         self.assertEqual(self.Adult.unique_id,  self.unique_id)
114         self.assertEqual(self.Adult.simulation, self.simulation)
115         self.assertEqual(self.Adult.location,   self.location)
```

```
        self.assertEqual(self.Adult.alive,      self.alive)

        self.assertEqual(self.Adult.mass,     self.mass)
        self.assertEqual(self.Adult.genotype, self.genotype)
        self.assertEqual(self.Adult.age,      self.age)
        self.assertEqual(self.Adult.death,    self.death)

        self.assertEqual(self.Adult.num_eggs, self.num_eggs)
        self.assertEqual(self.Adult.mate,     self.mate)
        self.assertEqual(self.Adult.survival, self.survival)
        self.assertEqual(self.Adult.movement, self.movement)
        self.assertEqual(self.Adult.lay,      self.lay)
        self.assertEqual(self.Adult.mating,   self.mating)

        # noinspection PyTypeChecker
        self.assertIsInstance(self.Adult._age_count, i_tools.count)
        # noinspection PyTypeChecker
        self.assertEqual(next(self.Adult._age_count),
                         next(i_tools.count(self.age + 1)))

        # noinspection PyTypeChecker
        self.assertIsInstance(self.Adult._id_count, i_tools.count)
        # noinspection PyTypeChecker
        self.assertEqual(next(self.Adult._id_count), next(i_tools.count()))

        self.assertTrue(dclass.is_dataclass(self.Adult))

    def test_survive(self):
        """test run survive behavior"""

        self.assertEqual(self.Adult.survive(), [])
        self.assertEqual(self.survival.survive.call_args_list,
                         [mk.call(self.Adult)])

    def test_move(self):
        """test run move behavior"""

        # Adult is not alive
        self.Adult.alive = False
```

```
155         self.assertEqual(self.Adult.move(), [])
156         self.assertEqual(self.movement.move.call_args_list, [])
157 
158         # Adult is alive
159         self.Adult.alive = True
160         self.assertEqual(self.Adult.move(), [])
161         self.assertEqual(self.movement.move.call_args_list,
162                          [mk.call(self.Adult)])
163 
164     def test_new_unique_id(self):
165         """test create a new unique_id for an egg_mass"""
166 
167         self.assertEqual(self.Adult.new_unique_id(),
168                          str(self.unique_id) + '_0' + keyword.egg_mass)
169         self.assertEqual(self.Adult.new_unique_id(),
170                          str(self.unique_id) + '_1' + keyword.egg_mass)
171         self.assertEqual(self.Adult.new_unique_id(),
172                          str(self.unique_id) + '_2' + keyword.egg_mass)
173 
174     def test_new_egg_location(self):
175         """test create a new egg_location"""
176 
177         self.simulation.space = mk.create_autospec(space.Space, spec_set=True)
178 
179         self.assertGreater(keyword.egg_depth - keyword.adult_depth, 0)
180         repeat = keyword.egg_depth - keyword.adult_depth
181 
182         self.assertEqual(self.Adult.new_egg_location(),
183                          self.simulation.space.extend_location.return_value)
184         self.assertEqual(len(self.simulation.space.
185                              extend_location.call_args_list), repeat)
186         call = self.simulation.space.extend_location.call_args_list.pop(0)
187         self.assertEqual(call,
188                          mk.call(self.location))
189         for call in self.simulation.space.extend_location.call_args_list:
190             self.assertEqual(call,
191                              mk.call(self.simulation.space.
192                                      extend_location.return_value))
193         self.assertEqual(len(self.simulation.space.
```

```
                            extend_location.call_args_list),
                        repeat - 1)

    def test_population(self):
        """test get the population of the plant"""

        self.simulation.agents = \
            mk.create_autospec(agents.Agents, spec_set=True)
        self.simulation.agents.__getitem__.return_value = \
            mk.create_autospec(agents.AgentsBin, spec_set=True)

        egg_bin = mk.create_autospec(agents.AgentBin, spec_set=True)
        larva_bin = mk.create_autospec(agents.AgentBin, spec_set=True)

        eggs   = [mk.create_autospec(egg_mass.EggMass, spec_set=True)
                  for _ in range(3)]
        larvae = [mk.create_autospec(larva.Larva, spec_set=True)
                  for _ in range(3)]
        egg_bin.  agents = eggs
        larva_bin.agents = larvae
        self.simulation.agents.__getitem__.return_value.\
            __getitem__.side_effect = [egg_bin, larva_bin,
                                       egg_bin, larva_bin]
        # Test calls
        with mk.patch.object(adult, 'len') as mkLen:
            self.assertEqual(self.Adult.population(),
                             mkLen.return_value.__add__.return_value)
            self.assertEqual(mkLen.return_value.__add__.call_args_list,
                             [mk.call(mkLen.return_value)])
            self.assertEqual(mkLen.call_args_list,
                             [mk.call(eggs), mk.call(larvae)])
            self.assertEqual(self.simulation.agents.__getitem__.return_value.
                                __getitem__.call_args_list,
                             [mk.call(keyword.egg_mass),
                              mk.call(keyword.larva)])
            self.assertEqual(self.simulation.agents.__getitem__.call_args_list,
                             [mk.call(self.location.location_key)])

        # Test practical
```

```
233         self.assertEqual(self.Adult.population(), 6)
234
235     def test_set_mate(self):
236         """test set a mate"""
237
238         self.simulation.models = \
239             mk.create_autospec(models.Models, spec_set=True)
240
241         self.simulation.models.__getitem__.side_effect = [False, False,
242                                                           False, True,
243                                                           True]
244
245         this = mk.create_autospec(AdultTest, spec_set=True)
246
247         with mk.patch.object(adult.Adult, 'deactivate',
248                              autospec=True) as mkDeactivate:
249             with mk.patch.object(adult.Adult, 'transition',
250                                  autospec=True) as mkTransition:
251                 # Not male
252                 self.Adult.set_mate(this)
253                 self.assertEqual(self.Adult.mate, this.genotype)
254                 self.assertEqual(mkDeactivate.call_args_list, [])
255                 self.assertEqual(mkTransition.call_args_list, [])
256                 self.assertEqual(self.simulation.models.
257                                  __getitem__.call_args_list, [])
258
259                 self.Adult.mate = self.mate
260                 # Is Male
261                 self.Adult.agent_key = keyword.male
262                 #   Unlimited male mating
263                 self.Adult.set_mate(this)
264                 self.assertEqual(self.Adult.mate, self.mate)
265                 self.assertEqual(mkDeactivate.call_args_list, [])
266                 self.assertEqual(mkTransition.call_args_list, [])
267                 self.assertEqual(self.simulation.models.
268                                  __getitem__.call_args_list,
269                                  [mk.call(keyword.lifetime_male),
270                                   mk.call(keyword.limited)])
271                 self.simulation.models.reset_mock()
```

```
                #   Limited male mating
                self.Adult.set_mate(this)
                self.assertEqual(self.Adult.mate, self.mate)
                self.assertEqual(mkDeactivate.call_args_list, [])
                self.assertEqual(mkTransition.call_args_list,
                                 [mk.call(self.Adult, keyword.mated)])
                self.assertEqual(self.simulation.models.
                                     __getitem__.call_args_list,
                                 [mk.call(keyword.lifetime_male),
                                  mk.call(keyword.limited)])
                self.simulation.models.reset_mock()
                mkTransition.reset_mock()
                #   Lifetime male mating
                self.Adult.set_mate(this)
                self.assertEqual(self.Adult.mate, self.mate)
                self.assertEqual(mkDeactivate.call_args_list,
                                 [mk.call(self.Adult)])
                self.assertEqual(mkTransition.call_args_list, [])
                self.assertEqual(self.simulation.models.
                                 __getitem__.call_args_list,
                                 [mk.call(keyword.lifetime_male)])

    def test__location_keys(self):
        """test get the location keys for reproduction"""

        kwargs = {'test': mk.MagicMock()}

        vertices = {mk.MagicMock(spec=int) for _ in range(3)}

        locs          = []
        location_keys = []
        for _ in vertices:
            loc = mk.create_autospec(location.Location, spec_set=True)
            location_keys.append(loc.location_key)
            locs.append(loc)

        self.location.copy.side_effect = locs

        with mk.patch.object(adult.Adult, 'vertices',
```

```
                            autospec=True) as mkVertices:
            mkVertices.return_value = vertices

            self.assertEqual(self.Adult._location_keys(**kwargs), location_keys)
            self.assertEqual(mkVertices.call_args_list,
                             [mk.call(self.Adult, **kwargs)])
            for index, vertex in enumerate(vertices):
                self.assertEqual(locs[index].__setitem__.call_args_list,
                                 [mk.call(keyword.adult_level, vertex)])
                self.assertEqual(self.location.copy.call_args_list[index],
                                 [mk.call()])
            self.assertEqual(len(self.location.copy.call_args_list), 3)

    def test_mates(self):
        """test get the mates"""

        kwargs = {'test': mk.MagicMock()}

        self.simulation.agents = mk.create_autospec(agents.Agents,
                                                    spec_set=True)

        location_keys = []
        agents_bins   = []
        mates         = []
        for _ in range(3):
            location_key = mk.MagicMock(spec=tuple)
            agents_bin   = mk.create_autospec(agents.AgentsBin, spec_set=True)
            male_bin     = mk.create_autospec(agents.AgentBin, spec_set=True)
            adults       = []
            for _ in range(3):
                adults.append(mk.create_autospec(adult.Adult, spec_set=True))
            male_bin.agents = adults
            mates.extend(adults)

            location_keys.append(location_key)
            agents_bin.__getitem__.return_value = male_bin
            agents_bins.append(agents_bin)

        self.simulation.agents.__getitem__.side_effect = agents_bins
```

```
350
351         with mk.patch.object(adult.Adult, '_location_keys',
352                              autospec=True) as mkKeys:
353             mkKeys.return_value = location_keys
354
355             new = self.Adult.mates(**kwargs)
356             self.assertEqual(mates, new)
357
358             self.assertEqual(mkKeys.call_args_list,
359                              [mk.call(self.Adult, **kwargs)])
360             for index, call in enumerate(self.simulation.agents.
361                                          __getitem__.call_args_list):
362                 self.assertEqual(call,
363                                  mk.call(location_keys[index]))
364             self.assertEqual(len(self.simulation.agents.
365                                  __getitem__.call_args_list), 3)
366             for index, agent_bin in enumerate(agents_bins):
367                 self.assertEqual(agent_bin.__getitem__.call_args_list,
368                                  [mk.call(keyword.male)])
369
370     def test_reproduce(self):
371         """test reproduce the agents"""
372
373         eggs = [mk.MagicMock(spec=egg_mass.EggMass)
374                 for _ in range(3)]
375         self.lay.lay.return_value = eggs
376
377         # Adult is not alive
378         self.Adult.alive = False
379         self.assertEqual(self.Adult.reproduce(), [])
380         self.assertEqual(self.mating.mate.call_args_list, [])
381         self.assertEqual(self.lay.lay.call_args_list, [])
382         for this in eggs:
383             self.assertEqual(this.activate.call_args_list, [])
384
385         # Adult is alive
386         self.Adult.alive = True
387         #   Test has a mate
388         self.assertEqual(self.Adult.reproduce(), [])
```

```
389         self.assertEqual(self.lay.lay.call_args_list,
390                          [mk.call(self.Adult)])
391         self.assertEqual(self.mating.mate.call_args_list, [])
392         for this in eggs:
393             self.assertEqual(this.activate.call_args_list,
394                              [mk.call()])
395             this.reset_mock()
396 
397         #   Test no mate
398         self.lay.reset_mock()
399         self.Adult.mate = None
400         self.assertEqual(self.Adult.reproduce(), [])
401         self.assertEqual(self.mating.mate.call_args_list,
402                          [mk.call(self.Adult)])
403         self.assertEqual(self.lay.lay.call_args_list, [])
404         for this in eggs:
405             self.assertEqual(this.activate.call_args_list, [])
406 
407     def test_reset(self):
408         """test reset the mock"""
409 
410         self.simulation.models = \
411             mk.create_autospec(models.Models, spec_set=True)
412 
413         self.simulation.models.__getitem__.side_effect = [True, False]
414 
415         with mk.patch.object(adult.Adult, 'transition',
416                              autospec=True) as mkTransition:
417             # Not female or mated
418             self.assertEqual(self.Adult.reset(), [])
419             self.assertEqual(self.Adult.mate, self.mate)
420             self.assertEqual(self.Adult.num_eggs, self.num_eggs)
421             self.assertEqual(self.lay.reset.call_args_list, [])
422             self.assertEqual(mkTransition.call_args_list, [])
423             self.assertEqual(self.simulation.models.
424                              __getitem__.call_args_list, [])
425 
426             # Is female
427             self.Adult.agent_key = keyword.female
```

```
428             #       Female lifetime mate
429             self.assertEqual(self.Adult.reset(), [])
430             self.assertEqual(self.Adult.mate, self.mate)
431             self.assertEqual(self.Adult.num_eggs,
432                              self.lay.reset.return_value)
433             self.assertEqual(self.lay.reset.call_args_list,
434                              [mk.call(self.Adult)])
435             self.assertEqual(mkTransition.call_args_list, [])
436             self.assertEqual(self.simulation.models.
437                                  __getitem__.call_args_list,
438                              [mk.call(keyword.lifetime_female)])
439             #       Not female lifetime mate
440             self.Adult.num_eggs = self.num_eggs
441             self.simulation.models.reset_mock()
442             self.lay.reset_mock()
443             self.assertEqual(self.Adult.reset(), [])
444             self.assertEqual(self.Adult.mate, None)
445             self.assertEqual(self.Adult.num_eggs,
446                              self.lay.reset.return_value)
447             self.assertEqual(self.lay.reset.call_args_list,
448                              [mk.call(self.Adult)])
449             self.assertEqual(mkTransition.call_args_list, [])
450             self.assertEqual(self.simulation.models.
451                                  __getitem__.call_args_list,
452                              [mk.call(keyword.lifetime_female)])
453
454             self.Adult.num_eggs = self.num_eggs
455             self.Adult.mate     = self.mate
456             self.simulation.models.reset_mock()
457             self.lay.reset_mock()
458             # Is mated and not alive
459             self.Adult.alive = False
460             self.Adult.agent_key = keyword.mated
461             self.assertEqual(self.Adult.reset(), [])
462             self.assertEqual(self.Adult.mate, self.mate)
463             self.assertEqual(self.Adult.num_eggs, self.num_eggs)
464             self.assertEqual(self.lay.reset.call_args_list, [])
465             self.assertEqual(mkTransition.call_args_list, [])
466             self.assertEqual(self.simulation.models.
```

```
                            __getitem__.call_args_list, [])
        # Is mated and alive
        self.Adult.alive = True
        self.Adult.agent_key = keyword.mated
        self.assertEqual(self.Adult.reset(), [])
        self.assertEqual(self.Adult.mate, self.mate)
        self.assertEqual(self.Adult.num_eggs, self.num_eggs)
        self.assertEqual(self.lay.reset.call_args_list, [])
        self.assertEqual(mkTransition.call_args_list,
                         [mk.call(self.Adult, keyword.male)])
        self.assertEqual(self.simulation.models.
                             __getitem__.call_args_list, [])

    def test__set_sex(self):
        """test get the sex of the adult"""

        self.simulation.models = mk.create_autospec(models.Models,
                                                    spec_set=True)
        self.simulation.models.__getitem__.return_value.side_effect = [True,
                                                                        False]

        # With mate
        self.Adult._set_sex()
        self.assertEqual(self.Adult.agent_key, keyword.female)
        self.assertEqual(self.Adult.num_eggs,
                         self.lay.reset.return_value)
        self.assertEqual(self.lay.reset.call_args_list,
                         [mk.call(self.Adult)])
        self.assertEqual(self.simulation.models.__getitem__.call_args_list, [])
        self.lay.reset_mock()
        self.Adult.num_eggs = self.num_eggs

        # No mate
        self.Adult.mate = None
        #       Set female
        self.Adult._set_sex()
        self.assertEqual(self.Adult.agent_key, keyword.female)
        self.assertEqual(self.Adult.num_eggs,  self.num_eggs)
        self.assertEqual(self.simulation.models.
```

```
                            __getitem__.return_value.call_args_list,
                        [mk.call(self.genotype)])
        self.assertEqual(self.simulation.models.__getitem__.call_args_list,
                        [mk.call(keyword.init_sex)])
        self.assertEqual(self.lay.reset.call_args_list, [])
        self.simulation.models.reset_mock()
        #       Set male
        self.Adult._set_sex()
        self.assertEqual(self.Adult.agent_key, keyword.male)
        self.assertEqual(self.Adult.num_eggs,  self.num_eggs)
        self.assertEqual(self.simulation.models.
                            __getitem__.return_value.call_args_list,
                        [mk.call(self.genotype)])
        self.assertEqual(self.simulation.models.__getitem__.call_args_list,
                        [mk.call(keyword.init_sex)])
        self.assertEqual(self.lay.reset.call_args_list, [])

    def test_initialize(self):
        """test initialize the agent"""

        self.simulation.behaviors = mk.create_autospec(BehaviorsTest,
                                                       spec_set=True)
        self.simulation.behaviors.survive_adult = self.survival
        self.simulation.behaviors.move_adult    = self.movement
        self.simulation.behaviors.lay           = self.lay
        self.simulation.behaviors.mate          = self.mating

        with mk.patch.object(adult.Adult, '_set_sex', autospec=True) as mkSex:
            # Test regular
            self.Adult = adult.Adult.initialize(self.unique_id,
                                                self.simulation,
                                                self.location,
                                                self.mass,
                                                self.genotype,
                                                self.mate)

            self.assertIsInstance(self.Adult, agent.Agent)
            self.assertIsInstance(self.Adult, insect.Insect)
            self.assertIsInstance(self.Adult, adult.Adult)
```

```
545
546             self.assertEqual(self.Adult.agent_key, '')
547             self.assertEqual(self.Adult.alive,     True)
548             self.assertEqual(self.Adult.age,       0)
549             self.assertEqual(self.Adult.death,     keyword.alive)
550             self.assertEqual(self.Adult.num_eggs,  0)
551             self.assertEqual(mkSex.call_args_list,
552                              [mk.call(self.Adult)])
553
554             self.assertEqual(self.Adult.unique_id,  self.unique_id)
555             self.assertEqual(self.Adult.simulation, self.simulation)
556             self.assertEqual(self.Adult.location,   self.location)
557
558             self.assertEqual(self.Adult.mass,     self.mass)
559             self.assertEqual(self.Adult.genotype, self.genotype)
560
561             self.assertEqual(self.Adult.mate,     self.mate)
562             self.assertEqual(self.Adult.survival, self.survival)
563             self.assertEqual(self.Adult.movement, self.movement)
564             self.assertEqual(self.Adult.lay,      self.lay)
565             self.assertEqual(self.Adult.mating,   self.mating)
566
567             # noinspection PyTypeChecker
568             self.assertIsInstance(self.Adult._age_count, i_tools.count)
569             # noinspection PyTypeChecker
570             self.assertEqual(next(self.Adult._age_count),
571                              next(i_tools.count(1)))
572
573             # noinspection PyTypeChecker
574             self.assertIsInstance(self.Adult._id_count, i_tools.count)
575             # noinspection PyTypeChecker
576             self.assertEqual(next(self.Adult._id_count), next(i_tools.count()))
577
578             self.assertTrue(dclass.is_dataclass(self.Adult))
579
580             mkSex.reset_mock()
581             # Test no mate
582             self.Adult = adult.Adult.initialize(self.unique_id,
583                                                 self.simulation,
```

```
                                            self.location,
                                            self.mass,
                                            self.genotype)

        self.assertIsInstance(self.Adult, agent.Agent)
        self.assertIsInstance(self.Adult, insect.Insect)
        self.assertIsInstance(self.Adult, adult.Adult)

        self.assertEqual(self.Adult.agent_key, '')
        self.assertEqual(self.Adult.alive,     True)
        self.assertEqual(self.Adult.age,       0)
        self.assertEqual(self.Adult.death,     keyword.alive)
        self.assertEqual(self.Adult.num_eggs,  0)
        self.assertEqual(mkSex.call_args_list,
                         [mk.call(self.Adult)])

        self.assertEqual(self.Adult.unique_id,  self.unique_id)
        self.assertEqual(self.Adult.simulation, self.simulation)
        self.assertEqual(self.Adult.location,   self.location)

        self.assertEqual(self.Adult.mass,     self.mass)
        self.assertEqual(self.Adult.genotype, self.genotype)

        self.assertEqual(self.Adult.mate,     None)
        self.assertEqual(self.Adult.survival, self.survival)
        self.assertEqual(self.Adult.movement, self.movement)
        self.assertEqual(self.Adult.lay,      self.lay)
        self.assertEqual(self.Adult.mating,   self.mating)

        # noinspection PyTypeChecker
        self.assertIsInstance(self.Adult._age_count, i_tools.count)
        # noinspection PyTypeChecker
        self.assertEqual(next(self.Adult._age_count),
                         next(i_tools.count(1)))

        # noinspection PyTypeChecker
        self.assertIsInstance(self.Adult._id_count, i_tools.count)
        # noinspection PyTypeChecker
        self.assertEqual(next(self.Adult._id_count), next(i_tools.count()))
```

```
623
624             self.assertTrue(dclass.is_dataclass(self.Adult))
625
626     def test_setup(self):
627         """test setup an initial adult"""
628
629         self.simulation.behaviors = mk.create_autospec(BehaviorsTest,
630                                                        spec_set=True)
631         self.simulation.behaviors.survive_adult = self.survival
632         self.simulation.behaviors.move_adult    = self.movement
633         self.simulation.behaviors.lay           = self.lay
634         self.simulation.behaviors.mate          = self.mating
635         self.simulation.space  = mk.create_autospec(space.Space, spec_set=True)
636         self.simulation.models = mk.create_autospec(models.Models,
637                                                     spec_set=True)
638
639         unique_id_num = mk.MagicMock(spec=int)
640         initial_key   = mk.MagicMock(spec=str)
641
642         with mk.patch.object(adult.Adult, '_set_sex', autospec=True) as mkSex:
643             # Test regular
644             self.Adult = adult.Adult.setup(unique_id_num,
645                                            initial_key,
646                                            self.simulation,
647                                            self.genotype,
648                                            self.mate)
649
650             self.assertIsInstance(self.Adult, agent.Agent)
651             self.assertIsInstance(self.Adult, insect.Insect)
652             self.assertIsInstance(self.Adult, adult.Adult)
653
654             self.assertEqual(self.Adult.unique_id,
655                              str(initial_key)   +
656                              str(unique_id_num) +
657                              keyword.adult)
658             self.assertEqual(self.Adult.location,
659                              self.simulation.space.new_location.return_value)
660             self.assertEqual(self.simulation.space.new_location.call_args_list,
661                              [mk.call(keyword.adult_depth)])
```

```
662             self.assertEqual(self.Adult.mass,
663                              self.simulation.models.
664                              __getitem__.return_value.return_value)
665             self.assertEqual(self.simulation.models.
666                              __getitem__.return_value.call_args_list,
667                              [mk.call(self.genotype)])
668             self.assertEqual(self.simulation.models.
669                              __getitem__.call_args_list,
670                              [mk.call(keyword.init_mature)])
671
672             self.assertEqual(self.Adult.agent_key, '')
673             self.assertEqual(self.Adult.alive,     True)
674             self.assertEqual(self.Adult.age,       0)
675             self.assertEqual(self.Adult.death,     keyword.alive)
676             self.assertEqual(self.Adult.num_eggs,  0)
677             self.assertEqual(mkSex.call_args_list,
678                              [mk.call(self.Adult)])
679
680             self.assertEqual(self.Adult.simulation, self.simulation)
681
682             self.assertEqual(self.Adult.genotype, self.genotype)
683
684             self.assertEqual(self.Adult.mate,     self.mate)
685             self.assertEqual(self.Adult.survival, self.survival)
686             self.assertEqual(self.Adult.movement, self.movement)
687             self.assertEqual(self.Adult.lay,      self.lay)
688             self.assertEqual(self.Adult.mating,   self.mating)
689
690             # noinspection PyTypeChecker
691             self.assertIsInstance(self.Adult._age_count, i_tools.count)
692             # noinspection PyTypeChecker
693             self.assertEqual(next(self.Adult._age_count),
694                              next(i_tools.count(1)))
695
696             self.assertTrue(dclass.is_dataclass(self.Adult))
697
698             mkSex.reset_mock()
699             self.simulation.space.reset_mock()
700             self.simulation.models.reset_mock()
```

```
701             # Test no mate
702             self.Adult = adult.Adult.setup(unique_id_num,
703                                            initial_key,
704                                            self.simulation,
705                                            self.genotype,
706                                            self.mate)
707
708             self.assertIsInstance(self.Adult, agent.Agent)
709             self.assertIsInstance(self.Adult, insect.Insect)
710             self.assertIsInstance(self.Adult, adult.Adult)
711
712             self.assertEqual(self.Adult.unique_id,
713                              str(initial_key)   +
714                              str(unique_id_num) +
715                              keyword.adult)
716             self.assertEqual(self.Adult.location,
717                              self.simulation.space.new_location.return_value)
718             self.assertEqual(self.simulation.space.new_location.call_args_list,
719                              [mk.call(keyword.adult_depth)])
720             self.assertEqual(self.Adult.mass,
721                              self.simulation.models.
722                              __getitem__.return_value.return_value)
723             self.assertEqual(self.simulation.models.
724                              __getitem__.return_value.call_args_list,
725                              [mk.call(self.genotype)])
726             self.assertEqual(self.simulation.models.
727                              __getitem__.call_args_list,
728                              [mk.call(keyword.init_mature)])
729
730             self.assertEqual(self.Adult.agent_key, '')
731             self.assertEqual(self.Adult.alive,     True)
732             self.assertEqual(self.Adult.age,       0)
733             self.assertEqual(self.Adult.death,     keyword.alive)
734             self.assertEqual(self.Adult.num_eggs,  0)
735             self.assertEqual(mkSex.call_args_list,
736                              [mk.call(self.Adult)])
737
738             self.assertEqual(self.Adult.simulation, self.simulation)
739
```

```
            self.assertEqual(self.Adult.genotype, self.genotype)

            self.assertEqual(self.Adult.mate,     self.mate)
            self.assertEqual(self.Adult.survival, self.survival)
            self.assertEqual(self.Adult.movement, self.movement)
            self.assertEqual(self.Adult.lay,      self.lay)
            self.assertEqual(self.Adult.mating,   self.mating)

            # noinspection PyTypeChecker
            self.assertIsInstance(self.Adult._age_count, i_tools.count)
            # noinspection PyTypeChecker
            self.assertEqual(next(self.Adult._age_count),
                             next(i_tools.count(1)))

            # noinspection PyTypeChecker
            self.assertIsInstance(self.Adult._id_count, i_tools.count)
            # noinspection PyTypeChecker
            self.assertEqual(next(self.Adult._id_count), next(i_tools.count()))

            self.assertTrue(dclass.is_dataclass(self.Adult))

    def test_advance(self):
        """test advance a pupa into an adult"""

        self.simulation.behaviors = mk.create_autospec(BehaviorsTest,
                                                       spec_set=True)
        self.simulation.behaviors.survive_adult = self.survival
        self.simulation.behaviors.move_adult    = self.movement
        self.simulation.behaviors.lay           = self.lay
        self.simulation.behaviors.mate          = self.mating

        pupa = mk.create_autospec(PupaTest, spec_set=True)
        pupa.unique_id  = self.unique_id
        pupa.simulation = self.simulation
        pupa.location   = self.location
        pupa.mass       = self.mass
        pupa.genotype   = self.genotype

        with mk.patch.object(adult.Adult, '_set_sex', autospec=True) as mkSex:
```

```
            self.Adult = adult.Adult.advance(pupa)

            self.assertIsInstance(self.Adult, agent.Agent)
            self.assertIsInstance(self.Adult, insect.Insect)
            self.assertIsInstance(self.Adult, adult.Adult)

            self.assertEqual(self.Adult.agent_key, '')
            self.assertEqual(self.Adult.alive,     True)
            self.assertEqual(self.Adult.age,       0)
            self.assertEqual(self.Adult.death,     keyword.alive)
            self.assertEqual(self.Adult.num_eggs,  0)
            self.assertEqual(mkSex.call_args_list,
                             [mk.call(self.Adult)])

            self.assertEqual(self.Adult.unique_id,  self.unique_id)
            self.assertEqual(self.Adult.simulation, self.simulation)
            self.assertEqual(self.Adult.location,   self.location)

            self.assertEqual(self.Adult.mass,     self.mass)
            self.assertEqual(self.Adult.genotype, self.genotype)

            self.assertEqual(self.Adult.mate,     None)
            self.assertEqual(self.Adult.survival, self.survival)
            self.assertEqual(self.Adult.movement, self.movement)
            self.assertEqual(self.Adult.lay,      self.lay)
            self.assertEqual(self.Adult.mating,   self.mating)

            # noinspection PyTypeChecker
            self.assertIsInstance(self.Adult._age_count, i_tools.count)
            # noinspection PyTypeChecker
            self.assertEqual(next(self.Adult._age_count),
                             next(i_tools.count(1)))

            # noinspection PyTypeChecker
            self.assertIsInstance(self.Adult._id_count, i_tools.count)
            # noinspection PyTypeChecker
            self.assertEqual(next(self.Adult._id_count), next(i_tools.count()))

            self.assertTrue(dclass.is_dataclass(self.Adult))
```

C.5.1.2 test_agent.py

```
1  import unittest      as ut
2  import unittest.mock as mk
3
4  import dataclasses as dclass
5
6  import source.keyword as keyword
7
8  import source.agents.agent as agent
9
10 import source.simulation.simulation as simulation
11
12 import source.space.agents   as agents
13 import source.space.location as location
14 import source.space.space    as space
15
16
17 class SimulationTest(simulation.Simulation):
18     """Class to add dynamic values for tests"""
19
20     agents = mk.create_autospec(agents.Agents, spec_set=True)
21     space  = mk.create_autospec(space.Space,   spec_set=True)
22
23
24 class TestAgent(ut.TestCase):
25     """test base Agent class"""
26
27     def setUp(self):
28         """Setup the tests"""
29
30         self.agent_key = mk.MagicMock(spec=str)
31         self.unique_id = mk.MagicMock(spec=str)
32         self.alive     = mk.MagicMock(spec=bool)
33
34         self.simulation = mk.create_autospec(SimulationTest,    spec_set=True)
35         self.location   = mk.create_autospec(location.Location, spec_set=True)
36
37         self.Agent = agent.Agent(self.agent_key,
```

```
38                                  self.unique_id,
39                                  self.simulation,
40                                  self.location,
41                                  self.alive)
42
43      def test___init__(self):
44          """test __init__ for class"""
45
46          self.assertIsInstance(self.Agent, agent.Agent)
47
48          self.assertEqual(self.Agent.agent_key,  self.agent_key)
49          self.assertEqual(self.Agent.unique_id,  self.unique_id)
50          self.assertEqual(self.Agent.simulation, self.simulation)
51          self.assertEqual(self.Agent.location,   self.location)
52          self.assertEqual(self.Agent.alive,      self.alive)
53
54          self.assertTrue(dclass.is_dataclass(self.Agent))
55
56      def test_activate(self):
57          """test activate the agent"""
58
59          self.simulation.agents = mk.create_autospec(agents.Agents,
60                                                      spec_set=True)
61
62          self.Agent.activate()
63          self.assertEqual(self.simulation.agents.activate.call_args_list,
64                           [mk.call(self.Agent)])
65
66      def test_deactivate(self):
67          """test deactivate the agent"""
68
69          self.simulation.agents = mk.create_autospec(agents.Agents,
70                                                      spec_set=True)
71
72          self.Agent.deactivate()
73          self.assertEqual(self.simulation.agents.deactivate.call_args_list,
74                           [mk.call(self.Agent)])
75
76      def test_transfer(self):
```

```
77          """test transfer to new vertex"""
78
79          self.simulation.agents = mk.create_autospec(agents.Agents,
80                                                      spec_set=True)
81
82          vertex = mk.MagicMock(spec=int)
83          level  = mk.MagicMock(spec=int)
84
85          master = mk.MagicMock()
86          master.attach_mock(self.simulation.agents, 'agents')
87          master.attach_mock(self.location,          'location')
88
89          self.Agent.transfer(vertex, level)
90          self.assertEqual(master.mock_calls,
91                           [mk.call.agents.deactivate(self.Agent),
92                            mk.call.location.__setitem__(level, vertex),
93                            mk.call.agents.activate(self.Agent)])
94
95      def test_transition(self):
96          """test transition agent_key of agent"""
97
98          self.simulation.agents = mk.create_autospec(agents.Agents,
99                                                      spec_set=True)
100
101         agent_key = mk.MagicMock(spec=str)
102
103         master = mk.MagicMock()
104         master.attach_mock(self.simulation.agents, 'agents')
105
106         self.Agent.transition(agent_key)
107         self.assertEqual(self.Agent.agent_key, agent_key)
108         self.assertEqual(master.mock_calls,
109                          [mk.call.agents.deactivate(self.Agent),
110                           mk.call.agents.activate(self.Agent)])
111
112     def test_vertices(self):
113         """test get the vertices for the agent's location at some distance"""
114
115         self.simulation.space = mk.create_autospec(space.Space, spec_set=True)
```

```
116
117         kwargs   = {keyword.upper: mk.MagicMock(spec=float),
118                     keyword.lower: mk.MagicMock(spec=float)}
119
120         self.assertEqual(self.Agent.vertices(**kwargs),
121                          self.simulation.space.neighborhood.return_value)
122         self.assertEqual(self.simulation.space.neighborhood.call_args_list,
123                          [mk.call(self.location, **kwargs)])
124
125     def test_die(self):
126         """test have agent die"""
127
128         with mk.patch.object(agent.Agent, 'deactivate',
129                              autospec=True) as mkDeactivate:
130             self.Agent.die()
131             self.assertEqual(self.Agent.alive, False)
132             self.assertEqual(mkDeactivate.call_args_list, [mk.call(self.Agent)])
133
134     def test_reset(self):
135         """test reset the agent"""
136
137         self.assertIsNone(self.Agent.reset())
```

C.5.1.3 test_egg.py

```
import unittest      as ut
import unittest.mock as mk

import dataclasses as dclass
import itertools   as i_tools

import source.keyword as keyword

import source.agents.agent    as agent
import source.agents.egg      as egg
import source.agents.egg_mass as egg_mass
import source.agents.insect   as insect

import source.simulation.behaviors  as behaviors
import source.simulation.models     as models
import source.simulation.simulation as simulation

import source.space.agents   as agents
import source.space.location as location
import source.space.space    as space

import source.survival.egg    as survival
import source.development.egg as development


class BehaviorsTest(behaviors.Behaviors):
    """Class to add dynamic values for tests"""

    survive_egg = mk.create_autospec(survival.Egg,    spec_set=True)
    develop_egg = mk.create_autospec(development.Egg, spec_set=True)


class SimulationTest(simulation.Simulation):
    """Class to add dynamic values for tests"""

    agents    = mk.create_autospec(agents.Agents, spec_set=True)
    behaviors = mk.create_autospec(BehaviorsTest, spec_set=True)
```

```
38      space     = mk.create_autospec(space.Space,   spec_set=True)
39      models    = mk.create_autospec(models.Models, spec_set=True)
40
41
42  class TestEgg(ut.TestCase):
43      """test base Egg class"""
44
45      def setUp(self):
46          """Setup the tests"""
47
48          self.agent_key = mk.MagicMock(spec=str)
49          self.unique_id = mk.MagicMock(spec=str)
50          self.alive     = mk.MagicMock(spec=bool)
51          self.mass      = mk.MagicMock(spec=float)
52          self.genotype  = mk.MagicMock(spec=str)
53          self.age       = mk.MagicMock(spec=int)
54          self.death     = mk.MagicMock(spec=str)
55
56          self.simulation  = mk.create_autospec(SimulationTest,    spec_set=True)
57          self.location    = mk.create_autospec(location.Location, spec_set=True)
58          self.egg_mass    = mk.create_autospec(egg_mass.EggMass,  spec_set=True)
59          self.survival    = mk.create_autospec(survival.Egg,      spec_set=True)
60          self.development = mk.create_autospec(development.Egg,   spec_set=True)
61
62          self.Egg = egg.Egg(self.agent_key,
63                             self.unique_id,
64                             self.simulation,
65                             self.location,
66                             self.alive,
67                             self.mass,
68                             self.genotype,
69                             self.age,
70                             self.death,
71                             self.egg_mass,
72                             self.survival,
73                             self.development)
74
75      def test___init__(self):
76          """test __init__ for class"""
```

```
77
78          self.assertIsInstance(self.Egg, agent.Agent)
79          self.assertIsInstance(self.Egg, insect.Insect)
80          self.assertIsInstance(self.Egg, egg.Egg)
81
82          self.assertEqual(self.Egg.agent_key,  self.agent_key)
83          self.assertEqual(self.Egg.unique_id,  self.unique_id)
84          self.assertEqual(self.Egg.simulation, self.simulation)
85          self.assertEqual(self.Egg.location,   self.location)
86          self.assertEqual(self.Egg.alive,      self.alive)
87
88          self.assertEqual(self.Egg.mass,     self.mass)
89          self.assertEqual(self.Egg.genotype, self.genotype)
90          self.assertEqual(self.Egg.age,      self.age)
91          self.assertEqual(self.Egg.death,    self.death)
92
93          self.assertEqual(self.Egg.egg_mass,    self.egg_mass)
94          self.assertEqual(self.Egg.survival,    self.survival)
95          self.assertEqual(self.Egg.development, self.development)
96
97          # noinspection PyTypeChecker
98          self.assertIsInstance(self.Egg._age_count, i_tools.count)
99          # noinspection PyTypeChecker
100         self.assertEqual(next(self.Egg._age_count),
101                          next(i_tools.count(self.age + 1)))
102
103         self.assertTrue(dclass.is_dataclass(self.Egg))
104
105     def test_deactivate(self):
106         """test deactivate the agent"""
107
108         self.simulation.agents = mk.create_autospec(agents.Agents,
109                                                     spec_set=True)
110         self.Egg.deactivate()
111         self.assertEqual(self.simulation.agents.deactivate.call_args_list,
112                          [mk.call(self.Egg)])
113         self.assertEqual(self.egg_mass.remove.call_args_list,
114                          [mk.call(self.Egg)])
115
```

```
116         self.simulation.agents.reset_mock()
117         self.egg_mass.reset_mock()
118         # Test deactivate is passed through to death
119         self.Egg.die(keyword.cannibalism)
120         self.assertEqual(self.simulation.agents.deactivate.call_args_list,
121                          [mk.call(self.Egg)])
122         self.assertEqual(self.egg_mass.remove.call_args_list,
123                          [mk.call(self.Egg)])
124         self.assertEqual(self.Egg.alive, False)
125         self.assertEqual(self.Egg.death, keyword.cannibalism)
126 
127     def test_survive(self):
128         """test run survive behavior"""
129 
130         # Egg is not alive
131         self.Egg.alive = False
132         self.assertEqual(self.Egg.survive(), [])
133         self.assertEqual(self.survival.survive.call_args_list, [])
134 
135         # Egg is alive
136         self.Egg.alive = True
137         self.assertEqual(self.Egg.survive(), [])
138         self.assertEqual(self.survival.survive.call_args_list,
139                          [mk.call(self.Egg)])
140 
141     def test_develop(self):
142         """test run develop behavior"""
143 
144         # Egg is not alive
145         self.Egg.alive = False
146         self.assertEqual(self.Egg.develop(), [])
147         self.assertEqual(self.development.develop.call_args_list, [])
148 
149         # Egg is alive
150         self.Egg.alive = True
151         self.assertEqual(self.Egg.develop(), [])
152         self.assertEqual(self.development.develop.call_args_list,
153                          [mk.call(self.Egg)])
154 
```

```
155     def test_initialize(self):
156         """test initialize a egg"""
157 
158         self.simulation.behaviors = mk.create_autospec(BehaviorsTest,
159                                                        spec_set=True)
160         self.simulation.behaviors.survive_egg = self.survival
161         self.simulation.behaviors.develop_egg = self.development
162 
163         self.Egg = egg.Egg.initialize(self.unique_id,
164                                       self.simulation,
165                                       self.location,
166                                       self.mass,
167                                       self.genotype,
168                                       self.egg_mass)
169 
170         self.assertIsInstance(self.Egg, agent.Agent)
171         self.assertIsInstance(self.Egg, insect.Insect)
172         self.assertIsInstance(self.Egg, egg.Egg)
173 
174         self.assertEqual(self.Egg.agent_key, keyword.egg)
175         self.assertEqual(self.Egg.alive,     True)
176         self.assertEqual(self.Egg.age,       0)
177         self.assertEqual(self.Egg.death,     keyword.alive)
178 
179         self.assertEqual(self.Egg.unique_id,  self.unique_id)
180         self.assertEqual(self.Egg.simulation, self.simulation)
181         self.assertEqual(self.Egg.location,   self.location)
182 
183         self.assertEqual(self.Egg.mass,     self.mass)
184         self.assertEqual(self.Egg.genotype, self.genotype)
185 
186         self.assertEqual(self.Egg.egg_mass,    self.egg_mass)
187         self.assertEqual(self.Egg.survival,    self.survival)
188         self.assertEqual(self.Egg.development, self.development)
189 
190         # noinspection PyTypeChecker
191         self.assertIsInstance(self.Egg._age_count, i_tools.count)
192         # noinspection PyTypeChecker
193         self.assertEqual(next(self.Egg._age_count), next(i_tools.count(1)))
```

```
194
195         self.assertTrue(dclass.is_dataclass(self.Egg))
```

C.5.1.4 test_egg_mass.py

```
import unittest      as ut
import unittest.mock as mk

import dataclasses  as dclass
import collections  as collect
import itertools    as i_tools
import numpy        as np
import numpy.random as rnd

import source.keyword as keyword

import source.agents.agent    as agent
import source.agents.egg      as agent_egg
import source.agents.egg_mass as egg_mass
import source.agents.adult    as agent_adult

import source.simulation.behaviors  as behaviors
import source.simulation.models     as models
import source.simulation.simulation as simulation

import source.space.agents   as agents
import source.space.location as location
import source.space.space    as space

import source.survival.egg    as survival
import source.development.egg as development


class BehaviorsTest(behaviors.Behaviors):
    """Class to add dynamic values for tests"""

    survive_egg = mk.create_autospec(survival.Egg,    spec_set=True)
    develop_egg = mk.create_autospec(development.Egg, spec_set=True)


class SimulationTest(simulation.Simulation):
    """Class to add dynamic values for tests"""
```

```
38
39      agents    = mk.create_autospec(agents.Agents, spec_set=True)
40      behaviors = mk.create_autospec(BehaviorsTest, spec_set=True)
41      space     = mk.create_autospec(space.Space,   spec_set=True)
42      models    = mk.create_autospec(models.Models, spec_set=True)
43
44
45  class AdultTest(agent_adult.Adult):
46      """Class to add dynamic values for tests"""
47
48      unique_id  = mk.MagicMock(spec=str)
49      simulation = mk.create_autospec(SimulationTest,    spec_set=True)
50      location   = mk.create_autospec(location.Location, spec_set=True)
51      genotype   = mk.MagicMock(spec=str)
52      mate       = mk.MagicMock(spec=str)
53
54
55  class EggTest(agent_egg.Egg):
56      """Class to add dynamic values for tests"""
57
58      unique_id  = mk.MagicMock(spec=str)
59      simulation = mk.create_autospec(SimulationTest,    spec_set=True)
60      location   = mk.create_autospec(location.Location, spec_set=True)
61      genotype   = mk.MagicMock(spec=str)
62
63
64  class EggMassTest(egg_mass.EggMass):
65      """Class to add dynamic values for tests"""
66
67      unique_id  = mk.MagicMock(spec=str)
68      simulation = mk.create_autospec(SimulationTest,    spec_set=True)
69      location   = mk.create_autospec(location.Location, spec_set=True)
70      mass       = mk.MagicMock(spec=float)
71      genotype   = mk.MagicMock(spec=str)
72
73
74  class EggsTest(egg_mass.Eggs):
75      """Class to add dynamic values for tests"""
76
```

```
    mass = mk.MagicMock(spec=float)


class TestEggs(ut.TestCase):
    """test the Eggs list class"""

    def setUp(self):
        """Setup the tests"""

        self.eggs = {mk.MagicMock(spec=str):
                         mk.create_autospec(agent_egg.Egg, spec_set=True)
                     for _ in range(3)}
        self.mass = mk.MagicMock(spec=float)

        self.Eggs = egg_mass.Eggs(self.eggs,
                                  self.mass)

    def test___init__(self):
        """test __init__ for class"""

        self.assertIsInstance(self.Eggs, collect.UserDict)
        self.assertIsInstance(self.Eggs, egg_mass.Eggs)

        self.assertEqual(self.Eggs.mass, self.mass)

        self.assertEqual(self.Eggs,      self.eggs)
        self.assertEqual(self.Eggs.data, self.eggs)

    def test_activate(self):
        """test activate the eggs"""

        self.Eggs.activate()

        for egg in self.eggs.values():
            self.assertEqual(egg.activate.call_args_list,
                             [mk.call()])

    def test_deactivate(self):
        """test deactivate the eggs"""
```

```
116
117         self.Eggs.deactivate()
118
119         for egg in self.eggs.values():
120             self.assertEqual(egg.deactivate.call_args_list,
121                              [mk.call()])
122
123     def test_cannibalize(self):
124         """test cannibalize the number of eggs"""
125
126         unique_ids = list(self.eggs.keys())
127
128         with mk.patch.object(rnd, 'shuffle') as mkRND:
129             for number in range(len(self.eggs)):
130                 self.Eggs.cannibalize(number)
131                 self.assertEqual(mkRND.call_args_list,
132                                  [mk.call(unique_ids)])
133                 mkRND.reset_mock()
134
135                 for index in range(number):
136                     unique_id = unique_ids[index]
137                     self.assertEqual(self.eggs[unique_id].die.call_args_list,
138                                      [mk.call(keyword.cannibalism)])
139                     self.eggs[unique_id].reset_mock()
140                 for egg in self.eggs.values():
141                     self.assertEqual(egg.die.call_args_list, [])
142
143     def test_initialize(self):
144         """test initialize a collection of eggs"""
145
146         survive = mk.create_autospec(survival.Egg,    spec_set=True)
147         develop = mk.create_autospec(development.Egg, spec_set=True)
148
149         loc = mk.create_autospec(location.Location, spec_set=True)
150         sim = mk.create_autospec(SimulationTest,    spec_set=True)
151
152         sim.behaviors = mk.create_autospec(BehaviorsTest,
153                                            spec_set=True)
154         sim.behaviors.survive_egg = survive
```

```
        sim.behaviors.develop_egg = develop

        genotypes  = [mk.MagicMock(spec=str) for _ in range(3)]
        unique_ids = [mk.MagicMock(spec=str) for _ in range(3)]

        mass = mk.create_autospec(EggMassTest, spec_set=True)
        mass.simulation = sim
        mass.location   = loc
        mass.new_unique_id.side_effect = unique_ids

        self.Eggs = egg_mass.Eggs.initialize(mass,
                                             genotypes,
                                             self.mass)
        self.assertIsInstance(self.Eggs, collect.UserDict)
        self.assertIsInstance(self.Eggs, egg_mass.Eggs)
        self.assertEqual(self.Eggs.mass, self.mass)

        for index, this in enumerate(self.Eggs.items()):
            unique_id, egg = this
            self.assertIsInstance(egg, agent_egg.Egg)

            self.assertEqual(egg.agent_key, keyword.egg)
            self.assertEqual(egg.alive, True)
            self.assertEqual(egg.age, 0)
            self.assertEqual(egg.death, keyword.alive)

            self.assertEqual(unique_id, egg.unique_id)
            self.assertEqual(egg.unique_id,
                             unique_ids[index])
            self.assertEqual(mass.new_unique_id.call_args_list[index],
                             mk.call())
            self.assertEqual(egg.location,
                             loc.copy.return_value)
            self.assertEqual(loc.copy.call_args_list,
                             mk.call())

            self.assertEqual(egg.simulation, sim)

            self.assertEqual(egg.mass,    self.mass)
```

```
            self.assertEqual(egg.genotype, genotypes[index])

            self.assertEqual(egg.egg_mass,    mass)
            self.assertEqual(egg.survival,    survive)
            self.assertEqual(egg.development, develop)

            # noinspection PyTypeChecker
            self.assertIsInstance(egg._age_count, i_tools.count)
            # noinspection PyTypeChecker
            self.assertEqual(next(egg._age_count), next(i_tools.count(1)))

            self.assertTrue(dclass.is_dataclass(egg))
        self.assertEqual(len(self.Eggs), 3)
        self.assertEqual(len(mass.new_unique_id.call_args_list), 3)
        self.assertEqual(len(loc.copy.call_args_list), 3)


class TestEggMass(ut.TestCase):
    """test the EggMass agent class"""

    def setUp(self):
        """Setup the tests"""

        self.agent_key = mk.MagicMock(spec=str)
        self.unique_id = mk.MagicMock(spec=str)
        self.alive     = mk.MagicMock(spec=bool)

        self.simulation = mk.create_autospec(SimulationTest,    spec_set=True)
        self.location   = mk.create_autospec(location.Location, spec_set=True)

        self.eggs = mk.create_autospec(EggsTest, spec_set=True)

        self.EggMass = egg_mass.EggMass(self.agent_key,
                                        self.unique_id,
                                        self.simulation,
                                        self.location,
                                        self.alive,
                                        self.eggs)

```

```
233     def test___init__(self):
234         """test __init__ for class"""
235 
236         self.assertIsInstance(self.EggMass, agent.Agent)
237         self.assertIsInstance(self.EggMass, egg_mass.EggMass)
238 
239         self.assertEqual(self.EggMass.agent_key,  self.agent_key)
240         self.assertEqual(self.EggMass.unique_id,  self.unique_id)
241         self.assertEqual(self.EggMass.simulation, self.simulation)
242         self.assertEqual(self.EggMass.location,   self.location)
243         self.assertEqual(self.EggMass.alive,      self.alive)
244 
245         # noinspection PyTypeChecker
246         self.assertIsInstance(self.EggMass._id_count, i_tools.count)
247         # noinspection PyTypeChecker
248         self.assertEqual(next(self.EggMass._id_count), next(i_tools.count()))
249 
250         self.assertTrue(dclass.is_dataclass(self.EggMass))
251 
252     def test_active(self):
253         """test if the egg_mass is active"""
254 
255         with mk.patch.object(egg_mass, 'len') as mkLen:
256             mkLen.return_value.__gt__.side_effect = [False, True, True]
257 
258             # Len is False
259             self.assertFalse(self.EggMass.active)
260             self.assertEqual(mkLen.return_value.__gt__.call_args_list,
261                              [mk.call(0)])
262             self.assertEqual(mkLen.call_args_list,
263                              [mk.call(self.eggs)])
264 
265             mkLen.reset_mock()
266             # Len is True, alive is False
267             self.EggMass.alive = False
268             self.assertFalse(self.EggMass.active)
269             self.assertEqual(mkLen.return_value.__gt__.call_args_list,
270                              [mk.call(0)])
271             self.assertEqual(mkLen.call_args_list,
```

```
                            [mk.call(self.eggs)])

        mkLen.reset_mock()
        # Len is True, alive is True
        self.EggMass.alive = True
        self.assertTrue(self.EggMass.active)
        self.assertEqual(mkLen.return_value.__gt__.call_args_list,
                         [mk.call(0)])
        self.assertEqual(mkLen.call_args_list,
                         [mk.call(self.eggs)])

    def test_inactive(self):
        """test if egg_mass is inactive"""

        with mk.patch.object(egg_mass, 'len') as mkLen:
            mkLen.return_value.__le__.side_effect = [False, True, True]

            # Len is False
            self.assertFalse(self.EggMass.inactive)
            self.assertEqual(mkLen.return_value.__le__.call_args_list,
                             [mk.call(0)])
            self.assertEqual(mkLen.call_args_list,
                             [mk.call(self.eggs)])

            mkLen.reset_mock()
            # Len is True, alive is False
            self.EggMass.alive = False
            self.assertFalse(self.EggMass.inactive)
            self.assertEqual(mkLen.return_value.__le__.call_args_list,
                             [mk.call(0)])
            self.assertEqual(mkLen.call_args_list,
                             [mk.call(self.eggs)])

            mkLen.reset_mock()
            # Len is True, alive is True
            self.EggMass.alive = True
            self.assertTrue(self.EggMass.inactive)
            self.assertEqual(mkLen.return_value.__le__.call_args_list,
                             [mk.call(0)])
```

```
311             self.assertEqual(mkLen.call_args_list,
312                              [mk.call(self.eggs)])
313
314     def test_activate(self):
315         """test activate the egg_mass"""
316
317         with mk.patch.object(agent.Agent, 'activate',
318                              autospec=True) as mkActivate:
319             self.assertEqual(self.EggMass.alive, self.alive)
320             self.EggMass.activate()
321             self.assertEqual(self.EggMass.alive, True)
322             self.assertEqual(mkActivate.call_args_list,
323                              [mk.call(self.EggMass)])
324             self.assertEqual(self.eggs.activate.call_args_list,
325                              [mk.call()])
326
327     def test_deactivate(self):
328         """test deactivate the egg_mass"""
329
330         with mk.patch.object(agent.Agent, 'deactivate',
331                              autospec=True) as mkDeactivate:
332             self.assertEqual(self.EggMass.alive, self.alive)
333             self.EggMass.deactivate()
334             self.assertEqual(self.EggMass.alive, False)
335             self.assertEqual(mkDeactivate.call_args_list,
336                              [mk.call(self.EggMass)])
337             self.assertEqual(self.eggs.deactivate.call_args_list,
338                              [mk.call()])
339
340     def test__feed_number(self):
341         """test get the number of eggs to feed on"""
342
343         amount = mk.MagicMock(spec=float)
344
345         ceil  = mk.MagicMock(spec=int)
346         floor = mk.MagicMock(spec=int)
347
348         ceil. __le__.side_effect = [True, False, False]
349         floor.__le__.side_effect = [      True,  False]
```

```
350
351         with mk.patch.object(np, 'ceil') as mkCeil:
352             with mk.patch.object(np, 'floor') as mkFloor:
353                 with mk.patch.object(egg_mass, 'len') as mkLen:
354                     with mk.patch.object(egg_mass, 'int') as mkInt:
355                         # Test get ceil
356                         mkInt.side_effect = [ceil, floor]
357                         self.assertEqual(self.EggMass._feed_number(amount),
358                                          ceil)
359                         self.assertEqual(mkInt.call_args_list,
360                                          [mk.call(mkCeil.return_value),
361                                           mk.call(mkFloor.return_value)])
362                         self.assertEqual(mkCeil.call_args_list,
363                                          [mk.call(amount.
364                                                   __truediv__.return_value)])
365                         self.assertEqual(mkFloor.call_args_list,
366                                          [mk.call(amount.
367                                                   __truediv__.return_value)])
368                         self.assertEqual(amount.__truediv__.call_args_list,
369                                          [mk.call(self.eggs.mass)])
370                         self.assertEqual(ceil.__le__.call_args_list,
371                                          [mk.call(mkLen.return_value)])
372                         self.assertEqual(floor.__le__.call_args_list, [])
373                         self.assertEqual(mkLen.call_args_list,
374                                          [mk.call(self.eggs)])
375
376                         mkInt.reset_mock()
377                         mkLen.reset_mock()
378                         mkCeil.reset_mock()
379                         mkFloor.reset_mock()
380                         ceil.reset_mock()
381                         amount.reset_mock()
382                         # Test get floor
383                         mkInt.side_effect = [ceil, floor]
384                         self.assertEqual(self.EggMass._feed_number(amount),
385                                          floor)
386                         self.assertEqual(mkInt.call_args_list,
387                                          [mk.call(mkCeil.return_value),
388                                           mk.call(mkFloor.return_value)])
```

```
                        self.assertEqual(mkCeil.call_args_list,
                                         [mk.call(amount.
                                                  __truediv__.return_value)])
                        self.assertEqual(mkFloor.call_args_list,
                                         [mk.call(amount.
                                                  __truediv__.return_value)])
                        self.assertEqual(amount.__truediv__.call_args_list,
                                         [mk.call(self.eggs.mass)])
                        self.assertEqual(ceil.__le__.call_args_list,
                                         [mk.call(mkLen.return_value)])
                        self.assertEqual(floor.__le__.call_args_list,
                                         [mk.call(mkLen.return_value)])
                        self.assertEqual(mkLen.call_args_list,
                                         [mk.call(self.eggs)])

                        mkInt.reset_mock()
                        mkLen.reset_mock()
                        mkCeil.reset_mock()
                        mkFloor.reset_mock()
                        ceil.reset_mock()
                        floor.reset_mock()
                        amount.reset_mock()
                        # Test get floor
                        mkInt.side_effect = [ceil, floor]
                        with self.assertRaisesRegex(RuntimeError,
                                                    'Tried to consume too much '
                                                    'egg'):
                            self.EggMass._feed_number(amount)
                        self.assertEqual(mkInt.call_args_list,
                                         [mk.call(mkCeil.return_value),
                                          mk.call(mkFloor.return_value)])
                        self.assertEqual(mkCeil.call_args_list,
                                         [mk.call(amount.
                                                  __truediv__.return_value)])
                        self.assertEqual(mkFloor.call_args_list,
                                         [mk.call(amount.
                                                  __truediv__.return_value)])
                        self.assertEqual(amount.__truediv__.call_args_list,
                                         [mk.call(self.eggs.mass)])
```

```
                        self.assertEqual(ceil.__le__.call_args_list,
                                         [mk.call(mkLen.return_value)])
                        self.assertEqual(floor.__le__.call_args_list,
                                         [mk.call(mkLen.return_value)])
                        self.assertEqual(mkLen.call_args_list,
                                         [mk.call(self.eggs)])

    def test_feed(self):
        """test feed on amount of eggs"""

        amount = mk.MagicMock(spec=float)

        with mk.patch.object(egg_mass.EggMass, '_feed_number',
                             autospec=True) as mkFeed:
            with mk.patch.object(egg_mass.EggMass, 'deactivate',
                                 autospec=True) as mkDeactivate:
                with mk.patch.object(egg_mass.EggMass, 'inactive',
                                     autospec=True) as mkInactive:
                    mkInactive.__get__ = mk.MagicMock(side_effect=[False, True])

                    # Test if no deactivate
                    self.EggMass.feed(amount)
                    self.assertEqual(self.eggs.cannibalize.call_args_list,
                                     [mk.call(mkFeed.return_value)])
                    self.assertEqual(mkFeed.call_args_list,
                                     [mk.call(self.EggMass, amount)])
                    self.assertEqual(mkDeactivate.call_args_list, [])

                    mkFeed.reset_mock()
                    self.eggs.reset_mock()
                    # Test if deactivate
                    self.EggMass.feed(amount)
                    self.assertEqual(self.eggs.cannibalize.call_args_list,
                                     [mk.call(mkFeed.return_value)])
                    self.assertEqual(mkFeed.call_args_list,
                                     [mk.call(self.EggMass, amount)])
                    self.assertEqual(mkDeactivate.call_args_list,
                                     [mk.call(self.EggMass)])

```

```
467     def test_remove(self):
468         """test remove an egg from egg_mass"""
469 
470         egg = mk.create_autospec(EggTest, spec_set=True)
471 
472         self.EggMass.remove(egg)
473         self.assertEqual(self.eggs.__delitem__.call_args_list,
474                          [mk.call(egg.unique_id)])
475 
476     def test_reset(self):
477         """test reset the egg_mass"""
478 
479         with mk.patch.object(egg_mass.EggMass, 'inactive',
480                              autospec=True) as mkInactive:
481             with mk.patch.object(egg_mass.EggMass, 'deactivate',
482                                  autospec=True) as mkDeactivate:
483                 mkInactive.__get__ = mk.MagicMock(side_effect=[False, True])
484 
485                 # Active
486                 self.EggMass.reset()
487                 self.assertEqual(mkDeactivate.call_args_list, [])
488 
489                 # Inactive
490                 self.EggMass.reset()
491                 self.assertEqual(mkDeactivate.call_args_list,
492                                  [mk.call(self.EggMass)])
493 
494     def test_new_unique_id(self):
495         """test generate a new unique_id"""
496 
497         counter = i_tools.count()
498 
499         self.assertEqual(self.EggMass.new_unique_id(),
500                          str(self.unique_id) + str(next(counter)))
501         self.assertEqual(self.EggMass.new_unique_id(),
502                          '{}1'.format(self.unique_id))
503         self.assertEqual(self.EggMass.new_unique_id(),
504                          '{}2'.format(self.unique_id))
505 
```

```
506     def test__alleles(self):
507         """test get the alleles for a genotype"""
508 
509         # homo_r
510         self.assertEqual(self.EggMass._alleles(keyword.homo_r),
511                          keyword.homo_r_alleles)
512         self.assertEqual(self.EggMass._alleles(keyword.homo_r),
513                          (0, 0))
514 
515         # hetero
516         self.assertEqual(self.EggMass._alleles(keyword.hetero),
517                          keyword.hetero_alleles)
518         self.assertEqual(self.EggMass._alleles(keyword.hetero),
519                          (0, 1))
520 
521         # homo_s
522         self.assertEqual(self.EggMass._alleles(keyword.homo_s),
523                          keyword.homo_s_alleles)
524         self.assertEqual(self.EggMass._alleles(keyword.homo_s),
525                          (1, 1))
526 
527         # Error
528         genotype = mk.MagicMock(spec=str)
529         with self.assertRaisesRegex(RuntimeError,
530                                     'Invalid genotype_key: '
531                                     '{}'.format(genotype)):
532             self.EggMass._alleles(genotype)
533 
534     def test__generate_genotype(self):
535         """test genotype from alleles"""
536 
537         # Test abstract
538         mother_alleles = mk.MagicMock(spec=tuple)
539         father_alleles = mk.MagicMock(spec=tuple)
540 
541         mother = mk.MagicMock(spec=int)
542         father = mk.MagicMock(spec=int)
543 
544         mother.__add__.side_effect = [keyword.homo_r_value,
```

```
                                    keyword.hetero_value,
                                    keyword.homo_s_value]
        with mk.patch.object(rnd, 'choice') as mkRND:
            mkRND.side_effect = [mother, father,
                                 mother, father,
                                 mother, father]

            # homo_r
            self.assertEqual(self.EggMass._generate_genotype(mother_alleles,
                                                             father_alleles),
                             keyword.homo_r)
            self.assertEqual(mkRND.call_args_list,
                             [mk.call(mother_alleles), mk.call(father_alleles)])
            self.assertEqual(mother.__add__.call_args_list,
                             [mk.call(father)])

            mother.reset_mock()
            mkRND.reset_mock()
            # hetero
            self.assertEqual(self.EggMass._generate_genotype(mother_alleles,
                                                             father_alleles),
                             keyword.hetero)
            self.assertEqual(mkRND.call_args_list,
                             [mk.call(mother_alleles), mk.call(father_alleles)])
            self.assertEqual(mother.__add__.call_args_list,
                             [mk.call(father)])

            mother.reset_mock()
            mkRND.reset_mock()
            # homo_s
            self.assertEqual(self.EggMass._generate_genotype(mother_alleles,
                                                             father_alleles),
                             keyword.homo_s)
            self.assertEqual(mkRND.call_args_list,
                             [mk.call(mother_alleles), mk.call(father_alleles)])
            self.assertEqual(mother.__add__.call_args_list,
                             [mk.call(father)])

        # Test practical
```

```
584         #       homo_r x homo_r
585         self.assertEqual(self.EggMass
586                          ._generate_genotype(keyword.homo_r_alleles,
587                                              keyword.homo_r_alleles),
588                          keyword.homo_r)
589         #       homo_r x homo_s
590         self.assertEqual(self.EggMass
591                          ._generate_genotype(keyword.homo_r_alleles,
592                                              keyword.homo_s_alleles),
593                          keyword.hetero)
594         #       homo_s x homo_r
595         self.assertEqual(self.EggMass
596                          ._generate_genotype(keyword.homo_s_alleles,
597                                              keyword.homo_r_alleles),
598                          keyword.hetero)
599         #       homo_s x homo_s
600         self.assertEqual(self.EggMass
601                          ._generate_genotype(keyword.homo_s_alleles,
602                                              keyword.homo_s_alleles),
603                          keyword.homo_s)
604
605     def test_genotypes(self):
606         """test generate genotypes"""
607
608         mother = mk.MagicMock(spec=str)
609         father = mk.MagicMock(spec=str)
610
611         mother_alleles = mk.MagicMock(spec=tuple)
612         father_alleles = mk.MagicMock(spec=tuple)
613
614         genotypes = []
615         for _ in range(3):
616             genotypes.append(mk.MagicMock(spec=str))
617
618         with mk.patch.object(egg_mass.EggMass, '_generate_genotype',
619                              autospec=True) as mkGenerate:
620             with mk.patch.object(egg_mass.EggMass, '_alleles',
621                                  autospec=True) as mkAlleles:
622                 mkGenerate.side_effect = genotypes
```

```
                mkAlleles. side_effect = [mother_alleles, father_alleles]

                self.assertEqual(self.EggMass.genotypes(3, mother, father),
                                 genotypes)
                for call in mkGenerate.call_args_list:
                    self.assertEqual(call,
                                     mk.call(mother_alleles, father_alleles))
                self.assertEqual(len(mkGenerate.call_args_list), 3)
                self.assertEqual(mkAlleles.call_args_list,
                                 [mk.call(mother), mk.call(father)])

    def test_empty(self):
        """test initialize egg_mass without eggs"""

        self.EggMass = egg_mass.EggMass.empty(self.unique_id,
                                              self.simulation,
                                              self.location)

        self.assertIsInstance(self.EggMass, agent.Agent)
        self.assertIsInstance(self.EggMass, egg_mass.EggMass)

        self.assertEqual(self.EggMass.agent_key, keyword.egg_mass)
        self.assertEqual(self.EggMass.alive,     True)

        self.assertEqual(self.EggMass.unique_id,  self.unique_id)
        self.assertEqual(self.EggMass.simulation, self.simulation)
        self.assertEqual(self.EggMass.location,   self.location)

        self.assertIsInstance(self.EggMass.eggs, egg_mass.Eggs)
        self.assertEqual(self.EggMass.eggs,      {})
        self.assertEqual(self.EggMass.eggs.mass, -1)

        # noinspection PyTypeChecker
        self.assertIsInstance(self.EggMass._id_count, i_tools.count)
        # noinspection PyTypeChecker
        self.assertEqual(next(self.EggMass._id_count), next(i_tools.count()))

        self.assertTrue(dclass.is_dataclass(self.EggMass))

```

```
662     def test_initialize(self):
663         """test initialize the egg_mass"""
664 
665         survive = mk.create_autospec(survival.Egg,    spec_set=True)
666         develop = mk.create_autospec(development.Egg, spec_set=True)
667         self.simulation.behaviors = mk.create_autospec(BehaviorsTest,
668                                                        spec_set=True)
669         self.simulation.behaviors.survive_egg = survive
670         self.simulation.behaviors.develop_egg = develop
671         self.simulation.models = mk.create_autospec(models.Models,
672                                                     spec_set=True)
673 
674         number    = mk.MagicMock(spec=int)
675         mass      = mk.MagicMock(spec=float)
676         genotypes = [mk.MagicMock(spec=str) for _ in range(3)]
677         locations = [mk.create_autospec(location.Location, spec_set=True)
678                      for _ in range(3)]
679 
680         self.simulation.models.\
681             __getitem__.return_value.side_effect = [number, mass]
682         self.location.copy.side_effect = locations
683 
684         mother = mk.MagicMock(spec=str)
685         father = mk.MagicMock(spec=str)
686 
687         with mk.patch.object(egg_mass.EggMass, 'genotypes',
688                              autospec=True) as mkGenotypes:
689             mkGenotypes.return_value = genotypes
690 
691             self.EggMass = egg_mass.EggMass.initialize(self.unique_id,
692                                                        self.simulation,
693                                                        self.location,
694                                                        mother, father)
695             self.assertEqual(self.EggMass.agent_key, keyword.egg_mass)
696             self.assertEqual(self.EggMass.alive,     True)
697 
698             self.assertEqual(self.EggMass.unique_id,  self.unique_id)
699             self.assertEqual(self.EggMass.simulation, self.simulation)
700             self.assertEqual(self.EggMass.location,   self.location)
```

```
701
702             self.assertIsInstance(self.EggMass.eggs, egg_mass.Eggs)
703             self.assertEqual(self.EggMass.eggs.mass, mass)
704
705             for index, this in enumerate(self.EggMass.eggs.items()):
706                 unique_id, egg = this
707                 self.assertIsInstance(egg, agent_egg.Egg)
708
709                 self.assertEqual(egg.agent_key, keyword.egg)
710                 self.assertEqual(egg.alive, True)
711                 self.assertEqual(egg.age, 0)
712                 self.assertEqual(egg.death, keyword.alive)
713
714                 self.assertEqual(unique_id, egg.unique_id)
715                 self.assertEqual(egg.unique_id,
716                                  str(self.unique_id) + str(index))
717                 self.assertEqual(egg.location, locations[index])
718
719                 self.assertEqual(egg.simulation, self.simulation)
720
721                 self.assertEqual(egg.mass,     mass)
722                 self.assertEqual(egg.genotype, genotypes[index])
723
724                 self.assertEqual(egg.egg_mass,    self.EggMass)
725                 self.assertEqual(egg.survival,    survive)
726                 self.assertEqual(egg.development, develop)
727             self.assertEqual(len(self.EggMass.eggs), 3)
728
729             self.assertEqual(self.simulation.models.
730                                __getitem__.return_value.call_args_list,
731                              [mk.call(mother),
732                               mk.call(mother)])
733             self.assertEqual(self.simulation.models.
734                                 __getitem__.call_args_list,
735                              [mk.call(keyword.init_num),
736                               mk.call(keyword.init_mass)])
737             self.assertEqual(mkGenotypes.call_args_list,
738                              [mk.call(self.EggMass, number, mother, father)])
739
```

```
740     def test_setup(self):
741         """test setup an initial egg_mass"""
742 
743         survive = mk.create_autospec(survival.Egg,    spec_set=True)
744         develop = mk.create_autospec(development.Egg, spec_set=True)
745         self.simulation.behaviors = mk.create_autospec(BehaviorsTest,
746                                                        spec_set=True)
747         self.simulation.behaviors.survive_egg = survive
748         self.simulation.behaviors.develop_egg = develop
749         self.simulation.space  = mk.create_autospec(space.Space, spec_set=True)
750         self.simulation.models = mk.create_autospec(models.Models,
751                                                     spec_set=True)
752 
753         self.simulation.space.new_location.return_value = self.location
754 
755         mass      = mk.MagicMock(spec=float)
756         locations = [mk.create_autospec(location.Location, spec_set=True)
757                      for _ in range(3)]
758 
759         unique_id_num = mk.MagicMock(spec=int)
760         initial_key   = mk.MagicMock(spec=str)
761 
762         parents = [[keyword.homo_r, keyword.homo_r],
763                    [keyword.homo_r, keyword.homo_s],
764                    [keyword.homo_s, keyword.homo_s]]
765 
766         with mk.patch.object(rnd, 'shuffle'):
767             for index_i, genotype in enumerate(keyword.genotype_keys):
768                 self.simulation.models.\
769                     __getitem__.return_value.side_effect = [3, mass]
770                 self.location.copy.side_effect = locations
771 
772                 self.EggMass = egg_mass.EggMass.setup(unique_id_num,
773                                                       initial_key,
774                                                       self.simulation,
775                                                       genotype)
776                 self.assertEqual(self.EggMass.agent_key, keyword.egg_mass)
777                 self.assertEqual(self.EggMass.alive,     True)
778 
```

```
779                 self.assertEqual(self.EggMass.unique_id,
780                                  str(initial_key) + str(unique_id_num)
781                                                   + keyword.egg_mass)
782                 self.assertEqual(self.EggMass.location,   self.location)
783                 self.assertEqual(self.simulation.space.
784                                  new_location.call_args_list,
785                                  [mk.call(keyword.egg_depth)])
786
787                 self.assertEqual(self.EggMass.simulation, self.simulation)
788
789                 self.assertIsInstance(self.EggMass.eggs, egg_mass.Eggs)
790                 self.assertEqual(self.EggMass.eggs.mass, mass)
791
792                 for index_j, this in enumerate(self.EggMass.eggs.items()):
793                     unique_id, egg = this
794                     self.assertIsInstance(egg, agent_egg.Egg)
795
796                     self.assertEqual(egg.agent_key, keyword.egg)
797                     self.assertEqual(egg.alive, True)
798                     self.assertEqual(egg.age, 0)
799                     self.assertEqual(egg.death, keyword.alive)
800
801                     self.assertEqual(unique_id, egg.unique_id)
802                     self.assertEqual(egg.unique_id,
803                                      self.EggMass.unique_id + str(index_j))
804                     self.assertEqual(egg.location, locations[index_j])
805
806                     self.assertEqual(egg.simulation, self.simulation)
807
808                     self.assertEqual(egg.mass,     mass)
809                     self.assertEqual(egg.genotype, genotype)
810
811                     self.assertEqual(egg.egg_mass,    self.EggMass)
812                     self.assertEqual(egg.survival,    survive)
813                     self.assertEqual(egg.development, develop)
814                 self.assertEqual(len(self.EggMass.eggs), 3)
815
816                 self.assertEqual(self.simulation.models.
817                                  __getitem__.return_value.call_args_list,
```

```
                                [mk.call(parents[index_i][0]),
                                 mk.call(parents[index_i][0])])
            self.assertEqual(self.simulation.models.
                                __getitem__.call_args_list,
                                [mk.call(keyword.init_num),
                                 mk.call(keyword.init_mass)])

            self.simulation.space.reset_mock()
            self.simulation.models.reset_mock()

    def test_birth(self):
        """test birth a new egg_mass"""

        survive = mk.create_autospec(survival.Egg,    spec_set=True)
        develop = mk.create_autospec(development.Egg, spec_set=True)
        self.simulation.behaviors = mk.create_autospec(BehaviorsTest,
                                                       spec_set=True)
        self.simulation.behaviors.survive_egg = survive
        self.simulation.behaviors.develop_egg = develop
        self.simulation.models = mk.create_autospec(models.Models,
                                                    spec_set=True)

        number    = mk.MagicMock(spec=int)
        mass      = mk.MagicMock(spec=float)
        genotypes = [mk.MagicMock(spec=str) for _ in range(3)]
        locations = [mk.create_autospec(location.Location, spec_set=True)
                     for _ in range(3)]

        self.simulation.models. \
            __getitem__.return_value.side_effect = [number, mass]
        self.location.copy.side_effect = locations

        adult = mk.create_autospec(AdultTest, spec_set=True)
        adult.simulation = self.simulation

        adult.new_unique_id.return_value    = self.unique_id
        adult.new_egg_location.return_value = self.location

        with mk.patch.object(egg_mass.EggMass, 'genotypes',
```

```
                                autospec=True) as mkGenotypes:
            mkGenotypes.return_value = genotypes

            self.EggMass = egg_mass.EggMass.birth(adult)
            self.assertEqual(self.EggMass.agent_key, keyword.egg_mass)
            self.assertEqual(self.EggMass.alive,     True)

            self.assertEqual(self.EggMass.unique_id,  self.unique_id)
            self.assertEqual(self.EggMass.simulation, self.simulation)
            self.assertEqual(self.EggMass.location,   self.location)

            self.assertIsInstance(self.EggMass.eggs, egg_mass.Eggs)
            self.assertEqual(self.EggMass.eggs.mass, mass)

            for index, this in enumerate(self.EggMass.eggs.items()):
                unique_id, egg = this
                self.assertIsInstance(egg, agent_egg.Egg)

                self.assertEqual(egg.agent_key, keyword.egg)
                self.assertEqual(egg.alive, True)
                self.assertEqual(egg.age, 0)
                self.assertEqual(egg.death, keyword.alive)

                self.assertEqual(unique_id, egg.unique_id)
                self.assertEqual(egg.unique_id,
                                 str(self.unique_id) + str(index))
                self.assertEqual(egg.location, locations[index])

                self.assertEqual(egg.simulation, self.simulation)

                self.assertEqual(egg.mass,     mass)
                self.assertEqual(egg.genotype, genotypes[index])

                self.assertEqual(egg.egg_mass,    self.EggMass)
                self.assertEqual(egg.survival,    survive)
                self.assertEqual(egg.development, develop)
            self.assertEqual(len(self.EggMass.eggs), 3)

            self.assertEqual(self.simulation.models.
```

```
                            __getitem__.return_value.call_args_list,
                            [mk.call(adult.genotype),
                             mk.call(adult.genotype)])
        self.assertEqual(self.simulation.models.
                            __getitem__.call_args_list,
                            [mk.call(keyword.init_num),
                             mk.call(keyword.init_mass)])
        self.assertEqual(mkGenotypes.call_args_list,
                            [mk.call(self.EggMass, number,
                                     adult.genotype, adult.mate)])
```

C.5.1.5 test_insect.py

```
import unittest      as ut
import unittest.mock as mk

import dataclasses as dclass
import itertools   as i_tools

import source.keyword as keyword

import source.agents.agent  as agent
import source.agents.insect as insect

import source.simulation.simulation as simulation

import source.space.agents      as agents
import source.space.environment as environment
import source.space.location    as location


class SimulationTest(simulation.Simulation):
    """Class to add dynamic values for tests"""

    agents = mk.create_autospec(agents.Agents, spec_set=True)


class EnvironmentTest(environment.Environment):
    """Class to add dynamic values for tests"""

    bt    = mk.MagicMock(spec=str)
    plant = mk.MagicMock(spec=float)


class AgentsBinTest(agents.AgentsBin):
    """Class to add dynamic values for tests"""

    environment = mk.create_autospec(EnvironmentTest, spec_set=True)


```

```
38 class TestInsect(ut.TestCase):
39     """test base Insect class"""
40 
41     def setUp(self):
42         """Setup the tests"""
43 
44         self.agent_key = mk.MagicMock(spec=str)
45         self.unique_id = mk.MagicMock(spec=str)
46         self.alive     = mk.MagicMock(spec=bool)
47         self.mass      = mk.MagicMock(spec=float)
48         self.genotype  = mk.MagicMock(spec=str)
49         self.age       = mk.MagicMock(spec=int)
50         self.death     = mk.MagicMock(spec=str)
51 
52         self.simulation = mk.create_autospec(SimulationTest,
53                                              spec_set=True)
54         self.location   = mk.create_autospec(location.Location,
55                                              spec_set=True)
56 
57         self.Insect = insect.Insect(self.agent_key,
58                                     self.unique_id,
59                                     self.simulation,
60                                     self.location,
61                                     self.alive,
62                                     self.mass,
63                                     self.genotype,
64                                     self.age,
65                                     self.death)
66 
67     def test___init__(self):
68         """test __init__ for class"""
69 
70         self.assertIsInstance(self.Insect, agent.Agent)
71         self.assertIsInstance(self.Insect, insect.Insect)
72 
73         self.assertEqual(self.Insect.agent_key,  self.agent_key)
74         self.assertEqual(self.Insect.unique_id,  self.unique_id)
75         self.assertEqual(self.Insect.simulation, self.simulation)
76         self.assertEqual(self.Insect.location,   self.location)
```

```
        self.assertEqual(self.Insect.alive,      self.alive)

        self.assertEqual(self.Insect.mass,       self.mass)
        self.assertEqual(self.Insect.genotype,   self.genotype)
        self.assertEqual(self.Insect.age,        self.age)
        self.assertEqual(self.Insect.death,      self.death)

        # noinspection PyTypeChecker
        self.assertIsInstance(self.Insect._age_count, i_tools.count)
        # noinspection PyTypeChecker
        self.assertEqual(next(self.Insect._age_count),
                         next(i_tools.count(self.age + 1)))

        self.assertTrue(dclass.is_dataclass(self.Insect))

    def test_bt(self):
        """test get the bt state of the plant"""

        self.simulation.agents = mk.create_autospec(agents.Agents,
                                                    spec_set=True)
        self.simulation.agents.__getitem__.return_value = \
            mk.create_autospec(AgentsBinTest, spec_set=True)
        self.simulation.agents.__getitem__.return_value.environment = \
            mk.create_autospec(EnvironmentTest, spec_set=True)
        self.location.__getitem__.return_value = \
            mk.create_autospec(location.Location, spec_set=True)

        self.assertEqual(self.Insect.bt,
                         self.simulation.agents.__getitem__.return_value.
                            environment.bt)
        self.assertEqual(self.simulation.agents.__getitem__.call_args_list,
                         [mk.call(self.location.__getitem__.return_value.
                                  location_key)])
        self.assertEqual(self.location.__getitem__.call_args_list,
                         [mk.call(slice(None, keyword.bt_depth, None))])

    def test_plant(self):
        """test get the plant mass"""

```

```
        self.simulation.agents = mk.create_autospec(agents.Agents,
                                                    spec_set=True)
        self.simulation.agents.__getitem__.return_value = \
            mk.create_autospec(AgentsBinTest, spec_set=True)
        self.simulation.agents.__getitem__.return_value.environment = \
            mk.create_autospec(EnvironmentTest, spec_set=True)
        self.location.__getitem__.return_value = \
            mk.create_autospec(location.Location, spec_set=True)

        self.assertEqual(self.Insect.plant,
                         self.simulation.agents.__getitem__.return_value.
                         environment.plant)
        self.assertEqual(self.simulation.agents.__getitem__.call_args_list,
                         [mk.call(self.location.__getitem__.return_value.
                                  location_key)])
        self.assertEqual(self.location.__getitem__.call_args_list,
                         [mk.call(slice(None, keyword.plant_depth, None))])

    def test_advance_age(self):
        """test age the agent"""

        counter = i_tools.count(self.age + 1)

        self.assertEqual(self.Insect.advance_age(), [])
        self.assertNotEqual(self.Insect.age, self.age)
        self.assertEqual(self.Insect.age,
                         next(counter))
        self.assertEqual(self.Insect.age,
                         self.age.__add__.return_value)

        self.assertEqual(self.Insect.advance_age(), [])
        self.assertNotEqual(self.Insect.age,
                            self.age.__add__.return_value)
        self.assertEqual(self.Insect.age,
                         next(counter))
        self.assertEqual(self.Insect.age,
                         self.age.__add__.return_value.__add__.return_value)

    def test_die(self):
```

```
        """test have agent die"""

        death = mk.MagicMock(spec=str)

        with mk.patch.object(agent.Agent, 'die', autospec=True) as mkDie:
            self.Insect.die(death)
            self.assertEqual(self.Insect.death, death)
            self.assertEqual(mkDie.call_args_list, [mk.call(self.Insect)])
```

C.5.1.6 test_larva.py

```
import unittest      as ut
import unittest.mock as mk

import dataclasses as dclass
import itertools   as i_tools

import source.keyword as keyword

import source.agents.agent    as agent
import source.agents.insect   as insect
import source.agents.egg      as agent_egg
import source.agents.egg_mass as agent_egg_mass
import source.agents.larva    as larva

import source.biomass.gut  as gut
import source.biomass.mass as mass

import source.simulation.behaviors  as behaviors
import source.simulation.models     as models
import source.simulation.simulation as simulation

import source.space.agents   as agents
import source.space.location as location
import source.space.space    as space

import source.development.larva  as development
import source.forage.cannibalism as cannibalism
import source.forage.egg         as forage_egg
import source.forage.larva       as forage_larva
import source.forage.plant       as forage_plant
import source.forage.target      as target_loss
import source.movement.larva     as movement
import source.survival.larva     as survival


class BehaviorsTest(behaviors.Behaviors):
    """Class to add dynamic values for tests"""
```

```
38
39      gut           = mk.create_autospec(gut.Gut,                 spec_set=True)
40      mass          = mk.create_autospec(mass.Mass,               spec_set=True)
41      survive_larva = mk.create_autospec(survival.Larva,          spec_set=True)
42      develop_larva = mk.create_autospec(development.Larva,       spec_set=True)
43      move_larva    = mk.create_autospec(movement.Larva,          spec_set=True)
44      cannibalism   = mk.create_autospec(cannibalism.Cannibalism, spec_set=True)
45      forage_plant  = mk.create_autospec(forage_plant.Plant,      spec_set=True)
46      forage_egg    = mk.create_autospec(forage_egg.Egg,          spec_set=True)
47      forage_larva  = mk.create_autospec(forage_larva.Larva,      spec_set=True)
48      loss          = mk.create_autospec(target_loss.Target,      spec_set=True)
49
50
51  class SimulationTest(simulation.Simulation):
52      """Class to add dynamic values for tests"""
53
54      agents    = mk.create_autospec(agents.Agents, spec_set=True)
55      behaviors = mk.create_autospec(BehaviorsTest, spec_set=True)
56      space     = mk.create_autospec(space.Space,   spec_set=True)
57      models    = mk.create_autospec(models.Models, spec_set=True)
58
59
60  class EggTest(agent_egg.Egg):
61      """Class to add dynamic values for tests"""
62
63      unique_id  = mk.MagicMock(spec=str)
64      simulation = mk.create_autospec(SimulationTest,    spec_set=True)
65      location   = mk.create_autospec(location.Location, spec_set=True)
66      mass       = mk.MagicMock(spec=float)
67      genotype   = mk.MagicMock(spec=str)
68
69
70  class EggMassTest(agent_egg_mass.EggMass):
71      """Class to add dynamic values for tests"""
72
73      agent_key = keyword.egg_mass
74
75
76  class TestLarva(ut.TestCase):
```

```
77      """test base Larva class"""
78
79      def setUp(self):
80          """Setup the tests"""
81
82          self.agent_key = mk.MagicMock(spec=str)
83          self.unique_id = mk.MagicMock(spec=str)
84          self.alive     = mk.MagicMock(spec=bool)
85          self.mass      = mk.MagicMock(spec=float)
86          self.genotype  = mk.MagicMock(spec=str)
87          self.age       = mk.MagicMock(spec=int)
88          self.death     = mk.MagicMock(spec=str)
89
90          self.simulation = mk.create_autospec(SimulationTest,    spec_set=True)
91          self.location   = mk.create_autospec(location.Location, spec_set=True)
92
93          self.plant_gut = mk.MagicMock(spec=float)
94          self.egg_gut   = mk.MagicMock(spec=float)
95          self.larva_gut = mk.MagicMock(spec=float)
96          self.full      = mk.MagicMock(spec=bool)
97          self.starve    = mk.MagicMock(spec=bool)
98
99          self.target      = mk.create_autospec(EggMassTest,       spec_set=True)
100         self.gut         = mk.create_autospec(gut.Gut,           spec_set=True)
101         self.biomass     = mk.create_autospec(mass.Mass,         spec_set=True)
102         self.survival    = mk.create_autospec(survival.Larva,    spec_set=True)
103         self.development = mk.create_autospec(development.Larva, spec_set=True)
104         self.movement    = mk.create_autospec(movement.Larva,    spec_set=True)
105
106         self.forage_plant = mk.create_autospec(forage_plant.Plant,
107                                                spec_set=True)
108         self.forage_egg   = mk.create_autospec(forage_egg.Egg,
109                                                spec_set=True)
110         self.forage_larva = mk.create_autospec(forage_larva.Larva,
111                                                spec_set=True)
112         self.loss         = mk.create_autospec(target_loss.Target,
113                                                spec_set=True)
114         self.cannibalism  = mk.create_autospec(cannibalism.Cannibalism,
115                                                spec_set=True)
```

```
116
117         self.Larva = larva.Larva(self.agent_key,
118                                  self.unique_id,
119                                  self.simulation,
120                                  self.location,
121                                  self.alive,
122                                  self.mass,
123                                  self.genotype,
124                                  self.age,
125                                  self.death,
126                                  self.plant_gut,
127                                  self.egg_gut,
128                                  self.larva_gut,
129                                  self.full,
130                                  self.starve,
131                                  self.gut,
132                                  self.biomass,
133                                  self.survival,
134                                  self.development,
135                                  self.movement,
136                                  self.forage_plant,
137                                  self.forage_egg,
138                                  self.forage_larva,
139                                  self.loss,
140                                  self.cannibalism,
141                                  self.target)
142
143     def test___init__(self):
144         """test __init__ for class"""
145
146         self.assertIsInstance(self.Larva, agent.Agent)
147         self.assertIsInstance(self.Larva, insect.Insect)
148         self.assertIsInstance(self.Larva, larva.Larva)
149
150         self.assertEqual(self.Larva.agent_key,  self.agent_key)
151         self.assertEqual(self.Larva.unique_id,  self.unique_id)
152         self.assertEqual(self.Larva.simulation, self.simulation)
153         self.assertEqual(self.Larva.location,   self.location)
154         self.assertEqual(self.Larva.alive,      self.alive)
```

```
155
156         self.assertEqual(self.Larva.mass,        self.mass)
157         self.assertEqual(self.Larva.genotype,    self.genotype)
158         self.assertEqual(self.Larva.age,         self.age)
159         self.assertEqual(self.Larva.death,       self.death)
160
161         self.assertEqual(self.Larva.plant_gut,     self.plant_gut)
162         self.assertEqual(self.Larva.egg_gut,       self.egg_gut)
163         self.assertEqual(self.Larva.larva_gut,     self.larva_gut)
164         self.assertEqual(self.Larva.full,          self.full)
165         self.assertEqual(self.Larva.starve,        self.starve)
166         self.assertEqual(self.Larva.gut,           self.gut)
167         self.assertEqual(self.Larva.biomass,       self.biomass)
168         self.assertEqual(self.Larva.survival,      self.survival)
169         self.assertEqual(self.Larva.development,   self.development)
170         self.assertEqual(self.Larva.movement,      self.movement)
171         self.assertEqual(self.Larva.forage_plant, self.forage_plant)
172         self.assertEqual(self.Larva.forage_egg,    self.forage_egg)
173         self.assertEqual(self.Larva.forage_larva, self.forage_larva)
174         self.assertEqual(self.Larva.loss,          self.loss)
175         self.assertEqual(self.Larva.cannibalism,   self.cannibalism,)
176         self.assertEqual(self.Larva.target,        self.target)
177
178         # noinspection PyTypeChecker
179         self.assertIsInstance(self.Larva._age_count, i_tools.count)
180         # noinspection PyTypeChecker
181         self.assertEqual(next(self.Larva._age_count),
182                          next(i_tools.count(self.age + 1)))
183
184         self.assertTrue(dclass.is_dataclass(self.Larva))
185
186     def test_active(self):
187         """test if this is active"""
188
189         self.mass.__gt__.side_effect = [True, False]
190
191         # test active
192         self.assertTrue(self.Larva.active)
193         self.assertEqual(self.mass.__gt__.call_args_list,
```

```
                        [mk.call(0)])

        self.mass.reset_mock()
        # test inactive
        self.assertFalse(self.Larva.active)
        self.assertEqual(self.mass.__gt__.call_args_list,
                         [mk.call(0)])

    def test__can_consume(self):
        """test if larva can consume"""

        # Not alive
        self.Larva.alive = False
        self.assertFalse(self.Larva._can_consume)

        # Alive and full
        self.Larva.alive = True
        self.assertTrue(self.Larva._can_consume)

    def test__has_target(self):
        """test if this larva has a target"""

        # has target given and active
        self.Larva.target.active = True
        self.assertTrue(self.Larva._has_target)

        # has target given but inactive
        self.Larva.target.active = False
        self.assertFalse(self.Larva._has_target)

        # no target given
        self.Larva.target = None
        self.assertFalse(self.Larva._has_target)

    def test_add_plant(self):
        """test add_plant"""

        available = mk.MagicMock(spec=float)
        amount    = mk.MagicMock(spec=float)
```

```
233
234         # Test not full
235         self.gut.amount.return_value = (amount, False)
236         self.assertEqual(self.Larva.add_plant(available),
237                          amount)
238         self.assertEqual(self.Larva.plant_gut,
239                          self.plant_gut.__add__.return_value)
240         self.assertEqual(self.plant_gut.__add__.call_args_list,
241                          [mk.call(amount)])
242         self.assertEqual(self.Larva.full, self.full)
243         self.assertEqual(self.gut.amount.call_args_list,
244                          [mk.call(self.Larva, available)])
245
246         self.Larva.plant_gut = self.plant_gut
247         self.plant_gut.reset_mock()
248         self.gut.amount.reset_mock()
249         # Test full
250         self.gut.amount.return_value = (amount, True)
251         self.assertEqual(self.Larva.add_plant(available),
252                          amount)
253         self.assertEqual(self.Larva.plant_gut,
254                          self.plant_gut.__add__.return_value)
255         self.assertEqual(self.plant_gut.__add__.call_args_list,
256                          [mk.call(amount)])
257         self.assertEqual(self.Larva.full, True)
258         self.assertNotEqual(self.full,    True)
259         self.assertEqual(self.gut.amount.call_args_list,
260                          [mk.call(self.Larva, available)])
261
262     def test_add_egg(self):
263         """test add_egg"""
264
265         available = mk.MagicMock(spec=float)
266         amount    = mk.MagicMock(spec=float)
267
268         # Test not full
269         self.gut.amount.return_value = (amount, False)
270         self.assertEqual(self.Larva.add_egg(available),
271                          amount)
```

```
        self.assertEqual(self.Larva.egg_gut,
                         self.egg_gut.__add__.return_value)
        self.assertEqual(self.egg_gut.__add__.call_args_list,
                         [mk.call(amount)])
        self.assertEqual(self.Larva.full, self.full)
        self.assertEqual(self.gut.amount.call_args_list,
                         [mk.call(self.Larva, available)])

        self.Larva.egg_gut = self.egg_gut
        self.egg_gut.reset_mock()
        self.gut.amount.reset_mock()
        # Test full
        self.gut.amount.return_value = (amount, True)
        self.assertEqual(self.Larva.add_egg(available),
                         amount)
        self.assertEqual(self.Larva.egg_gut,
                         self.egg_gut.__add__.return_value)
        self.assertEqual(self.egg_gut.__add__.call_args_list,
                         [mk.call(amount)])
        self.assertEqual(self.Larva.full, True)
        self.assertNotEqual(self.full,    True)
        self.assertEqual(self.gut.amount.call_args_list,
                         [mk.call(self.Larva, available)])

    def test_add_larva(self):
        """test add_larva"""

        available = mk.MagicMock(spec=float)
        amount    = mk.MagicMock(spec=float)

        # Test not full
        self.gut.amount.return_value = (amount, False)
        self.assertEqual(self.Larva.add_larva(available),
                         amount)
        self.assertEqual(self.Larva.larva_gut,
                         self.larva_gut.__add__.return_value)
        self.assertEqual(self.larva_gut.__add__.call_args_list,
                         [mk.call(amount)])
        self.assertEqual(self.Larva.full, self.full)
```

```
311         self.assertEqual(self.gut.amount.call_args_list,
312                          [mk.call(self.Larva, available)])
313 
314         self.Larva.larva_gut = self.larva_gut
315         self.larva_gut.reset_mock()
316         self.gut.amount.reset_mock()
317         # Test full
318         self.gut.amount.return_value = (amount, True)
319         self.assertEqual(self.Larva.add_larva(available),
320                          amount)
321         self.assertEqual(self.Larva.larva_gut,
322                          self.larva_gut.__add__.return_value)
323         self.assertEqual(self.larva_gut.__add__.call_args_list,
324                          [mk.call(amount)])
325         self.assertEqual(self.Larva.full, True)
326         self.assertNotEqual(self.full,    True)
327         self.assertEqual(self.gut.amount.call_args_list,
328                          [mk.call(self.Larva, available)])
329 
330     def test_grow(self):
331         """test grow the larva"""
332 
333         self.biomass.grow.return_value.__lt__.side_effect = [False, True]
334 
335         # Larva is not alive
336         self.Larva.alive = False
337         #   Test that it does not starve
338         self.assertEqual(self.Larva.grow(), [])
339         self.assertEqual(self.Larva.mass, self.mass)
340         self.assertEqual(self.mass.__add__.call_args_list, [])
341         self.assertEqual(self.biomass.grow.call_args_list, [])
342         self.assertEqual(self.Larva.starve, self.starve)
343 
344         # Larva is alive
345         self.Larva.alive = True
346         #   Test that it does not starve
347         self.assertEqual(self.Larva.grow(), [])
348         self.assertEqual(self.Larva.mass,
349                          self.mass.__add__.return_value)
```

```
350         self.assertEqual(self.mass.__add__.call_args_list,
351                          [mk.call(self.biomass.grow.return_value)])
352         self.assertEqual(self.biomass.grow.call_args_list,
353                          [mk.call(self.Larva)])
354         self.assertEqual(self.Larva.starve, self.starve)
355         self.assertEqual(self.biomass.grow.return_value.__lt__.call_args_list,
356                          [mk.call(0)])
357 
358         self.biomass.reset_mock()
359         self.mass.reset_mock()
360         self.Larva.mass = self.mass
361         self.mass.reset_mock()
362 
363         #   Test that it does starve
364         self.assertEqual(self.Larva.grow(), [])
365         self.assertEqual(self.Larva.mass, self.mass)
366         self.assertEqual(self.mass.__add__.call_args_list, [])
367         self.assertEqual(self.biomass.grow.call_args_list,
368                          [mk.call(self.Larva)])
369         self.assertEqual(self.Larva.starve, True)
370         self.assertEqual(self.biomass.grow.return_value.__lt__.call_args_list,
371                          [mk.call(0)])
372 
373     def test_survive(self):
374         """test run survive behavior"""
375 
376         # Larva is not alive
377         self.Larva.alive = False
378         self.assertEqual(self.Larva.survive(), [])
379         self.assertEqual(self.survival.survive.call_args_list, [])
380 
381         # Larva is alive
382         self.Larva.alive = True
383         self.assertEqual(self.Larva.survive(), [])
384         self.assertEqual(self.survival.survive.call_args_list,
385                          [mk.call(self.Larva)])
386 
387     def test_develop(self):
388         """test run develop behavior"""
```

```
389
390         # Larva is not alive
391         self.Larva.alive = False
392         self.assertEqual(self.Larva.develop(), [])
393         self.assertEqual(self.development.develop.call_args_list, [])
394
395         # Larva is alive
396         self.Larva.alive = True
397         self.assertEqual(self.Larva.develop(), [])
398         self.assertEqual(self.development.develop.call_args_list,
399                          [mk.call(self.Larva)])
400
401     def test_move(self):
402         """test run move behavior"""
403
404         with mk.patch.object(larva.Larva, '_can_consume',
405                              autospec=True) as mkCan:
406             with mk.patch.object(larva.Larva, '_has_target',
407                                  autospec=True) as mkHas:
408                 mkCan.__get__ = mk.MagicMock(side_effect=[False, True, True])
409                 mkHas.__get__ = mk.MagicMock(side_effect=[       True, False])
410
411                 # Cannot consume
412                 self.assertEqual(self.Larva.move(), [])
413                 self.assertEqual(self.movement.move.call_args_list, [])
414
415                 # Can consume and has target
416                 self.assertEqual(self.Larva.move(), [])
417                 self.assertEqual(self.movement.move.call_args_list, [])
418
419                 # Can consume and does not have target
420                 self.assertEqual(self.Larva.move(), [])
421                 self.assertEqual(self.movement.move.call_args_list,
422                                  [mk.call(self.Larva)])
423
424     def test__consume_plant(self):
425         """test consume the plant"""
426
427         with mk.patch.object(larva.Larva, '_can_consume',
```

```
                            autospec=True) as mkCan:
            mkCan.__get__ = mk.MagicMock(side_effect=[False, True])

            # Cannot consume
            self.Larva._consume_plant()
            self.assertEqual(self.forage_plant.consume.call_args_list, [])

            # Can consume
            self.Larva._consume_plant()
            self.assertEqual(self.forage_plant.consume.call_args_list,
                             [mk.call(self.Larva)])

    def test_consume_egg(self):
        """test consume the egg"""

        egg_mass = mk.create_autospec(agent_egg_mass.EggMass, spec_set=True)

        self.Larva.consume_egg(egg_mass)
        self.assertEqual(self.Larva.target, egg_mass)
        self.assertEqual(self.forage_egg.consume.call_args_list,
                         [mk.call(self.Larva, egg_mass)])

    def test_consume_larva(self):
        """test consume the larva"""

        target = mk.create_autospec(larva.Larva, spec_set=True)

        self.Larva.consume_larva(target)
        self.assertEqual(self.Larva.target, target)
        self.assertEqual(self.forage_larva.consume.call_args_list,
                         [mk.call(self.Larva, target)])

    def test__consume_target(self):
        """test consume the target"""

        with mk.patch.object(larva.Larva, '_has_target',
                             autospec=True) as mkHas:
            mkHas.__get__ = mk.MagicMock(side_effect=[False, True, True])

```

```
            # Has no target
            self.Larva._consume_target()
            self.assertEqual(self.loss.consume.call_args_list, [])

            # Has target and is not alive
            self.Larva.alive = False
            self.Larva._consume_target()
            self.assertEqual(self.loss.consume.call_args_list, [])

            # Has target and is alive
            self.Larva.alive = True
            self.Larva._consume_target()
            self.assertEqual(self.loss.consume.call_args_list,
                             [mk.call(self.Larva)])

    def test__location_keys(self):
        """test get the location keys for cannibalism"""

        kwargs = {'test': mk.MagicMock()}

        vertices = {mk.MagicMock(spec=int) for _ in range(3)}

        locs          = []
        location_keys = []
        for _ in vertices:
            loc = mk.create_autospec(location.Location, spec_set=True)
            location_keys.append(loc.location_key)
            locs.append(loc)

        self.location.copy.side_effect = locs

        with mk.patch.object(larva.Larva, 'vertices',
                             autospec=True) as mkVertices:
            mkVertices.return_value = vertices

            self.assertEqual(self.Larva._location_keys(**kwargs), location_keys)
            self.assertEqual(mkVertices.call_args_list,
                             [mk.call(self.Larva, **kwargs)])
            for index, vertex in enumerate(vertices):
```

```
506                 self.assertEqual(locs[index].__setitem__.call_args_list,
507                                  [mk.call(keyword.larva_level, vertex)])
508                 self.assertEqual(self.location.copy.call_args_list[index],
509                                  [mk.call()])
510             self.assertEqual(len(self.location.copy.call_args_list), 3)
511
512     def test_targets(self):
513         """test get the targets"""
514
515         kwargs = {'test': mk.MagicMock()}
516
517         self.simulation.agents = mk.create_autospec(agents.Agents,
518                                                     spec_set=True)
519
520         location_keys = []
521         agents_bins   = []
522         targets       = []
523         for index in range(3):
524             location_key = mk.MagicMock(spec=tuple)
525             agents_bin   = mk.create_autospec(agents.AgentsBin, spec_set=True)
526             egg_bin      = mk.create_autospec(agents.AgentBin, spec_set=True)
527             larva_bin    = mk.create_autospec(agents.AgentBin, spec_set=True)
528
529             eggs   = []
530             larvae = []
531             if index == 0:
532                 larvae.append(self.Larva)
533
534             for _ in range(3):
535                 eggs.append(mk.create_autospec(EggMassTest, spec_set=True))
536                 larvae.append(mk.create_autospec(larva.Larva, spec_set=True))
537             egg_bin.  agents = eggs
538             larva_bin.agents = larvae
539             targets.extend(eggs)
540             targets.extend(larvae)
541             agents_bin.__getitem__.side_effect = [egg_bin, larva_bin]
542
543             location_keys.append(location_key)
544             agents_bins.  append(agents_bin)
```

```
545
546         self.simulation.agents.__getitem__.side_effect = agents_bins
547
548         self.assertIn(self.Larva, targets)
549         with mk.patch.object(larva.Larva, '_location_keys',
550                              autospec=True) as mkKeys:
551             mkKeys.return_value = location_keys
552
553             new = self.Larva.targets(**kwargs)
554             self.assertEqual(len(targets), len(new) + 1)
555             targets.remove(self.Larva)
556             self.assertEqual(targets, new)
557
558             self.assertEqual(mkKeys.call_args_list,
559                              [mk.call(self.Larva, **kwargs)])
560             for index, call in enumerate(self.simulation.agents.
561                                          __getitem__.call_args_list):
562                 self.assertEqual(call,
563                                  mk.call(location_keys[index]))
564             self.assertEqual(len(self.simulation.agents.
565                                  __getitem__.call_args_list), 3)
566             for index, agent_bin in enumerate(agents_bins):
567                 self.assertEqual(agent_bin.__getitem__.call_args_list,
568                                  [mk.call(keyword.egg_mass),
569                                   mk.call(keyword.larva)])
570
571     def test_consume(self):
572         """test run the consume system"""
573
574         with mk.patch.object(larva.Larva, '_consume_target') as mkTarget:
575             with mk.patch.object(larva.Larva, '_consume_plant') as mkPlant:
576                 master = mk.MagicMock()
577                 master.attach_mock(mkTarget,         'target')
578                 master.attach_mock(mkPlant,          'plant')
579                 master.attach_mock(self.cannibalism, 'cannibalism')
580
581                 self.assertEqual(self.Larva.consume(), [])
582                 self.assertEqual(master.mock_calls,
583                                  [mk.call.target(),
```

```
                                mk.call.cannibalism.cannibalism(self.Larva),
                                mk.call.plant()])

    def test_reset(self):
        """test empty the gut system"""

        self.assertEqual(self.Larva.reset(), [])
        self.assertEqual(self.Larva.plant_gut, 0)
        self.assertEqual(self.Larva.egg_gut,   0)
        self.assertEqual(self.Larva.larva_gut, 0)
        self.assertEqual(self.Larva.full,      False)

    def test_initialize(self):
        """test initialize a larva"""

        self.simulation.behaviors = mk.create_autospec(BehaviorsTest,
                                                       spec_set=True)
        self.simulation.behaviors.gut           = self.gut
        self.simulation.behaviors.mass          = self.biomass
        self.simulation.behaviors.survive_larva = self.survival
        self.simulation.behaviors.develop_larva = self.development
        self.simulation.behaviors.move_larva    = self.movement
        self.simulation.behaviors.cannibalism   = self.cannibalism
        self.simulation.behaviors.forage_plant  = self.forage_plant
        self.simulation.behaviors.forage_egg    = self.forage_egg
        self.simulation.behaviors.forage_larva  = self.forage_larva

        self.Larva = larva.Larva.initialize(self.unique_id,
                                            self.simulation,
                                            self.location,
                                            self.mass,
                                            self.genotype)

        self.assertIsInstance(self.Larva, agent.Agent)
        self.assertIsInstance(self.Larva, insect.Insect)
        self.assertIsInstance(self.Larva, larva.Larva)

        self.assertEqual(self.Larva.agent_key, keyword.larva)
        self.assertEqual(self.Larva.alive,     True)
```

```
        self.assertEqual(self.Larva.age,        0)
        self.assertEqual(self.Larva.death,      keyword.alive)

        self.assertEqual(self.Larva.plant_gut, 0)
        self.assertEqual(self.Larva.egg_gut,   0)
        self.assertEqual(self.Larva.larva_gut, 0)
        self.assertEqual(self.Larva.full,       False)
        self.assertEqual(self.Larva.starve,     False)
        self.assertEqual(self.Larva.target,     None)

        self.assertEqual(self.Larva.unique_id,  self.unique_id)
        self.assertEqual(self.Larva.simulation, self.simulation)
        self.assertEqual(self.Larva.location,   self.location)

        self.assertEqual(self.Larva.mass,       self.mass)
        self.assertEqual(self.Larva.genotype,   self.genotype)

        self.assertEqual(self.Larva.gut,          self.gut)
        self.assertEqual(self.Larva.biomass,      self.biomass)
        self.assertEqual(self.Larva.survival,     self.survival)
        self.assertEqual(self.Larva.development,  self.development)
        self.assertEqual(self.Larva.movement,     self.movement)
        self.assertEqual(self.Larva.forage_plant, self.forage_plant)
        self.assertEqual(self.Larva.forage_egg,   self.forage_egg)
        self.assertEqual(self.Larva.forage_larva, self.forage_larva)
        self.assertEqual(self.Larva.cannibalism,  self.cannibalism,)

        # noinspection PyTypeChecker
        self.assertIsInstance(self.Larva._age_count, i_tools.count)
        # noinspection PyTypeChecker
        self.assertEqual(next(self.Larva._age_count), next(i_tools.count(1)))

        self.assertTrue(dclass.is_dataclass(self.Larva))

    def test_setup(self):
        """test setup an initial larva"""

        self.simulation.behaviors = mk.create_autospec(BehaviorsTest,
                                                       spec_set=True)
```

```
662         self.simulation.behaviors.gut           = self.gut
663         self.simulation.behaviors.mass          = self.biomass
664         self.simulation.behaviors.survive_larva = self.survival
665         self.simulation.behaviors.develop_larva = self.development
666         self.simulation.behaviors.move_larva    = self.movement
667         self.simulation.behaviors.cannibalism   = self.cannibalism
668         self.simulation.behaviors.forage_plant  = self.forage_plant
669         self.simulation.behaviors.forage_egg    = self.forage_egg
670         self.simulation.behaviors.forage_larva  = self.forage_larva
671         self.simulation.space  = mk.create_autospec(space.Space, spec_set=True)
672         self.simulation.models = mk.create_autospec(models.Models,
673                                                     spec_set=True)
674
675         unique_id_num = mk.MagicMock(spec=int)
676         initial_key   = mk.MagicMock(spec=str)
677
678         self.Larva = larva.Larva.setup(unique_id_num,
679                                        initial_key,
680                                        self.simulation,
681                                        self.genotype)
682
683         self.assertIsInstance(self.Larva, agent.Agent)
684         self.assertIsInstance(self.Larva, insect.Insect)
685         self.assertIsInstance(self.Larva, larva.Larva)
686
687         self.assertEqual(self.Larva.unique_id,
688                          str(initial_key)   +
689                          str(unique_id_num) +
690                          keyword.larva)
691         self.assertEqual(self.Larva.location,
692                          self.simulation.space.new_location.return_value)
693         self.assertEqual(self.simulation.space.new_location.call_args_list,
694                          [mk.call(keyword.larva_depth)])
695         self.assertEqual(self.Larva.mass,
696                          self.simulation.models.
697                          __getitem__.return_value.return_value)
698         self.assertEqual(self.simulation.models.
699                          __getitem__.return_value.call_args_list,
700                          [mk.call(self.genotype)])
```

```
701         self.assertEqual(self.simulation.models.
702                          __getitem__.call_args_list,
703                          [mk.call(keyword.init_juvenile)])
704 
705         self.assertEqual(self.Larva.agent_key, keyword.larva)
706         self.assertEqual(self.Larva.alive,     True)
707         self.assertEqual(self.Larva.age,       0)
708         self.assertEqual(self.Larva.death,     keyword.alive)
709 
710         self.assertEqual(self.Larva.plant_gut, 0)
711         self.assertEqual(self.Larva.egg_gut,   0)
712         self.assertEqual(self.Larva.larva_gut, 0)
713         self.assertEqual(self.Larva.full,      False)
714         self.assertEqual(self.Larva.starve,    False)
715         self.assertEqual(self.Larva.target,    None)
716 
717         self.assertEqual(self.Larva.simulation, self.simulation)
718 
719         self.assertEqual(self.Larva.genotype,   self.genotype)
720 
721         self.assertEqual(self.Larva.gut,          self.gut)
722         self.assertEqual(self.Larva.biomass,      self.biomass)
723         self.assertEqual(self.Larva.survival,     self.survival)
724         self.assertEqual(self.Larva.development,  self.development)
725         self.assertEqual(self.Larva.movement,     self.movement)
726         self.assertEqual(self.Larva.forage_plant, self.forage_plant)
727         self.assertEqual(self.Larva.forage_egg,   self.forage_egg)
728         self.assertEqual(self.Larva.forage_larva, self.forage_larva)
729         self.assertEqual(self.Larva.cannibalism,  self.cannibalism,)
730 
731         # noinspection PyTypeChecker
732         self.assertIsInstance(self.Larva._age_count, i_tools.count)
733         # noinspection PyTypeChecker
734         self.assertEqual(next(self.Larva._age_count), next(i_tools.count(1)))
735 
736         self.assertTrue(dclass.is_dataclass(self.Larva))
737 
738     def test_advance(self):
739         """test advance a egg into a larva"""
```

```
740
741         self.simulation.behaviors = mk.create_autospec(BehaviorsTest,
742                                                        spec_set=True)
743         self.simulation.behaviors.gut           = self.gut
744         self.simulation.behaviors.mass          = self.biomass
745         self.simulation.behaviors.survive_larva = self.survival
746         self.simulation.behaviors.develop_larva = self.development
747         self.simulation.behaviors.move_larva    = self.movement
748         self.simulation.behaviors.cannibalism   = self.cannibalism
749         self.simulation.behaviors.forage_plant  = self.forage_plant
750         self.simulation.behaviors.forage_egg    = self.forage_egg
751         self.simulation.behaviors.forage_larva  = self.forage_larva
752         self.simulation.behaviors.target        = self.loss
753
754         egg = mk.create_autospec(EggTest, spec_set=True)
755         egg.unique_id  = self.unique_id
756         egg.simulation = self.simulation
757         egg.location   = self.location
758         egg.mass       = self.mass
759         egg.genotype   = self.genotype
760
761         self.Larva = larva.Larva.advance(egg)
762
763         self.assertIsInstance(self.Larva, agent.Agent)
764         self.assertIsInstance(self.Larva, insect.Insect)
765         self.assertIsInstance(self.Larva, larva.Larva)
766
767         self.assertEqual(self.Larva.agent_key, keyword.larva)
768         self.assertEqual(self.Larva.alive,     True)
769         self.assertEqual(self.Larva.age,       0)
770         self.assertEqual(self.Larva.death,     keyword.alive)
771
772         self.assertEqual(self.Larva.plant_gut, 0)
773         self.assertEqual(self.Larva.egg_gut,   0)
774         self.assertEqual(self.Larva.larva_gut, 0)
775         self.assertEqual(self.Larva.full,      False)
776         self.assertEqual(self.Larva.starve,    False)
777         self.assertEqual(self.Larva.target,    None)
778
```

```
        self.assertEqual(self.Larva.unique_id,  self.unique_id)
        self.assertEqual(self.Larva.simulation, self.simulation)
        self.assertEqual(self.Larva.location,   self.location)

        self.assertEqual(self.Larva.mass,       self.mass)
        self.assertEqual(self.Larva.genotype,   self.genotype)

        self.assertEqual(self.Larva.gut,          self.gut)
        self.assertEqual(self.Larva.biomass,      self.biomass)
        self.assertEqual(self.Larva.survival,     self.survival)
        self.assertEqual(self.Larva.development,  self.development)
        self.assertEqual(self.Larva.movement,     self.movement)
        self.assertEqual(self.Larva.forage_plant, self.forage_plant)
        self.assertEqual(self.Larva.forage_egg,   self.forage_egg)
        self.assertEqual(self.Larva.forage_larva, self.forage_larva)
        self.assertEqual(self.Larva.loss,         self.loss)
        self.assertEqual(self.Larva.cannibalism,  self.cannibalism)

        # noinspection PyTypeChecker
        self.assertIsInstance(self.Larva._age_count, i_tools.count)
        # noinspection PyTypeChecker
        self.assertEqual(next(self.Larva._age_count), next(i_tools.count(1)))

        self.assertTrue(dclass.is_dataclass(self.Larva))
```

C.5.1.7 test_pupa.py

```
import unittest      as ut
import unittest.mock as mk

import dataclasses as dclass
import itertools   as i_tools

import source.keyword as keyword

import source.agents.agent  as agent
import source.agents.larva  as agent_larva
import source.agents.pupa   as pupa
import source.agents.insect as insect

import source.simulation.behaviors  as behaviors
import source.simulation.models     as models
import source.simulation.simulation as simulation

import source.space.agents   as agents
import source.space.location as location
import source.space.space    as space

import source.survival.pupa    as survival
import source.development.pupa as development


class BehaviorsTest(behaviors.Behaviors):
    """Class to add dynamic values for tests"""

    survive_pupa = mk.create_autospec(survival.Pupa,    spec_set=True)
    develop_pupa = mk.create_autospec(development.Pupa, spec_set=True)


class SimulationTest(simulation.Simulation):
    """Class to add dynamic values for tests"""

    agents    = mk.create_autospec(agents.Agents, spec_set=True)
    behaviors = mk.create_autospec(BehaviorsTest, spec_set=True)
```

```
    space     = mk.create_autospec(space.Space,   spec_set=True)
    models    = mk.create_autospec(models.Models, spec_set=True)


class LarvaTest(agent_larva.Larva):
    """Class to add dynamic values for tests"""

    unique_id  = mk.MagicMock(spec=str)
    simulation = mk.create_autospec(SimulationTest,    spec_set=True)
    location   = mk.create_autospec(location.Location, spec_set=True)
    mass       = mk.MagicMock(spec=float)
    genotype   = mk.MagicMock(spec=str)


class TestPupa(ut.TestCase):
    """test base Pupa class"""

    def setUp(self):
        """Setup the tests"""

        self.agent_key = mk.MagicMock(spec=str)
        self.unique_id = mk.MagicMock(spec=str)
        self.alive     = mk.MagicMock(spec=bool)
        self.mass      = mk.MagicMock(spec=float)
        self.genotype  = mk.MagicMock(spec=str)
        self.age       = mk.MagicMock(spec=int)
        self.death     = mk.MagicMock(spec=str)

        self.simulation  = mk.create_autospec(SimulationTest,    spec_set=True)
        self.location    = mk.create_autospec(location.Location, spec_set=True)

        self.survival    = mk.create_autospec(survival.Pupa,    spec_set=True)
        self.development = mk.create_autospec(development.Pupa, spec_set=True)

        self.Pupa = pupa.Pupa(self.agent_key,
                              self.unique_id,
                              self.simulation,
                              self.location,
                              self.alive,
```

```
                             self.mass,
                             self.genotype,
                             self.age,
                             self.death,
                             self.survival,
                             self.development)

    def test___init__(self):
        """test __init__ for class"""

        self.assertIsInstance(self.Pupa, agent.Agent)
        self.assertIsInstance(self.Pupa, insect.Insect)
        self.assertIsInstance(self.Pupa, pupa.Pupa)

        self.assertEqual(self.Pupa.agent_key,  self.agent_key)
        self.assertEqual(self.Pupa.unique_id,  self.unique_id)
        self.assertEqual(self.Pupa.simulation, self.simulation)
        self.assertEqual(self.Pupa.location,   self.location)
        self.assertEqual(self.Pupa.alive,      self.alive)

        self.assertEqual(self.Pupa.mass,     self.mass)
        self.assertEqual(self.Pupa.genotype, self.genotype)
        self.assertEqual(self.Pupa.age,      self.age)
        self.assertEqual(self.Pupa.death,    self.death)

        self.assertEqual(self.Pupa.survival,    self.survival)
        self.assertEqual(self.Pupa.development, self.development)

        # noinspection PyTypeChecker
        self.assertIsInstance(self.Pupa._age_count, i_tools.count)
        # noinspection PyTypeChecker
        self.assertEqual(next(self.Pupa._age_count),
                         next(i_tools.count(self.age + 1)))

        self.assertTrue(dclass.is_dataclass(self.Pupa))

    def test_survive(self):
        """test run survive behavior"""

```

```
116         # Pupa is not alive
117         self.Pupa.alive = False
118         self.assertEqual(self.Pupa.survive(), [])
119         self.assertEqual(self.survival.survive.call_args_list, [])
120 
121         # Pupa is alive
122         self.Pupa.alive = True
123         self.assertEqual(self.Pupa.survive(), [])
124         self.assertEqual(self.survival.survive.call_args_list,
125                          [mk.call(self.Pupa)])
126 
127     def test_develop(self):
128         """test run develop behavior"""
129 
130         # Pupa is not alive
131         self.Pupa.alive = False
132         self.assertEqual(self.Pupa.develop(), [])
133         self.assertEqual(self.development.develop.call_args_list, [])
134 
135         # Pupa is alive
136         self.Pupa.alive = True
137         self.assertEqual(self.Pupa.develop(), [])
138         self.assertEqual(self.development.develop.call_args_list,
139                          [mk.call(self.Pupa)])
140 
141     def test_initialize(self):
142         """test initialize a pupa"""
143 
144         self.simulation.behaviors = mk.create_autospec(BehaviorsTest,
145                                                        spec_set=True)
146         self.simulation.behaviors.survive_pupa = self.survival
147         self.simulation.behaviors.develop_pupa = self.development
148 
149         self.Pupa = pupa.Pupa.initialize(self.unique_id,
150                                          self.simulation,
151                                          self.location,
152                                          self.mass,
153                                          self.genotype)
154 
```

```
        self.assertIsInstance(self.Pupa, agent.Agent)
        self.assertIsInstance(self.Pupa, insect.Insect)
        self.assertIsInstance(self.Pupa, pupa.Pupa)

        self.assertEqual(self.Pupa.agent_key, keyword.pupa)
        self.assertEqual(self.Pupa.alive,     True)
        self.assertEqual(self.Pupa.age,       0)
        self.assertEqual(self.Pupa.death,     keyword.alive)

        self.assertEqual(self.Pupa.unique_id,  self.unique_id)
        self.assertEqual(self.Pupa.simulation, self.simulation)
        self.assertEqual(self.Pupa.location,   self.location)

        self.assertEqual(self.Pupa.mass,     self.mass)
        self.assertEqual(self.Pupa.genotype, self.genotype)

        self.assertEqual(self.Pupa.survival,    self.survival)
        self.assertEqual(self.Pupa.development, self.development)

        # noinspection PyTypeChecker
        self.assertIsInstance(self.Pupa._age_count, i_tools.count)
        # noinspection PyTypeChecker
        self.assertEqual(next(self.Pupa._age_count), next(i_tools.count(1)))

        self.assertTrue(dclass.is_dataclass(self.Pupa))

    def test_setup(self):
        """test setup an initial pupa"""

        self.simulation.behaviors = mk.create_autospec(BehaviorsTest,
                                                       spec_set=True)
        self.simulation.behaviors.survive_pupa = self.survival
        self.simulation.behaviors.develop_pupa = self.development
        self.simulation.space  = mk.create_autospec(space.Space, spec_set=True)
        self.simulation.models = mk.create_autospec(models.Models,
                                                    spec_set=True)

        unique_id_num = mk.MagicMock(spec=int)
        initial_key   = mk.MagicMock(spec=str)
```

```
194
195         self.Pupa = pupa.Pupa.setup(unique_id_num,
196                                     initial_key,
197                                     self.simulation,
198                                     self.genotype)
199
200         self.assertIsInstance(self.Pupa, agent.Agent)
201         self.assertIsInstance(self.Pupa, insect.Insect)
202         self.assertIsInstance(self.Pupa, pupa.Pupa)
203
204         self.assertEqual(self.Pupa.unique_id,
205                          str(initial_key)   +
206                          str(unique_id_num) +
207                          keyword.pupa)
208         self.assertEqual(self.Pupa.location,
209                          self.simulation.space.new_location.return_value)
210         self.assertEqual(self.simulation.space.new_location.call_args_list,
211                          [mk.call(keyword.pupa_depth)])
212         self.assertEqual(self.Pupa.mass,
213                          self.simulation.models.
214                             __getitem__.return_value.return_value)
215         self.assertEqual(self.simulation.models.
216                             __getitem__.return_value.call_args_list,
217                          [mk.call(self.genotype)])
218         self.assertEqual(self.simulation.models.
219                             __getitem__.call_args_list,
220                          [mk.call(keyword.init_mature)])
221
222         self.assertEqual(self.Pupa.agent_key, keyword.pupa)
223         self.assertEqual(self.Pupa.alive,     True)
224         self.assertEqual(self.Pupa.age,       0)
225         self.assertEqual(self.Pupa.death,     keyword.alive)
226
227         self.assertEqual(self.Pupa.simulation, self.simulation)
228
229         self.assertEqual(self.Pupa.genotype, self.genotype)
230
231         self.assertEqual(self.Pupa.survival,    self.survival)
232         self.assertEqual(self.Pupa.development, self.development)
```

```
233
234         # noinspection PyTypeChecker
235         self.assertIsInstance(self.Pupa._age_count, i_tools.count)
236         # noinspection PyTypeChecker
237         self.assertEqual(next(self.Pupa._age_count), next(i_tools.count(1)))
238
239         self.assertTrue(dclass.is_dataclass(self.Pupa))
240
241     def test_advance(self):
242         """test advance a larva into a pupa"""
243
244         self.simulation.behaviors = mk.create_autospec(BehaviorsTest,
245                                                        spec_set=True)
246         self.simulation.behaviors.survive_pupa = self.survival
247         self.simulation.behaviors.develop_pupa = self.development
248
249         larva = mk.create_autospec(LarvaTest, spec_set=True)
250         larva.unique_id  = self.unique_id
251         larva.simulation = self.simulation
252         larva.location   = self.location
253         larva.mass       = self.mass
254         larva.genotype   = self.genotype
255
256         self.Pupa = pupa.Pupa.advance(larva)
257
258         self.assertIsInstance(self.Pupa, agent.Agent)
259         self.assertIsInstance(self.Pupa, insect.Insect)
260         self.assertIsInstance(self.Pupa, pupa.Pupa)
261
262         self.assertEqual(self.Pupa.agent_key, keyword.pupa)
263         self.assertEqual(self.Pupa.alive,     True)
264         self.assertEqual(self.Pupa.age,       0)
265         self.assertEqual(self.Pupa.death,     keyword.alive)
266
267         self.assertEqual(self.Pupa.unique_id,  self.unique_id)
268         self.assertEqual(self.Pupa.simulation, self.simulation)
269         self.assertEqual(self.Pupa.location,
270                          self.location.__getitem__.return_value)
271         self.assertEqual(self.location.__getitem__.call_args_list,
```

```
                        [mk.call(slice(None, keyword.pupa_depth, None))])

        self.assertEqual(self.Pupa.mass,     self.mass)
        self.assertEqual(self.Pupa.genotype, self.genotype)

        self.assertEqual(self.Pupa.survival,    self.survival)
        self.assertEqual(self.Pupa.development, self.development)

        # noinspection PyTypeChecker
        self.assertIsInstance(self.Pupa._age_count, i_tools.count)
        # noinspection PyTypeChecker
        self.assertEqual(next(self.Pupa._age_count), next(i_tools.count(1)))

        self.assertTrue(dclass.is_dataclass(self.Pupa))
```

C.5.2 test_biomass

```
FallArmyworm
└── test
    └── test_biomass
        ├── test_gut.py
        ├── test_mass.py
        └── test_models.py
```

C.5.2.1 test_gut.py

```
import unittest      as ut
import unittest.mock as mk

import dataclasses as dclass

import source.keyword as keyword

import source.agents.larva as agent_larva

import source.biomass.gut as gut


class LarvaTest(agent_larva.Larva):
    """Class to add dynamic values for tests"""

    mass      = mk.MagicMock(spec=float)
    plant_gut = mk.MagicMock(spec=float)
    egg_gut   = mk.MagicMock(spec=float)
    larva_gut = mk.MagicMock(spec=float)


class TestGut(ut.TestCase):
    """test the Gut system class"""

    def setUp(self):
        """Setup the tests"""

        self.max_gut = mk.MagicMock(spec=callable)

        self.Gut = gut.Gut(self.max_gut)

    def test___init__(self):
        """test __init__ for class"""

        self.assertIsInstance(self.Gut, gut.Gut)

        self.assertEqual(self.Gut.max_gut, self.max_gut)
```

```
38
39          self.assertTrue(dclass.is_dataclass(self.Gut))
40
41      def test__volume(self):
42          """test get the volume of the larva"""
43
44          larva = mk.create_autospec(LarvaTest, spec_set=True)
45
46          self.assertEqual(self.Gut._volume(larva),
47                           larva.plant_gut.__add__.return_value.
48                              __add__.return_value)
49          self.assertEqual(larva.plant_gut.__add__.return_value.
50                              __add__.call_args_list,
51                           [mk.call(larva.larva_gut)])
52          self.assertEqual(larva.plant_gut.__add__.call_args_list,
53                           [mk.call(larva.egg_gut)])
54
55      def test__remaining(self):
56          """test get the remaining gut volume for the larva"""
57
58          larva = mk.create_autospec(LarvaTest, spec_set=True)
59
60          with mk.patch.object(gut.Gut, '_volume', autospec=True) as mkVolume:
61              self.assertEqual(self.Gut._remaining(larva),
62                               self.max_gut.return_value.__sub__.return_value)
63              self.assertEqual(self.max_gut.return_value.__sub__.call_args_list,
64                               [mk.call(mkVolume.return_value)])
65              self.assertEqual(self.max_gut.call_args_list,
66                               [mk.call(larva.mass)])
67              self.assertEqual(mkVolume.call_args_list,
68                               [mk.call(larva)])
69
70      def test_amount(self):
71          """test get the amount of mass to remove"""
72
73          larva     = mk.create_autospec(LarvaTest, spec_set=True)
74          available = mk.MagicMock(spec=float)
75
76          available.__ge__.side_effect = [True, False]
```

```
77
78          with mk.patch.object(gut.Gut, '_remaining',
79                               autospec=True) as mkRemaining:
80              # Test if available is too much
81              self.assertEqual(self.Gut.amount(larva, available),
82                               (mkRemaining.return_value, True))
83              self.assertEqual(available.__ge__.call_args_list,
84                               [mk.call(mkRemaining.return_value)])
85              self.assertEqual(mkRemaining.call_args_list,
86                               [mk.call(self.Gut, larva)])
87
88              available.reset_mock()
89              mkRemaining.reset_mock()
90              # Test if available not enough
91              self.assertEqual(self.Gut.amount(larva, available),
92                               (available, False))
93              self.assertEqual(available.__ge__.call_args_list,
94                               [mk.call(mkRemaining.return_value)])
95              self.assertEqual(mkRemaining.call_args_list,
96                               [mk.call(self.Gut, larva)])
97
98      def test_setup(self):
99          """test setup the class"""
100
101         kwargs = {keyword.max_gut: self.max_gut}
102
103         self.Gut = gut.Gut.setup(**kwargs)
104         self.assertIsInstance(self.Gut, gut.Gut)
105         self.assertEqual(self.Gut.max_gut, self.max_gut)
```

C.5.2.2 test_mass.py

```
import unittest      as ut
import unittest.mock as mk

import dataclasses as dclass

import source.keyword as keyword

import source.agents.larva as agent_larva

import source.biomass.mass as mass


class LarvaTest(agent_larva.Larva):
    """Class to add dynamic values for tests"""

    genotype  = mk.MagicMock(spec=str)
    mass      = mk.MagicMock(spec=float)
    plant_gut = mk.MagicMock(spec=float)
    egg_gut   = mk.MagicMock(spec=float)
    larva_gut = mk.MagicMock(spec=float)


class TestMass(ut.TestCase):
    """test the mass system class"""

    def setUp(self):
        """Setup the tests"""

        self.max_gut = mk.MagicMock(spec=callable)
        self.growth  = mk.MagicMock(spec=callable)

        self.Mass = mass.Mass(self.max_gut,
                              self.growth)

    def test___init__(self):
        """test __init__ for class"""

```

```
38          self.assertIsInstance(self.Mass, mass.Mass)
39
40          self.assertEqual(self.Mass.max_gut, self.max_gut)
41          self.assertEqual(self.Mass.growth,  self.growth)
42
43          self.assertTrue(dclass.is_dataclass(self.Mass))
44
45      def test__volume(self):
46          """test get the volume of the larva"""
47
48          larva = mk.create_autospec(LarvaTest, spec_set=True)
49
50          self.assertEqual(self.Mass._volume(larva),
51                           larva.plant_gut.__add__.return_value.
52                           __add__.return_value)
53          self.assertEqual(larva.plant_gut.__add__.return_value.
54                           __add__.call_args_list,
55                           [mk.call(larva.larva_gut)])
56          self.assertEqual(larva.plant_gut.__add__.call_args_list,
57                           [mk.call(larva.egg_gut)])
58
59      def test__ratio(self):
60          """test get the ratio of gut volume"""
61
62          larva  = mk.create_autospec(LarvaTest, spec_set=True)
63          volume = mk.MagicMock(spec=float)
64
65          with mk.patch.object(mass.Mass, '_volume', autospec=True) as mkVolume:
66              mkVolume.return_value = volume
67
68              self.assertEqual(self.Mass._ratio(larva),
69                               volume.__truediv__.return_value)
70              self.assertEqual(volume.__truediv__.call_args_list,
71                               [mk.call(self.max_gut.return_value)])
72              self.assertEqual(self.max_gut.call_args_list,
73                               [mk.call(larva.mass)])
74              self.assertEqual(mkVolume.call_args_list,
75                               [mk.call(larva)])
76
```

```
77      def test__energy(self):
78          """test get the energy for the simulation"""
79
80          larva_mass = mk.MagicMock(spec=float)
81          ratio      = mk.MagicMock(spec=float)
82
83          self.assertEqual(self.Mass._energy(larva_mass, ratio),
84                           ratio.__mul__.return_value)
85          self.assertEqual(ratio.__mul__.call_args_list,
86                           [mk.call(self.max_gut.return_value)])
87          self.assertEqual(self.max_gut.call_args_list,
88                           [mk.call(larva_mass)])
89
90      def test__rhs(self):
91          """test get the right hand side of the growth for RH4"""
92
93          larva = mk.create_autospec(LarvaTest, spec_set=True)
94          ratio = mk.MagicMock(spec=float)
95          shift = mk.MagicMock(spec=float)
96
97          with mk.patch.object(mass.Mass, '_energy', autospec=True) as mkEnergy:
98              self.assertEqual(self.Mass._rhs(larva, ratio, shift),
99                               self.growth.return_value)
100             self.assertEqual(self.growth.call_args_list,
101                              [mk.call(larva.mass.__add__.return_value,
102                                       mkEnergy.return_value,
103                                       larva.genotype)])
104             self.assertEqual(mkEnergy.call_args_list,
105                              [mk.call(self.Mass,
106                                       larva.mass.__add__.return_value,
107                                       ratio)])
108             self.assertEqual(larva.mass.__add__.call_args_list,
109                              [mk.call(shift)])
110
111     def test_grow(self):
112         """test get the amount we grow"""
113
114         larva = mk.create_autospec(LarvaTest, spec_set=True)
115
```

```
        ratio = mk.MagicMock(spec=float)

        k1 = mk.MagicMock(spec=float)
        k2 = mk.MagicMock(spec=float)
        k3 = mk.MagicMock(spec=float)
        k4 = mk.MagicMock(spec=float)

        effects = [k1, k2, k3, k4]

        with mk.patch.object(mass.Mass, '_ratio', autospec=True) as mkRatio:
            with mk.patch.object(mass.Mass, '_rhs', autospec=True) as mkRhs:
                mkRatio.return_value = ratio
                mkRhs.side_effect    = effects

                self.assertEqual(self.Mass.grow(larva),
                                 k1.__add__.return_value.
                                    __add__.return_value.
                                    __add__.return_value.
                                    __truediv__.return_value)
                self.assertEqual(k1.__add__.return_value.
                                    __add__.return_value.
                                    __add__.return_value.
                                    __truediv__.call_args_list,
                                 [mk.call(6)])
                self.assertEqual(k1.__add__.return_value.
                                    __add__.return_value.
                                    __add__.call_args_list,
                                 [mk.call(k4)])
                self.assertEqual(k1.__add__.return_value.
                                     __add__.call_args_list,
                                 [mk.call(k3.__mul__.return_value)])
                self.assertEqual(k1.__add__.call_args_list,
                                 [mk.call(k2.__mul__.return_value)])
                self.assertEqual(k2.__mul__.call_args_list,
                                 [mk.call(2)])
                self.assertEqual(k3.__mul__.call_args_list,
                                 [mk.call(2)])

                self.assertEqual(mkRhs.call_args_list,
```

```
                                [mk.call(self.Mass, larva, ratio, 0),
                                 mk.call(self.Mass, larva, ratio,
                                         k1.__truediv__.return_value),
                                 mk.call(self.Mass, larva, ratio,
                                         k2.__truediv__.return_value),
                                 mk.call(self.Mass, larva, ratio, k3)])
            self.assertEqual(k1.__truediv__.call_args_list,
                             [mk.call(2)])
            self.assertEqual(k2.__truediv__.call_args_list,
                             [mk.call(2)])
            self.assertEqual(mkRatio.call_args_list,
                             [mk.call(self.Mass, larva)])

    def test_setup(self):
        """test setup the class"""

        kwargs = {keyword.max_gut: self.max_gut,
                  keyword.growth:  self.growth}

        self.Mass = mass.Mass.setup(**kwargs)
        self.assertIsInstance(self.Mass, mass.Mass)
        self.assertEqual(self.Mass.max_gut, self.max_gut)
        self.assertEqual(self.Mass.growth,  self.growth)
```

C.5.2.3 test_models.py

```
import unittest      as ut
import unittest.mock as mk

import dataclasses as dclass
import numpy       as np
import scipy.stats as stats

import source.keyword as keyword

import source.biomass.models as model

import source.simulation.models as models


class TestMaxGut(ut.TestCase):
    """test the MaxGut mathematical model"""

    def setUp(self):
        """Setup the tests"""

        self.MaxGut = model.MaxGut()

    def test___init__(self):
        """test __init__ for class"""

        self.assertIsInstance(self.MaxGut, models.Model)
        self.assertIsInstance(self.MaxGut, model.MaxGut)

        self.assertEqual(self.MaxGut.model_key, keyword.max_gut)

        self.assertTrue(dclass.is_dataclass(self.MaxGut))

    def test___call__(self):
        """test call to get maximum gut volume"""

        mass = mk.MagicMock(spec=float)

```

```
        self.assertEqual(self.MaxGut(mass),
                         mass.__pow__.return_value)
        self.assertEqual(mass.__pow__.call_args_list,
                         [mk.call(0.75)])


class TestGrowth(ut.TestCase):
    """test the Growth mathematical model"""

    def setUp(self):
        """Setup the tests"""

        self.alpha = mk.MagicMock(spec=dict)
        self.beta  = mk.MagicMock(spec=dict)

        self.Growth = model.Growth(self.alpha,
                                   self.beta)

    def test___init__(self):
        """test __init__ for class"""

        self.assertIsInstance(self.Growth, models.Model)
        self.assertIsInstance(self.Growth, model.Growth)

        self.assertEqual(self.Growth.alpha, self.alpha)
        self.assertEqual(self.Growth.beta,  self.beta)

        self.assertEqual(self.Growth.model_key, keyword.growth)

        self.assertTrue(dclass.is_dataclass(self.Growth))

    def test___call__(self):
        """test call the mathematical model"""

        mass     = mk.MagicMock(spec=float)
        energy   = mk.MagicMock(spec=float)
        genotype = mk.MagicMock(spec=str)

        self.assertEqual(self.Growth(mass, energy, genotype),
```

```
                        self.alpha.__getitem__.return_value.
                            __mul__.return_value.__sub__.return_value)
        self.assertEqual(self.alpha.__getitem__.return_value.
                            __mul__.return_value.__sub__.call_args_list,
                        [mk.call(self.beta.__getitem__.return_value.
                                        __mul__.return_value)])
        self.assertEqual(self.alpha.__getitem__.return_value.
                            __mul__.call_args_list,
                        [mk.call(energy)])
        self.assertEqual(self.alpha.__getitem__.call_args_list,
                        [mk.call(genotype)])
        self.assertEqual(self.beta.__getitem__.return_value.
                            __mul__.call_args_list,
                        [mk.call(mass)])
        self.assertEqual(self.beta.__getitem__.call_args_list,
                        [mk.call(genotype)])


class TestInitNum(ut.TestCase):
    """test the InitNum mathematical model"""

    def setUp(self):
        """Setup the tests"""

        self.lam = mk.MagicMock(spec=float)

        self.InitNum = model.InitNum(self.lam)

    def test___init__(self):
        """test __init__ for class"""

        self.assertIsInstance(self.InitNum, models.Model)
        self.assertIsInstance(self.InitNum, model.InitNum)

        self.assertEqual(self.InitNum.lam, self.lam)

        self.assertEqual(self.InitNum.model_key, keyword.init_num)

        self.assertTrue(dclass.is_dataclass(self.InitNum))
```

```
116
117     def test___call__(self):
118         """test call the model"""
119
120         genotype = mk.MagicMock(spec=str)
121
122         with mk.patch.object(stats.poisson, 'rvs', autospec=True) as mkRVS:
123             with mk.patch.object(model, 'int') as mkInt:
124                 self.assertEqual(self.InitNum(genotype),
125                                  mkInt.return_value)
126                 self.assertEqual(mkInt.call_args_list,
127                                  [mk.call(mkRVS.return_value)])
128                 self.assertEqual(mkRVS.call_args_list,
129                                  [mk.call(self.lam)])
130
131
132 class TestInitMass(ut.TestCase):
133     """test the InitMass mathematical model"""
134
135     def setUp(self):
136         """Setup the tests"""
137
138         self.mu    = mk.MagicMock(spec=dict)
139         self.sigma = mk.MagicMock(spec=dict)
140
141         self.InitMass = model.InitMass(self.mu,
142                                        self.sigma)
143
144     def test___init__(self):
145         """test __init__ for class"""
146
147         self.assertIsInstance(self.InitMass, models.Model)
148         self.assertIsInstance(self.InitMass, model.InitMass)
149
150         self.assertEqual(self.InitMass.mu,    self.mu)
151         self.assertEqual(self.InitMass.sigma, self.sigma)
152
153         self.assertEqual(self.InitMass.model_key, keyword.init_mass)
154
```

```
155         self.assertTrue(dclass.is_dataclass(self.InitMass))
156
157     def test___call__(self):
158         """test call the model"""
159
160         genotype = mk.MagicMock(spec=str)
161
162         with mk.patch.object(stats.truncnorm, 'rvs',
163                              autospec=True) as mkRVS:
164             with mk.patch.object(model, 'float') as mkFloat:
165                 self.assertEqual(self.InitMass(genotype),
166                                  mkFloat.return_value)
167                 self.assertEqual(mkFloat.call_args_list,
168                                  [mk.call(mkRVS.return_value)])
169                 self.assertEqual(mkRVS.call_args_list,
170                                  [mk.call(0,
171                                           np.inf,
172                                           loc=self.mu.
173                                             __getitem__.return_value,
174                                           scale=self.sigma.
175                                             __getitem__.return_value)])
176                 self.assertEqual(self.mu.__getitem__.call_args_list,
177                                  [mk.call(genotype)])
178                 self.assertEqual(self.sigma.__getitem__.call_args_list,
179                                  [mk.call(genotype)])
180
181
182 class TestInitJuvenile(ut.TestCase):
183     """test InitJuvenile mathematical models"""
184
185     def setUp(self):
186         """Setup the tests"""
187
188         self.mu    = mk.MagicMock(spec=dict)
189         self.sigma = mk.MagicMock(spec=dict)
190
191         self.InitJuvenile = model.InitJuvenile(self.mu,
192                                                self.sigma)
193
```

```
194     def test___init__(self):
195         """test __init__ for class"""
196 
197         self.assertIsInstance(self.InitJuvenile, models.Model)
198         self.assertIsInstance(self.InitJuvenile, model.InitJuvenile)
199 
200         self.assertEqual(self.InitJuvenile.mu,    self.mu)
201         self.assertEqual(self.InitJuvenile.sigma, self.sigma)
202 
203         self.assertEqual(self.InitJuvenile.model_key, keyword.init_juvenile)
204 
205         self.assertTrue(dclass.is_dataclass(self.InitJuvenile))
206 
207     def test___call__(self):
208         """test call the model"""
209 
210         genotype = mk.MagicMock(spec=str)
211 
212         with mk.patch.object(stats.truncnorm, 'rvs',
213                              autospec=True) as mkRVS:
214             with mk.patch.object(model, 'float') as mkFloat:
215                 self.assertEqual(self.InitJuvenile(genotype),
216                                  mkFloat.return_value)
217                 self.assertEqual(mkFloat.call_args_list,
218                                  [mk.call(mkRVS.return_value)])
219                 self.assertEqual(mkRVS.call_args_list,
220                                  [mk.call(0,
221                                           np.inf,
222                                           loc=self.mu.
223                                           __getitem__.return_value,
224                                           scale=self.sigma.
225                                           __getitem__.return_value)])
226                 self.assertEqual(self.mu.__getitem__.call_args_list,
227                                  [mk.call(genotype)])
228                 self.assertEqual(self.sigma.__getitem__.call_args_list,
229                                  [mk.call(genotype)])
230 
231 
232 class TestInitMature(ut.TestCase):
```

```
233     """test the InitMature mathematical model"""
234
235     def setUp(self):
236         """Setup the tests"""
237
238         self.mu    = mk.MagicMock(spec=dict)
239         self.sigma = mk.MagicMock(spec=dict)
240
241         self.InitMature = model.InitMature(self.mu,
242                                            self.sigma)
243
244     def test___init__(self):
245         """test __init__ for class"""
246
247         self.assertIsInstance(self.InitMature, models.Model)
248         self.assertIsInstance(self.InitMature, model.InitMature)
249
250         self.assertEqual(self.InitMature.mu,    self.mu)
251         self.assertEqual(self.InitMature.sigma, self.sigma)
252
253         self.assertEqual(self.InitMature.model_key, keyword.init_mature)
254
255         self.assertTrue(dclass.is_dataclass(self.InitMature))
256
257     def test___call__(self):
258         """test call the model"""
259
260         genotype = mk.MagicMock(spec=str)
261
262         with mk.patch.object(stats.truncnorm, 'rvs',
263                              autospec=True) as mkRVS:
264             with mk.patch.object(model, 'float') as mkFloat:
265                 self.assertEqual(self.InitMature(genotype),
266                                  mkFloat.return_value)
267                 self.assertEqual(mkFloat.call_args_list,
268                                  [mk.call(mkRVS.return_value)])
269                 self.assertEqual(mkRVS.call_args_list,
270                                  [mk.call(0,
271                                           np.inf,
```

```
                                        loc=self.mu.
                                        __getitem__.return_value,
                                        scale=self.sigma.
                                        __getitem__.return_value)])
                self.assertEqual(self.mu.__getitem__.call_args_list,
                                 [mk.call(genotype)])
                self.assertEqual(self.sigma.__getitem__.call_args_list,
                                 [mk.call(genotype)])


class TestInitPlant(ut.TestCase):
    """test the InitPlant mathematical model"""

    def setUp(self):
        """Setup the tests"""

        self.mu    = mk.MagicMock(spec=float)
        self.sigma = mk.MagicMock(spec=float)

        self.InitPlant = model.InitPlant(self.mu,
                                         self.sigma)

    def test___init__(self):
        """test __init__ for class"""

        self.assertIsInstance(self.InitPlant, models.Model)
        self.assertIsInstance(self.InitPlant, model.InitPlant)

        self.assertEqual(self.InitPlant.mu,    self.mu)
        self.assertEqual(self.InitPlant.sigma, self.sigma)

        self.assertEqual(self.InitPlant.model_key, keyword.init_plant)

        self.assertTrue(dclass.is_dataclass(self.InitPlant))

    def test___call__(self):
        """test call the model"""

        bt = mk.MagicMock(spec=str)
```

```
311
312         with mk.patch.object(stats.truncnorm, 'rvs',
313                              autospec=True) as mkRVS:
314             with mk.patch.object(model, 'float') as mkFloat:
315                 self.assertEqual(self.InitPlant(bt),
316                                  mkFloat.return_value)
317                 self.assertEqual(mkFloat.call_args_list,
318                                  [mk.call(mkRVS.return_value)])
319                 self.assertEqual(mkRVS.call_args_list,
320                                  [mk.call(0, np.inf,
321                                           loc=self.mu,
322                                           scale=self.sigma)])
```

C.5.3 test_data

```
FallArmyworm
└── test
    └── test_data
        ├── test_counter.py
        └── test_database.py
```

C.5.3.1 test_counter.py

```
import unittest      as ut
import unittest.mock as mk

import collections as collect
import dataclasses as d_class

import pandas as pd

import source.keyword as keyword

import source.agents.agent as main_agent

import source.data.counter as counter


class CountTest(counter.Count):
    """Class to add dynamic values for agent tests"""

    data_columns = mk.create_autospec(counter.DataColumns, spec_set=True)


@d_class.dataclass
class InsectTest(main_agent.Agent):
    """calls to define a new agent"""

    death:    str
    genotype: str


class TestDataColumn(ut.TestCase):
    """test the DataColumn class"""

    def setUp(self):
        """Setup the tests"""

        self.data = [mk.MagicMock(spec=int) for _ in range(3)]

```

```
        self.attr_value = mk.MagicMock(spec=str)

        self.DataColumn = counter.DataColumn(self.data,
                                             self.attr_value)

    def test___init__(self):
        """test __init__ for class"""

        self.assertIsInstance(self.DataColumn, collect.UserList)
        self.assertIsInstance(self.DataColumn, counter.DataColumn)

        self.assertEqual(self.DataColumn.attr_value, self.attr_value)

        self.assertEqual(self.DataColumn,      self.data)
        self.assertEqual(self.DataColumn.data, self.data)

    def test_record(self):
        """test record a count"""

        count = mk.create_autospec(counter.Count, spec_set=True)

        with mk.patch.object(counter.DataColumn, 'append',
                             autospec=True) as mkAppend:
            self.DataColumn.record(count)
            self.assertEqual(mkAppend.call_args_list,
                             [mk.call(self.data,
                                      count.__getitem__.return_value)])
            self.assertEqual(count.__getitem__.call_args_list,
                             [mk.call(self.attr_value)])

        count.reset_mock()
        self.DataColumn.record(count)
        self.assertEqual(self.DataColumn[-1], count.__getitem__.return_value)
        self.assertNotEqual(self.data, self.DataColumn)

    def test_empty(self):
        """test create an empty data column"""

        self.DataColumn = counter.DataColumn.empty(self.attr_value)
```

```
        self.assertIsInstance(self.DataColumn, counter.DataColumn)
        self.assertEqual(self.DataColumn.attr_value, self.attr_value)
        self.assertEqual(self.DataColumn, [])


class TestDataColumns(ut.TestCase):
    """test DataColumns class"""

    def setUp(self):
        """Setup the tests"""

        self.data = {mk.MagicMock(spec=str):
                         mk.create_autospec(counter.DataColumn, spec_set=True)
                     for _ in range(3)}

        self.attr = mk.MagicMock(spec=str)

        self.DataColumns = counter.DataColumns(self.data,
                                               self.attr)

    def test___init__(self):
        """test __init__ for class"""

        self.assertIsInstance(self.DataColumns, collect.UserDict)
        self.assertIsInstance(self.DataColumns, counter.DataColumns)

        self.assertEqual(self.DataColumns.attr, self.attr)

        self.assertEqual(self.DataColumns,      self.data)
        self.assertEqual(self.DataColumns.data, self.data)

    def test_record(self):
        """test record all counts for attr"""

        count = mk.create_autospec(counter.Count, spec_set=True)

        self.DataColumns.record(count)
        for column in self.data.values():
            self.assertEqual(column.record.call_args_list,
```

```
                            [mk.call(count)])

    def test_refresh(self):
        """test refresh the data columns"""

        self.DataColumns.refresh()
        for column in self.data.values():
            self.assertEqual(column.clear.call_args_list,
                             [mk.call()])

    def test_columns(self):
        """test generate all the columns of data"""

        columns = self.DataColumns.columns()

        self.assertIsInstance(columns, dict)

        for key, value in self.data.items():
            column_key = '{}_{}'.format(self.attr, key)
            self.assertIn(column_key, columns)
            self.assertEqual(columns[column_key], value)
        self.assertEqual(len(columns), len(self.data))

    def test_empty(self):
        """test create an empty set of data columns"""

        attr_values = [mk.MagicMock(spec=str) for _ in range(3)]

        self.DataColumns = counter.DataColumns.empty(self.attr,
                                                     attr_values)
        self.assertIsInstance(self.DataColumns, counter.DataColumns)
        self.assertEqual(self.DataColumns.attr, self.attr)

        for key, value in self.DataColumns.items():
            self.assertIn(key, attr_values)
            self.assertIsInstance(value, counter.DataColumn)
            self.assertEqual(value.attr_value, key)
            self.assertEqual(value, [])
        for attr_value in attr_values:
```

```
            self.assertIn(attr_value, self.DataColumns)
            self.assertIsInstance(self.DataColumns[attr_value],
                                  counter.DataColumn)
            self.assertEqual(self.DataColumns[attr_value].attr_value,
                             attr_value)
            self.assertEqual(self.DataColumns[attr_value], [])


class TestBaseCount(ut.TestCase):
    """test BaseCount class"""

    def setUp(self):
        """Setup the tests"""

        self.counts = {mk.MagicMock(spec=str): mk.MagicMock(spec=int)
                       for _ in range(3)}

        self.attr = mk.MagicMock(spec=str)

        self.Count = counter.BaseCount(self.counts,
                                       self.attr)

    def test___init__(self):
        """test __init__ for class"""

        self.assertIsInstance(self.Count, collect.UserDict)
        self.assertIsInstance(self.Count, counter.BaseCount)

        self.assertEqual(self.Count.attr, self.attr)

        self.assertEqual(self.Count,      self.counts)
        self.assertEqual(self.Count.data, self.counts)

    def test_add(self):
        """test add count"""

        agent = mk.create_autospec(main_agent.Agent, spec_set=True)

        self.assertIsNone(self.Count.add(agent))
```

```
194
195     def test_sub(self):
196         """test sub count"""
197
198         agent = mk.create_autospec(main_agent.Agent, spec_set=True)
199
200         self.assertIsNone(self.Count.sub(agent))
201
202     def test_record(self):
203         """test record counts"""
204
205         self.assertIsNone(self.Count.record())
206
207     def test_refresh(self):
208         """test refresh counts"""
209
210         self.assertIsNone(self.Count.refresh())
211
212     def test_get_data_columns(self):
213         """test get the data columns"""
214
215         self.assertIsNone(self.Count.get_data_columns())
216
217
218 class TestCount(ut.TestCase):
219     """test Count class"""
220
221     def setUp(self):
222         """Setup the tests"""
223
224         self.counts = {mk.MagicMock(spec=str): mk.MagicMock(spec=int)
225                        for _ in range(3)}
226
227         self.attr    = mk.MagicMock(spec=str)
228         self.removal = mk.MagicMock(spec=bool)
229
230         self.data_columns = mk.create_autospec(counter.DataColumns,
231                                                spec_set=True)
232
```

```
233         self.Count = counter.Count(self.counts,
234                                    self.attr,
235                                    self.removal,
236                                    self.data_columns)
237 
238     def test___init__(self):
239         """test __init__ for class"""
240 
241         self.assertIsInstance(self.Count, collect.UserDict)
242         self.assertIsInstance(self.Count, counter.BaseCount)
243         self.assertIsInstance(self.Count, counter.Count)
244 
245         self.assertEqual(self.Count.attr,    self.attr)
246         self.assertEqual(self.Count.removal, self.removal)
247 
248         self.assertEqual(self.Count.data_columns, self.data_columns)
249 
250         self.assertEqual(self.Count,      self.counts)
251         self.assertEqual(self.Count.data, self.counts)
252 
253     def test_add(self):
254         """test add agent to counter"""
255 
256         # Call test
257         agent = mk.create_autospec(main_agent.Agent, spec_set=True)
258 
259         with mk.patch.object(counter, 'getattr') as mkGet:
260             # Removal is True
261             self.Count.removal = True
262             self.Count.add(agent)
263             self.assertEqual(mkGet.call_args_list, [])
264             for key, value in self.Count.items():
265                 self.assertEqual(self.counts[key], value)
266 
267             # Removal is False
268             self.Count.removal = False
269             for value in self.counts.keys():
270                 mkGet.return_value = value
271                 self.Count.add(agent)
```

```
                self.assertEqual(mkGet.call_args_list,
                                 [mk.call(agent, self.attr)])
                mkGet.reset_mock()

                self.assertNotEqual(self.Count[value], self.counts[value])
                self.assertEqual(self.Count[value],
                                 self.counts[value].__add__.return_value)
                self.assertEqual(self.counts[value].__add__.call_args_list,
                                 [mk.call(1)])
            for value in self.counts.keys():
                self.assertEqual(self.Count[value],
                                 self.counts[value].__add__.return_value)

        # Practical test
        # noinspection PyTypeChecker
        agent = main_agent.Agent('test', 'test0', None, [0], True)
        # noinspection PyTypeChecker
        count = counter.Count.empty('alive', [True, False], False)
        self.assertEqual(count[True], 0)
        count.add(agent)
        self.assertEqual(count[True], 1)
        # noinspection PyTypeChecker
        agent = main_agent.Agent('test', 'test0', None, [0], False)
        self.assertEqual(count[False], 0)
        count.add(agent)
        self.assertEqual(count[False], 1)
        # noinspection PyTypeChecker
        agent = main_agent.Agent('test', 'test0', None, [0], True)
        # noinspection PyTypeChecker
        count = counter.Count.empty('alive', [True, False], True)
        self.assertEqual(count[True], 0)
        count.add(agent)
        self.assertEqual(count[True], 0)
        # noinspection PyTypeChecker
        agent = main_agent.Agent('test', 'test0', None, [0], False)
        self.assertEqual(count[False], 0)
        count.add(agent)
        self.assertEqual(count[False], 0)

```

```
311     def test_sub(self):
312         """test sub agent to counter"""
313 
314         agent = mk.create_autospec(main_agent.Agent, spec_set=True)
315 
316         with mk.patch.object(counter, 'getattr') as mkGet:
317             # Removal is True
318             self.Count.removal = True
319             #   Test remove non-present
320             self.Count.sub(agent)
321             self.assertEqual(mkGet.call_args_list,
322                              [mk.call(agent, self.attr)])
323             mkGet.reset_mock()
324             self.assertEqual(self.Count, self.counts)
325             #   Test remove present
326             for value in self.counts.keys():
327                 mkGet.return_value = value
328                 self.Count.sub(agent)
329                 self.assertEqual(mkGet.call_args_list,
330                                  [mk.call(agent, self.attr)])
331                 mkGet.reset_mock()
332 
333                 self.assertNotEqual(self.Count[value], self.counts[value])
334                 self.assertEqual(self.Count[value],
335                                  self.counts[value].__add__.return_value)
336                 self.assertEqual(self.counts[value].__add__.call_args_list,
337                                  [mk.call(1)])
338             for value in self.counts.keys():
339                 self.assertEqual(self.Count[value],
340                                  self.counts[value].__add__.return_value)
341 
342             self.Count = counter.Count(self.counts,
343                                        self.attr,
344                                        self.removal,
345                                        self.data_columns)
346             # Removal is False
347             self.Count.removal = False
348             for value in self.counts.keys():
349                 mkGet.return_value = value
```

```
                self.Count.sub(agent)
                self.assertEqual(mkGet.call_args_list,
                                 [mk.call(agent, self.attr)])
                mkGet.reset_mock()

                self.assertNotEqual(self.Count[value], self.counts[value])
                self.assertEqual(self.Count[value],
                                 self.counts[value].__sub__.return_value)
                self.assertEqual(self.counts[value].__sub__.call_args_list,
                                 [mk.call(1)])
            for value in self.counts.keys():
                self.assertEqual(self.Count[value],
                                 self.counts[value].__sub__.return_value)

        # Practical test
        # noinspection PyTypeChecker
        agent = main_agent.Agent('test', 'test0', None, [0], True)
        # noinspection PyTypeChecker
        count = counter.Count.empty('alive', [True, False], False)
        self.assertEqual(count[True], 0)
        count.sub(agent)
        self.assertEqual(count[True], -1)
        # noinspection PyTypeChecker
        agent = main_agent.Agent('test', 'test0', None, [0], False)
        self.assertEqual(count[False], 0)
        count.sub(agent)
        self.assertEqual(count[False], -1)
        # noinspection PyTypeChecker
        agent = main_agent.Agent('test', 'test0', None, [0], True)
        # noinspection PyTypeChecker
        count = counter.Count.empty('alive', [True, False], True)
        self.assertEqual(count[True], 0)
        count.sub(agent)
        self.assertEqual(count[True], 1)
        # noinspection PyTypeChecker
        agent = main_agent.Agent('test', 'test0', None, [0], False)
        self.assertEqual(count[False], 0)
        count.sub(agent)
        self.assertEqual(count[False], 1)
```

```
389
390     def test__reset(self):
391         """test reset the count"""
392
393         self.assertEqual(self.Count, self.counts)
394         self.Count._reset()
395         self.assertNotEqual(self.Count, self.counts)
396
397         for key in self.counts:
398             self.assertIn(key, self.Count)
399             self.assertEqual(self.Count[key], 0)
400
401     def test_record(self):
402         """test record the counts"""
403
404         with mk.patch.object(counter.Count, '_reset', autospec=True) as mkReset:
405             # Removal is False
406             self.Count.removal = False
407             self.Count.record()
408             self.assertEqual(self.data_columns.record.call_args_list,
409                              [mk.call(self.Count)])
410             self.assertEqual(mkReset.call_args_list, [])
411
412             self.data_columns.reset_mock()
413             # Removal is true
414             self.Count.removal = True
415             self.Count.record()
416             self.assertEqual(self.data_columns.record.call_args_list,
417                              [mk.call(self.Count)])
418             self.assertEqual(mkReset.call_args_list,
419                              [mk.call(self.Count)])
420
421     def test_refresh(self):
422         """test refresh the stored values"""
423
424         master = mk.MagicMock()
425         master.attach_mock(self.data_columns, 'columns')
426
427         self.Count.refresh()
```

```
428         self.assertEqual(master.mock_calls,
429                          [mk.call.columns.refresh(),
430                           mk.call.columns.record(self.Count)])
431 
432     def test_get_data_columns(self):
433         """test get the data columns"""
434 
435         self.assertEqual(self.Count.get_data_columns(),
436                          self.data_columns.columns.return_value)
437         self.assertEqual(self.data_columns.columns.call_args_list,
438                          [mk.call()])
439 
440     def test_empty(self):
441         """test create empty class"""
442 
443         values = [mk.MagicMock(spec=str) for _ in range(3)]
444 
445         self.Count = counter.Count.empty(self.attr, values, self.removal)
446         self.assertIsInstance(self.Count, counter.Count)
447         self.assertEqual(self.Count.attr,    self.attr)
448         self.assertEqual(self.Count.removal, self.removal)
449         for key, value in self.Count.items():
450             self.assertIn(key, values)
451             self.assertEqual(value, 0)
452         for value in values:
453             self.assertIn(value, self.Count)
454             self.assertEqual(self.Count[value], 0)
455 
456         self.assertIsInstance(self.Count.data_columns, counter.DataColumns)
457         self.assertEqual(self.Count.data_columns.attr, self.attr)
458         for key, column in self.Count.data_columns.items():
459             self.assertIn(key, values)
460             self.assertIsInstance(column, counter.DataColumn)
461             self.assertEqual(column.attr_value, key)
462             self.assertEqual(column, [])
463         for value in values:
464             self.assertIn(value, self.Count.data_columns)
465             self.assertIsInstance(self.Count.data_columns[value],
466                                   counter.DataColumn)
```

```
            self.assertEqual(self.Count.data_columns[value].attr_value, value)
            self.assertEqual(self.Count.data_columns[value], [])


class TestCountFilter(ut.TestCase):
    """test CountFilter class"""

    def setUp(self):
        """Setup the tests"""

        self.counts = {}
        for _ in range(3):
            self.counts[mk.MagicMock(spec=str)] = \
                mk.create_autospec(CountTest, spec_set=True)
            self.counts[mk.MagicMock(spec=str)] = \
                mk.create_autospec(counter.CountFilter, spec_set=True)

        self.attr = mk.MagicMock(spec=str)

        self.Count = counter.CountFilter(self.counts,
                                         self.attr)

    def test___init__(self):
        """test __init__ for class"""

        self.assertIsInstance(self.Count, collect.UserDict)
        self.assertIsInstance(self.Count, counter.BaseCount)
        self.assertIsInstance(self.Count, counter.CountFilter)

        self.assertEqual(self.Count.attr, self.attr)

        self.assertEqual(self.Count,      self.counts)
        self.assertEqual(self.Count.data, self.counts)

    def test_add(self):
        """test add agent to counter"""

        # Call test
        agent = mk.create_autospec(main_agent.Agent, spec_set=True)
```

```
506
507         with mk.patch.object(counter, 'getattr') as mkGet:
508             for value in self.counts.keys():
509                 mkGet.return_value = value
510                 self.Count.add(agent)
511                 self.assertEqual(mkGet.call_args_list,
512                                  [mk.call(agent, self.attr)])
513                 mkGet.reset_mock()
514
515                 self.assertEqual(self.Count[value].add.call_args_list,
516                                  [mk.call(agent)])
517                 self.Count[value].reset_mock()
518             for value in self.counts.keys():
519                 self.assertEqual(self.Count[value].add.call_args_list, [])
520
521         # Practical test
522         # noinspection PyTypeChecker
523         agent_sur_homo_r = InsectTest('test', 'test0', None, None, True,
524                                       keyword.survival, keyword.homo_r)
525         # noinspection PyTypeChecker
526         agent_sur_hetero = InsectTest('test', 'test0', None, None, True,
527                                       keyword.survival, keyword.hetero)
528         # noinspection PyTypeChecker
529         agent_sur_homo_s = InsectTest('test', 'test0', None, None, True,
530                                       keyword.survival, keyword.homo_s)
531         # noinspection PyTypeChecker
532         agent_can_homo_r = InsectTest('test', 'test0', None, None, True,
533                                       keyword.cannibalism, keyword.homo_r)
534         # noinspection PyTypeChecker
535         agent_can_hetero = InsectTest('test', 'test0', None, None, True,
536                                       keyword.cannibalism, keyword.hetero)
537         # noinspection PyTypeChecker
538         agent_can_homo_s = InsectTest('test', 'test0', None, None, True,
539                                       keyword.cannibalism, keyword.homo_s)
540         attrs = {keyword.genotype: (keyword.genotype_keys, False)}
541         count = counter.CountFilter.empty('death',
542                                           keyword.death_keys,
543                                           attrs)
544         count.add(agent_sur_homo_r)
```

```
        count.add(agent_sur_hetero)
        count.add(agent_sur_homo_s)
        count.add(agent_can_homo_r)
        count.add(agent_can_hetero)
        count.add(agent_can_homo_s)
        count.record()

        count_data = count.get_data_columns()

        actual_deaths = [keyword.cannibalism, keyword.survival]
        other_deaths  = [key for key in keyword.death_keys
                         if key not in actual_deaths]

        for death_key in actual_deaths:
            for genotype_key in keyword.genotype_keys:
                key = '{}_{}_{}_{}'.format('death', death_key,
                                           'genotype', genotype_key)
                self.assertIn(key, count_data)
                self.assertEqual(count_data[key], [1])

        for death_key in other_deaths:
            for genotype_key in keyword.genotype_keys:
                key = '{}_{}_{}_{}'.format('death', death_key,
                                           'genotype', genotype_key)
                self.assertIn(key, count_data)
                self.assertEqual(count_data[key], [0])

    def test_sub(self):
        """test sub agent to counter"""

        # Call test
        agent = mk.create_autospec(main_agent.Agent, spec_set=True)

        with mk.patch.object(counter, 'getattr') as mkGet:
            for value in self.counts.keys():
                mkGet.return_value = value
                self.Count.sub(agent)
                self.assertEqual(mkGet.call_args_list,
                                 [mk.call(agent, self.attr)])
```

```
                mkGet.reset_mock()

                self.assertEqual(self.Count[value].sub.call_args_list,
                                 [mk.call(agent)])
                self.Count[value].reset_mock()
            for value in self.counts.keys():
                self.assertEqual(self.Count[value].sub.call_args_list, [])

        # Practical test
        # noinspection PyTypeChecker
        agent_sur_homo_r = InsectTest('test', 'test0', None, None, True,
                                      keyword.survival, keyword.homo_r)
        # noinspection PyTypeChecker
        agent_sur_hetero = InsectTest('test', 'test0', None, None, True,
                                      keyword.survival, keyword.hetero)
        # noinspection PyTypeChecker
        agent_sur_homo_s = InsectTest('test', 'test0', None, None, True,
                                      keyword.survival, keyword.homo_s)
        # noinspection PyTypeChecker
        agent_can_homo_r = InsectTest('test', 'test0', None, None, True,
                                      keyword.cannibalism, keyword.homo_r)
        # noinspection PyTypeChecker
        agent_can_hetero = InsectTest('test', 'test0', None, None, True,
                                      keyword.cannibalism, keyword.hetero)
        # noinspection PyTypeChecker
        agent_can_homo_s = InsectTest('test', 'test0', None, None, True,
                                      keyword.cannibalism, keyword.homo_s)
        attrs = {keyword.genotype: (keyword.genotype_keys, False)}
        count = counter.CountFilter.empty('death',
                                          keyword.death_keys,
                                          attrs)
        count.sub(agent_sur_homo_r)
        count.sub(agent_sur_hetero)
        count.sub(agent_sur_homo_s)
        count.sub(agent_can_homo_r)
        count.sub(agent_can_hetero)
        count.sub(agent_can_homo_s)
        count.record()

```

```
        count_data = count.get_data_columns()

        actual_deaths = [keyword.cannibalism, keyword.survival]
        other_deaths  = [key for key in keyword.death_keys
                         if key not in actual_deaths]

        for death_key in actual_deaths:
            for genotype_key in keyword.genotype_keys:
                key = '{}_{}_{}_{}'.format('death', death_key,
                                           'genotype', genotype_key)
                self.assertIn(key, count_data)
                self.assertEqual(count_data[key], [-1])

        for death_key in other_deaths:
            for genotype_key in keyword.genotype_keys:
                key = '{}_{}_{}_{}'.format('death', death_key,
                                           'genotype', genotype_key)
                self.assertIn(key, count_data)
                self.assertEqual(count_data[key], [0])

    def test_record(self):
        """test record data"""

        self.Count.record()
        for count in self.counts.values():
            self.assertEqual(count.record.call_args_list,
                             [mk.call()])

    def test_refresh(self):
        """test refresh data"""

        self.Count.refresh()
        for count in self.counts.values():
            self.assertEqual(count.refresh.call_args_list,
                             [mk.call()])

    def test_get_data_columns(self):
        """test get the data columns"""

```

```
        data_columns = {}
        for attr_value, count in self.counts.items():
            data_column = {mk.MagicMock(spec=str):
                           mk.create_autospec(counter.DataColumn,
                                              spec_set=True)
                           for _ in range(3)}
            count.get_data_columns.return_value = data_column

            for key, value in data_column.items():
                data_key = '{}_{}_{}'.format(self.attr, attr_value, key)
                data_columns[data_key] = value

        self.assertEqual(self.Count.get_data_columns(), data_columns)
        for count in self.Count.values():
            self.assertEqual(count.get_data_columns.call_args_list,
                             [mk.call()])

    def test_empty(self):
        """test build an empty class"""

        values = [mk.MagicMock(spec=str) for _ in range(3)]

        # Single layer of filtering
        attr  = mk.MagicMock(spec=str)
        attrs = {attr:
                     ([mk.MagicMock(spec=str) for _ in range(3)],
                      mk.MagicMock(spec=bool))}
        attr_data = attrs[attr]

        self.Count = counter.CountFilter.empty(self.attr,
                                               values,
                                               attrs)
        self.assertIsInstance(self.Count, counter.CountFilter)
        self.assertEqual(self.Count.attr, self.attr)
        for value_key, count in self.Count.items():
            self.assertIn(value_key, values)
            self.assertIsInstance(count, counter.Count)
            self.assertEqual(count.attr,    attr)
            self.assertEqual(count.removal, attr_data[1])
```

```
701
702             for key, value in count.items():
703                 self.assertIn(key, attr_data[0])
704                 self.assertEqual(value, 0)
705             for key in attr_data[0]:
706                 self.assertIn(key, count)
707                 self.assertEqual(count[key], 0)
708
709             self.assertIsInstance(count.data_columns, counter.DataColumns)
710             self.assertEqual(count.data_columns.attr, attr)
711             for key, column in count.data_columns.items():
712                 self.assertIn(key, attr_data[0])
713                 self.assertIsInstance(column, counter.DataColumn)
714                 self.assertEqual(column.attr_value, key)
715                 self.assertEqual(column, [])
716             for value in attr_data[0]:
717                 self.assertIn(value, count.data_columns)
718                 self.assertIsInstance(count.data_columns[value],
719                                       counter.DataColumn)
720                 self.assertEqual(count.data_columns[value].attr_value, value)
721                 self.assertEqual(count.data_columns[value], [])
722
723         self.assertEqual(len(self.Count), 3)
724         for value_key in values:
725             self.assertIn(value_key, self.Count)
726             count = self.Count[value_key]
727             self.assertIsInstance(count, counter.Count)
728             self.assertEqual(count.attr,    attr)
729             self.assertEqual(count.removal, attr_data[1])
730
731             for key, value in count.items():
732                 self.assertIn(key, attr_data[0])
733                 self.assertEqual(value, 0)
734             for key in attr_data[0]:
735                 self.assertIn(key, count)
736                 self.assertEqual(count[key], 0)
737
738             self.assertIsInstance(count.data_columns, counter.DataColumns)
739             self.assertEqual(count.data_columns.attr, attr)
```

```
            for key, column in count.data_columns.items():
                self.assertIn(key, attr_data[0])
                self.assertIsInstance(column, counter.DataColumn)
                self.assertEqual(column.attr_value, key)
                self.assertEqual(column, [])
            for value in attr_data[0]:
                self.assertIn(value, count.data_columns)
                self.assertIsInstance(count.data_columns[value],
                                      counter.DataColumn)
                self.assertEqual(count.data_columns[value].attr_value, value)
                self.assertEqual(count.data_columns[value], [])

        # Two layers of filtering
        attrs = {mk.MagicMock(spec=str):
                     ([mk.MagicMock(spec=str) for _ in range(3)],
                      mk.MagicMock(spec=bool))
                 for _ in range(2)}
        attr_list = list(attrs.keys())
        attr_data_list = [attrs[key] for key in attr_list]

        self.Count = counter.CountFilter.empty(self.attr,
                                               values,
                                               attrs)
        self.assertIsInstance(self.Count, counter.CountFilter)
        self.assertEqual(self.Count.attr, self.attr)
        for value_key0, count0 in self.Count.items():
            attr0      = attr_list[0]
            attr_data0 = attr_data_list[0]
            self.assertIn(value_key0, values)
            self.assertIsInstance(count0, counter.CountFilter)
            self.assertEqual(count0.attr, attr0)
            for value_key1, count1 in count0.items():
                self.assertIn(value_key1, attr_data0[0])
                self.assertIsInstance(count1, counter.Count)
                attr1      = attr_list[1]
                attr_data1 = attr_data_list[1]
                self.assertEqual(count1.attr,    attr1)
                self.assertEqual(count1.removal, attr_data1[1])

```

```
                for key, value in count1.items():
                    self.assertIn(key, attr_data1[0])
                    self.assertEqual(value, 0)
                for key in attr_data1[0]:
                    self.assertIn(key, count1)
                    self.assertEqual(count1[key], 0)

                self.assertIsInstance(count1.data_columns, counter.DataColumns)
                self.assertEqual(count1.data_columns.attr, attr1)
                for key, column in count1.data_columns.items():
                    self.assertIn(key, attr_data1[0])
                    self.assertIsInstance(column, counter.DataColumn)
                    self.assertEqual(column.attr_value, key)
                    self.assertEqual(column, [])
                for value in attr_data1[0]:
                    self.assertIn(value, count1.data_columns)
                    self.assertIsInstance(count1.data_columns[value],
                                          counter.DataColumn)
                    self.assertEqual(count1.data_columns[value].attr_value,
                                     value)
                    self.assertEqual(count1.data_columns[value], [])

        self.assertEqual(len(self.Count), 3)
        for value_key in values:
            self.assertIn(value_key, self.Count)
            count0     = self.Count[value_key]
            attr0      = attr_list[0]
            attr_data0 = attr_data_list[0]
            self.assertIsInstance(count0, counter.CountFilter)
            self.assertEqual(count0.attr, attr0)
            for value_key1, count1 in count0.items():
                self.assertIn(value_key1, attr_data0[0])
                self.assertIsInstance(count1, counter.Count)
                attr1      = attr_list[1]
                attr_data1 = attr_data_list[1]
                self.assertEqual(count1.attr,    attr1)
                self.assertEqual(count1.removal, attr_data1[1])

                for key, value in count1.items():
```

```
818                     self.assertIn(key, attr_data1[0])
819                     self.assertEqual(value, 0)
820                 for key in attr_data1[0]:
821                     self.assertIn(key, count1)
822                     self.assertEqual(count1[key], 0)
823
824                 self.assertIsInstance(count1.data_columns, counter.DataColumns)
825                 self.assertEqual(count1.data_columns.attr, attr1)
826                 for key, column in count1.data_columns.items():
827                     self.assertIn(key, attr_data1[0])
828                     self.assertIsInstance(column, counter.DataColumn)
829                     self.assertEqual(column.attr_value, key)
830                     self.assertEqual(column, [])
831                 for value in attr_data1[0]:
832                     self.assertIn(value, count1.data_columns)
833                     self.assertIsInstance(count1.data_columns[value],
834                                           counter.DataColumn)
835                     self.assertEqual(count1.data_columns[value].attr_value,
836                                      value)
837                     self.assertEqual(count1.data_columns[value], [])
838
839
840 class TestCounts(ut.TestCase):
841     """test the Counts class"""
842
843     def setUp(self):
844         """Setup the tests"""
845
846         self.counts = {}
847         for _ in range(3):
848             self.counts[mk.MagicMock(spec=str)] = \
849                 mk.create_autospec(CountTest, spec_set=True)
850             self.counts[mk.MagicMock(spec=str)] = \
851                 mk.create_autospec(counter.CountFilter, spec_set=True)
852
853         self.Counts = counter.Counts(self.counts)
854
855     def test___init__(self):
856         """test __init__ for class"""
```

```
        self.assertIsInstance(self.Counts, collect.UserDict)
        self.assertIsInstance(self.Counts, counter.Counts)

        self.assertEqual(self.Counts,      self.counts)
        self.assertEqual(self.Counts.data, self.counts)

    def test_add(self):
        """test add to counts of all things"""

        agent = mk.create_autospec(main_agent.Agent, spec_set=True)

        self.Counts.add(agent)
        for count in self.counts.values():
            self.assertEqual(count.add.call_args_list,
                             [mk.call(agent)])

    def test_sub(self):
        """test sub from counts of all things"""

        agent = mk.create_autospec(main_agent.Agent, spec_set=True)

        self.Counts.sub(agent)
        for count in self.counts.values():
            self.assertEqual(count.sub.call_args_list,
                             [mk.call(agent)])

    def test_record(self):
        """test record the counts"""

        self.Counts.record()
        for count in self.counts.values():
            self.assertEqual(count.record.call_args_list,
                             [mk.call()])

    def test_refresh(self):
        """test refresh the stored data"""

        self.Counts.refresh()
```

```
896         for count in self.counts.values():
897             self.assertEqual(count.refresh.call_args_list,
898                              [mk.call()])
899 
900     def test_columns(self):
901         """test create all the columns of data"""
902 
903         columns = {}
904         for count in self.counts.values():
905             column = {}
906             for _ in range(3):
907                 key   = mk.MagicMock(spec=str)
908                 value = mk.create_autospec(counter.DataColumn, spec_set=True)
909 
910                 column[ key] = value
911                 columns[key] = value
912 
913             count.get_data_columns.return_value = column
914 
915         self.assertEqual(self.Counts.columns(), columns)
916         for count in self.Counts.values():
917             self.assertEqual(count.get_data_columns.call_args_list,
918                              [mk.call()])
919 
920     def test_dataframe(self):
921         """test create a dataframe of the data"""
922 
923         with mk.patch.object(counter.Counts, 'columns',
924                              autospec=True) as mkColumns:
925             with mk.patch.object(pd.DataFrame, 'from_dict',
926                                  autospec=True) as mkDataFrame:
927                 self.assertEqual(self.Counts.dataframe(),
928                                  mkDataFrame.return_value)
929                 self.assertEqual(mkDataFrame.call_args_list,
930                                  [mk.call(mkColumns.return_value)])
931                 self.assertEqual(mkColumns.call_args_list,
932                                  [mk.call(self.Counts)])
933 
934     def test_count(self):
```

```
        """test add a count to system"""

        attr_key = mk.MagicMock(spec=str)
        attr     = mk.MagicMock(spec=str)
        values   = [mk.MagicMock(spec=str) for _ in range(3)]

        # Add standard
        removal = mk.MagicMock(spec=bool)
        self.assertNotIn(attr_key, self.Counts)

        with mk.patch.object(counter.Count, 'empty', autospec=True) as mkEmpty:
            self.Counts.count(attr_key, attr, values, removal)
            self.assertIn(attr_key, self.Counts)
            self.assertEqual(self.Counts[attr_key],
                             mkEmpty.return_value)
            self.assertEqual(mkEmpty.call_args_list,
                             [mk.call(attr, values, removal)])

        del self.Counts[attr_key]

        # Add default
        self.assertNotIn(attr_key, self.Counts)

        with mk.patch.object(counter.Count, 'empty', autospec=True) as mkEmpty:
            self.Counts.count(attr_key, attr, values, {})
            self.assertIn(attr_key, self.Counts)
            self.assertEqual(self.Counts[attr_key],
                             mkEmpty.return_value)
            self.assertEqual(mkEmpty.call_args_list,
                             [mk.call(attr, values, False)])

        del self.Counts[attr_key]

        # Add filter stack
        attrs = {mk.MagicMock(spec=str): mk.MagicMock(spec=tuple)}
        self.assertNotIn(attr_key, self.Counts)

        with mk.patch.object(counter.CountFilter, 'empty',
                             autospec=True) as mkEmpty:
```

```
            self.Counts.count(attr_key, attr, values, attrs)
            self.assertIn(attr_key, self.Counts)
            self.assertEqual(self.Counts[attr_key],
                             mkEmpty.return_value)
            self.assertEqual(mkEmpty.call_args_list,
                             [mk.call(attr, values, attrs)])


    def test_empty(self):
        """test create an empty counts class"""

        # Test calls
        attrs = {}
        for _ in range(3):
            attr   = mk.MagicMock(spec=str)
            values = [mk.MagicMock(spec=str) for _ in range(3)]
            removal = mk.MagicMock(spec=bool)
            attrs[mk.MagicMock(spec=str)] = (attr, values, removal)

        attr_keys = list(attrs.keys())

        with mk.patch.object(counter.Counts, 'count', autospec=True) as mkCount:
            self.Counts = counter.Counts.empty(attrs)
            self.assertIsInstance(self.Counts, counter.Counts)
            self.assertEqual(self.Counts, {})

            for index, this in enumerate(attrs.items()):
                attr_key, attr_filter = this
                self.assertEqual(mkCount.call_args_list[index],
                                 mk.call(self.Counts, attr_key, *attr_filter))
            for index, call in enumerate(mkCount.call_args_list):
                self.assertEqual(call,
                                 mk.call(self.Counts, attr_keys[index],
                                         *attrs[attr_keys[index]]))
            self.assertEqual(len(mkCount.call_args_list), 3)


        # test practical
        attrs = {keyword.genotype:    (keyword.genotype,
```

```
                                    keyword.genotype_keys,
                                    False),
                keyword.death_track: (keyword.death,
                                    keyword.death_keys,
                                    {keyword.genotype:
                                         (keyword.genotype_keys,
                                          True)})}

        self.Counts = counter.Counts.empty(attrs)
        self.assertIsInstance(self.Counts, counter.Counts)

        # check the genotype
        self.assertIn(keyword.genotype, self.Counts)
        count = self.Counts[keyword.genotype]
        self.assertIsInstance(count, counter.Count)
        self.assertEqual(count.attr,    keyword.genotype)
        self.assertEqual(count.removal, False)
        for key, value in count.items():
            self.assertIn(key, keyword.genotype_keys)
            self.assertEqual(value, 0)
        for key in keyword.genotype_keys:
            self.assertIn(key, count)
            self.assertEqual(count[key], 0)
        self.assertEqual(len(count), 3)
        datacolumns = count.data_columns
        self.assertIsInstance(datacolumns, counter.DataColumns)
        self.assertEqual(datacolumns.attr, keyword.genotype)
        for key, column in datacolumns.items():
            self.assertIn(key, keyword.genotype_keys)
            self.assertIsInstance(column, counter.DataColumn)
            self.assertEqual(column.attr_value, key)
            self.assertEqual(column, [])
        for key in keyword.genotype_keys:
            self.assertIn(key, datacolumns)
            column = datacolumns[key]
            self.assertIsInstance(column, counter.DataColumn)
            self.assertEqual(column.attr_value, key)
            self.assertEqual(column, [])
        self.assertEqual(len(datacolumns), 3)
```

```
1052
1053        # check the death
1054        self.assertIn(keyword.death_track, self.Counts)
1055        death_count = self.Counts[keyword.death_track]
1056        self.assertIsInstance(death_count, counter.CountFilter)
1057        self.assertEqual(death_count.attr, keyword.death)
1058        for death_key, count in death_count.items():
1059            self.assertIn(death_key, keyword.death_keys)
1060            self.assertIsInstance(count, counter.Count)
1061            self.assertEqual(count.attr,    keyword.genotype)
1062            self.assertEqual(count.removal, True)
1063            for key, value in count.items():
1064                self.assertIn(key, keyword.genotype_keys)
1065                self.assertEqual(value, 0)
1066            for key in keyword.genotype_keys:
1067                self.assertIn(key, count)
1068                self.assertEqual(count[key], 0)
1069            self.assertEqual(len(count), 3)
1070            datacolumns = count.data_columns
1071            self.assertIsInstance(datacolumns, counter.DataColumns)
1072            self.assertEqual(datacolumns.attr, keyword.genotype)
1073            for key, column in datacolumns.items():
1074                self.assertIn(key, keyword.genotype_keys)
1075                self.assertIsInstance(column, counter.DataColumn)
1076                self.assertEqual(column.attr_value, key)
1077                self.assertEqual(column, [])
1078            for key in keyword.genotype_keys:
1079                self.assertIn(key, datacolumns)
1080                column = datacolumns[key]
1081                self.assertIsInstance(column, counter.DataColumn)
1082                self.assertEqual(column.attr_value, key)
1083                self.assertEqual(column, [])
1084            self.assertEqual(len(datacolumns), 3)
1085        for death_key in keyword.death_keys:
1086            self.assertIn(death_key, death_count)
1087            count = death_count[death_key]
1088            self.assertIsInstance(count, counter.Count)
1089            self.assertEqual(count.attr,    keyword.genotype)
1090            self.assertEqual(count.removal, True)
```

```
            for key, value in count.items():
                self.assertIn(key, keyword.genotype_keys)
                self.assertEqual(value, 0)
            for key in keyword.genotype_keys:
                self.assertIn(key, count)
                self.assertEqual(count[key], 0)
            self.assertEqual(len(count), 3)
            datacolumns = count.data_columns
            self.assertIsInstance(datacolumns, counter.DataColumns)
            self.assertEqual(datacolumns.attr, keyword.genotype)
            for key, column in datacolumns.items():
                self.assertIn(key, keyword.genotype_keys)
                self.assertIsInstance(column, counter.DataColumn)
                self.assertEqual(column.attr_value, key)
                self.assertEqual(column, [])
            for key in keyword.genotype_keys:
                self.assertIn(key, datacolumns)
                column = datacolumns[key]
                self.assertIsInstance(column, counter.DataColumn)
                self.assertEqual(column.attr_value, key)
                self.assertEqual(column, [])
            self.assertEqual(len(datacolumns), 3)
        self.assertEqual(len(death_count), 5)

        # test the data columns
        columns = self.Counts.columns()
        #   genotype columns
        for genotype_key in keyword.genotype_keys:
            geno_key = '{}_{}'.format(keyword.genotype,
                                      genotype_key)
            self.assertIn(geno_key, columns)
            self.assertEqual(columns[geno_key], [])
            for death_key in keyword.death_keys:
                key = '{}_{}_{}'.format(keyword.death,
                                        death_key,
                                        geno_key)
                self.assertIn(key, columns)
                self.assertEqual(columns[key], [])
```

C.5.3.2 test_database.py

```
import unittest      as ut
import unittest.mock as mk

import dataclasses as dclass
import sqlalchemy  as sql
import pandas      as pd

import source.data.database as database

import source.simulation.simulation as main_simulation

import source.space.agents as main_agents


class SimulationTest(main_simulation.Simulation):
    """Class to add dynamic values for tests"""

    agents   = mk.create_autospec(main_agents.Agents, spec_set=True)
    timestep = mk.MagicMock(spec=int)


class TestDatabase(ut.TestCase):
    """test the Database system class"""

    def setUp(self):
        """Setup the tests"""

        self.spacing   = mk.MagicMock(spec=int)
        self.file_path = mk.MagicMock(spec=str)
        self.file_name = mk.MagicMock(spec=str)
        self.prev_dump = mk.MagicMock(spec=int)

        self.Database = database.Database(self.spacing,
                                          self.file_name,
                                          self.file_path,
                                          self.prev_dump)

```

```
    def test___init__(self):
        """test __init__ for class"""

        self.assertIsInstance(self.Database, database.Database)

        self.assertEqual(self.Database.spacing,   self.spacing)
        self.assertEqual(self.Database.file_path, self.file_path)
        self.assertEqual(self.Database.file_name, self.file_name)
        self.assertEqual(self.Database.prev_dump, self.prev_dump)

        self.assertEqual(self.Database.next_dump,
                         self.prev_dump.__add__.return_value)
        self.assertEqual(self.prev_dump.__add__.call_args_list,
                         [mk.call(self.spacing)])

        self.assertTrue(dclass.is_dataclass(self.Database))

        # Test default
        self.Database = database.Database(self.spacing)

        self.assertIsInstance(self.Database, database.Database)

        self.assertEqual(self.Database.spacing,   self.spacing)
        self.assertEqual(self.Database.file_path, '')
        self.assertEqual(self.Database.file_name, ':memory:')
        self.assertEqual(self.Database.prev_dump, 0)

        self.assertEqual(self.Database.next_dump,
                         self.spacing.__radd__.return_value)
        self.assertEqual(self.spacing.__radd__.call_args_list,
                         [mk.call(0)])

        self.assertTrue(dclass.is_dataclass(self.Database))

    def test_sql_file_name(self):
        """test get the file_name for sql engine"""

        simulation = mk.create_autospec(SimulationTest, spec_set=True)

```

```
        dialect = 'sqlite:///'
        time    = str(self.file_path)      + '/'    + \
                  str(self.prev_dump)      + '_to_' + \
                  str(simulation.timestep) + '_'

        self.assertEqual(self.Database.sql_file_name(simulation),
                         dialect + time + str(self.file_name))

    def test__save(self):
        """test save data to file"""

        simulation = mk.create_autospec(SimulationTest, spec_set=True)
        simulation.agents = mk.create_autospec(main_agents.Agents,
                                               spec_set=True)
        dataframes = {mk.MagicMock(spec=str): mk.create_autospec(pd.DataFrame,
                                                                 spec_set=True)
                      for _ in range(3)}
        simulation.agents.dataframes.return_value = dataframes

        with mk.patch.object(database.Database, 'sql_file_name',
                             autospec=True) as mkName:
            with mk.patch.object(sql, 'create_engine') as mkSQL:
                self.Database._save(simulation)
                for table_name, dataframe in dataframes.items():
                    self.assertEqual(dataframe.to_sql.call_args_list,
                                     [mk.call(table_name, mkSQL.return_value)])
                self.assertEqual(mkSQL.call_args_list,
                                 [mk.call(mkName.return_value)])
                self.assertEqual(mkName.call_args_list,
                                 [mk.call(self.Database, simulation)])
                self.assertEqual(simulation.agents.dataframes.call_args_list,
                                 [mk.call()])

    def test_dump(self):
        """test dump the data"""

        simulation = mk.create_autospec(SimulationTest, spec_set=True)
        simulation.agents = mk.create_autospec(main_agents.Agents,
                                               spec_set=True)
```

```
116
117         with mk.patch.object(database.Database, '_save') as mkSave:
118             master = mk.MagicMock()
119             master.attach_mock(mkSave,            'save')
120             master.attach_mock(simulation.agents, 'agents')
121
122             self.Database.dump(simulation)
123             self.assertEqual(master.mock_calls,
124                              [mk.call.save(simulation),
125                               mk.call.agents.refresh()])
126
127             self.assertEqual(self.Database.prev_dump, simulation.timestep)
128             self.assertEqual(self.Database.next_dump,
129                              simulation.timestep.__add__.return_value)
130             self.assertEqual(simulation.timestep.__add__.call_args_list,
131                              [mk.call(self.spacing)])
132
133     def test_save(self):
134         """test save the simulation"""
135
136         simulation = mk.create_autospec(SimulationTest, spec_set=True)
137
138         with mk.patch.object(database.Database, 'dump',
139                              autospec=True) as mkDump:
140             simulation.timestep.__eq__.side_effect = [False, True]
141
142             # Test no save
143             self.Database.save(simulation)
144             self.assertEqual(mkDump.call_args_list, [])
145             self.assertEqual(simulation.timestep.__eq__.call_args_list,
146                              [mk.call(self.Database.next_dump)])
147
148             simulation.timestep.__eq__.reset_mock()
149             # Test save
150             self.Database.save(simulation)
151             self.assertEqual(mkDump.call_args_list,
152                              [mk.call(self.Database, simulation)])
153             self.assertEqual(simulation.timestep.__eq__.call_args_list,
154                              [mk.call(self.Database.next_dump)])
```

```
155
156     def test_setup(self):
157         """test save the class"""
158
159         # Just spacing
160         data_tuple = (self.spacing,)
161         self.Database = database.Database.setup(data_tuple)
162         self.assertIsInstance(self.Database, database.Database)
163         self.assertEqual(self.Database.spacing,   self.spacing)
164         self.assertEqual(self.Database.file_path, '')
165         self.assertEqual(self.Database.file_name, ':memory:')
166         self.assertEqual(self.Database.prev_dump, 0)
167         self.assertEqual(self.Database.next_dump,
168                          self.spacing.__radd__.return_value)
169         self.assertEqual(self.spacing.__radd__.call_args_list,
170                          [mk.call(0)])
171
172         self.spacing.reset_mock()
173         # Spacing and file name
174         data_tuple = (self.spacing, self.file_name, self.file_path)
175         self.Database = database.Database.setup(data_tuple)
176         self.assertIsInstance(self.Database, database.Database)
177         self.assertEqual(self.Database.spacing,   self.spacing)
178         self.assertEqual(self.Database.file_path, self.file_path)
179         self.assertEqual(self.Database.file_name, self.file_name)
180         self.assertEqual(self.Database.prev_dump, 0)
181         self.assertEqual(self.Database.next_dump,
182                          self.spacing.__radd__.return_value)
183         self.assertEqual(self.spacing.__radd__.call_args_list,
184                          [mk.call(0)])
```

C.5.4 test_development

```
FallArmyworm
└── test
    └── test_development
        ├── test_egg.py
        ├── test_larva.py
        ├── test_models.py
        └── test_pupa.py
```

C.5.4.1 test_egg.py

```
import unittest      as ut
import unittest.mock as mk

import dataclasses as dclass

import source.keyword as keyword

import source.agents.egg   as agent_egg
import source.agents.larva as agent_larva

import source.development.egg as development


class EggTest(agent_egg.Egg):
    """Class to add dynamic values for tests"""

    mass     = mk.MagicMock(spec=float)
    genotype = mk.MagicMock(spec=str)
    age      = mk.MagicMock(spec=int)


class TestEgg(ut.TestCase):
    """test the Egg development class"""

    def setUp(self):
        """Setup the tests"""

        self.development = mk.MagicMock(spec=callable)

        self.Egg = development.Egg(self.development)

    def test___init__(self):
        """test __init__ for class"""

        self.assertIsInstance(self.Egg, development.Egg)

        self.assertEqual(self.Egg.development, self.development)
```

```
38
39          self.assertTrue(dclass.is_dataclass(self.Egg))
40
41      def test__use_development(self):
42          """test if we use the development system"""
43
44          self.assertTrue(self.Egg._use_development)
45
46          self.Egg.development = None
47          self.assertFalse(self.Egg._use_development)
48
49      def test__develop(self):
50          """test determine if egg develops"""
51
52          egg = mk.create_autospec(EggTest, spec_set=True)
53
54          with mk.patch.object(development.Egg, '_use_development',
55                               autospec=True) as mkUse:
56              mkUse.__get__ = mk.MagicMock(side_effect=[False, True])
57
58              # Test when we don't have a model
59              self.assertFalse(self.Egg._develop(egg))
60              self.assertEqual(self.development.call_args_list, [])
61
62              # Test when have a model
63              self.assertEqual(self.Egg._develop(egg),
64                               self.development.return_value)
65              self.assertEqual(self.development.call_args_list,
66                               [mk.call(egg.mass, egg.age, egg.genotype)])
67
68      def test__make_larva(self):
69          """test make a larva"""
70
71          egg   = mk.create_autospec(EggTest, spec_set=True)
72          larva = mk.create_autospec(agent_larva.Larva, spec_set=True)
73
74          with mk.patch.object(agent_larva.Larva, 'advance',
75                               autospec=True) as mkAdvance:
76              mkAdvance.return_value = larva
```

```
77
78              self.Egg._make_larva(egg)
79              self.assertEqual(larva.activate.call_args_list,
80                               [mk.call()])
81              self.assertEqual(mkAdvance.call_args_list,
82                               [mk.call(egg)])
83
84      def test_develop(self):
85          """test run development system"""
86
87          egg = mk.create_autospec(EggTest, spec_set=True)
88
89          with mk.patch.object(development.Egg, '_develop',
90                               autospec=True) as mkDevelop:
91              with mk.patch.object(development.Egg, '_make_larva',
92                                   autospec=True) as mkMake:
93                  mkDevelop.side_effect = [False, True]
94                  master = mk.MagicMock()
95                  master.attach_mock(egg,    'egg')
96                  master.attach_mock(mkMake, 'make')
97
98                  # Does not develop
99                  self.Egg.develop(egg)
100                 self.assertEqual(mkDevelop.call_args_list,
101                                  [mk.call(self.Egg, egg)])
102                 self.assertEqual(master.mock_calls, [])
103
104                 mkDevelop.reset_mock()
105                 # Does develop
106                 self.Egg.develop(egg)
107                 self.assertEqual(mkDevelop.call_args_list,
108                                  [mk.call(self.Egg, egg)])
109                 self.assertEqual(master.mock_calls,
110                                  [mk.call.egg.deactivate(),
111                                   mk.call.make(egg)])
112
113     def test_setup(self):
114         """test setup the class"""
115
```

```
        # Test if have the model
        kwargs = {keyword.egg_development: self.development}
        self.Egg = development.Egg.setup(**kwargs)
        self.assertIsInstance(self.Egg, development.Egg)
        self.assertEqual(self.Egg.development, self.development)

        # Test if have the model
        kwargs = {}
        self.Egg = development.Egg.setup(**kwargs)
        self.assertIsInstance(self.Egg, development.Egg)
        self.assertEqual(self.Egg.development, None)
```

C.5.4.2 test_larva.py

```
import unittest      as ut
import unittest.mock as mk

import dataclasses as dclass

import source.keyword as keyword

import source.agents.larva   as agent_larva
import source.agents.pupa as agent_pupa

import source.development.larva as development


class LarvaTest(agent_larva.Larva):
    """Class to add dynamic values for tests"""

    mass     = mk.MagicMock(spec=float)
    genotype = mk.MagicMock(spec=str)
    age      = mk.MagicMock(spec=int)


class TestLarva(ut.TestCase):
    """test the Larva development class"""

    def setUp(self):
        """Setup the tests"""

        self.development = mk.MagicMock(spec=callable)

        self.Larva = development.Larva(self.development)

    def test___init__(self):
        """test __init__ for class"""

        self.assertIsInstance(self.Larva, development.Larva)

        self.assertEqual(self.Larva.development, self.development)
```

```
38
39          self.assertTrue(dclass.is_dataclass(self.Larva))
40
41      def test__use_development(self):
42          """test if we use the development system"""
43
44          self.assertTrue(self.Larva._use_development)
45
46          self.Larva.development = None
47          self.assertFalse(self.Larva._use_development)
48
49      def test__develop(self):
50          """test determine if larva develops"""
51
52          larva = mk.create_autospec(LarvaTest, spec_set=True)
53
54          with mk.patch.object(development.Larva, '_use_development',
55                               autospec=True) as mkUse:
56              mkUse.__get__ = mk.MagicMock(side_effect=[False, True])
57
58              # Test when we don't have a model
59              self.assertFalse(self.Larva._develop(larva))
60              self.assertEqual(self.development.call_args_list, [])
61
62              # Test when have a model
63              self.assertEqual(self.Larva._develop(larva),
64                               self.development.return_value)
65              self.assertEqual(self.development.call_args_list,
66                               [mk.call(larva.mass, larva.age, larva.genotype)])
67
68      def test__make_pupa(self):
69          """test make a pupa"""
70
71          larva   = mk.create_autospec(LarvaTest, spec_set=True)
72          pupa = mk.create_autospec(agent_pupa.Pupa, spec_set=True)
73
74          with mk.patch.object(agent_pupa.Pupa, 'advance',
75                               autospec=True) as mkAdvance:
76              mkAdvance.return_value = pupa
```

```
77
78              self.Larva._make_pupa(larva)
79              self.assertEqual(pupa.activate.call_args_list,
80                               [mk.call()])
81              self.assertEqual(mkAdvance.call_args_list,
82                               [mk.call(larva)])
83
84      def test_develop(self):
85          """test run development system"""
86
87          larva = mk.create_autospec(LarvaTest, spec_set=True)
88
89          with mk.patch.object(development.Larva, '_develop',
90                               autospec=True) as mkDevelop:
91              with mk.patch.object(development.Larva, '_make_pupa',
92                                   autospec=True) as mkMake:
93                  mkDevelop.side_effect = [False, True]
94                  master = mk.MagicMock()
95                  master.attach_mock(larva,    'larva')
96                  master.attach_mock(mkMake, 'make')
97
98                  # Does not develop
99                  self.Larva.develop(larva)
100                 self.assertEqual(mkDevelop.call_args_list,
101                                  [mk.call(self.Larva, larva)])
102                 self.assertEqual(master.mock_calls, [])
103
104                 mkDevelop.reset_mock()
105                 # Does develop
106                 self.Larva.develop(larva)
107                 self.assertEqual(mkDevelop.call_args_list,
108                                  [mk.call(self.Larva, larva)])
109                 self.assertEqual(master.mock_calls,
110                                  [mk.call.larva.deactivate(),
111                                   mk.call.make(larva)])
112
113     def test_setup(self):
114         """test setup the class"""
115
```

```
        # Test if have the model
        kwargs = {keyword.larva_development: self.development}
        self.Larva = development.Larva.setup(**kwargs)
        self.assertIsInstance(self.Larva, development.Larva)
        self.assertEqual(self.Larva.development, self.development)

        # Test if have the model
        kwargs = {}
        self.Larva = development.Larva.setup(**kwargs)
        self.assertIsInstance(self.Larva, development.Larva)
        self.assertEqual(self.Larva.development, None)
```

C.5.4.3 test_models.py

```
import unittest      as ut
import unittest.mock as mk

import dataclasses  as dclass
import numpy.random as rnd
import scipy.stats  as stats

import source.keyword as keyword

import source.development.models as model

import source.simulation.models as models


class TestBaseTime(ut.TestCase):
    """test the BaseTime development mathematical model"""

    def setUp(self):
        """Setup the tests"""

        self.mu    = mk.MagicMock(spec=float)
        self.sigma = mk.MagicMock(spec=float)

        self.BaseTime = model.BaseTime(self.mu,
                                       self.sigma)

    def test___init__(self):
        """test __init__ for class"""

        self.assertIsInstance(self.BaseTime, models.Model)
        self.assertIsInstance(self.BaseTime, model.BaseTime)

        self.assertEqual(self.BaseTime.mu,    self.mu)
        self.assertEqual(self.BaseTime.sigma, self.sigma)

        self.assertEqual(self.BaseTime.model_key, None)

```

```
38          self.assertTrue(dclass.is_dataclass(self.BaseTime))
39
40      def test___call__(self):
41          """test call the model"""
42
43          mass     = mk.MagicMock(spec=float)
44          age      = mk.MagicMock(spec=int)
45          genotype = mk.MagicMock(spec=str)
46
47          with mk.patch.object(stats.norm, 'cdf', autospec=True) as mkCDF:
48              with mk.patch.object(rnd, 'random') as mkRND:
49                  mkRND.return_value.__le__.side_effect = [True, False]
50
51                  # Return True
52                  self.assertTrue(self.BaseTime(mass, age, genotype))
53                  self.assertEqual(mkRND.return_value.__le__.call_args_list,
54                                   [mk.call(mkCDF.return_value)])
55                  self.assertEqual(mkRND.call_args_list,
56                                   [mk.call()])
57                  self.assertEqual(mkCDF.call_args_list,
58                                   [mk.call(age,
59                                            loc=self.mu, scale=self.sigma)])
60
61                  mkRND.reset_mock()
62                  mkCDF.reset_mock()
63                  # Return False
64                  self.assertFalse(self.BaseTime(mass, age, genotype))
65                  self.assertEqual(mkRND.return_value.__le__.call_args_list,
66                                   [mk.call(mkCDF.return_value)])
67                  self.assertEqual(mkRND.call_args_list,
68                                   [mk.call()])
69                  self.assertEqual(mkCDF.call_args_list,
70                                   [mk.call(age,
71                                            loc=self.mu, scale=self.sigma)])
72
73
74  class TestEgg(ut.TestCase):
75      """test the Egg development mathematical model"""
76
```

```
77      def setUp(self):
78          """Setup the tests"""
79
80          self.mu    = mk.MagicMock(spec=float)
81          self.sigma = mk.MagicMock(spec=float)
82
83          self.Egg = model.Egg(self.mu,
84                               self.sigma)
85
86      def test___init__(self):
87          """test __init__ for class"""
88
89          self.assertIsInstance(self.Egg, models.Model)
90          self.assertIsInstance(self.Egg, model.BaseTime)
91          self.assertIsInstance(self.Egg, model.Egg)
92
93          self.assertEqual(self.Egg.mu,    self.mu)
94          self.assertEqual(self.Egg.sigma, self.sigma)
95
96          self.assertEqual(self.Egg.model_key, keyword.egg_development)
97
98          self.assertTrue(dclass.is_dataclass(self.Egg))
99
100
101 class TestPupa(ut.TestCase):
102     """test the Pupa development mathematical model"""
103
104     def setUp(self):
105         """Setup the tests"""
106
107         self.mu    = mk.MagicMock(spec=float)
108         self.sigma = mk.MagicMock(spec=float)
109
110         self.Pupa = model.Pupa(self.mu,
111                                self.sigma)
112
113     def test___init__(self):
114         """test __init__ for class"""
115
```

```
116         self.assertIsInstance(self.Pupa, models.Model)
117         self.assertIsInstance(self.Pupa, model.BaseTime)
118         self.assertIsInstance(self.Pupa, model.Pupa)
119 
120         self.assertEqual(self.Pupa.mu,    self.mu)
121         self.assertEqual(self.Pupa.sigma, self.sigma)
122 
123         self.assertEqual(self.Pupa.model_key, keyword.pupa_development)
124 
125         self.assertTrue(dclass.is_dataclass(self.Pupa))
126 
127 
128 class TestLarva(ut.TestCase):
129     """test the Larva development mathematical model"""
130 
131     def setUp(self):
132         """Setup the tests"""
133 
134         self.mu    = mk.MagicMock(spec=dict)
135         self.sigma = mk.MagicMock(spec=dict)
136 
137         self.Larva = model.Larva(self.mu,
138                                  self.sigma)
139 
140     def test___init__(self):
141         """test __init__ for class"""
142 
143         self.assertIsInstance(self.Larva, models.Model)
144         self.assertIsInstance(self.Larva, model.Larva)
145 
146         self.assertEqual(self.Larva.mu,    self.mu)
147         self.assertEqual(self.Larva.sigma, self.sigma)
148 
149         self.assertEqual(self.Larva.model_key, keyword.larva_development)
150 
151         self.assertTrue(dclass.is_dataclass(self.Larva))
152 
153     def test___call__(self):
154         """test call the model"""
```

```
155
156         mass     = mk.MagicMock(spec=float)
157         age      = mk.MagicMock(spec=int)
158         genotype = mk.MagicMock(spec=str)
159
160         with mk.patch.object(stats.norm, 'cdf', autospec=True) as mkCDF:
161             with mk.patch.object(rnd, 'random') as mkRND:
162                 mkRND.return_value.__le__.side_effect = [True, False]
163
164                 # Return True
165                 self.assertTrue(self.Larva(mass, age, genotype))
166                 self.assertEqual(mkRND.return_value.__le__.call_args_list,
167                                  [mk.call(mkCDF.return_value)])
168                 self.assertEqual(mkRND.call_args_list,
169                                  [mk.call()])
170                 self.assertEqual(mkCDF.call_args_list,
171                                  [mk.call(mass,
172                                           loc=self.mu.
173                                             __getitem__.return_value,
174                                           scale=self.sigma.
175                                             __getitem__.return_value)])
176                 self.assertEqual(self.mu.__getitem__.call_args_list,
177                                  [mk.call(genotype)])
178                 self.assertEqual(self.sigma.__getitem__.call_args_list,
179                                  [mk.call(genotype)])
180
181                 mkRND.reset_mock()
182                 mkCDF.reset_mock()
183                 self.mu.reset_mock()
184                 self.sigma.reset_mock()
185                 # Return False
186                 self.assertFalse(self.Larva(mass, age, genotype))
187                 self.assertEqual(mkRND.return_value.__le__.call_args_list,
188                                  [mk.call(mkCDF.return_value)])
189                 self.assertEqual(mkRND.call_args_list,
190                                  [mk.call()])
191                 self.assertEqual(mkCDF.call_args_list,
192                                  [mk.call(mass,
193                                           loc=self.mu.
```

```
                                        __getitem__.return_value,
                                        scale=self.sigma.
                                        __getitem__.return_value)])
            self.assertEqual(self.mu.__getitem__.call_args_list,
                             [mk.call(genotype)])
            self.assertEqual(self.sigma.__getitem__.call_args_list,
                             [mk.call(genotype)])
```

C.5.4.4 test_pupa.py

```
import unittest      as ut
import unittest.mock as mk

import dataclasses as dclass

import source.keyword as keyword

import source.agents.pupa   as agent_pupa
import source.agents.adult as agent_adult

import source.development.pupa as development


class PupaTest(agent_pupa.Pupa):
    """Class to add dynamic values for tests"""

    mass     = mk.MagicMock(spec=float)
    genotype = mk.MagicMock(spec=str)
    age      = mk.MagicMock(spec=int)


class TestPupa(ut.TestCase):
    """test the Pupa development class"""

    def setUp(self):
        """Setup the tests"""

        self.development = mk.MagicMock(spec=callable)

        self.Pupa = development.Pupa(self.development)

    def test___init__(self):
        """test __init__ for class"""

        self.assertIsInstance(self.Pupa, development.Pupa)

        self.assertEqual(self.Pupa.development, self.development)
```

```
38
39          self.assertTrue(dclass.is_dataclass(self.Pupa))
40
41      def test__use_development(self):
42          """test if we use the development system"""
43
44          self.assertTrue(self.Pupa._use_development)
45
46          self.Pupa.development = None
47          self.assertFalse(self.Pupa._use_development)
48
49      def test__develop(self):
50          """test determine if pupa develops"""
51
52          pupa = mk.create_autospec(PupaTest, spec_set=True)
53
54          with mk.patch.object(development.Pupa, '_use_development',
55                               autospec=True) as mkUse:
56              mkUse.__get__ = mk.MagicMock(side_effect=[False, True])
57
58              # Test when we don't have a model
59              self.assertFalse(self.Pupa._develop(pupa))
60              self.assertEqual(self.development.call_args_list, [])
61
62              # Test when have a model
63              self.assertEqual(self.Pupa._develop(pupa),
64                               self.development.return_value)
65              self.assertEqual(self.development.call_args_list,
66                               [mk.call(pupa.mass, pupa.age, pupa.genotype)])
67
68      def test__make_adult(self):
69          """test make a adult"""
70
71          pupa   = mk.create_autospec(PupaTest, spec_set=True)
72          adult = mk.create_autospec(agent_adult.Adult, spec_set=True)
73
74          with mk.patch.object(agent_adult.Adult, 'advance',
75                               autospec=True) as mkAdvance:
76              mkAdvance.return_value = adult
```

```
77
78              self.Pupa._make_adult(pupa)
79              self.assertEqual(adult.activate.call_args_list,
80                               [mk.call()])
81              self.assertEqual(mkAdvance.call_args_list,
82                               [mk.call(pupa)])
83
84      def test_develop(self):
85          """test run development system"""
86
87          pupa = mk.create_autospec(PupaTest, spec_set=True)
88
89          with mk.patch.object(development.Pupa, '_develop',
90                               autospec=True) as mkDevelop:
91              with mk.patch.object(development.Pupa, '_make_adult',
92                                   autospec=True) as mkMake:
93                  mkDevelop.side_effect = [False, True]
94                  master = mk.MagicMock()
95                  master.attach_mock(pupa,    'pupa')
96                  master.attach_mock(mkMake, 'make')
97
98                  # Does not develop
99                  self.Pupa.develop(pupa)
100                 self.assertEqual(mkDevelop.call_args_list,
101                                  [mk.call(self.Pupa, pupa)])
102                 self.assertEqual(master.mock_calls, [])
103
104                 mkDevelop.reset_mock()
105                 # Does develop
106                 self.Pupa.develop(pupa)
107                 self.assertEqual(mkDevelop.call_args_list,
108                                  [mk.call(self.Pupa, pupa)])
109                 self.assertEqual(master.mock_calls,
110                                  [mk.call.pupa.deactivate(),
111                                   mk.call.make(pupa)])
112
113     def test_setup(self):
114         """test setup the class"""
115
```

```
        # Test if have the model
        kwargs = {keyword.pupa_development: self.development}
        self.Pupa = development.Pupa.setup(**kwargs)
        self.assertIsInstance(self.Pupa, development.Pupa)
        self.assertEqual(self.Pupa.development, self.development)

        # Test if have the model
        kwargs = {}
        self.Pupa = development.Pupa.setup(**kwargs)
        self.assertIsInstance(self.Pupa, development.Pupa)
        self.assertEqual(self.Pupa.development, None)
```

C.5.5 test_forage

```
FallArmyworm
└── test
    └── test_forage
        ├── test_cannibalism.py
        ├── test_egg.py
        ├── test_larva.py
        ├── test_models.py
        ├── test_plant.py
        └── test_target.py
```

C.5.5.1 test_cannibalism.py

```
import unittest      as ut
import unittest.mock as mk

import dataclasses  as dclass
import numpy.random as rnd

import source.keyword as keyword

import source.agents.egg_mass as agent_egg_mass
import source.agents.larva    as agent_larva

import source.forage.cannibalism as cannibalism


class LarvaTest(agent_larva.Larva):
    """Class to add dynamic values for tests"""

    agent_key = mk.MagicMock(spec=str)
    mass      = mk.MagicMock(spec=float)
    genotype  = mk.MagicMock(spec=str)
    alive     = mk.MagicMock(spec=bool)
    full      = mk.MagicMock(spec=bool)


class EggMassTest(agent_egg_mass.EggMass):
    """Class to add dynamic values for tests"""

    agent_key = keyword.egg_mass


class TestCannibalism(ut.TestCase):
    """test the Cannibalism behavior class"""

    def setUp(self):
        """Setup the tests"""

        self.fight     = mk.MagicMock(spec=callable)
```

```
38          self.encounter = mk.MagicMock(spec=callable)
39          self.radius    = mk.MagicMock(spec=callable)
40
41          self.Cannibalism = cannibalism.Cannibalism(self.fight,
42                                                     self.encounter,
43                                                     self.radius)
44
45      def test___init__(self):
46          """test __init__ for class"""
47
48          self.assertIsInstance(self.Cannibalism, cannibalism.Cannibalism)
49
50          self.assertEqual(self.Cannibalism.fight,     self.fight)
51          self.assertEqual(self.Cannibalism.encounter, self.encounter)
52          self.assertEqual(self.Cannibalism.radius,    self.radius)
53
54          self.assertTrue(dclass.is_dataclass(self.Cannibalism))
55
56      def test__use_fight(self):
57          """test if we use the fight system"""
58
59          self.assertTrue(self.Cannibalism._use_fight)
60
61          self.Cannibalism.fight = None
62          self.assertFalse(self.Cannibalism._use_fight)
63
64      def test__winner(self):
65          """test determine if larva is winner of fight with target"""
66
67          larva  = mk.create_autospec(LarvaTest, spec_set=True)
68          target = mk.create_autospec(LarvaTest, spec_set=True)
69
70          self.assertEqual(self.Cannibalism._winner(larva, target),
71                           self.fight.return_value)
72          self.assertEqual(self.fight.call_args_list,
73                           [mk.call(larva.mass, target.mass)])
74
75      def test__fight(self):
76          """test run fight"""
```

```
77
78          larva  = mk.create_autospec(LarvaTest, spec_set=True)
79          target = mk.create_autospec(LarvaTest, spec_set=True)
80
81          with mk.patch.object(cannibalism.Cannibalism, '_winner',
82                               autospec=True) as mkWinner:
83              mkWinner.side_effect = [True, False]
84
85              # Test if larva wins
86              self.Cannibalism._fight(larva, target)
87              self.assertEqual(larva.consume_larva.call_args_list,
88                               [mk.call(target)])
89              self.assertEqual(target.consume_larva.call_args_list, [])
90              self.assertEqual(mkWinner.call_args_list,
91                               [mk.call(self.Cannibalism, larva, target)])
92
93              mkWinner.reset_mock()
94              larva.reset_mock()
95              # Test if target wins
96              self.Cannibalism._fight(larva, target)
97              self.assertEqual(target.consume_larva.call_args_list,
98                               [mk.call(larva)])
99              self.assertEqual(larva.consume_larva.call_args_list, [])
100             self.assertEqual(mkWinner.call_args_list,
101                              [mk.call(self.Cannibalism, larva, target)])
102
103     def test__contest(self):
104         """test run a contest"""
105
106         larva  = mk.create_autospec(LarvaTest, spec_set=True)
107         target = mk.create_autospec(LarvaTest, spec_set=True)
108
109         with mk.patch.object(cannibalism.Cannibalism, '_use_fight',
110                              autospec=True) as mkUse:
111             with mk.patch.object(cannibalism.Cannibalism, '_fight',
112                                  autospec=True) as mkFight:
113                 mkUse.__get__ = mk.MagicMock(side_effect=[False, True])
114
115                 # Test if no fight model
```

```
                self.Cannibalism._contest(larva, target)
                self.assertEqual(mkFight.call_args_list, [])

                # Test if fight model
                self.Cannibalism._contest(larva, target)
                self.assertEqual(mkFight.call_args_list,
                                 [mk.call(self.Cannibalism, larva, target)])

    def test__use_radius(self):
        """test if we use the radius system"""

        self.assertTrue(self.Cannibalism._use_radius)

        self.Cannibalism.radius = None
        self.assertFalse(self.Cannibalism._use_radius)

    def test__bounds(self):
        """test get the bounds on the radius"""

        larva = mk.create_autospec(LarvaTest, spec_set=True)

        with mk.patch.object(cannibalism.Cannibalism, '_use_radius',
                             autospec=True) as mkUse:
            mkUse.__get__ = mk.MagicMock(side_effect=[False, True])

            # Test if no radius model
            self.assertEqual(self.Cannibalism._bounds(larva),
                             {keyword.upper: 0,
                              keyword.lower: 0})
            self.assertEqual(self.radius.call_args_list, [])

            # Test if radius model
            self.assertEqual(self.Cannibalism._bounds(larva),
                             {keyword.upper: self.radius.return_value,
                              keyword.lower: 0})
            self.assertEqual(self.radius.call_args_list,
                             [mk.call(larva.mass, larva.genotype)])

    def test__targets(self):
```

```
155         """test get list of targets for cannibalism"""
156 
157         larva = mk.create_autospec(LarvaTest, spec_set=True)
158 
159         bounds = {'test': mk.MagicMock()}
160 
161         with mk.patch.object(cannibalism.Cannibalism, '_bounds',
162                              autospec=True) as mkBounds:
163             mkBounds.return_value = bounds
164 
165             self.assertEqual(self.Cannibalism._targets(larva),
166                              larva.targets.return_value)
167             self.assertEqual(larva.targets.call_args_list,
168                              [mk.call(**bounds)])
169             self.assertEqual(mkBounds.call_args_list,
170                              [mk.call(self.Cannibalism, larva)])
171 
172     def test__get_target(self):
173         """test get insect to encounter"""
174 
175         target  = mk.create_autospec(LarvaTest, spec_set=True)
176         targets = mk.MagicMock(spec=list)
177 
178         with mk.patch.object(rnd, 'choice') as mkRND:
179             mkRND.return_value = target
180 
181             self.assertEqual(self.Cannibalism._get_target(targets), target)
182             self.assertEqual(targets.remove.call_args_list,
183                              [mk.call(target)])
184             self.assertEqual(mkRND.call_args_list,
185                              [mk.call(targets)])
186 
187     def test__can_encounter(self):
188         """test if larva can encounter"""
189 
190         larva = mk.create_autospec(LarvaTest, spec_set=True)
191 
192         # Has encounter, larva is alive, larva is full
193         larva.alive = True
```

```
194         larva.full  = True
195         self.assertFalse(self.Cannibalism._can_encounter(larva))
196 
197         # Has encounter, larva is alive, larva is not full
198         larva.alive = True
199         larva.full  = False
200         self.assertTrue(self.Cannibalism._can_encounter(larva))
201 
202         # Has encounter, larva is  not alive, larva is full
203         larva.alive = False
204         larva.full  = False
205         self.assertFalse(self.Cannibalism._can_encounter(larva))
206 
207         # Has no encounter
208         self.Cannibalism.encounter = None
209         larva.alive = True
210         larva.full  = False
211         self.assertFalse(self.Cannibalism._can_encounter(larva))
212 
213     def test__encounter(self):
214         """test determine if encounter occurs"""
215 
216         larva   = mk.create_autospec(LarvaTest, spec_set=True)
217         targets = mk.MagicMock(spec=list)
218 
219         with mk.patch.object(cannibalism.Cannibalism, '_can_encounter',
220                              autospec=True) as mkCan:
221             with mk.patch.object(cannibalism, 'len') as mkLen:
222                 mkCan.side_effect = [False, True]
223 
224                 # Test cannot
225                 self.assertFalse(self.Cannibalism._encounter(larva, targets))
226                 self.assertEqual(self.encounter.call_args_list, [])
227                 self.assertEqual(mkLen.call_args_list, [])
228                 self.assertEqual(mkCan.call_args_list,
229                                  [mk.call(self.Cannibalism, larva)])
230 
231                 mkCan.reset_mock()
232                 # Test can
```

```
                self.assertEqual(self.Cannibalism._encounter(larva, targets),
                                 self.encounter.return_value)
                self.assertEqual(self.encounter.call_args_list,
                                 [mk.call(mkLen.return_value,
                                          larva.mass, larva.genotype)])
                self.assertEqual(mkLen.call_args_list,
                                 [mk.call(targets)])
                self.assertEqual(mkCan.call_args_list,
                                 [mk.call(self.Cannibalism, larva)])

    def test__cannibalize(self):
        """test perform cannibalism on target"""

        larva   = mk.create_autospec(LarvaTest, spec_set=True)
        targets = mk.MagicMock(spec=list)

        target = mk.create_autospec(LarvaTest, spec_set=True)

        with mk.patch.object(cannibalism.Cannibalism, '_get_target',
                             autospec=True) as mkGet:
            with mk.patch.object(cannibalism.Cannibalism, '_contest',
                                 autospec=True) as mkContest:
                mkGet.return_value = target

                # Test target is larva
                self.Cannibalism._cannibalize(larva, targets)
                self.assertEqual(mkContest.call_args_list,
                                 [mk.call(self.Cannibalism, larva, target)])
                self.assertEqual(larva.consume_egg.call_args_list, [])
                self.assertEqual(mkGet.call_args_list,
                                 [mk.call(targets)])

                mkGet.reset_mock()
                mkContest.reset_mock()
                # Test target is egg
                target.agent_key = keyword.egg_mass
                self.Cannibalism._cannibalize(larva, targets)
                self.assertEqual(mkContest.call_args_list, [])
                self.assertEqual(larva.consume_egg.call_args_list,
```

```
                                [mk.call(target)])
                self.assertEqual(mkGet.call_args_list,
                                [mk.call(targets)])

    def test__cannibalism(self):
        """test run cannibalism step"""

        larva   = mk.create_autospec(LarvaTest, spec_set=True)
        targets = mk.MagicMock(spec=list)

        with mk.patch.object(cannibalism.Cannibalism, '_encounter',
                             autospec=True) as mkEncounter:
            with mk.patch.object(cannibalism.Cannibalism, '_cannibalize',
                                 autospec=True) as mkCannibalize:
                mkEncounter.side_effect = [False, True]

                # Test if no encounter
                self.assertFalse(self.Cannibalism._cannibalism(larva, targets))
                self.assertEqual(mkEncounter.call_args_list,
                                 [mk.call(self.Cannibalism, larva, targets)])
                self.assertEqual(mkCannibalize.call_args_list, [])

                mkEncounter.reset_mock()
                # Test if encounter
                self.assertTrue(self.Cannibalism._cannibalism(larva, targets))
                self.assertEqual(mkEncounter.call_args_list,
                                 [mk.call(self.Cannibalism, larva, targets)])
                self.assertEqual(mkCannibalize.call_args_list,
                                 [mk.call(self.Cannibalism, larva, targets)])

    def test__run_cannibalism(self):
        """test run cannibalism"""

        larva = mk.create_autospec(LarvaTest, spec_set=True)

        with mk.patch.object(cannibalism.Cannibalism, '_cannibalism',
                             autospec=True) as mkCannibalism:
            with mk.patch.object(cannibalism.Cannibalism, '_targets',
                                 autospec=True) as mkTargets:
```

```
                # Four time repeat
                mkCannibalism.side_effect = [True, True, True, False]
                self.Cannibalism._run_cannibalism(larva)
                for call in mkCannibalism.call_args_list:
                    self.assertEqual(call,
                                     mk.call(self.Cannibalism,
                                             larva, mkTargets.return_value))
                self.assertEqual(len(mkCannibalism.call_args_list), 4)
                self.assertEqual(mkTargets.call_args_list,
                                 [mk.call(self.Cannibalism, larva)])

                mkTargets.reset_mock()
                mkCannibalism.reset_mock()
                # Three time repeat
                mkCannibalism.side_effect = [True, True, False]
                self.Cannibalism._run_cannibalism(larva)
                for call in mkCannibalism.call_args_list:
                    self.assertEqual(call,
                                     mk.call(self.Cannibalism,
                                             larva, mkTargets.return_value))
                self.assertEqual(len(mkCannibalism.call_args_list), 3)
                self.assertEqual(mkTargets.call_args_list,
                                 [mk.call(self.Cannibalism, larva)])

                mkTargets.reset_mock()
                mkCannibalism.reset_mock()
                # Two time repeat
                mkCannibalism.side_effect = [True, False]
                self.Cannibalism._run_cannibalism(larva)
                for call in mkCannibalism.call_args_list:
                    self.assertEqual(call,
                                     mk.call(self.Cannibalism,
                                             larva, mkTargets.return_value))
                self.assertEqual(len(mkCannibalism.call_args_list), 2)
                self.assertEqual(mkTargets.call_args_list,
                                 [mk.call(self.Cannibalism, larva)])

                mkTargets.reset_mock()
                mkCannibalism.reset_mock()
```

```
350                 # One time repeat
351                 mkCannibalism.side_effect = [False]
352                 self.Cannibalism._run_cannibalism(larva)
353                 for call in mkCannibalism.call_args_list:
354                     self.assertEqual(call,
355                                      mk.call(self.Cannibalism,
356                                              larva, mkTargets.return_value))
357                 self.assertEqual(len(mkCannibalism.call_args_list), 1)
358                 self.assertEqual(mkTargets.call_args_list,
359                                  [mk.call(self.Cannibalism, larva)])
360
361     def test_cannibalism(self):
362         """test run cannibalism system"""
363
364         larva = mk.create_autospec(LarvaTest, spec_set=True)
365
366         with mk.patch.object(cannibalism.Cannibalism, '_can_encounter',
367                              autospec=True) as mkCan:
368             with mk.patch.object(cannibalism.Cannibalism, '_run_cannibalism',
369                                  autospec=True) as mkRun:
370                 mkCan.side_effect = [False, True]
371
372                 # Test cannot
373                 self.Cannibalism.cannibalism(larva)
374                 self.assertEqual(mkCan.call_args_list,
375                                  [mk.call(self.Cannibalism, larva)])
376                 self.assertEqual(mkRun.call_args_list, [])
377
378                 mkCan.reset_mock()
379                 # Test can
380                 self.Cannibalism.cannibalism(larva)
381                 self.assertEqual(mkCan.call_args_list,
382                                  [mk.call(self.Cannibalism, larva)])
383                 self.assertEqual(mkRun.call_args_list,
384                                  [mk.call(self.Cannibalism, larva)])
385
386     def test_setup(self):
387         """test setup the class"""
388
```

```
        # Test if have the models
        kwargs = {keyword.fight:     self.fight,
                  keyword.encounter: self.encounter,
                  keyword.radius:    self.radius}
        self.Cannibalism = cannibalism.Cannibalism.setup(**kwargs)
        self.assertIsInstance(self.Cannibalism, cannibalism.Cannibalism)
        self.assertEqual(self.Cannibalism.fight,     self.fight)
        self.assertEqual(self.Cannibalism.encounter, self.encounter)
        self.assertEqual(self.Cannibalism.radius,    self.radius)

        # Test if have no models
        kwargs = {}
        self.Cannibalism = cannibalism.Cannibalism.setup(**kwargs)
        self.assertIsInstance(self.Cannibalism, cannibalism.Cannibalism)
        self.assertEqual(self.Cannibalism.fight,     None)
        self.assertEqual(self.Cannibalism.encounter, None)
        self.assertEqual(self.Cannibalism.radius,    None)
```

C.5.5.2 test_egg.py

```
import unittest      as ut
import unittest.mock as mk

import dataclasses as dclass

import source.keyword as keyword

import source.agents.egg_mass as agent_egg_mass
import source.agents.larva    as agent_larva

import source.forage.egg as forage


class LarvaTest(agent_larva.Larva):
    """Class to add dynamic values for tests"""

    mass     = mk.MagicMock(spec=float)
    genotype = mk.MagicMock(spec=str)


class EggMassTest(agent_egg_mass.EggMass):
    """Class to add dynamic values for tests"""

    agent_key = keyword.egg_mass


class TestEgg(ut.TestCase):
    """test the Egg forage class"""

    def setUp(self):
        """Setup the tests"""

        self.forage = mk.MagicMock(spec=callable)

        self.Egg = forage.Egg(self.forage)

    def test___init__(self):
```

```
        """test __init__ for class"""

        self.assertIsInstance(self.Egg, forage.Egg)

        self.assertEqual(self.Egg.forage, self.forage)

        self.assertTrue(dclass.is_dataclass(self.Egg))

    def test__use_forage(self):
        """test if we use the forage system"""

        self.assertTrue(self.Egg._use_forage)

        self.Egg.forage = None
        self.assertFalse(self.Egg._use_forage)

    def test__available(self):
        """test get the available amount of mass"""

        larva    = mk.create_autospec(LarvaTest, spec_set=True)
        egg_mass = mk.create_autospec(EggMassTest, spec_set=True)

        self.assertEqual(self.Egg._available(larva, egg_mass),
                         self.forage.return_value)
        self.assertEqual(self.forage.call_args_list,
                         [mk.call(egg_mass.mass, larva.mass, larva.genotype)])

    def test__consume(self):
        """test consume the egg"""

        larva    = mk.create_autospec(LarvaTest, spec_set=True)
        egg_mass = mk.create_autospec(EggMassTest, spec_set=True)

        with mk.patch.object(forage.Egg, '_available',
                             autospec=True) as mkAvailable:
            self.Egg._consume(larva, egg_mass)
            self.assertEqual(egg_mass.feed.call_args_list,
                             [mk.call(larva.add_egg.return_value)])
            self.assertEqual(larva.add_egg.call_args_list,
```

```
                            [mk.call(mkAvailable.return_value)])
        self.assertEqual(mkAvailable.call_args_list,
                            [mk.call(self.Egg, larva, egg_mass)])

def test_consume(self):
    """test consume the larva"""

    larva    = mk.create_autospec(LarvaTest, spec_set=True)
    egg_mass = mk.create_autospec(EggMassTest, spec_set=True)

    with mk.patch.object(forage.Egg, '_use_forage', autospec=True) as mkUse:
        with mk.patch.object(forage.Egg, '_consume',
                             autospec=True) as mkConsume:
            mkUse.__get__ = mk.MagicMock(side_effect=[False, True])

            # No forage model is given
            self.Egg.consume(larva, egg_mass)
            self.assertEqual(mkConsume.call_args_list, [])

            # Forage model is given
            self.Egg.consume(larva, egg_mass)
            self.assertEqual(mkConsume.call_args_list,
                             [mk.call(self.Egg, larva, egg_mass)])

def test_setup(self):
    """test setup the class"""

    # Test if have the model
    kwargs = {keyword.egg_forage: self.forage}
    self.Egg = forage.Egg.setup(**kwargs)
    self.assertIsInstance(self.Egg, forage.Egg)
    self.assertEqual(self.Egg.forage, self.forage)

    # Test if have the model
    kwargs = {}
    self.Egg = forage.Egg.setup(**kwargs)
    self.assertIsInstance(self.Egg, forage.Egg)
    self.assertEqual(self.Egg.forage, None)
```

C.5.5.3 test_larva.py

```
import unittest      as ut
import unittest.mock as mk

import dataclasses as dclass

import source.keyword as keyword

import source.agents.larva as agent_larva

import source.forage.larva as forage


class LarvaTest(agent_larva.Larva):
    """Class to add dynamic values for tests"""

    mass     = mk.MagicMock(spec=float)
    genotype = mk.MagicMock(spec=str)
    alive    = mk.MagicMock(spec=bool)


class TestLarva(ut.TestCase):
    """test the Larva forage class"""

    def setUp(self):
        """Setup the tests"""

        self.forage = mk.MagicMock(spec=callable)

        self.Larva = forage.Larva(self.forage)

    def test___init__(self):
        """test __init__ for class"""

        self.assertIsInstance(self.Larva, forage.Larva)

        self.assertEqual(self.Larva.forage, self.forage)

```

```
        self.assertTrue(dclass.is_dataclass(self.Larva))

    def test__use_forage(self):
        """test if we use the forage system"""

        self.assertTrue(self.Larva._use_forage)

        self.Larva.forage = None
        self.assertFalse(self.Larva._use_forage)

    def test__available(self):
        """test get the available amount of mass"""

        larva  = mk.create_autospec(LarvaTest, spec_set=True)
        target = mk.create_autospec(LarvaTest, spec_set=True)

        self.assertEqual(self.Larva._available(larva, target),
                         self.forage.return_value)
        self.assertEqual(self.forage.call_args_list,
                         [mk.call(target.mass, larva.mass, larva.genotype)])

    def test__consume(self):
        """test consume the larva"""

        larva  = mk.create_autospec(LarvaTest, spec_set=True)
        target = mk.create_autospec(LarvaTest, spec_set=True)

        mass = mk.MagicMock(spec=float)
        target.mass = mass

        with mk.patch.object(forage.Larva, '_available',
                             autospec=True) as mkAvailable:
            # Target is dead
            target.alive = False
            self.Larva._consume(larva, target)
            self.assertEqual(target.die.call_args_list, [])
            self.assertEqual(target.mass, mass.__sub__.return_value)
            self.assertEqual(mass.__sub__.call_args_list,
                             [mk.call(larva.add_larva.return_value)])
```

```
            self.assertEqual(larva.add_larva.call_args_list,
                             [mk.call(mkAvailable.return_value)])
            self.assertEqual(mkAvailable.call_args_list,
                             [mk.call(self.Larva, larva, target)])

            larva.reset_mock()
            target.mass = mass
            mass.reset_mock()
            mkAvailable.reset_mock()
            # Target is alive
            target.alive = True
            self.Larva._consume(larva, target)
            self.assertEqual(target.die.call_args_list,
                             [mk.call(keyword.cannibalism)])
            self.assertEqual(target.mass, mass.__sub__.return_value)
            self.assertEqual(mass.__sub__.call_args_list,
                             [mk.call(larva.add_larva.return_value)])
            self.assertEqual(larva.add_larva.call_args_list,
                             [mk.call(mkAvailable.return_value)])
            self.assertEqual(mkAvailable.call_args_list,
                             [mk.call(self.Larva, larva, target)])

    def test_consume(self):
        """test consume the larva"""

        larva  = mk.create_autospec(LarvaTest, spec_set=True)
        target = mk.create_autospec(LarvaTest, spec_set=True)

        with mk.patch.object(forage.Larva, '_use_forage',
                             autospec=True) as mkUse:
            with mk.patch.object(forage.Larva, '_consume',
                                 autospec=True) as mkConsume:
                mkUse.__get__ = mk.MagicMock(side_effect=[False, True])

                # No forage model is given
                self.Larva.consume(larva, target)
                self.assertEqual(mkConsume.call_args_list, [])

                # Forage model is given
```

```
                self.Larva.consume(larva, target)
                self.assertEqual(mkConsume.call_args_list,
                                 [mk.call(self.Larva, larva, target)])

    def test_setup(self):
        """test setup the class"""

        # Test if have the model
        kwargs = {keyword.larva_forage: self.forage}
        self.Larva = forage.Larva.setup(**kwargs)
        self.assertIsInstance(self.Larva, forage.Larva)
        self.assertEqual(self.Larva.forage, self.forage)

        # Test if have the model
        kwargs = {}
        self.Larva = forage.Larva.setup(**kwargs)
        self.assertIsInstance(self.Larva, forage.Larva)
        self.assertEqual(self.Larva.forage, None)
```

C.5.5.4 test_models.py

```
import unittest      as ut
import unittest.mock as mk

import dataclasses   as dclass
import scipy.stats   as stats
import scipy.special as spcl
import numpy         as np
import numpy.random  as rnd

import source.keyword as keyword

import source.biomass.models as biomass

import source.forage.models as model

import source.simulation.models as models


class TestPlantBase(ut.TestCase):
    """test the PlantBase mathematical model"""

    def setUp(self):
        """Setup the tests"""

        self.steps = mk.MagicMock(spec=int)

        self.PlantBase = model.PlantBase(self.steps)

    def test___init__(self):
        """test __init__ for class"""

        self.assertIsInstance(self.PlantBase, models.Model)
        self.assertIsInstance(self.PlantBase, model.PlantBase)

        self.assertEqual(self.PlantBase.steps, self.steps)

        self.assertEqual(self.PlantBase.model_key, keyword.plant_forage)
```

```
38
39          self.assertTrue(dclass.is_dataclass(self.PlantBase))
40
41      def test___call__(self):
42          """test call the model"""
43
44          mass     = mk.MagicMock(spec=float)
45          plant    = mk.MagicMock(spec=float)
46          genotype = mk.MagicMock(spec=str)
47          bt       = mk.MagicMock(spec=str)
48
49          self.assertIsNone(self.PlantBase(mass, plant, genotype, bt))
50
51
52  class TestPlantAdLibitum(ut.TestCase):
53      """test the PlantAdLibitum mathematical model"""
54
55      def setUp(self):
56          """Setup the tests"""
57
58          self.steps   = mk.MagicMock(spec=int)
59          self.max_gut = mk.MagicMock(spec=biomass.MaxGut)
60
61          self.PlantAdLibitum = model.PlantAdLibitum(self.steps,
62                                                     self.max_gut)
63
64      def test___init__(self):
65          """test __init__ for class"""
66
67          self.assertIsInstance(self.PlantAdLibitum, models.Model)
68          self.assertIsInstance(self.PlantAdLibitum, model.PlantBase)
69          self.assertIsInstance(self.PlantAdLibitum, model.PlantAdLibitum)
70
71          self.assertEqual(self.PlantAdLibitum.steps,   self.steps)
72          self.assertEqual(self.PlantAdLibitum.max_gut, self.max_gut)
73
74          self.assertEqual(self.PlantAdLibitum.model_key, keyword.plant_forage)
75
76          self.assertTrue(dclass.is_dataclass(self.PlantAdLibitum))
```

```
77
78      def test___call__(self):
79          """test call the model"""
80
81          mass     = mk.MagicMock(spec=float)
82          plant    = mk.MagicMock(spec=float)
83          genotype = mk.MagicMock(spec=str)
84          bt       = mk.MagicMock(spec=str)
85
86          self.assertEqual(self.PlantAdLibitum(mass, plant, genotype, bt),
87                           self.max_gut.return_value.__truediv__.return_value)
88          self.assertEqual(self.max_gut.return_value.__truediv__.call_args_list,
89                           [mk.call(self.steps)])
90          self.assertEqual(self.max_gut.call_args_list,
91                           [mk.call(mass)])
92
93
94  class TestPlantStarve(ut.TestCase):
95      """test PlantStarve mathematical model"""
96
97      def setUp(self):
98          """Setup the tests"""
99
100         self.steps   = mk.MagicMock(spec=int)
101         self.theta   = mk.MagicMock(spec=float)
102         self.sigma   = mk.MagicMock(spec=float)
103         self.max_gut = mk.MagicMock(spec=biomass.MaxGut)
104
105         self.PlantStarve = model.PlantStarve(self.steps,
106                                              self.theta,
107                                              self.sigma,
108                                              self.max_gut)
109
110     def test___init__(self):
111         """test __init__ for class"""
112
113         self.assertIsInstance(self.PlantStarve, models.Model)
114         self.assertIsInstance(self.PlantStarve, model.PlantBase)
115         self.assertIsInstance(self.PlantStarve, model.PlantStarve)
```

```
116
117         self.assertEqual(self.PlantStarve.steps,   self.steps)
118         self.assertEqual(self.PlantStarve.theta,   self.theta)
119         self.assertEqual(self.PlantStarve.sigma,   self.sigma)
120         self.assertEqual(self.PlantStarve.max_gut, self.max_gut)
121
122         self.assertEqual(self.PlantStarve.model_key, keyword.plant_forage)
123
124         self.assertTrue(dclass.is_dataclass(self.PlantStarve))
125
126     def test__mu(self):
127         """test get the mean for the distribution"""
128
129         mass = mk.MagicMock(spec=float)
130
131         self.assertEqual(self.PlantStarve._mu(mass),
132                          self.theta.__truediv__.return_value.
133                             __mul__.return_value)
134         self.assertEqual(self.theta.__truediv__.return_value.
135                             __mul__.call_args_list,
136                          [mk.call(self.max_gut.return_value)])
137         self.assertEqual(self.theta.__truediv__.call_args_list,
138                          [mk.call(self.steps)])
139         self.assertEqual(self.max_gut.call_args_list,
140                          [mk.call(mass)])
141
142     def test___call__(self):
143         """test call the model"""
144
145         mass     = mk.MagicMock(spec=float)
146         plant    = mk.MagicMock(spec=float)
147         genotype = mk.MagicMock(spec=str)
148         bt       = mk.MagicMock(spec=str)
149
150         with mk.patch.object(model.PlantStarve, '_mu', autospec=True) as mkMu:
151             with mk.patch.object(stats.truncnorm, 'rvs',
152                                  autospec=True) as mkRVS:
153                 with mk.patch.object(model, 'float') as mkFloat:
154                     self.assertEqual(self.PlantStarve(mass, plant,
```

```
                                                    genotype, bt),
                                    mkFloat.return_value)
                    self.assertEqual(mkFloat.call_args_list,
                                    [mk.call(mkRVS.return_value)])
                    self.assertEqual(mkRVS.call_args_list,
                                    [mk.call(0, np.inf,
                                             loc=mkMu.return_value,
                                             scale=self.sigma)])
                    self.assertEqual(mkMu.call_args_list,
                                    [mk.call(self.PlantStarve, mass)])


class TestEgg(ut.TestCase):
    """test Egg forage mathematical model"""

    def setUp(self):
        """Setup the tests"""

        self.factor = mk.MagicMock(spec=float)

        self.Egg = model.Egg(self.factor)

    def test___init__(self):
        """test __init__ for class"""

        self.assertIsInstance(self.Egg, models.Model)
        self.assertIsInstance(self.Egg, model.Egg)

        self.assertEqual(self.Egg.factor, self.factor)

        self.assertEqual(self.Egg.model_key, keyword.egg_forage)

        self.assertTrue(dclass.is_dataclass(self.Egg))

    def test___call__(self):
        """test call the class"""

        egg_mass = mk.MagicMock(spec=float)
        mass     = mk.MagicMock(spec=float)
```

```
        genotype = mk.MagicMock(spec=str)

        self.assertEqual(self.Egg(egg_mass, mass, genotype),
                         self.factor.__mul__.return_value)
        self.assertEqual(self.factor.__mul__.call_args_list,
                         [mk.call(egg_mass)])


class TestLarva(ut.TestCase):
    """test Larva forage mathematical model"""

    def setUp(self):
        """Setup the tests"""

        self.factor = mk.MagicMock(spec=float)

        self.Larva = model.Larva(self.factor)

    def test___init__(self):
        """test __init__ for class"""

        self.assertIsInstance(self.Larva, models.Model)
        self.assertIsInstance(self.Larva, model.Larva)

        self.assertEqual(self.Larva.factor, self.factor)

        self.assertEqual(self.Larva.model_key, keyword.larva_forage)

        self.assertTrue(dclass.is_dataclass(self.Larva))

    def test___call__(self):
        """test call the class"""

        target_mass = mk.MagicMock(spec=float)
        mass        = mk.MagicMock(spec=float)
        genotype    = mk.MagicMock(spec=str)

        self.assertEqual(self.Larva(target_mass, mass, genotype),
                         self.factor.__mul__.return_value)
```

```
        self.assertEqual(self.factor.__mul__.call_args_list,
                         [mk.call(target_mass)])


class TestLoss(ut.TestCase):
    """test target Loss mathematical model"""

    def setUp(self):
        """Setup the tests"""

        self.slope = mk.MagicMock(spec=float)
        self.mid   = mk.MagicMock(spec=dict)

        self.max_gut      = mk.MagicMock(spec=biomass.MaxGut)
        self.forage_egg   = mk.MagicMock(spec=model.Egg)
        self.forage_larva = mk.MagicMock(spec=model.Larva)

        self.Loss = model.Loss(self.slope,
                               self.mid,
                               self.max_gut,
                               self.forage_egg,
                               self.forage_larva)

    def test___init__(self):
        """test __init__ for class"""

        self.assertIsInstance(self.Loss, models.Model)
        self.assertIsInstance(self.Loss, model.Loss)

        self.assertEqual(self.Loss.slope,        self.slope)
        self.assertEqual(self.Loss.mid,          self.mid)
        self.assertEqual(self.Loss.max_gut,      self.max_gut)
        self.assertEqual(self.Loss.forage_egg,   self.forage_egg)
        self.assertEqual(self.Loss.forage_larva, self.forage_larva)

        self.assertEqual(self.Loss.model_key, keyword.loss)

        self.assertTrue(dclass.is_dataclass(self.Loss))

```

```
272     def test__diff(self):
273         """test find the difference in food and gut"""
274 
275         mass        = mk.MagicMock(spec=float)
276         target_mass = mk.MagicMock(spec=float)
277         genotype    = mk.MagicMock(spec=str)
278 
279         # Is egg mass
280         self.assertEqual(self.Loss._diff(mass, target_mass, genotype,
281                                          keyword.egg_mass),
282                          self.forage_egg.return_value.__sub__.return_value)
283         self.assertEqual(self.forage_egg.return_value.__sub__.call_args_list,
284                          [mk.call(self.max_gut.return_value)])
285         self.assertEqual(self.forage_egg.call_args_list,
286                          [mk.call(target_mass, mass, genotype)])
287         self.assertEqual(self.max_gut.call_args_list,
288                          [mk.call(mass)])
289         self.assertEqual(self.forage_larva.call_args_list, [])
290 
291         self.forage_egg.reset_mock()
292         self.max_gut.reset_mock()
293         # Is larva
294         self.assertEqual(self.Loss._diff(mass, target_mass, genotype,
295                                          keyword.larva),
296                          self.forage_larva.return_value.__sub__.return_value)
297         self.assertEqual(self.forage_larva.return_value.__sub__.call_args_list,
298                          [mk.call(self.max_gut.return_value)])
299         self.assertEqual(self.forage_larva.call_args_list,
300                          [mk.call(target_mass, mass, genotype)])
301         self.assertEqual(self.max_gut.call_args_list,
302                          [mk.call(mass)])
303         self.assertEqual(self.forage_egg.call_args_list, [])
304 
305     def test__prob(self):
306         """test find the probability"""
307 
308         mass        = mk.MagicMock(spec=float)
309         target_mass = mk.MagicMock(spec=float)
310         genotype    = mk.MagicMock(spec=str)
```

```
        target_key  = mk.MagicMock(spec=str)

        with mk.patch.object(model.Loss, '_diff', autospec=True) as mkDiff:
            with mk.patch.object(np, 'exp') as mkExp:
                self.assertEqual(self.Loss._prob(mass, target_mass,
                                                 genotype, target_key),
                                 self.mid.__getitem__.return_value.
                                    __mul__.return_value.
                                    __add__.return_value.
                                    __rtruediv__.return_value)
                self.assertEqual(self.mid.__getitem__.return_value.
                                    __mul__.return_value.
                                    __add__.return_value.
                                    __rtruediv__.call_args_list,
                                  [mk.call(1)])
                self.assertEqual(self.mid.__getitem__.return_value.
                                    __mul__.return_value.
                                    __add__.call_args_list,
                                 [mk.call(1)])
                self.assertEqual(self.mid.__getitem__.return_value.
                                    __mul__.call_args_list,
                                 [mk.call(mkExp.return_value)])
                self.assertEqual(self.mid.__getitem__.call_args_list,
                                 [mk.call(target_key)])
                self.assertEqual(mkExp.call_args_list,
                                 [mk.call(self.slope.
                                             __neg__.return_value.
                                             __mul__.return_value)])
                self.assertEqual(self.slope.__neg__.return_value.
                                    __mul__.call_args_list,
                                 [mk.call(mkDiff.return_value)])
                self.assertEqual(self.slope.__neg__.call_args_list,
                                 [mk.call()])
                self.assertEqual(mkDiff.call_args_list,
                                 [mk.call(self.Loss, mass, target_mass,
                                          genotype, target_key)])

    def test___call__(self):
        """test call the model"""
```

```
350
351         mass        = mk.MagicMock(spec=float)
352         target_mass = mk.MagicMock(spec=float)
353         genotype    = mk.MagicMock(spec=str)
354         target_key  = mk.MagicMock(spec=str)
355
356         with mk.patch.object(model.Loss, '_prob', autospec=True) as mkProb:
357             with mk.patch.object(rnd, 'random') as mkRND:
358                 mkRND.return_value.__le__.side_effect = [True, False]
359
360                 # test true
361                 self.assertTrue(self.Loss(mass, target_mass, genotype, target_key))
362                 self.assertEqual(mkRND.return_value.__le__.call_args_list,
363                                  [mk.call(mkProb.return_value)])
364                 self.assertEqual(mkRND.call_args_list,
365                                  [mk.call()])
366                 self.assertEqual(mkProb.call_args_list,
367                                  [mk.call(self.Loss, mass, target_mass,
368                                           genotype, target_key)])
369
370                 mkRND.reset_mock()
371                 mkProb.reset_mock()
372                 # test false
373                 self.assertFalse(self.Loss(mass, target_mass, genotype, target_key))
374                 self.assertEqual(mkRND.return_value.__le__.call_args_list,
375                                  [mk.call(mkProb.return_value)])
376                 self.assertEqual(mkRND.call_args_list,
377                                  [mk.call()])
378                 self.assertEqual(mkProb.call_args_list,
379                                  [mk.call(self.Loss, mass, target_mass,
380                                           genotype, target_key)])
381
382
383 class TestFight(ut.TestCase):
384     """test Fight mathematical model"""
385
386     def setUp(self):
387         """Setup the tests"""
388
```

```
        self.slope = mk.MagicMock(spec=float)

        self.Fight = model.Fight(self.slope)

    def test___init__(self):
        """test __init__ for class"""

        self.assertIsInstance(self.Fight, models.Model)
        self.assertIsInstance(self.Fight, model.Fight)

        self.assertEqual(self.Fight.slope, self.slope)

        self.assertEqual(self.Fight.model_key, keyword.fight)

        self.assertTrue(dclass.is_dataclass(self.Fight))

    def test_prob(self):
        """test get the probability"""

        mass0 = mk.MagicMock(spec=float)
        mass1 = mk.MagicMock(spec=float)

        with mk.patch.object(spcl, 'expit', autospec=True) as mkExpit:
            self.assertEqual(self.Fight.prob(mass0, mass1),
                             mkExpit.return_value)
            self.assertEqual(mkExpit.call_args_list,
                             [mk.call(self.slope.__mul__.return_value)])
            self.assertEqual(self.slope.__mul__.call_args_list,
                             [mk.call(mass0.__sub__.return_value)])
            self.assertEqual(mass0.__sub__.call_args_list,
                             [mk.call(mass1)])

    def test___call__(self):
        """test __call__ the model"""

        mass0 = mk.MagicMock(spec=float)
        mass1 = mk.MagicMock(spec=float)

        with mk.patch.object(model.Fight, 'prob', autospec=True) as mkProb:
```

```
428             with mk.patch.object(rnd, 'random') as mkRND:
429                 mkRND.return_value.__le__.side_effect = [True, False]
430 
431                 # Test True
432                 self.assertTrue(self.Fight(mass0, mass1))
433                 self.assertEqual(mkRND.return_value.__le__.call_args_list,
434                                  [mk.call(mkProb.return_value)])
435                 self.assertEqual(mkRND.call_args_list,
436                                  [mk.call()])
437                 self.assertEqual(mkProb.call_args_list,
438                                  [mk.call(self.Fight, mass0, mass1)])
439 
440                 mkRND.reset_mock()
441                 mkProb.reset_mock()
442                 # Test False
443                 self.assertFalse(self.Fight(mass0, mass1))
444                 self.assertEqual(mkRND.return_value.__le__.call_args_list,
445                                  [mk.call(mkProb.return_value)])
446                 self.assertEqual(mkRND.call_args_list,
447                                  [mk.call()])
448                 self.assertEqual(mkProb.call_args_list,
449                                  [mk.call(self.Fight, mass0, mass1)])
450 
451 
452 class TestEncounter(ut.TestCase):
453     """test Encounter mathematical model"""
454 
455     def setUp(self):
456         """Setup the tests"""
457 
458         self.factor = mk.MagicMock(spec=float)
459 
460         self.Encounter = model.Encounter(self.factor)
461 
462     def test___init__(self):
463         """test __init__ for class"""
464 
465         self.assertIsInstance(self.Encounter, models.Model)
466         self.assertIsInstance(self.Encounter, model.Encounter)
```

```
467
468         self.assertEqual(self.Encounter.factor, self.factor)
469
470         self.assertEqual(self.Encounter.model_key, keyword.encounter)
471
472         self.assertTrue(dclass.is_dataclass(self.Encounter))
473
474     def test__prob(self):
475         """test get probability"""
476
477         number = mk.MagicMock(spec=int)
478
479         with mk.patch.object(np, 'exp', autospec=True) as mkExp:
480             self.assertEqual(self.Encounter._prob(number),
481                              mkExp.return_value.__rsub__.return_value)
482             self.assertEqual(mkExp.return_value.__rsub__.call_args_list,
483                              [mk.call(1)])
484             self.assertEqual(mkExp.call_args_list,
485                              [mk.call(self.factor.__neg__.return_value.
486                                       __mul__.return_value)])
487             self.assertEqual(self.factor.__neg__.return_value.
488                                 __mul__.call_args_list,
489                              [mk.call(number)])
490             self.assertEqual(self.factor.__neg__.call_args_list,
491                              [mk.call()])
492
493     def test___call__(self):
494         """test call the model"""
495
496         number   = mk.MagicMock(spec=int)
497         mass     = mk.MagicMock(spec=float)
498         genotype = mk.MagicMock(spec=str)
499
500         with mk.patch.object(model.Encounter, '_prob', autospec=True) as mkProb:
501             with mk.patch.object(rnd, 'random') as mkRND:
502                 mkRND.return_value.__le__.side_effect = [True, False]
503
504                 # Test True
505                 self.assertTrue(self.Encounter(number, mass, genotype))
```

```
                self.assertEqual(mkRND.return_value.__le__.call_args_list,
                                 [mk.call(mkProb.return_value)])
                self.assertEqual(mkRND.call_args_list,
                                 [mk.call()])
                self.assertEqual(mkProb.call_args_list,
                                 [mk.call(self.Encounter, number)])

                mkRND.reset_mock()
                mkProb.reset_mock()
                # Test False
                self.assertFalse(self.Encounter(number, mass, genotype))
                self.assertEqual(mkRND.return_value.__le__.call_args_list,
                                 [mk.call(mkProb.return_value)])
                self.assertEqual(mkRND.call_args_list,
                                 [mk.call()])
                self.assertEqual(mkProb.call_args_list,
                                 [mk.call(self.Encounter, number)])


class TestRadius(ut.TestCase):
    """test Radius mathematical model"""

    def setUp(self):
        """Setup the tests"""

        self.radius = mk.MagicMock(spec=int)

        self.Radius = model.Radius(self.radius)

    def test___init__(self):
        """test __init__ for class"""

        self.assertIsInstance(self.Radius, models.Model)
        self.assertIsInstance(self.Radius, model.Radius)

        self.assertEqual(self.Radius.radius, self.radius)

        self.assertEqual(self.Radius.model_key, keyword.radius)

```

```
        self.assertTrue(dclass.is_dataclass(self.Radius))

    def test___call__(self):
        """test call the model"""

        mass     = mk.MagicMock(spec=float)
        genotype = mk.MagicMock(spec=str)

        self.assertEqual(self.Radius(mass, genotype), self.radius)
```

C.5.5.5 test_plant.py

```
import unittest      as ut
import unittest.mock as mk

import dataclasses as dclass

import source.keyword as keyword

import source.agents.larva as agent_larva

import source.forage.plant as forage


class LarvaTest(agent_larva.Larva):
    """Class to add dynamic values for tests"""

    mass     = mk.MagicMock(spec=float)
    genotype = mk.MagicMock(spec=str)


class TestPlant(ut.TestCase):
    """test the Plant forage class"""

    def setUp(self):
        """Setup the tests"""

        self.forage = mk.MagicMock(spec=callable)

        self.Plant = forage.Plant(self.forage)

    def test___init__(self):
        """test __init__ for class"""

        self.assertIsInstance(self.Plant, forage.Plant)

        self.assertEqual(self.Plant.forage, self.forage)

        self.assertTrue(dclass.is_dataclass(self.Plant))
```

```
38
39      def test__use_forage(self):
40          """test if we use the forage system"""
41
42          self.assertTrue(self.Plant._use_forage)
43
44          self.Plant.forage = None
45          self.assertFalse(self.Plant._use_forage)
46
47      def test__available(self):
48          """test get the available amount of mass"""
49
50          larva = mk.create_autospec(LarvaTest, spec_set=True)
51
52          self.assertEqual(self.Plant._available(larva),
53                           self.forage.return_value)
54          self.assertEqual(self.forage.call_args_list,
55                           [mk.call(larva.mass, larva.plant,
56                                    larva.genotype, larva.bt)])
57
58      def test__consume(self):
59          """test run consume """
60
61          larva = mk.create_autospec(LarvaTest, spec_set=True)
62
63          with mk.patch.object(forage.Plant, '_available',
64                               autospec=True) as mkAvailable:
65              self.Plant._consume(larva)
66              self.assertEqual(larva.add_plant.call_args_list,
67                               [mk.call(mkAvailable.return_value)])
68              self.assertEqual(mkAvailable.call_args_list,
69                               [mk.call(self.Plant, larva)])
70
71      def test_consume(self):
72          """test consume the larva"""
73
74          larva = mk.create_autospec(LarvaTest, spec_set=True)
75
76          with mk.patch.object(forage.Plant, '_use_forage', autospec=True) as mkUse:
```

```
            with mk.patch.object(forage.Plant, '_consume',
                                 autospec=True) as mkConsume:
                mkUse.__get__ = mk.MagicMock(side_effect=[False, True])

                # No forage model is given
                self.Plant.consume(larva)
                self.assertEqual(mkConsume.call_args_list, [])

                # Forage model is given
                self.Plant.consume(larva)
                self.assertEqual(mkConsume.call_args_list,
                                 [mk.call(self.Plant, larva)])

    def test_setup(self):
        """test setup the class"""

        # Test if have the model
        kwargs = {keyword.plant_forage: self.forage}
        self.Plant = forage.Plant.setup(**kwargs)
        self.assertIsInstance(self.Plant, forage.Plant)
        self.assertEqual(self.Plant.forage, self.forage)

        # Test if have the model
        kwargs = {}
        self.Plant = forage.Plant.setup(**kwargs)
        self.assertIsInstance(self.Plant, forage.Plant)
        self.assertEqual(self.Plant.forage, None)
```

C.5.5.6 test_target.py

```
import unittest      as ut
import unittest.mock as mk

import dataclasses as dclass

import source.keyword as keyword

import source.agents.egg_mass as agent_egg_mass
import source.agents.larva    as agent_larva

import source.forage.target as forage


class LarvaTest(agent_larva.Larva):
    """Class to add dynamic values for tests"""


    agent_key = keyword.larva
    mass      = mk.MagicMock(spec=float)
    genotype  = mk.MagicMock(spec=str)
    target    = mk.MagicMock()


class EggMassTest(agent_egg_mass.EggMass):
    """Class to add dynamic values for tests"""

    agent_key = keyword.egg_mass


class TestTarget(ut.TestCase):
    """test Target foraging class"""

    def setUp(self):
        """Setup the tests"""

        self.loss = mk.MagicMock(spec=callable)

```

```
        self.Target = forage.Target(self.loss)

    def test___init__(self):
        """test __init__ for class"""

        self.assertIsInstance(self.Target, forage.Target)

        self.assertEqual(self.Target.loss, self.loss)

        self.assertTrue(dclass.is_dataclass(self.Target))

    def test__use_loss(self):
        """test if we use the loss system"""

        self.assertTrue(self.Target._use_loss)

        self.Target.loss = None
        self.assertFalse(self.Target._use_loss)

    def test__keep_target(self):
        """test if we loose the target"""

        larva = mk.create_autospec(LarvaTest, spec_set=True)

        with mk.patch.object(forage.Target, '_use_loss',
                             autospec=True) as mkUse:
            mkUse.__get__ = mk.MagicMock(side_effect=[False, True, True])

            # Test no loss function
            self.assertFalse(self.Target._keep_target(larva, larva))
            self.assertEqual(self.loss.call_args_list, [])

            # Test target is egg_mass
            target = mk.create_autospec(EggMassTest, spec_set=True)
            self.assertEqual(self.Target._keep_target(larva, target),
                             self.loss.return_value)
            self.assertEqual(self.loss.call_args_list,
                             [mk.call(larva.mass, target.mass,
                                      larva.genotype, target.agent_key)])
```

```
77
78              self.loss.reset_mock()
79              # Test target is larva
80              target = mk.create_autospec(LarvaTest, spec_set=True)
81              self.assertEqual(self.Target._keep_target(larva, target),
82                               self.loss.return_value)
83              self.assertEqual(self.loss.call_args_list,
84                               [mk.call(larva.mass, target.mass,
85                                        larva.genotype, target.agent_key)])
86
87      def test__consume_target(self):
88          """test consume the target larva"""
89
90          larva = mk.create_autospec(LarvaTest, spec_set=True)
91
92          # Target is egg_mass
93          target = mk.create_autospec(EggMassTest, spec_set=True)
94          target.agent_key = keyword.egg_mass
95          self.Target._consume_target(larva, target)
96          self.assertEqual(larva.consume_egg.call_args_list,
97                           [mk.call(target)])
98          self.assertEqual(larva.consume_larva.call_args_list, [])
99
100         larva.reset_mock()
101         # Target is egg_mass
102         target = mk.create_autospec(LarvaTest, spec_set=True)
103         target.agent_key = keyword.larva
104         self.Target._consume_target(larva, target)
105         self.assertEqual(larva.consume_egg.call_args_list, [])
106         self.assertEqual(larva.consume_larva.call_args_list,
107                          [mk.call(target)])
108
109     def test_consume(self):
110         """test consume the target"""
111
112         larva  = mk.create_autospec(LarvaTest, spec_set=True)
113         target = mk.MagicMock()
114
115         larva.target = target
```

```
116
117         with mk.patch.object(forage.Target, '_keep_target',
118                              autospec=True) as mkKeep:
119             with mk.patch.object(forage.Target, '_consume_target',
120                                  autospec=True) as mkConsume:
121                 mkKeep.side_effect = [True, False]
122
123                 # Consume target
124                 self.Target.consume(larva)
125                 self.assertEqual(larva.target, target)
126                 self.assertEqual(mkKeep.call_args_list,
127                                  [mk.call(self.Target, larva, target)])
128                 self.assertEqual(mkConsume.call_args_list,
129                                  [mk.call(larva, target)])
130
131                 mkKeep.reset_mock()
132                 mkConsume.reset_mock()
133                 # Keep target
134                 self.Target.consume(larva)
135                 self.assertEqual(larva.target, None)
136                 self.assertEqual(mkKeep.call_args_list,
137                                  [mk.call(self.Target, larva, target)])
138                 self.assertEqual(mkConsume.call_args_list, [])
139
140     def test_setup(self):
141         """test setup the class"""
142
143         # Test if have the model
144         kwargs = {keyword.loss: self.loss}
145         self.Target = forage.Target.setup(**kwargs)
146         self.assertIsInstance(self.Target, forage.Target)
147         self.assertEqual(self.Target.loss, self.loss)
148
149         # Test if have the model
150         kwargs = {}
151         self.Target = forage.Target.setup(**kwargs)
152         self.assertIsInstance(self.Target, forage.Target)
153         self.assertEqual(self.Target.loss, None)
```

C.5.6 test_migration

```
FallArmyworm
└── test
    └── test_migration
        ├── test_emigration.py
        └── test_immigration.py
```

C.5.6.1 test_emigration.py

```
import unittest      as ut
import unittest.mock as mk

import dataclasses  as dclass
import collections  as collect
import numpy.random as rnd
import scipy.stats  as stats

import source.keyword as keyword

import source.agents.agent as main_agent

import source.migration.emigration as emigration

import source.space.agents as main_agents


class TestEmigration(ut.TestCase):
    """test the Emigration class"""

    def setUp(self):
        """Setup the tests"""

        self.mu    = mk.MagicMock(spec=float)
        self.sigma = mk.MagicMock(spec=float)

        self.agent_keys = [mk.MagicMock(spec=str) for _ in range(3)]

        self.Emigration = emigration.Emigration(self.mu,
                                                self.sigma,
                                                self.agent_keys)

    def test___init__(self):
        """test __init__ for class"""

        self.assertIsInstance(self.Emigration, emigration.Emigration)

```

```
38          self.assertEqual(self.Emigration.mu,          self.mu)
39          self.assertEqual(self.Emigration.sigma,       self.sigma)
40          self.assertEqual(self.Emigration.agent_keys, self.agent_keys)
41
42          self.assertEqual(self.Emigration.location, (0,))
43
44          self.assertTrue(dclass.is_dataclass(self.Emigration))
45
46      def test_remove(self):
47          """test get the probability of agent emigrating"""
48
49          population = mk.MagicMock(spec=float)
50
51          with mk.patch.object(stats.norm, 'cdf') as mkCDF:
52              with mk.patch.object(rnd, 'random') as mkRND:
53                  mkRND.return_value.__le__.side_effect = [True, False]
54
55                  # Test if True
56                  self.assertTrue(self.Emigration._remove(population))
57                  self.assertEqual(mkRND.return_value.__le__.call_args_list,
58                                   [mk.call(mkCDF.return_value)])
59                  self.assertEqual(mkRND.call_args_list,
60                                   [mk.call()])
61                  self.assertEqual(mkCDF.call_args_list,
62                                   [mk.call(population,
63                                            loc=self.mu, scale=self.sigma)])
64
65                  mkRND.reset_mock()
66                  mkCDF.reset_mock()
67                  # Test if False
68                  self.assertFalse(self.Emigration._remove(population))
69                  self.assertEqual(mkRND.return_value.__le__.call_args_list,
70                                   [mk.call(mkCDF.return_value)])
71                  self.assertEqual(mkRND.call_args_list,
72                                   [mk.call()])
73                  self.assertEqual(mkCDF.call_args_list,
74                                   [mk.call(population,
75                                            loc=self.mu, scale=self.sigma)])
76
```

```
77      def test__emigrate(self):
78          """test run emigrate on agent"""
79
80          agent      = mk.create_autospec(main_agent.Agent, spec_set=True)
81          population = mk.MagicMock(spec=float)
82
83          with mk.patch.object(emigration.Emigration, '_remove',
84                               autospec=True) as mkRemove:
85              mkRemove.side_effect = [False, True]
86
87              # Test not remove
88              self.assertEqual(self.Emigration._emigrate(agent, population),
89                               population)
90              self.assertEqual(population.__sub__.call_args_list, [])
91              self.assertEqual(mkRemove.call_args_list,
92                               [mk.call(self.Emigration, population)])
93              self.assertEqual(agent.die.call_args_list, [])
94
95              mkRemove.reset_mock()
96              # Test not remove
97              self.assertEqual(self.Emigration._emigrate(agent, population),
98                               population.__sub__.return_value)
99              self.assertEqual(population.__sub__.call_args_list,
100                              [mk.call(1)])
101             self.assertEqual(mkRemove.call_args_list,
102                              [mk.call(self.Emigration, population)])
103             self.assertEqual(agent.die.call_args_list,
104                              [mk.call(keyword.emigrate)])
105
106     def test__agents(self):
107         """test create the agents"""
108
109         agents = mk.create_autospec(main_agents.Agents, spec_set=True)
110         agents.__getitem__.return_value = \
111             mk.create_autospec(main_agents.AgentsBin, spec_set=True)
112
113         population = []
114         agent_bins = []
115         for agent_key in self.agent_keys:
```

```
            pop       = {mk.MagicMock(spec=str):
                                mk.create_autospec(main_agent.Agent, spec_set=True)
                            for _ in range(3)}
            agent_bin = main_agents.AgentBin(pop, mk.MagicMock(), agent_key)
            population.extend(agent_bin.agents)
            agent_bins.append(agent_bin)

        agents.__getitem__.return_value.__getitem__.side_effect = agent_bins

        self.assertEqual(self.Emigration._agents(agents), population)
        for index, agent_key in enumerate(self.agent_keys):
            self.assertEqual(agents.__getitem__.return_value.
                                __getitem__.call_args_list[index],
                                mk.call(agent_key))
            self.assertEqual(agents.__getitem__.call_args_list[index],
                                mk.call(self.Emigration.location))
        self.assertEqual(len(agents.__getitem__.return_value.
                                __getitem__.call_args_list), 3)
        self.assertEqual(len(agents.__getitem__.call_args_list), 3)

    def test_emigration(self):
        """test run emigration"""

        agents = mk.create_autospec(main_agents.Agents, spec_set=True)

        population = [mk.create_autospec(main_agent.Agent, spec_set=True)
                      for _ in range(3)]
        pop        = [mk.MagicMock(spec=int) for _ in range(3)]

        with mk.patch.object(emigration.Emigration, '_agents',
                                autospec=True) as mkAgents:
            with mk.patch.object(emigration.Emigration, '_emigrate',
                                    autospec=True) as mkEmigrate:
                with mk.patch.object(emigration, 'len') as mkLen:
                    mkAgents.return_value  = population
                    mkEmigrate.side_effect = pop

                    self.Emigration.emigration(agents)
                    self.assertEqual(len(mkEmigrate.call_args_list), 3)
```

```
                    call = mkEmigrate.call_args_list.pop(0)
                    self.assertEqual(call,
                                     mk.call(self.Emigration,
                                             population[0], mkLen.return_value))
                    self.assertEqual(mkLen.call_args_list,
                                     [mk.call(population)])
                    self.assertEqual(mkAgents.call_args_list,
                                     [mk.call(self.Emigration, agents)])

                    for index, call in enumerate(mkEmigrate.call_args_list):
                        self.assertEqual(call,
                                         mk.call(self.Emigration,
                                                 population[index + 1],
                                                 pop[index]))
                    self.assertEqual(len(mkEmigrate.call_args_list), 2)


class TestEmigrations(ut.TestCase):
    """test Emigrations class"""

    def setUp(self):
        """Setup the tests"""

        self.emigrations = [mk.create_autospec(emigration.Emigration,
                                               spec_set=True)
                            for _ in range(3)]

        self.Emigrations = emigration.Emigrations(self.emigrations)

    def test___init__(self):
        """test __init__ for class"""

        self.assertIsInstance(self.Emigrations, collect.UserList)
        self.assertIsInstance(self.Emigrations, emigration.Emigrations)

        self.assertEqual(self.Emigrations,      self.emigrations)
        self.assertEqual(self.Emigrations.data, self.emigrations)

    def test_emigration(self):
```

```
        """test run emigration"""

        agents = mk.create_autospec(main_agents.Agents, spec_set=True)

        self.Emigrations.emigration(agents)
        for this in self.emigrations:
            self.assertEqual(this.emigration.call_args_list,
                             [mk.call(agents)])

    def test_setup(self):
        """test setup the class"""

        setup_tuples = [(mk.MagicMock(spec=float), mk.MagicMock(spec=float),
                         [mk.MagicMock(spec=str) for _ in range(3)])
                        for _ in range(3)]

        self.Emigrations = emigration.Emigrations.setup(setup_tuples)
        self.assertIsInstance(self.Emigrations, emigration.Emigrations)

        for index, emigrate in enumerate(self.Emigrations):
            self.assertIsInstance(emigrate, emigration.Emigration)
            self.assertEqual(emigrate.mu,         setup_tuples[index][0])
            self.assertEqual(emigrate.sigma,      setup_tuples[index][1])
            self.assertEqual(emigrate.agent_keys, setup_tuples[index][2])

            self.assertEqual(emigrate.location, (0,))
        self.assertEqual(len(self.Emigrations), 3)
```

C.5.6.2 test_immigration.py

```
import unittest      as ut
import unittest.mock as mk

import dataclasses  as dclass
import collections  as collect
import numpy.random as rnd
import scipy.stats  as stats

import source.keyword as keyword

import source.agents.adult    as adult
import source.agents.egg_mass as egg_mass
import source.agents.larva    as larva
import source.agents.pupa     as pupa

import source.migration.immigration as immigration

import source.simulation.simulation as main_simulation


class TestImmigration(ut.TestCase):
    """test Immigration class"""

    def setUp(self):
        """Setup the tests"""

        self.lam       = mk.MagicMock(spec=float)
        self.genotype  = mk.MagicMock(spec=str)
        self.agent_key = mk.MagicMock(spec=str)

        self.Immigration = immigration.Immigration(self.lam,
                                                   self.genotype,
                                                   self.agent_key)

    def test___init__(self):
        """test __init__ for class"""

```

```
38          self.assertIsInstance(self.Immigration, immigration.Immigration)
39
40          self.assertEqual(self.Immigration.lam,       self.lam)
41          self.assertEqual(self.Immigration.genotype,  self.genotype)
42          self.assertEqual(self.Immigration.agent_key, self.agent_key)
43
44          self.assertTrue(dclass.is_dataclass(self.Immigration))
45
46      def test__number(self):
47          """test get the number of immigrants"""
48
49          with mk.patch.object(stats.poisson, 'rvs') as mkRVS:
50              with mk.patch.object(immigration, 'int') as mkInt:
51                  self.assertEqual(self.Immigration._number(),
52                                   mkInt.return_value)
53                  self.assertEqual(mkInt.call_args_list,
54                                   [mk.call(mkRVS.return_value)])
55                  self.assertEqual(mkRVS.call_args_list,
56                                   [mk.call(self.lam)])
57
58      def test__immigrate_egg_masses(self):
59          """test immigrate egg_masses into simulation"""
60
61          simulation = mk.create_autospec(main_simulation.Simulation,
62                                          spec_set=True)
63          new = [mk.create_autospec(egg_mass.EggMass, spec_set=True)
64                 for _ in range(3)]
65
66          with mk.patch.object(immigration.Immigration, '_number',
67                               autospec=True) as mkNumber:
68              with mk.patch.object(egg_mass.EggMass, 'setup',
69                                   autospec=True) as mkSetup:
70                  mkNumber.return_value = 3
71                  mkSetup.side_effect = new
72
73                  self.Immigration._immigrate_egg_masses(simulation)
74                  self.assertEqual(mkNumber.call_args_list,
75                                   [mk.call(self.Immigration)])
76                  for index, agent in enumerate(new):
```

```
                self.assertEqual(mkSetup.call_args_list[index],
                                 mk.call(simulation.
                                             new_unique_id.return_value,
                                         keyword.immigrant,
                                         simulation,
                                         self.genotype))
                self.assertEqual(simulation.
                                 new_unique_id.call_args_list[index],
                                 mk.call())
                self.assertEqual(agent.activate.call_args_list,
                                 [mk.call()])
            self.assertEqual(len(mkSetup.call_args_list), 3)
            self.assertEqual(len(simulation.
                                    new_unique_id.call_args_list), 3)

    def test__immigrate_larvae(self):
        """test immigrate larvae into simulation"""

        simulation = mk.create_autospec(main_simulation.Simulation,
                                        spec_set=True)
        new = [mk.create_autospec(larva.Larva, spec_set=True)
               for _ in range(3)]

        with mk.patch.object(immigration.Immigration, '_number',
                             autospec=True) as mkNumber:
            with mk.patch.object(larva.Larva, 'setup',
                                 autospec=True) as mkSetup:
                mkNumber.return_value = 3
                mkSetup.side_effect = new

                self.Immigration._immigrate_larvae(simulation)
                self.assertEqual(mkNumber.call_args_list,
                                 [mk.call(self.Immigration)])
                for index, agent in enumerate(new):
                    self.assertEqual(mkSetup.call_args_list[index],
                                     mk.call(simulation.
                                             new_unique_id.return_value,
                                             keyword.immigrant,
                                             simulation,
```

```
                                            self.genotype))
                self.assertEqual(simulation.
                                 new_unique_id.call_args_list[index],
                                 mk.call())
                self.assertEqual(agent.activate.call_args_list,
                                 [mk.call()])
            self.assertEqual(len(mkSetup.call_args_list), 3)
            self.assertEqual(len(simulation.
                                 new_unique_id.call_args_list), 3)

    def test__immigrate_pupae(self):
        """test immigrate pupae into simulation"""

        simulation = mk.create_autospec(main_simulation.Simulation,
                                        spec_set=True)
        new = [mk.create_autospec(pupa.Pupa, spec_set=True)
               for _ in range(3)]

        with mk.patch.object(immigration.Immigration, '_number',
                             autospec=True) as mkNumber:
            with mk.patch.object(pupa.Pupa, 'setup',
                                 autospec=True) as mkSetup:
                mkNumber.return_value = 3
                mkSetup.side_effect = new

                self.Immigration._immigrate_pupae(simulation)
                self.assertEqual(mkNumber.call_args_list,
                                 [mk.call(self.Immigration)])
                for index, agent in enumerate(new):
                    self.assertEqual(mkSetup.call_args_list[index],
                                     mk.call(simulation.
                                             new_unique_id.return_value,
                                             keyword.immigrant,
                                             simulation,
                                             self.genotype))
                    self.assertEqual(simulation.
                                     new_unique_id.call_args_list[index],
                                     mk.call())
                    self.assertEqual(agent.activate.call_args_list,
```

```
                                    [mk.call()])
                self.assertEqual(len(mkSetup.call_args_list), 3)
                self.assertEqual(len(simulation.
                                    new_unique_id.call_args_list), 3)

    def test__immigrate_adults(self):
        """test immigrate adults into simulation"""

        simulation = mk.create_autospec(main_simulation.Simulation,
                                        spec_set=True)
        new = [mk.create_autospec(adult.Adult, spec_set=True)
               for _ in range(3)]

        with mk.patch.object(immigration.Immigration, '_number',
                             autospec=True) as mkNumber:
            with mk.patch.object(adult.Adult, 'setup',
                                 autospec=True) as mkSetup:
                mkNumber.return_value = 3
                mkSetup.side_effect = new

                self.Immigration._immigrate_adults(simulation)
                self.assertEqual(mkNumber.call_args_list,
                                 [mk.call(self.Immigration)])
                for index, agent in enumerate(new):
                    self.assertEqual(mkSetup.call_args_list[index],
                                     mk.call(simulation.
                                             new_unique_id.return_value,
                                             keyword.immigrant,
                                             simulation,
                                             self.genotype))
                    self.assertEqual(simulation.
                                     new_unique_id.call_args_list[index],
                                     mk.call())
                    self.assertEqual(agent.activate.call_args_list,
                                     [mk.call()])
                self.assertEqual(len(mkSetup.call_args_list), 3)
                self.assertEqual(len(simulation.
                                     new_unique_id.call_args_list), 3)

```

```
    def test__immigrate_pregnant(self):
        """test immigrate pregnant into simulation"""

        simulation = mk.create_autospec(main_simulation.Simulation,
                                        spec_set=True)
        new = [mk.create_autospec(adult.Adult, spec_set=True)
               for _ in range(3)]

        # Homo_r
        self.Immigration.genotype = keyword.homo_r
        with mk.patch.object(immigration.Immigration, '_number',
                             autospec=True) as mkNumber:
            with mk.patch.object(adult.Adult, 'setup',
                                 autospec=True) as mkSetup:
                with mk.patch.object(rnd, 'shuffle') as mkRND:
                    mkNumber.return_value = 3
                    mkSetup.side_effect = new

                    self.Immigration._immigrate_pregnant(simulation)
                    self.assertEqual(mkNumber.call_args_list,
                                     [mk.call(self.Immigration)])
                    for index, agent in enumerate(new):
                        self.assertEqual(mkSetup.call_args_list[index],
                                         mk.call(simulation.
                                                 new_unique_id.return_value,
                                                 keyword.immigrant,
                                                 simulation,
                                                 keyword.homo_r,
                                                 keyword.homo_r))
                        self.assertEqual(simulation.
                                         new_unique_id.call_args_list[index],
                                         mk.call())
                        self.assertEqual(mkRND.call_args_list[index],
                                         mk.call([keyword.homo_r,
                                                  keyword.homo_r]))
                        self.assertEqual(agent.activate.call_args_list,
                                         [mk.call()])
                        agent.reset_mock()
                    self.assertEqual(len(mkSetup.call_args_list), 3)
```

```
                    self.assertEqual(len(simulation.
                                         new_unique_id.call_args_list), 3)
                    self.assertEqual(len(mkRND.call_args_list), 3)
                    simulation.reset_mock()

        # Hetero
        self.Immigration.genotype = keyword.hetero
        with mk.patch.object(immigration.Immigration, '_number',
                             autospec=True) as mkNumber:
            with mk.patch.object(adult.Adult, 'setup',
                                 autospec=True) as mkSetup:
                with mk.patch.object(rnd, 'shuffle') as mkRND:
                    mkNumber.return_value = 3
                    mkSetup.side_effect = new

                    self.Immigration._immigrate_pregnant(simulation)
                    self.assertEqual(mkNumber.call_args_list,
                                     [mk.call(self.Immigration)])
                    for index, agent in enumerate(new):
                        self.assertEqual(mkSetup.call_args_list[index],
                                         mk.call(simulation.
                                                 new_unique_id.return_value,
                                                 keyword.immigrant,
                                                 simulation,
                                                 keyword.homo_r,
                                                 keyword.homo_s))
                        self.assertEqual(simulation.
                                         new_unique_id.call_args_list[index],
                                         mk.call())
                        self.assertEqual(mkRND.call_args_list[index],
                                         mk.call([keyword.homo_r,
                                                  keyword.homo_s]))
                        self.assertEqual(agent.activate.call_args_list,
                                         [mk.call()])
                        agent.reset_mock()
                    self.assertEqual(len(mkSetup.call_args_list), 3)
                    self.assertEqual(len(simulation.
                                         new_unique_id.call_args_list), 3)
                    self.assertEqual(len(mkRND.call_args_list), 3)
```

```
                    simulation.reset_mock()

        # Homo_s
        self.Immigration.genotype = keyword.homo_s
        with mk.patch.object(immigration.Immigration, '_number',
                             autospec=True) as mkNumber:
            with mk.patch.object(adult.Adult, 'setup',
                                 autospec=True) as mkSetup:
                with mk.patch.object(rnd, 'shuffle') as mkRND:
                    mkNumber.return_value = 3
                    mkSetup.side_effect = new

                    self.Immigration._immigrate_pregnant(simulation)
                    self.assertEqual(mkNumber.call_args_list,
                                     [mk.call(self.Immigration)])
                    for index, agent in enumerate(new):
                        self.assertEqual(mkSetup.call_args_list[index],
                                         mk.call(simulation.
                                                 new_unique_id.return_value,
                                                 keyword.immigrant,
                                                 simulation,
                                                 keyword.homo_s,
                                                 keyword.homo_s))
                        self.assertEqual(simulation.
                                         new_unique_id.call_args_list[index],
                                         mk.call())
                        self.assertEqual(mkRND.call_args_list[index],
                                         mk.call([keyword.homo_s,
                                                  keyword.homo_s]))
                        self.assertEqual(agent.activate.call_args_list,
                                         [mk.call()])
                        agent.reset_mock()
                    self.assertEqual(len(mkSetup.call_args_list), 3)
                    self.assertEqual(len(simulation.
                                         new_unique_id.call_args_list), 3)
                    self.assertEqual(len(mkRND.call_args_list), 3)
                    simulation.reset_mock()

    def test_immigration(self):
```

```
        """test run the immigration"""

        simulation = mk.create_autospec(main_simulation.Simulation,
                                        spec_set=True)

        with mk.patch.object(immigration.Immigration, '_immigrate_egg_masses',
                             autospec=True) as mkEggs:
            with mk.patch.object(immigration.Immigration, '_immigrate_larvae',
                                 autospec=True) as mkLarvae:
                with mk.patch.object(immigration.Immigration, '_immigrate_pupae',
                                     autospec=True) as mkPupae:
                    with mk.patch.object(immigration.Immigration,
                                         '_immigrate_adults',
                                         autospec=True) as mkAdults:
                        with mk.patch.object(immigration.Immigration,
                                             '_immigrate_pregnant',
                                             autospec=True) as mkPregnant:
                            # Not used agent_key
                            self.Immigration.immigration(simulation)
                            self.assertEqual(mkEggs.call_args_list, [])
                            self.assertEqual(mkLarvae.call_args_list, [])
                            self.assertEqual(mkPupae.call_args_list, [])
                            self.assertEqual(mkAdults.call_args_list, [])
                            self.assertEqual(mkPregnant.call_args_list, [])

                            # Eggs
                            self.Immigration.agent_key = keyword.egg_mass
                            self.Immigration.immigration(simulation)
                            self.assertEqual(mkEggs.call_args_list,
                                             [mk.call(self.Immigration,
                                                      simulation)])
                            self.assertEqual(mkLarvae.call_args_list, [])
                            self.assertEqual(mkPupae.call_args_list, [])
                            self.assertEqual(mkAdults.call_args_list, [])
                            self.assertEqual(mkPregnant.call_args_list, [])

                            mkEggs.reset_mock()
                            # Larvae
                            self.Immigration.agent_key = keyword.larva
```

```
                            self.Immigration.immigration(simulation)
                            self.assertEqual(mkEggs.call_args_list, [])
                            self.assertEqual(mkLarvae.call_args_list,
                                             [mk.call(self.Immigration,
                                                      simulation)])
                            self.assertEqual(mkPupae.call_args_list, [])
                            self.assertEqual(mkAdults.call_args_list, [])
                            self.assertEqual(mkPregnant.call_args_list, [])

                            mkLarvae.reset_mock()
                            # Pupae
                            self.Immigration.agent_key = keyword.pupa
                            self.Immigration.immigration(simulation)
                            self.assertEqual(mkEggs.call_args_list, [])
                            self.assertEqual(mkLarvae.call_args_list, [])
                            self.assertEqual(mkPupae.call_args_list,
                                             [mk.call(self.Immigration,
                                                      simulation)])
                            self.assertEqual(mkAdults.call_args_list, [])
                            self.assertEqual(mkPregnant.call_args_list, [])

                            mkPupae.reset_mock()
                            # Adults
                            self.Immigration.agent_key = keyword.adult
                            self.Immigration.immigration(simulation)
                            self.assertEqual(mkEggs.call_args_list, [])
                            self.assertEqual(mkLarvae.call_args_list, [])
                            self.assertEqual(mkPupae.call_args_list, [])
                            self.assertEqual(mkAdults.call_args_list,
                                             [mk.call(self.Immigration,
                                                      simulation)])
                            self.assertEqual(mkPregnant.call_args_list, [])

                            mkAdults.reset_mock()
                            # Pregnant
                            self.Immigration.agent_key = keyword.pregnant
                            self.Immigration.immigration(simulation)
                            self.assertEqual(mkEggs.call_args_list, [])
                            self.assertEqual(mkLarvae.call_args_list, [])
```

```
                            self.assertEqual(mkPupae.call_args_list, [])
                            self.assertEqual(mkAdults.call_args_list, [])
                            self.assertEqual(mkPregnant.call_args_list,
                                             [mk.call(self.Immigration,
                                                      simulation)])


class TestImmigrations(ut.TestCase):
    """test the Immigrations system class"""

    def setUp(self):
        """Setup the tests"""

        self.immigrations = [mk.create_autospec(immigration.Immigration,
                                                spec_set=True)
                             for _ in range(3)]

        self.Immigrations = immigration.Immigrations(self.immigrations)

    def test___init__(self):
        """test __init__ for class"""

        self.assertIsInstance(self.Immigrations, collect.UserList)
        self.assertIsInstance(self.Immigrations, immigration.Immigrations)

        self.assertEqual(self.Immigrations,      self.immigrations)
        self.assertEqual(self.Immigrations.data, self.immigrations)

    def test_immigration(self):
        """test run immigration"""

        simulation = mk.create_autospec(main_simulation.Simulation,
                                        spec_set=True)

        self.Immigrations.immigration(simulation)
        for this in self.immigrations:
            self.assertEqual(this.immigration.call_args_list,
                             [mk.call(simulation)])

```

```
    def test_setup(self):
        """test setup the class"""

        setup_tuples = [(mk.MagicMock(spec=float),
                         mk.MagicMock(spec=str), mk.MagicMock(spec=str))
                        for _ in range(3)]

        self.Immigrations = immigration.Immigrations.setup(setup_tuples)
        self.assertIsInstance(self.Immigrations, immigration.Immigrations)
        for index, immigrate in enumerate(self.Immigrations):
            self.assertIsInstance(immigrate, immigration.Immigration)
            self.assertEqual(immigrate.lam,       setup_tuples[index][0])
            self.assertEqual(immigrate.genotype,  setup_tuples[index][1])
            self.assertEqual(immigrate.agent_key, setup_tuples[index][2])
        self.assertEqual(len(self.Immigrations), 3)
```

C.5.7 test_movement

```
FallArmyworm
└── test
    └── test_movement
        ├── test_adult.py
        ├── test_larva.py
        └── test_models.py
```

C.5.7.1 test_adult.py

```
import unittest      as ut
import unittest.mock as mk

import dataclasses  as dclass
import numpy.random as rnd

import source.keyword as keyword

import source.agents.adult as agent_adult

import source.movement.adult as movement


class AdultTest(agent_adult.Adult):
    """Class to add dynamic values for tests"""

    mass     = mk.MagicMock(spec=float)
    genotype = mk.MagicMock(spec=str)


class TestAdult(ut.TestCase):
    """test the Adult movement class"""

    def setUp(self):
        """Setup the tests"""

        self.movement = mk.MagicMock(spec=callable)

        self.Adult = movement.Adult(self.movement)

    def test___init__(self):
        """test __init__ for class"""

        self.assertIsInstance(self.Adult, movement.Adult)

        self.assertEqual(self.Adult.movement, self.movement)

```

```
        self.assertTrue(dclass.is_dataclass(self.Adult))

    def test__use_movement(self):
        """test if we use the movement system"""

        self.assertTrue(self.Adult._use_movement)

        self.Adult.movement = None
        self.assertFalse(self.Adult._use_movement)

    def test__distance(self):
        """test determine if adult distances"""

        adult = mk.create_autospec(AdultTest, spec_set=True)

        self.assertEqual(self.Adult._distance(adult),
                         self.movement.return_value)
        self.assertEqual(self.movement.call_args_list,
                         [mk.call(adult.mass, adult.genotype)])

    def test__vertex(self):
        """test get the vertex to move to"""

        adult = mk.create_autospec(AdultTest, spec_set=True)

        with mk.patch.object(movement.Adult, '_distance',
                             autospec=True) as mkDistance:
            with mk.patch.object(movement, 'list') as mkList:
                with mk.patch.object(rnd, 'choice') as mkRND:
                    kwargs = {keyword.upper: mkDistance.return_value,
                              keyword.lower: mkDistance.return_value}

                    self.assertEqual(self.Adult._vertex(adult),
                                     mkRND.return_value)
                    self.assertEqual(mkRND.call_args_list,
                                     [mk.call(mkList.return_value)])
                    self.assertEqual(mkList.call_args_list,
                                     [mk.call(adult.vertices.return_value)])
                    self.assertEqual(adult.vertices.call_args_list,
```

```
                                    [mk.call(**kwargs)])
                self.assertEqual(mkDistance.call_args_list,
                                    [mk.call(self.Adult, adult)])

    def test_move(self):
        """test move the adult"""

        adult = mk.create_autospec(AdultTest, spec_set=True)

        with mk.patch.object(movement.Adult, '_use_movement',
                             autospec=True) as mkUse:
            with mk.patch.object(movement.Adult, '_vertex',
                                 autospec=True) as mkVertex:
                mkUse.__get__ = mk.MagicMock(side_effect=[False, True])

                # Test if no movement
                self.Adult.move(adult)
                self.assertEqual(adult.transfer.call_args_list, [])
                self.assertEqual(mkVertex.call_args_list, [])

                # Test if movement
                self.Adult.move(adult)
                self.assertEqual(adult.transfer.call_args_list,
                                 [mk.call(mkVertex.return_value,
                                          keyword.adult_level)])
                self.assertEqual(mkVertex.call_args_list,
                                 [mk.call(self.Adult, adult)])

    def test_setup(self):
        """test setup the class"""

        # Test if have the model
        kwargs = {keyword.adult_movement: self.movement}
        self.Adult = movement.Adult.setup(**kwargs)
        self.assertIsInstance(self.Adult, movement.Adult)
        self.assertEqual(self.Adult.movement, self.movement)

        # Test if have the model
        kwargs = {}
```

```
        self.Adult = movement.Adult.setup(**kwargs)
        self.assertIsInstance(self.Adult, movement.Adult)
        self.assertEqual(self.Adult.movement, None)
```

C.5.7.2 test_larva.py

```
import unittest      as ut
import unittest.mock as mk

import dataclasses  as dclass
import numpy.random as rnd

import source.keyword as keyword

import source.agents.larva as agent_larva

import source.movement.larva as movement


class LarvaTest(agent_larva.Larva):
    """Class to add dynamic values for tests"""

    mass     = mk.MagicMock(spec=float)
    genotype = mk.MagicMock(spec=str)


class TestLarva(ut.TestCase):
    """test the Larva movement class"""

    def setUp(self):
        """Setup the tests"""

        self.movement = mk.MagicMock(spec=callable)

        self.Larva = movement.Larva(self.movement)

    def test___init__(self):
        """test __init__ for class"""

        self.assertIsInstance(self.Larva, movement.Larva)

        self.assertEqual(self.Larva.movement, self.movement)

```

```
38          self.assertTrue(dclass.is_dataclass(self.Larva))
39
40      def test__use_movement(self):
41          """test if we use the movement system"""
42
43          self.assertTrue(self.Larva._use_movement)
44
45          self.Larva.movement = None
46          self.assertFalse(self.Larva._use_movement)
47
48      def test__distance(self):
49          """test determine if larva distances"""
50
51          larva = mk.create_autospec(LarvaTest, spec_set=True)
52
53          self.assertEqual(self.Larva._distance(larva),
54                           self.movement.return_value)
55          self.assertEqual(self.movement.call_args_list,
56                           [mk.call(larva.mass, larva.genotype)])
57
58      def test__vertex(self):
59          """test get the vertex to move to"""
60
61          larva = mk.create_autospec(LarvaTest, spec_set=True)
62
63          with mk.patch.object(movement.Larva, '_distance',
64                               autospec=True) as mkDistance:
65              with mk.patch.object(movement, 'list') as mkList:
66                  with mk.patch.object(rnd, 'choice') as mkRND:
67                      kwargs = {keyword.upper: mkDistance.return_value,
68                                keyword.lower: mkDistance.return_value}
69
70                      self.assertEqual(self.Larva._vertex(larva),
71                                       mkRND.return_value)
72                      self.assertEqual(mkRND.call_args_list,
73                                       [mk.call(mkList.return_value)])
74                      self.assertEqual(mkList.call_args_list,
75                                       [mk.call(larva.vertices.return_value)])
76                      self.assertEqual(larva.vertices.call_args_list,
```

```
                                    [mk.call(**kwargs)])
                self.assertEqual(mkDistance.call_args_list,
                                    [mk.call(self.Larva, larva)])

    def test_move(self):
        """test move the larva"""

        larva = mk.create_autospec(LarvaTest, spec_set=True)

        with mk.patch.object(movement.Larva, '_use_movement',
                             autospec=True) as mkUse:
            with mk.patch.object(movement.Larva, '_vertex',
                                 autospec=True) as mkVertex:
                mkUse.__get__ = mk.MagicMock(side_effect=[False, True])

                # Test if no movement
                self.Larva.move(larva)
                self.assertEqual(larva.transfer.call_args_list, [])
                self.assertEqual(mkVertex.call_args_list, [])

                # Test if movement
                self.Larva.move(larva)
                self.assertEqual(larva.transfer.call_args_list,
                                 [mk.call(mkVertex.return_value,
                                          keyword.larva_level)])
                self.assertEqual(mkVertex.call_args_list,
                                 [mk.call(self.Larva, larva)])

    def test_setup(self):
        """test setup the class"""

        # Test if have the model
        kwargs = {keyword.larva_movement: self.movement}
        self.Larva = movement.Larva.setup(**kwargs)
        self.assertIsInstance(self.Larva, movement.Larva)
        self.assertEqual(self.Larva.movement, self.movement)

        # Test if have the model
        kwargs = {}
```

```
        self.Larva = movement.Larva.setup(**kwargs)
        self.assertIsInstance(self.Larva, movement.Larva)
        self.assertEqual(self.Larva.movement, None)
```

C.5.7.3 test_models.py

```
import unittest      as ut
import unittest.mock as mk

import dataclasses as dclass
import scipy.stats as stats

import source.keyword as keyword

import source.movement.models as model

import source.simulation.models as models


class TestLevy(ut.TestCase):
    """test the Levy flight mathematical model"""

    def setUp(self):
        """Setup the tests"""

        self.scale = mk.MagicMock(spec=float)
        self.shape = mk.MagicMock(spec=float)

        self.Levy = model.Levy(self.scale,
                               self.shape)

    def test___init__(self):
        """test __init__ for class"""

        self.assertIsInstance(self.Levy, models.Model)
        self.assertIsInstance(self.Levy, model.Levy)

        self.assertEqual(self.Levy.scale, self.scale)
        self.assertEqual(self.Levy.shape, self.shape)

        self.assertTrue(dclass.is_dataclass(self.Levy))

    def test___call__(self):
```

```
38          """test call the model"""
39
40          mass     = mk.MagicMock(spec=float)
41          genotype = mk.MagicMock(spec=str)
42
43          with mk.patch.object(stats.pareto, 'rvs', autospec=True) as mkRVS:
44              with mk.patch.object(model, 'float') as mkFloat:
45                  self.assertEqual(self.Levy(mass, genotype),
46                                   mkFloat.return_value)
47                  self.assertEqual(mkFloat.call_args_list,
48                                   [mk.call(mkRVS.return_value)])
49                  self.assertEqual(mkRVS.call_args_list,
50                                   [mk.call(self.shape, self.scale)])
51
52
53  class TestLarva(ut.TestCase):
54      """test the Larva movement mathematical model"""
55
56      def setUp(self):
57          """Setup the tests"""
58
59          self.scale = mk.MagicMock(spec=float)
60          self.shape = mk.MagicMock(spec=float)
61
62          self.Larva = model.Larva(self.scale,
63                                   self.shape)
64
65      def test___init__(self):
66          """test __init__ for class"""
67
68          self.assertIsInstance(self.Larva, models.Model)
69          self.assertIsInstance(self.Larva, model.Levy)
70          self.assertIsInstance(self.Larva, model.Larva)
71
72          self.assertEqual(self.Larva.scale, self.scale)
73          self.assertEqual(self.Larva.shape, self.shape)
74
75          self.assertEqual(self.Larva.model_key, keyword.larva_movement)
76
```

```
        self.assertTrue(dclass.is_dataclass(self.Larva))


class TestAdult(ut.TestCase):
    """test the Adult movement mathematical model"""

    def setUp(self):
        """Setup the tests"""

        self.scale = mk.MagicMock(spec=float)
        self.shape = mk.MagicMock(spec=float)

        self.Adult = model.Adult(self.scale,
                                 self.shape)

    def test___init__(self):
        """test __init__ for class"""

        self.assertIsInstance(self.Adult, models.Model)
        self.assertIsInstance(self.Adult, model.Levy)
        self.assertIsInstance(self.Adult, model.Adult)

        self.assertEqual(self.Adult.scale, self.scale)
        self.assertEqual(self.Adult.shape, self.shape)

        self.assertEqual(self.Adult.model_key, keyword.adult_movement)

        self.assertTrue(dclass.is_dataclass(self.Adult))
```

C.5.8 test_reproduction

```
FallArmyworm
└── test
    └── test_reproduction
        ├── test_lay.py
        ├── test_mate.py
        └── test_models.py
```

C.5.8.1 test_lay.py

```
import unittest      as ut
import unittest.mock as mk

import dataclasses as dclass

import source.keyword as keyword

import source.agents.adult    as agent_adult
import source.agents.egg_mass as egg_mass

import source.reproduction.lay as lay


class AdultTest(agent_adult.Adult):
    """Class to add dynamic values for tests"""

    mass     = mk.MagicMock(spec=float)
    age      = mk.MagicMock(spec=int)
    genotype = mk.MagicMock(spec=str)
    num_eggs = mk.MagicMock(spec=int)


class TestLay(ut.TestCase):
    """test Lay behavior class"""

    def setUp(self):
        """Setup the tests"""

        self.fecundity = mk.MagicMock(spec=callable)
        self.density   = mk.MagicMock(spec=callable)

        self.Lay = lay.Lay(self.fecundity,
                           self.density)

    def test___init__(self):
        """test __init__ for class"""

```

```
        self.assertIsInstance(self.Lay, lay.Lay)

        self.assertEqual(self.Lay.fecundity, self.fecundity)
        self.assertEqual(self.Lay.density,   self.density)

        self.assertTrue(dclass.is_dataclass(self.Lay))

    def test__use_fecundity(self):
        """test if we use the fecundity system"""

        self.assertTrue(self.Lay._use_fecundity)

        self.Lay.fecundity = None
        self.assertFalse(self.Lay._use_fecundity)

    def test__use_density(self):
        """test if we use the density system"""

        self.assertTrue(self.Lay._use_density)

        self.Lay.density = None
        self.assertFalse(self.Lay._use_density)

    def test_reset(self):
        """test reset the num_eggs"""

        adult = mk.create_autospec(AdultTest, spec_set=True)

        with mk.patch.object(lay.Lay, '_use_fecundity', autospec=True) as mkUse:
            mkUse.__get__ = mk.MagicMock(side_effect=[False, True])

            # No fecundity
            self.assertEqual(self.Lay.reset(adult), 0)
            self.assertEqual(self.fecundity.call_args_list, [])

            # Has fecundity
            self.assertEqual(self.Lay.reset(adult),
                             self.fecundity.return_value)
            self.assertEqual(self.fecundity.call_args_list,
```

```
                            [mk.call(adult.age, adult.mass, adult.genotype)])

    def test__check_density(self):
        """test the density checker"""

        adult  = mk.create_autospec(AdultTest, spec_set=True)
        number = mk.MagicMock(spec=int)

        with mk.patch.object(lay.Lay, '_use_density', autospec=True) as mkUse:
            mkUse.__get__ = mk.MagicMock(side_effect=[False, True])

            # Test use is false
            self.assertTrue(self.Lay._check_density(adult, number))
            self.assertEqual(self.density.call_args_list, [])

            # Test use is True
            self.assertEqual(self.Lay._check_density(adult, number),
                             self.density.return_value)
            self.assertEqual(self.density.call_args_list,
                             [mk.call(number, adult.mass, adult.genotype)])

    def test__lay_egg_mass(self):
        """test lay an egg_mass"""

        adult  = mk.create_autospec(AdultTest, spec_set=True)
        number = mk.MagicMock(spec=int)

        num_eggs       = mk.MagicMock(spec=int)
        adult.num_eggs = num_eggs

        with mk.patch.object(lay.Lay, '_check_density',
                             autospec=True) as mkCheck:
            with mk.patch.object(egg_mass.EggMass, 'birth',
                                 autospec=True) as mkBirth:
                mkCheck.side_effect = [False, True]

                # Test Fail check
                self.assertEqual(self.Lay._lay_egg_mass(adult, number),
                                 ([], number, True))
```

```
                self.assertEqual(mkCheck.call_args_list,
                                 [mk.call(self.Lay, adult, number)])
                self.assertEqual(mkBirth.call_args_list, [])
                self.assertEqual(adult.num_eggs, num_eggs)
                self.assertEqual(number.__add__.call_args_list, [])
                self.assertEqual(num_eggs.__sub__.call_args_list, [])

                mkCheck.reset_mock()
                # Test pass check
                self.assertEqual(self.Lay._lay_egg_mass(adult, number),
                                 ([mkBirth.return_value],
                                  number.__add__.return_value,
                                  False))
                self.assertEqual(mkCheck.call_args_list,
                                 [mk.call(self.Lay, adult, number)])
                self.assertEqual(mkBirth.call_args_list,
                                 [mk.call(adult)])
                self.assertEqual(adult.num_eggs,
                                 num_eggs.__sub__.return_value)
                self.assertEqual(number.__add__.call_args_list,
                                 [mk.call(1)])
                self.assertEqual(num_eggs.__sub__.call_args_list,
                                 [mk.call(1)])

    def test_lay(self):
        """test lay loop"""

        adult = mk.create_autospec(AdultTest, spec_set=True)
        adult.num_eggs = 3

        with mk.patch.object(lay.Lay, '_lay_egg_mass', autospec=True) as mkLay:
            # No Break
            numbers    = []
            effects    = []
            egg_masses = []
            for _ in range(3):
                number = mk.MagicMock(spec=int)
                new    = [mk.create_autospec(egg_mass.EggMass, spec_set=True)
                          for _ in range(3)]
```

```
                effects.append((new, number, False))
                numbers.append(number)
                egg_masses.extend(new)
            mkLay.side_effect = effects
            self.assertEqual(self.Lay.lay(adult), egg_masses)
            self.assertEqual(len(mkLay.call_args_list), 3)
            call = mkLay.call_args_list.pop(0)
            self.assertEqual(call,
                             mk.call(self.Lay,
                                     adult, adult.population.return_value))
            self.assertEqual(adult.population.call_args_list,
                             [mk.call()])
            for index, call in enumerate(mkLay.call_args_list):
                self.assertEqual(call,
                                 mk.call(self.Lay,
                                         adult, numbers[index]))
            self.assertEqual(len(mkLay.call_args_list), 2)

            adult.reset_mock()
            mkLay.reset_mock()
            # Break on 3rd round
            stop_lay = [False, True]
            numbers    = []
            effects    = []
            egg_masses = []
            for index in range(2):
                number = mk.MagicMock(spec=int)
                new    = [mk.create_autospec(egg_mass.EggMass, spec_set=True)
                          for _ in range(3)]
                effects.append((new, number, stop_lay[index]))
                numbers.append(number)
                egg_masses.extend(new)
            mkLay.side_effect = effects
            self.assertEqual(self.Lay.lay(adult), egg_masses)
            self.assertEqual(len(mkLay.call_args_list), 2)
            call = mkLay.call_args_list.pop(0)
            self.assertEqual(call,
                             mk.call(self.Lay,
                                     adult, adult.population.return_value))
```

```
            self.assertEqual(adult.population.call_args_list,
                             [mk.call()])
            for index, call in enumerate(mkLay.call_args_list):
                self.assertEqual(call,
                                 mk.call(self.Lay,
                                         adult, numbers[index]))
            self.assertEqual(len(mkLay.call_args_list), 1)

            adult.reset_mock()
            mkLay.reset_mock()
            # Break on 2nd round
            numbers    = []
            effects    = []
            egg_masses = []
            for _ in range(1):
                number = mk.MagicMock(spec=int)
                new    = [mk.create_autospec(egg_mass.EggMass, spec_set=True)
                          for _ in range(3)]
                effects.append((new, number, True))
                numbers.append(number)
                egg_masses.extend(new)
            mkLay.side_effect = effects
            self.assertEqual(self.Lay.lay(adult), egg_masses)
            self.assertEqual(len(mkLay.call_args_list), 1)
            call = mkLay.call_args_list.pop(0)
            self.assertEqual(call,
                             mk.call(self.Lay,
                                     adult, adult.population.return_value))
            self.assertEqual(adult.population.call_args_list,
                             [mk.call()])
            self.assertEqual(len(mkLay.call_args_list), 0)

            adult.reset_mock()
            mkLay.reset_mock()
            # Break on 1st round
            adult.num_eggs = 0
            effects    = []
            egg_masses = []
            mkLay.side_effect = effects
```

```
            self.assertEqual(self.Lay.lay(adult), egg_masses)
            self.assertEqual(len(mkLay.call_args_list), 0)
            self.assertEqual(adult.population.call_args_list,
                             [mk.call()])

    def test_setup(self):
        """test setup the class"""

        # Test if have the models
        kwargs = {keyword.fecundity: self.fecundity,
                  keyword.density:   self.density}
        self.Lay = lay.Lay.setup(**kwargs)
        self.assertIsInstance(self.Lay, lay.Lay)
        self.assertEqual(self.Lay.fecundity, self.fecundity)
        self.assertEqual(self.Lay.density,   self.density)

        # Test if have no models
        kwargs = {}
        self.Lay = lay.Lay.setup(**kwargs)
        self.assertIsInstance(self.Lay, lay.Lay)
        self.assertEqual(self.Lay.fecundity, None)
        self.assertEqual(self.Lay.density,   None)
```

C.5.8.2 test_mate.py

```
import unittest      as ut
import unittest.mock as mk

import dataclasses  as dclass
import numpy.random as rnd

import source.keyword as keyword

import source.agents.adult as agent_adult

import source.reproduction.mate as mating


class AdultTest(agent_adult.Adult):
    """Class to add dynamic values for tests"""

    mass     = mk.MagicMock(spec=float)
    age      = mk.MagicMock(spec=int)
    genotype = mk.MagicMock(spec=str)
    num_eggs = mk.MagicMock(spec=int)


class TestMate(ut.TestCase):
    """test Mate behavior class"""

    def setUp(self):
        """Setup the tests"""

        self.mating = mk.MagicMock(spec=callable)
        self.radius = mk.MagicMock(spec=callable)

        self.Mate = mating.Mate(self.mating,
                                self.radius)

    def test___init__(self):
        """test __init__ for class"""

```

```
        self.assertIsInstance(self.Mate, mating.Mate)

        self.assertEqual(self.Mate.mating, self.mating)
        self.assertEqual(self.Mate.radius, self.radius)

        self.assertTrue(dclass.is_dataclass(self.Mate))

    def test__use_mating(self):
        """test if we use the mating system"""

        self.assertTrue(self.Mate._use_mating)

        self.Mate.mating = None
        self.assertFalse(self.Mate._use_mating)

    def test__use_radius(self):
        """test if we use the radius system"""

        self.assertTrue(self.Mate._use_radius)

        self.Mate.radius = None
        self.assertFalse(self.Mate._use_radius)

    def test__mate_with(self):
        """test mate with the other adult"""

        adult = mk.create_autospec(AdultTest, spec_set=True)
        mate  = mk.create_autospec(AdultTest, spec_set=True)

        self.Mate._mate_with(adult, mate)
        self.assertEqual(adult.set_mate.call_args_list,
                         [mk.call(mate)])
        self.assertEqual(mate.set_mate.call_args_list,
                         [mk.call(adult)])

    def test__bounds(self):
        """test get the bounds on the radius"""

        adult = mk.create_autospec(AdultTest, spec_set=True)
```

```
77
78          with mk.patch.object(mating.Mate, '_use_radius',
79                               autospec=True) as mkUse:
80              mkUse.__get__ = mk.MagicMock(side_effect=[False, True])
81
82              # Test if no radius model
83              self.assertEqual(self.Mate._bounds(adult),
84                               {keyword.upper: 0,
85                                keyword.lower: 0})
86              self.assertEqual(self.radius.call_args_list, [])
87
88              # Test if radius model
89              self.assertEqual(self.Mate._bounds(adult),
90                               {keyword.upper: self.radius.return_value,
91                                keyword.lower: 0})
92              self.assertEqual(self.radius.call_args_list,
93                               [mk.call(adult.mass, adult.genotype)])
94
95      def test__mates(self):
96          """test get the adults to mate"""
97
98          adult = mk.create_autospec(AdultTest, spec_set=True)
99
100         bounds = {'test': mk.MagicMock()}
101
102         with mk.patch.object(mating.Mate, '_bounds',
103                              autospec=True) as mkBounds:
104             mkBounds.return_value = bounds
105
106             self.assertEqual(self.Mate._mates(adult),
107                              adult.mates.return_value)
108             self.assertEqual(adult.mates.call_args_list,
109                              [mk.call(**bounds)])
110             self.assertEqual(mkBounds.call_args_list,
111                              [mk.call(self.Mate, adult)])
112
113     def test__encounter(self):
114         """test determine if we encounter a mate"""
115
```

```
116         adult = mk.create_autospec(AdultTest, spec_set=True)
117         mates = mk.MagicMock(spec=list)
118 
119         with mk.patch.object(mating, 'len') as mkLen:
120             self.assertEqual(self.Mate._encounter(adult, mates),
121                              self.mating.return_value)
122             self.assertEqual(self.mating.call_args_list,
123                              [mk.call(mkLen.return_value,
124                                       adult.mass, adult.genotype)])
125             self.assertEqual(mkLen.call_args_list,
126                              [mk.call(mates)])
127 
128     def test__perform(self):
129         """test perform a mating ritual"""
130 
131         adult = mk.create_autospec(AdultTest, spec_set=True)
132 
133         with mk.patch.object(mating.Mate, '_mates', autospec=True) as mkMates:
134             with mk.patch.object(mating.Mate, '_encounter',
135                                  autospec=True) as mkEncounter:
136                 with mk.patch.object(mating.Mate, '_mate_with',
137                                      autospec=True) as mkMate:
138                     with mk.patch.object(rnd, 'choice') as mkRND:
139                         mkEncounter.side_effect = [False, True]
140 
141                         # No encounter
142                         self.Mate._perform(adult)
143                         self.assertEqual(mkMate.call_args_list, [])
144                         self.assertEqual(mkRND.call_args_list, [])
145                         self.assertEqual(mkEncounter.call_args_list,
146                                          [mk.call(self.Mate,
147                                                   adult, mkMates.return_value)])
148                         self.assertEqual(mkMates.call_args_list,
149                                          [mk.call(self.Mate, adult)])
150 
151                         mkEncounter.reset_mock()
152                         mkMates.reset_mock()
153                         # Has encounter
154                         self.Mate._perform(adult)
```

```
                        self.assertEqual(mkMate.call_args_list,
                                         [mk.call(adult, mkRND.return_value)])
                        self.assertEqual(mkRND.call_args_list,
                                         [mk.call(mkMates.return_value)])
                        self.assertEqual(mkEncounter.call_args_list,
                                         [mk.call(self.Mate,
                                                  adult, mkMates.return_value)])
                        self.assertEqual(mkMates.call_args_list,
                                         [mk.call(self.Mate, adult)])

    def test_mate(self):
        """test run mate behavior"""

        adult = mk.create_autospec(AdultTest, spec_set=True)

        with mk.patch.object(mating.Mate, '_use_mating',
                             autospec=True) as mkUse:
            with mk.patch.object(mating.Mate, '_perform',
                                 autospec=True) as mkPerform:
                mkUse.__get__ = mk.MagicMock(side_effect=[False, True])

                # Test use is False
                self.Mate.mate(adult)
                self.assertEqual(mkPerform.call_args_list, [])

                # Test use is True
                self.Mate.mate(adult)
                self.assertEqual(mkPerform.call_args_list,
                                 [mk.call(self.Mate, adult)])

    def test_setup(self):
        """test setup the class"""

        # Test if have the models
        kwargs = {keyword.mating:      self.mating,
                  keyword.mate_radius: self.radius}
        self.Mate = mating.Mate.setup(**kwargs)
        self.assertIsInstance(self.Mate, mating.Mate)
        self.assertEqual(self.Mate.mating, self.mating)
```

```
        self.assertEqual(self.Mate.radius, self.radius)

        # Test if have no models
        kwargs = {}
        self.Mate = mating.Mate.setup(**kwargs)
        self.assertIsInstance(self.Mate, mating.Mate)
        self.assertEqual(self.Mate.mating, None)
        self.assertEqual(self.Mate.radius, None)
```

C.5.8.3 test_models.py

```
import unittest      as ut
import unittest.mock as mk

import dataclasses  as dclass
import numpy        as np
import numpy.random as rnd
import scipy.stats  as stats

import source.keyword as keyword

import source.reproduction.models as model

import source.simulation.models as models


class TestInitSex(ut.TestCase):
    """test InitSex mathematical model class"""

    def setUp(self):
        """Setup the tests"""

        self.prob = mk.MagicMock(spec=float)

        self.InitSex = model.InitSex(self.prob)

    def test___init__(self):
        """test __init__ for class"""

        self.assertIsInstance(self.InitSex, models.Model)
        self.assertIsInstance(self.InitSex, model.InitSex)

        self.assertEqual(self.InitSex.prob, self.prob)

        self.assertEqual(self.InitSex.model_key, keyword.init_sex)

        self.assertTrue(dclass.is_dataclass(self.InitSex))

```

```
38      def test___call__(self):
39          """test call the model"""
40
41          genotype = mk.MagicMock(spec=str)
42
43          with mk.patch.object(rnd, 'random') as mkRND:
44              mkRND.return_value.__le__.side_effect = [True, False]
45
46              # Test is true
47              self.assertTrue(self.InitSex(genotype))
48              self.assertEqual(mkRND.return_value.__le__.call_args_list,
49                               [mk.call(self.prob)])
50              self.assertEqual(mkRND.call_args_list,
51                               [mk.call()])
52
53              mkRND.reset_mock()
54              # Test is false
55              self.assertFalse(self.InitSex(genotype))
56              self.assertEqual(mkRND.return_value.__le__.call_args_list,
57                               [mk.call(self.prob)])
58              self.assertEqual(mkRND.call_args_list,
59                               [mk.call()])
60
61
62  class TestMating(ut.TestCase):
63      """test Mating mathematical model class"""
64
65      def setUp(self):
66          """Setup the tests"""
67
68          self.factor = mk.MagicMock(spec=float)
69
70          self.Mating = model.Mating(self.factor)
71
72      def test___init__(self):
73          """test __init__ for class"""
74
75          self.assertIsInstance(self.Mating, models.Model)
76          self.assertIsInstance(self.Mating, model.Mating)
```

```
77
78          self.assertEqual(self.Mating.factor, self.factor)
79
80          self.assertEqual(self.Mating.model_key, keyword.mating)
81
82          self.assertTrue(dclass.is_dataclass(self.Mating))
83
84      def test__prob(self):
85          """test get the probability"""
86
87          number = mk.MagicMock(spec=int)
88
89          with mk.patch.object(np, 'exp', autospec=True) as mkExp:
90              self.assertEqual(self.Mating._prob(number),
91                               mkExp.return_value.__rsub__.return_value)
92              self.assertEqual(mkExp.return_value.__rsub__.call_args_list,
93                               [mk.call(1)])
94              self.assertEqual(mkExp.call_args_list,
95                               [mk.call(self.factor.__neg__.return_value.
96                                        __mul__.return_value)])
97              self.assertEqual(self.factor.__neg__.return_value.
98                                  __mul__.call_args_list,
99                               [mk.call(number)])
100             self.assertEqual(self.factor.__neg__.call_args_list,
101                              [mk.call()])
102
103     def test___call__(self):
104         """test call the model"""
105
106         number   = mk.MagicMock(spec=int)
107         mass     = mk.MagicMock(spec=float)
108         genotype = mk.MagicMock(spec=str)
109
110         with mk.patch.object(model.Mating, '_prob', autospec=True) as mkProb:
111             with mk.patch.object(rnd, 'random') as mkRND:
112                 mkRND.return_value.__le__.side_effect = [True, False]
113
114                 # Test True
115                 self.assertTrue(self.Mating(number, mass, genotype))
```

```
                self.assertEqual(mkRND.return_value.__le__.call_args_list,
                                 [mk.call(mkProb.return_value)])
                self.assertEqual(mkRND.call_args_list,
                                 [mk.call()])
                self.assertEqual(mkProb.call_args_list,
                                 [mk.call(self.Mating, number)])

                mkRND.reset_mock()
                mkProb.reset_mock()
                # Test False
                self.assertFalse(self.Mating(number, mass, genotype))
                self.assertEqual(mkRND.return_value.__le__.call_args_list,
                                 [mk.call(mkProb.return_value)])
                self.assertEqual(mkRND.call_args_list,
                                 [mk.call()])
                self.assertEqual(mkProb.call_args_list,
                                 [mk.call(self.Mating, number)])


class TestRadius(ut.TestCase):
    """test Radius mathematical model"""

    def setUp(self):
        """Setup the tests"""

        self.radius = mk.MagicMock(spec=int)

        self.Radius = model.Radius(self.radius)

    def test___init__(self):
        """test __init__ for class"""

        self.assertIsInstance(self.Radius, models.Model)
        self.assertIsInstance(self.Radius, model.Radius)

        self.assertEqual(self.Radius.radius, self.radius)

        self.assertEqual(self.Radius.model_key, keyword.mate_radius)

```

```
155         self.assertTrue(dclass.is_dataclass(self.Radius))
156
157     def test___call__(self):
158         """test call the model"""
159
160         mass     = mk.MagicMock(spec=float)
161         genotype = mk.MagicMock(spec=str)
162
163         self.assertEqual(self.Radius(mass, genotype), self.radius)
164
165
166 class TestFecundity(ut.TestCase):
167     """test Fecundity mathematical model class"""
168
169     def setUp(self):
170         """Setup the tests"""
171
172         self.maximum = mk.MagicMock(spec=float)
173         self.decay   = mk.MagicMock(spec=float)
174
175         self.Fecundity = model.Fecundity(self.maximum,
176                                          self.decay)
177
178     def test___init__(self):
179         """test __init__ for class"""
180
181         self.assertIsInstance(self.Fecundity, models.Model)
182         self.assertIsInstance(self.Fecundity, model.Fecundity)
183
184         self.assertEqual(self.Fecundity.maximum, self.maximum)
185         self.assertEqual(self.Fecundity.decay,   self.decay)
186
187         self.assertEqual(self.Fecundity.model_key, keyword.fecundity)
188
189         self.assertTrue(dclass.is_dataclass(self.Fecundity))
190
191     def test__lam(self):
192         """test get mean of distribution"""
193
```

```
194         time = mk.MagicMock(spec=int)
195
196         with mk.patch.object(np, 'exp', autospec=True) as mkExp:
197             self.assertEqual(self.Fecundity._lam(time),
198                              self.maximum.__mul__.return_value.
199                                 __truediv__.return_value)
200             self.assertEqual(self.maximum.__mul__.return_value.
201                                 __truediv__.call_args_list,
202                              [mk.call(mkExp.return_value.__add__.return_value)])
203             self.assertEqual(self.maximum.__mul__.call_args_list,
204                              [mk.call(2)])
205             self.assertEqual(mkExp.return_value.__add__.call_args_list,
206                              [mk.call(1)])
207             self.assertEqual(mkExp.call_args_list,
208                              [mk.call(self.decay.__mul__.return_value)])
209             self.assertEqual(self.decay.__mul__.call_args_list,
210                              [mk.call(time)])
211
212     def test___call__(self):
213         """test call the model"""
214
215         age      = mk.MagicMock(spec=int)
216         mass     = mk.MagicMock(spec=float)
217         genotype = mk.MagicMock(spec=str)
218
219         with mk.patch.object(model.Fecundity, '_lam', autospec=True) as mkLam:
220             with mk.patch.object(stats.poisson, 'rvs', autospec=True) as mkRVS:
221                 with mk.patch.object(model, 'int') as mkInt:
222                     self.assertEqual(self.Fecundity(age, mass, genotype),
223                                      mkInt.return_value)
224                     self.assertEqual(mkInt.call_args_list,
225                                      [mk.call(mkRVS.return_value)])
226                     self.assertEqual(mkRVS.call_args_list,
227                                      [mk.call(mkLam.return_value)])
228                     self.assertEqual(mkLam.call_args_list,
229                                      [mk.call(self.Fecundity, age)])
230
231
232 class TestDensity(ut.TestCase):
```

```
    """test Density mathematical model class"""

    def setUp(self):
        """Setup the tests"""

        self.eta   = mk.MagicMock(spec=float)
        self.gamma = mk.MagicMock(spec=float)

        self.Density = model.Density(self.eta,
                                     self.gamma)

    def test___init__(self):
        """test __init__ for class"""

        self.assertIsInstance(self.Density, models.Model)
        self.assertIsInstance(self.Density, model.Density)

        self.assertEqual(self.Density.eta,   self.eta)
        self.assertEqual(self.Density.gamma, self.gamma)

        self.assertEqual(self.Density.model_key, keyword.density)

        self.assertTrue(dclass.is_dataclass(self.Density))

    def test__prob(self):
        """test get the probability"""

        number = mk.MagicMock(spec=int)

        with mk.patch.object(np, 'exp', autospec=True) as mkExp:
            self.assertEqual(self.Density._prob(number),
                             mkExp.return_value)
            self.assertEqual(mkExp.call_args_list,
                             [mk.call(number.__truediv__.return_value.
                                      __pow__.return_value.
                                      __neg__.return_value)])
            self.assertEqual(number.__truediv__.return_value.
                                __pow__.return_value.__neg__.call_args_list,
                             [mk.call()])
```

```
            self.assertEqual(number.__truediv__.return_value.
                                __pow__.call_args_list,
                             [mk.call(self.gamma)])
            self.assertEqual(number.__truediv__.call_args_list,
                             [mk.call(self.eta)])

    def test___call__(self):
        """test call the model"""

        number   = mk.MagicMock(spec=int)
        mass     = mk.MagicMock(spec=float)
        genotype = mk.MagicMock(spec=str)

        with mk.patch.object(model.Density, '_prob', autospec=True) as mkProb:
            with mk.patch.object(rnd, 'random') as mkRND:
                mkRND.return_value.__le__.side_effect = [True, False]

                # Test True
                self.assertTrue(self.Density(number, mass, genotype))
                self.assertEqual(mkRND.return_value.__le__.call_args_list,
                                 [mk.call(mkProb.return_value)])
                self.assertEqual(mkRND.call_args_list,
                                 [mk.call()])
                self.assertEqual(mkProb.call_args_list,
                                 [mk.call(self.Density, number)])

                mkRND.reset_mock()
                mkProb.reset_mock()
                # Test False
                self.assertFalse(self.Density(number, mass, genotype))
                self.assertEqual(mkRND.return_value.__le__.call_args_list,
                                 [mk.call(mkProb.return_value)])
                self.assertEqual(mkRND.call_args_list,
                                 [mk.call()])
                self.assertEqual(mkProb.call_args_list,
                                 [mk.call(self.Density, number)])
```

C.5.9 test_schedule

```
FallArmyworm
└── test
    └── test_schedule
        ├── test_actions.py
        ├── test_schedule.py
        └── test_step.py
```

C.5.9.1 test_actions.py

```
import unittest      as ut
import unittest.mock as mk

import collections as collect
import dataclasses as dclass

import source.agents.agent as main_agent

import source.schedule.actions as actions


class TestAction(ut.TestCase):
    """test the Action class"""

    def setUp(self):
        """Setup the tests"""

        self.action = mk.MagicMock(spec=str)

        self.Action = actions.Action(self.action)

    def test___init__(self):
        """test __init__ for class"""

        self.assertIsInstance(self.Action, actions.Action)

        self.assertEqual(self.Action.action, self.action)

        self.assertTrue(dclass.is_dataclass(self.Action))

    def test_perform(self):
        """test have agent perform action"""

        agent = mk.create_autospec(main_agent.Agent, spec_set=True)

        with mk.patch.object(actions, 'getattr') as mkGet:
            self.assertEqual(self.Action.perform(agent),
```

```
                                mkGet.return_value.return_value)
            self.assertEqual(mkGet.return_value.call_args_list,
                                [mk.call()])
            self.assertEqual(mkGet.call_args_list,
                                [mk.call(agent, self.action)])


class TestActions(ut.TestCase):
    """test the Actions class"""

    def setUp(self):
        """Setup the tests"""

        self.actions   = [mk.create_autospec(actions.Action, spec_set=True)
                          for _ in range(3)]
        self.agent_key = mk.MagicMock(spec=str)

        self.Actions = actions.Actions(self.actions,
                                       self.agent_key)

    def test___init__(self):
        """test __init__ for class"""

        self.assertIsInstance(self.Actions, collect.UserList)
        self.assertIsInstance(self.Actions, actions.Actions)

        self.assertEqual(self.Actions.agent_key, self.agent_key)

        self.assertEqual(self.Actions,      self.actions)
        self.assertEqual(self.Actions.data, self.actions)

    def test_perform(self):
        """test perform actions on agent"""

        agent = mk.create_autospec(main_agent.Agent, spec_set=True)

        results = []
        for action in self.actions:
            result = [mk.create_autospec(main_agent.Agent, spec_set=True)
```

```
                    for _ in range(3)]
            action.perform.return_value = result
            results.extend(result)

        self.assertEqual(self.Actions.perform(agent),
                         results)
        for action in self.actions:
            self.assertEqual(action.perform.call_args_list,
                             [mk.call(agent)])

    def test_setup(self):
        """test setup the agent"""

        action_keys = [mk.MagicMock(spec=str) for _ in range(3)]

        self.Actions = actions.Actions.setup(self.agent_key, action_keys)
        self.assertIsInstance(self.Actions, actions.Actions)
        self.assertEqual(self.Actions.agent_key, self.agent_key)

        for index, action in enumerate(self.Actions):
            self.assertIsInstance(action, actions.Action)
            self.assertEqual(action.action, action_keys[index])
        for index, action in enumerate(action_keys):
            self.assertIsInstance(self.Actions[index], actions.Action)
            self.assertEqual(self.Actions[index].action, action)
        self.assertEqual(len(self.Actions), 3)
```

C.5.9.2 test_schedule.py

```
import unittest      as ut
import unittest.mock as mk

import collections as collect

import source.agents.agent as main_agent

import source.schedule.actions  as agent_actions
import source.schedule.schedule as schedule
import source.schedule.step     as agent_step

import source.space.agents   as main_agents
import source.space.space    as agent_space


class TestSchedule(ut.TestCase):
    """test the Schedule class"""

    def setUp(self):
        """Setup the tests"""

        self.steps = [mk.create_autospec(agent_step.Step, spec_set=True)
                      for _ in range(3)]

        self.Schedule = schedule.Schedule(self.steps)

    def test___init__(self):
        """test __init__ for class"""

        self.assertIsInstance(self.Schedule, collect.UserList)
        self.assertIsInstance(self.Schedule, schedule.Schedule)

        self.assertEqual(self.Schedule,      self.steps)
        self.assertEqual(self.Schedule.data, self.steps)

    def test__activate(self):
        """test activate the results"""
```

```
38
39          results = [mk.create_autospec(main_agent.Agent, spec_set=True)
40                     for _ in range(3)]
41
42          self.Schedule._activate(results)
43          for result in results:
44              self.assertEqual(result.activate.call_args_list,
45                               [mk.call()])
46
47      def test__perform(self):
48          """test perform the schedule and get results"""
49
50          space  = mk.create_autospec(agent_space.Space, spec_set=True)
51          agents = mk.create_autospec(main_agents.Agents, spec_set=True)
52
53          results = []
54          for step in self.steps:
55              result = [mk.MagicMock() for _ in range(3)]
56              step.perform.return_value = result
57              results.extend(result)
58
59          self.assertEqual(self.Schedule._perform(space, agents), results)
60
61          for step in self.steps:
62              self.assertEqual(step.perform.call_args_list,
63                               [mk.call(space, agents)])
64          self.assertEqual(len(self.steps), 3)
65
66      def test_perform(self):
67          """test perform the schedule"""
68
69          space  = mk.create_autospec(agent_space.Space, spec_set=True)
70          agents = mk.create_autospec(main_agents.Agents, spec_set=True)
71
72          with mk.patch.object(schedule.Schedule, '_perform',
73                               autospec=True) as mkPerform:
74              with mk.patch.object(schedule.Schedule, '_activate',
75                                   autospec=True) as mkActivate:
76                  self.Schedule.perform(space, agents)
```

```
                self.assertEqual(mkActivate.call_args_list,
                                 [mk.call(mkPerform.return_value)])
                self.assertEqual(mkPerform.call_args_list,
                                 [mk.call(self.Schedule, space, agents)])

    def test_setup(self):
        """test setup an entire schedule"""

        basic_actions = {}
        for _ in range(3):
            agent_key   = mk.MagicMock(spec=str)
            action_keys = [mk.MagicMock(spec=str) for _ in range(3)]

            basic_actions[agent_key] = action_keys
        tuple_basic = (basic_actions,)

        repeat_actions = {}
        for _ in range(3):
            agent_key   = mk.MagicMock(spec=str)
            action_keys = [mk.MagicMock(spec=str) for _ in range(3)]

            repeat_actions[agent_key] = action_keys
        tuple_repeat = (repeat_actions, 10)

        shuffle_actions_0 = {}
        for _ in range(3):
            agent_key   = mk.MagicMock(spec=str)
            action_keys = [mk.MagicMock(spec=str) for _ in range(3)]

            shuffle_actions_0[agent_key] = action_keys
        tuple_shuffle_0 = (shuffle_actions_0, 10, True)

        shuffle_actions_1 = {}
        for _ in range(3):
            agent_key   = mk.MagicMock(spec=str)
            action_keys = [mk.MagicMock(spec=str) for _ in range(3)]

            shuffle_actions_1[agent_key] = action_keys
        tuple_shuffle_1 = (shuffle_actions_1, 10, True, True)
```

```
        reg_actions = {}
        for _ in range(3):
            agent_key   = mk.MagicMock(spec=str)
            action_keys = [mk.MagicMock(spec=str) for _ in range(3)]

            reg_actions[agent_key] = action_keys
        tuple_reg = (reg_actions, 10, True, True, True)

        loc_actions = {}
        for _ in range(3):
            agent_key   = mk.MagicMock(spec=str)
            action_keys = [mk.MagicMock(spec=str) for _ in range(3)]

            loc_actions[agent_key] = action_keys
        tuple_loc = (loc_actions, 10, True, True, False, True, 1)

        step_tuples = [tuple_basic,
                       tuple_repeat,
                       tuple_shuffle_0,
                       tuple_shuffle_1,
                       tuple_reg,
                       tuple_loc]

        self.Schedule = schedule.Schedule.setup(step_tuples)
        self.assertIsInstance(self.Schedule, schedule.Schedule)

        self.assertEqual(len(self.Schedule), 6)

        # basic tuple
        self.assertIsInstance(self.Schedule[0], agent_step.Step)
        self.assertEqual(self.Schedule[0].number,          1)
        self.assertEqual(self.Schedule[0].shuffle_agents,  False)
        self.assertEqual(self.Schedule[0].shuffle_actions, False)
        self.assertEqual(self.Schedule[0].parallel_reg,    False)
        self.assertEqual(self.Schedule[0].parallel_loc,    False)
        self.assertEqual(self.Schedule[0].level,           0)
        for index_i, thing in enumerate(basic_actions.items()):
            agent_key, action_keys = thing
```

```
            self.assertIsInstance(self.Schedule[0][index_i],
                                  agent_actions.Actions)
            self.assertEqual(self.Schedule[0][index_i].agent_key, agent_key)
            for index_j, action in enumerate(action_keys):
                self.assertIsInstance(self.Schedule[0][index_i][index_j],
                                      agent_actions.Action)
                self.assertEqual(self.Schedule[0][index_i][index_j].action,
                                 action)
            for index_j, action in enumerate(self.Schedule[0][index_i]):
                self.assertIsInstance(action, agent_actions.Action)
                self.assertEqual(action.action, action_keys[index_j])
            self.assertEqual(len(self.Schedule[0][index_i]), 3)
        self.assertEqual(len(self.Schedule[0]), 3)

        # repeat tuple
        self.assertIsInstance(self.Schedule[1], agent_step.Step)
        self.assertEqual(self.Schedule[1].number,          10)
        self.assertEqual(self.Schedule[1].shuffle_agents,  False)
        self.assertEqual(self.Schedule[1].shuffle_actions, False)
        self.assertEqual(self.Schedule[1].parallel_reg,    False)
        self.assertEqual(self.Schedule[1].parallel_loc,    False)
        self.assertEqual(self.Schedule[1].level,           0)
        for index_i, thing in enumerate(repeat_actions.items()):
            agent_key, action_keys = thing
            self.assertIsInstance(self.Schedule[1][index_i],
                                  agent_actions.Actions)
            self.assertEqual(self.Schedule[1][index_i].agent_key, agent_key)
            for index_j, action in enumerate(action_keys):
                self.assertIsInstance(self.Schedule[1][index_i][index_j],
                                      agent_actions.Action)
                self.assertEqual(self.Schedule[1][index_i][index_j].action,
                                 action)
            for index_j, action in enumerate(self.Schedule[1][index_i]):
                self.assertIsInstance(action, agent_actions.Action)
                self.assertEqual(action.action, action_keys[index_j])
            self.assertEqual(len(self.Schedule[1][index_i]), 3)
        self.assertEqual(len(self.Schedule[1]), 3)

        # shuffle tuple 0
```

```
        self.assertIsInstance(self.Schedule[2], agent_step.Step)
        self.assertEqual(self.Schedule[2].number,          10)
        self.assertEqual(self.Schedule[2].shuffle_agents,  True)
        self.assertEqual(self.Schedule[2].shuffle_actions, False)
        self.assertEqual(self.Schedule[2].parallel_reg,    False)
        self.assertEqual(self.Schedule[2].parallel_loc,    False)
        self.assertEqual(self.Schedule[2].level,           0)
        for index_i, thing in enumerate(shuffle_actions_0.items()):
            agent_key, action_keys = thing
            self.assertIsInstance(self.Schedule[2][index_i],
                                  agent_actions.Actions)
            self.assertEqual(self.Schedule[2][index_i].agent_key, agent_key)
            for index_j, action in enumerate(action_keys):
                self.assertIsInstance(self.Schedule[2][index_i][index_j],
                                      agent_actions.Action)
                self.assertEqual(self.Schedule[2][index_i][index_j].action,
                                 action)
            for index_j, action in enumerate(self.Schedule[2][index_i]):
                self.assertIsInstance(action, agent_actions.Action)
                self.assertEqual(action.action, action_keys[index_j])
            self.assertEqual(len(self.Schedule[2][index_i]), 3)
        self.assertEqual(len(self.Schedule[2]), 3)

        # shuffle tuple 1
        self.assertIsInstance(self.Schedule[3], agent_step.Step)
        self.assertEqual(self.Schedule[3].number,          10)
        self.assertEqual(self.Schedule[3].shuffle_agents,  True)
        self.assertEqual(self.Schedule[3].shuffle_actions, True)
        self.assertEqual(self.Schedule[3].parallel_reg,    False)
        self.assertEqual(self.Schedule[3].parallel_loc,    False)
        self.assertEqual(self.Schedule[3].level,           0)
        for index_i, thing in enumerate(shuffle_actions_1.items()):
            agent_key, action_keys = thing
            self.assertIsInstance(self.Schedule[3][index_i],
                                  agent_actions.Actions)
            self.assertEqual(self.Schedule[3][index_i].agent_key, agent_key)
            for index_j, action in enumerate(action_keys):
                self.assertIsInstance(self.Schedule[3][index_i][index_j],
                                      agent_actions.Action)
```

```
                self.assertEqual(self.Schedule[3][index_i][index_j].action,
                                 action)
            for index_j, action in enumerate(self.Schedule[3][index_i]):
                self.assertIsInstance(action, agent_actions.Action)
                self.assertEqual(action.action, action_keys[index_j])
            self.assertEqual(len(self.Schedule[3][index_i]), 3)
        self.assertEqual(len(self.Schedule[3]), 3)

        # reg tuple
        self.assertIsInstance(self.Schedule[4], agent_step.Step)
        self.assertEqual(self.Schedule[4].number,          10)
        self.assertEqual(self.Schedule[4].shuffle_agents,  True)
        self.assertEqual(self.Schedule[4].shuffle_actions, True)
        self.assertEqual(self.Schedule[4].parallel_reg,    True)
        self.assertEqual(self.Schedule[4].parallel_loc,    False)
        self.assertEqual(self.Schedule[4].level,           0)
        for index_i, thing in enumerate(reg_actions.items()):
            agent_key, action_keys = thing
            self.assertIsInstance(self.Schedule[4][index_i],
                                  agent_actions.Actions)
            self.assertEqual(self.Schedule[4][index_i].agent_key, agent_key)
            for index_j, action in enumerate(action_keys):
                self.assertIsInstance(self.Schedule[4][index_i][index_j],
                                      agent_actions.Action)
                self.assertEqual(self.Schedule[4][index_i][index_j].action,
                                 action)
            for index_j, action in enumerate(self.Schedule[4][index_i]):
                self.assertIsInstance(action, agent_actions.Action)
                self.assertEqual(action.action, action_keys[index_j])
            self.assertEqual(len(self.Schedule[4][index_i]), 3)
        self.assertEqual(len(self.Schedule[4]), 3)

        # loc tuple
        self.assertIsInstance(self.Schedule[5], agent_step.Step)
        self.assertEqual(self.Schedule[5].number,          10)
        self.assertEqual(self.Schedule[5].shuffle_agents,  True)
        self.assertEqual(self.Schedule[5].shuffle_actions, True)
        self.assertEqual(self.Schedule[5].parallel_reg,    False)
        self.assertEqual(self.Schedule[5].parallel_loc,    True)
```

```
        self.assertEqual(self.Schedule[5].level,           1)
        for index_i, thing in enumerate(loc_actions.items()):
            agent_key, action_keys = thing
            self.assertIsInstance(self.Schedule[5][index_i],
                                  agent_actions.Actions)
            self.assertEqual(self.Schedule[5][index_i].agent_key, agent_key)
            for index_j, action in enumerate(action_keys):
                self.assertIsInstance(self.Schedule[5][index_i][index_j],
                                      agent_actions.Action)
                self.assertEqual(self.Schedule[5][index_i][index_j].action,
                                 action)
            for index_j, action in enumerate(self.Schedule[5][index_i]):
                self.assertIsInstance(action, agent_actions.Action)
                self.assertEqual(action.action, action_keys[index_j])
            self.assertEqual(len(self.Schedule[5][index_i]), 3)
        self.assertEqual(len(self.Schedule[5]), 3)
```

C.5.9.3 test_step.py

```
import unittest      as ut
import unittest.mock as mk

import collections  as collect
import itertools    as i_tools
import numpy.random as rnd

import source.agents.agent as main_agent

import source.schedule.actions as agent_actions
import source.schedule.step    as step

import source.space.agents   as main_agents
import source.space.location as location
import source.space.space    as agent_space


step.num_cpu = 4


class AgentBinTest(main_agents.AgentBin):
    """Class to contain the agent bin"""

    data = mk.MagicMock(spec=list)


class ActionsTest(agent_actions.Actions):
    """Class to add dynamic values for tests"""

    agent_key = mk.MagicMock(spec=str)


class SpaceTest(agent_space.Space):
    """Class to add dynamic values for tests"""

    locations     = mk.MagicMock(spec=list)
    location_keys = mk.MagicMock(spec=dict)
```

```
38
39
40 class AgentParallel(main_agent.Agent):
41     """Class to contain a basic agent for parallel"""
42
43     def __init__(self, agent_key, unique_id, loc):
44         super().__init__(agent_key, unique_id, None, loc, True)
45
46         self.results0 = [mk.MagicMock() for _ in range(3)]
47         self.results1 = [mk.MagicMock() for _ in range(3)]
48         self.results2 = [mk.MagicMock() for _ in range(3)]
49
50     def test0(self) -> list:
51         """Test action to perform"""
52
53         return self.results0
54
55     def test1(self) -> list:
56         """Test action to perform"""
57
58         return self.results1
59
60     def test2(self) -> list:
61         """Test action to perform"""
62
63         return self.results2
64
65
66 class TestStep(ut.TestCase):
67     """test Step class"""
68
69     def setUp(self):
70         """Setup the tests"""
71
72         self.actions = [mk.create_autospec(ActionsTest, spec_set=True)
73                         for _ in range(3)]
74
75         self.number          = mk.MagicMock(spec=int)
76         self.shuffle_agents  = mk.MagicMock(spec=bool)
```

```
        self.shuffle_actions = mk.MagicMock(spec=bool)
        self.parallel_reg    = mk.MagicMock(spec=bool)
        self.parallel_loc    = mk.MagicMock(spec=bool)
        self.level           = mk.MagicMock(spec=int)

        self.Step = step.Step(self.actions,
                              self.number,
                              self.shuffle_agents,
                              self.shuffle_actions,
                              self.parallel_reg,
                              self.parallel_loc,
                              self.level)

    def test___init__(self):
        """test __init__ for class"""

        self.assertIsInstance(self.Step, collect.UserList)
        self.assertIsInstance(self.Step, step.Step)

        self.assertEqual(self.Step.number,          self.number)
        self.assertEqual(self.Step.shuffle_agents,  self.shuffle_agents)
        self.assertEqual(self.Step.shuffle_actions, self.shuffle_actions)
        self.assertEqual(self.Step.parallel_reg,    self.parallel_reg)
        self.assertEqual(self.Step.parallel_loc,    self.parallel_loc)
        self.assertEqual(self.Step.level,           self.level)

        self.assertEqual(self.Step,      self.actions)
        self.assertEqual(self.Step.data, self.actions)

    def test__perform_agent_action_regular(self):
        """test perform an action in regular state"""

        action = mk.create_autospec(ActionsTest, spec_set=True)
        agents = [mk.create_autospec(main_agent.Agent, spec_set=True)
                  for _ in range(3)]

        results = []
        effects = []
        for _ in agents:
```

```
            result = [mk.MagicMock() for _ in range(3)]
            effects.append(result)
            results.extend(result)
        action.perform.side_effect = effects

        self.assertEqual(self.Step._perform_agent_action_regular(action,
                                                                 agents),
                         results)
        for index, call in enumerate(action.perform.call_args_list):
            self.assertEqual(call,
                             mk.call(agents[index]))
        self.assertEqual(len(action.perform.call_args_list), 3)

    def test__perform_agent_action_parallel(self):
        """test perform an action in parallel state"""

        loc = mk.create_autospec(location.Location, spec_set=True)

        agent = AgentParallel('test', 0, loc)

        action0 = agent_actions.Action('test0')
        self.assertEqual(action0.perform(agent), agent.results0)
        action1 = agent_actions.Action('test1')
        self.assertEqual(action1.perform(agent), agent.results1)
        action2 = agent_actions.Action('test2')
        self.assertEqual(action2.perform(agent), agent.results2)

        results_i_tools = list(i_tools.chain.from_iterable([agent.results0,
                                                            agent.results1,
                                                            agent.results2]))
        results  = []
        results += agent.results0
        results += agent.results1
        results += agent.results2
        self.assertEqual(results, results_i_tools)

        actions = agent_actions.Actions([action0, action1, action2], 'test')
        self.assertEqual(actions.perform(agent), results)

```

```
        agents = [AgentParallel('test', index, loc) for index in range(40)]
        regular_results  = self.Step._perform_agent_action_regular( actions,
                                                                    agents)
        self.assertEqual(len(set(regular_results)), 40 * 9)
        parallel_results = self.Step._perform_agent_action_parallel(actions,
                                                                    agents)
        self.assertEqual(set(regular_results), set(parallel_results))

    def test__perform_agent_action(self):
        """test perform action on agent_bin"""

        action     = mk.create_autospec(ActionsTest, spec_set=True)
        agents_bin = mk.create_autospec(main_agents.AgentsBin, spec_set=True)
        agent_bin  = mk.create_autospec(AgentBinTest, spec_set=True)
        agents     = [mk.create_autospec(main_agent.Agent, spec_set=True)
                      for _ in range(3)]

        agents_bin.__getitem__.return_value = agent_bin
        agent_bin.agents                    = mk.MagicMock(spec=list)
        agent_bin.agents.copy.return_value = agents

        with mk.patch.object(step.Step, '_perform_agent_action_parallel',
                             autospec=True) as mkParallel:
            with mk.patch.object(step.Step,
                                 '_perform_agent_action_regular') as mkRegular:
                with mk.patch.object(rnd, 'shuffle') as mkRnd:
                    # Test run parallel
                    #      Test with shuffle
                    self.Step.shuffle_agents = True
                    self.Step.parallel_reg   = True
                    self.assertEqual(self.Step.
                                     _perform_agent_action(action,
                                                           agents_bin),
                                     mkParallel.return_value)
                    self.assertEqual(mkParallel.call_args_list,
                                     [mk.call(self.Step, action, agents)])
                    self.assertEqual(mkRegular.call_args_list, [])
                    self.assertEqual(agents_bin.__getitem__.call_args_list,
                                     [mk.call(action.agent_key)])
```

```
                    self.assertEqual(agent_bin.agents.copy.call_args_list,
                                     [mk.call()])
                    self.assertEqual(mkRnd.call_args_list,
                                     [mk.call(agents)])

                    mkRnd.reset_mock()
                    mkParallel.reset_mock()
                    agents_bin.reset_mock()
                    agent_bin.agents.reset_mock()
                    #       Test without shuffle
                    self.Step.shuffle_agents = False
                    self.Step.parallel_reg   = True
                    self.assertEqual(self.Step.
                                        _perform_agent_action(action,
                                                              agents_bin),
                                     mkParallel.return_value)
                    self.assertEqual(mkParallel.call_args_list,
                                     [mk.call(self.Step, action, agents)])
                    self.assertEqual(mkRegular.call_args_list, [])
                    self.assertEqual(agents_bin.__getitem__.call_args_list,
                                     [mk.call(action.agent_key)])
                    self.assertEqual(agent_bin.agents.copy.call_args_list,
                                     [mk.call()])
                    self.assertEqual(mkRnd.call_args_list, [])

                    mkParallel.reset_mock()
                    agent_bin.reset_mock()
                    agents_bin.reset_mock()
                    agent_bin.agents.reset_mock()
                    # Test run regular
                    #       Test with shuffle
                    self.Step.shuffle_agents = True
                    self.Step.parallel_reg   = False
                    self.assertEqual(self.Step.
                                        _perform_agent_action(action,
                                                              agents_bin),
                                     mkRegular.return_value)
                    self.assertEqual(mkRegular.call_args_list,
                                     [mk.call(action, agents)])
```

```
233                     self.assertEqual(mkParallel.call_args_list, [])
234                     self.assertEqual(agents_bin.__getitem__.call_args_list,
235                                      [mk.call(action.agent_key)])
236                     self.assertEqual(agent_bin.agents.copy.call_args_list,
237                                      [mk.call()])
238                     self.assertEqual(mkRnd.call_args_list,
239                                      [mk.call(agents)])
240 
241                     mkRnd.reset_mock()
242                     mkRegular.reset_mock()
243                     agent_bin.reset_mock()
244                     agents_bin.reset_mock()
245                     agent_bin.agents.reset_mock()
246                     #      Test with shuffle
247                     self.Step.shuffle_agents = False
248                     self.Step.parallel_reg   = False
249                     self.assertEqual(self.Step.
250                                         _perform_agent_action(action,
251                                                               agents_bin),
252                                      mkRegular.return_value)
253                     self.assertEqual(mkRegular.call_args_list,
254                                      [mk.call(action, agents)])
255                     self.assertEqual(mkParallel.call_args_list, [])
256                     self.assertEqual(agents_bin.__getitem__.call_args_list,
257                                      [mk.call(action.agent_key)])
258                     self.assertEqual(agent_bin.agents.copy.call_args_list,
259                                      [mk.call()])
260                     self.assertEqual(mkRnd.call_args_list, [])
261 
262     def test__perform_actions_step(self):
263         """test perform actions at each location"""
264 
265         location_key = mk.MagicMock(spec=str)
266 
267         agents    = mk.create_autospec(main_agents.Agents, spec_set=True)
268         agent_bin = mk.create_autospec(main_agents.AgentsBin, spec_set=True)
269         agents.__getitem__.return_value = agent_bin
270 
271         results = []
```

```
        effects = []
        for _ in self.actions:
            result = [mk.MagicMock() for _ in range(3)]
            effects.append(result)
            results.extend(result)

        with mk.patch.object(step.Step, '_perform_agent_action',
                             autospec=True) as mkPerform:
            mkPerform.side_effect = effects

            self.assertEqual(self.Step._perform_actions_step(location_key,
                                                             agents),
                             results)

            for index, call in enumerate(mkPerform.call_args_list):
                self.assertEqual(call,
                                 mk.call(self.Step,
                                         self.actions[index], agent_bin))
            for index, action in enumerate(self.actions):
                self.assertEqual(mkPerform.call_args_list[index],
                                 mk.call(self.Step, action, agent_bin))
            self.assertEqual(len(mkPerform.call_args_list), 3)

            self.assertEqual(agents.__getitem__.call_args_list,
                             [mk.call(location_key)])

    def test__perform_regular_step(self):
        """test perform a regular step"""

        location_keys = [mk.MagicMock(spec=str) for _ in range(3)]
        agents        = mk.create_autospec(main_agents.Agents, spec_set=True)

        results = []
        effects = []
        for _ in self.actions:
            result = [mk.MagicMock() for _ in range(3)]
            effects.append(result)
            results.extend(result)

```

```
311         with mk.patch.object(step.Step, '_perform_actions_step',
312                              autospec=True) as mkPerform:
313             mkPerform.side_effect = effects
314 
315             self.assertEqual(self.Step._perform_regular_step(location_keys,
316                                                              agents),
317                              results)
318 
319             for index, call in enumerate(mkPerform.call_args_list):
320                 self.assertEqual(call,
321                                  mk.call(self.Step,
322                                          location_keys[index], agents))
323             for index, location_key in enumerate(location_keys):
324                 self.assertEqual(mkPerform.call_args_list[index],
325                                  mk.call(self.Step,
326                                          location_key, agents))
327             self.assertEqual(len(mkPerform.call_args_list), 3)
328 
329     def test__perform_parallel_step(self):
330         """test perform a parallel step"""
331 
332         space = mk.create_autospec(SpaceTest, spec_set=True)
333 
334         locations     = []
335         location_keys = []
336         for index in range(40):
337             loc = location.Location([0, index])
338             locations.    append(loc)
339             location_keys.append(loc.location_key)
340 
341         locs = locations.copy()
342         locs.append(location.Location([0]))
343         space.locations = locs
344 
345         agent_keys  = ['test0', 'test1', 'test2']
346         environment = (0, mk.MagicMock())
347         agents      = main_agents.Agents.empty(space, agent_keys,
348                                                {}, environment)
349 
```

```
        unique_id = 0
        for loc in locations:
            for agent_key in agent_keys:
                for _ in range(3):
                    agent = AgentParallel(agent_key, unique_id, loc)
                    agents.activate(agent)
                    unique_id += 1
        for agent_key in agent_keys:
            self.assertEqual(len(agents.agents(agent_key)), 120)
            for location_key in location_keys:
                self.assertEqual(len(agents[location_key][agent_key]), 3)

        action_keys = ['test0', 'test1', 'test2']
        actions = []
        for agent_key in agent_keys:
            action = agent_actions.Actions.setup(agent_key, action_keys)
            actions.append(action)

        self.Step = step.Step(actions)

        regular_results = self.Step._perform_regular_step(location_keys,
                                                          agents)
        self.assertEqual(len(regular_results), 40 * 3 * 3 * 9)
        parallel_results = self.Step._perform_parallel_step(location_keys,
                                                            agents)
        self.assertEqual(len(regular_results), len(parallel_results))
        set_regular = set(regular_results)
        self.assertEqual(len(regular_results), len(set_regular))
        set_parallel = set(parallel_results)
        self.assertEqual(len(parallel_results), len(set_parallel))
        self.assertEqual(set_regular, set_parallel)

    def test__perform_step(self):
        """test perform a single set of actions"""

        space         = mk.create_autospec(SpaceTest, spec_set=True)
        agents        = mk.create_autospec(main_agents.Agents, spec_set=True)
        location_keys = [mk.MagicMock(spec=str) for _ in range(3)]

```

```
        space.location_keys.__getitem__.return_value = location_keys

        with mk.patch.object(step.Step, '_perform_parallel_step',
                             autospec=True) as mkParallel:
            with mk.patch.object(step.Step, '_perform_regular_step',
                                 autospec=True) as mkRegular:
                with mk.patch.object(rnd, 'shuffle') as mkRnd:
                    # Parallel No shuffle
                    self.Step.shuffle_actions = False
                    self.Step.parallel_loc    = True
                    self.assertEqual(self.Step._perform_step(space, agents),
                                     mkParallel.return_value)
                    self.assertEqual(mkParallel.call_args_list,
                                     [mk.call(self.Step,
                                              location_keys, agents)])
                    self.assertEqual(mkRegular.call_args_list, [])
                    self.assertEqual(space.location_keys.
                                        __getitem__.call_args_list,
                                     [mk.call(self.level)])
                    self.assertEqual(mkRnd.call_args_list, [])

                    mkParallel.reset_mock()
                    space.location_keys.__getitem__.reset_mock()
                    # Parallel With  shuffle
                    self.Step.shuffle_actions = True
                    self.Step.parallel_loc    = True
                    self.assertEqual(self.Step._perform_step(space, agents),
                                     mkParallel.return_value)
                    self.assertEqual(mkParallel.call_args_list,
                                     [mk.call(self.Step,
                                              location_keys, agents)])
                    self.assertEqual(mkRegular.call_args_list, [])
                    self.assertEqual(space.location_keys.
                                     __getitem__.call_args_list,
                                     [mk.call(self.level)])
                    self.assertEqual(mkRnd.call_args_list,
                                     [mk.call(self.Step)])

                    mkParallel.reset_mock()
```

```
                    space.location_keys.__getitem__.reset_mock()
                    mkRnd.reset_mock()
                    # No Parallel No shuffle
                    self.Step.shuffle_actions = False
                    self.Step.parallel_loc    = False
                    self.assertEqual(self.Step._perform_step(space, agents),
                                     mkRegular.return_value)
                    self.assertEqual(mkRegular.call_args_list,
                                     [mk.call(self.Step,
                                              location_keys, agents)])
                    self.assertEqual(mkParallel.call_args_list, [])
                    self.assertEqual(space.location_keys.
                                     __getitem__.call_args_list,
                                     [mk.call(self.level)])
                    self.assertEqual(mkRnd.call_args_list, [])

                    mkRegular.reset_mock()
                    space.location_keys.__getitem__.reset_mock()
                    # No Parallel With  shuffle
                    self.Step.shuffle_actions = True
                    self.Step.parallel_loc    = False
                    self.assertEqual(self.Step._perform_step(space, agents),
                                     mkRegular.return_value)
                    self.assertEqual(mkRegular.call_args_list,
                                     [mk.call(self.Step,
                                              location_keys, agents)])
                    self.assertEqual(mkParallel.call_args_list, [])
                    self.assertEqual(space.location_keys.
                                     __getitem__.call_args_list,
                                     [mk.call(self.level)])
                    self.assertEqual(mkRnd.call_args_list,
                                     [mk.call(self.Step)])

    def test_perform(self):
        """test perform the step"""

        space  = mk.create_autospec(SpaceTest, spec_set=True)
        agents = mk.create_autospec(main_agents.Agents, spec_set=True)

```

```
        results = []
        effects = []
        for _ in self.actions:
            result = [mk.MagicMock() for _ in range(3)]
            effects.append(result)
            results.extend(result)

        self.Step.number = 3
        with mk.patch.object(step.Step, '_perform_step',
                             autospec=True) as mkPerform:
            mkPerform.side_effect = effects

            self.assertEqual(self.Step.perform(space,
                                               agents),
                             results)

            for index, call in enumerate(mkPerform.call_args_list):
                self.assertEqual(call,
                                 mk.call(self.Step,
                                         space, agents))
            self.assertEqual(len(mkPerform.call_args_list), 3)

    def test_setup(self):
        """test setup the class"""

        actions = {}
        for _ in range(3):
            agent_key   = mk.MagicMock(spec=str)
            action_keys = [mk.MagicMock(spec=str) for _ in range(3)]

            actions[agent_key] = action_keys

        # Test default
        self.Step = step.Step.setup(actions)
        self.assertIsInstance(self.Step, step.Step)
        self.assertEqual(self.Step.number,          1)
        self.assertEqual(self.Step.shuffle_agents,  False)
        self.assertEqual(self.Step.shuffle_actions, False)
        self.assertEqual(self.Step.parallel_reg,    False)
```

```
506         self.assertEqual(self.Step.parallel_loc,    False)
507         self.assertEqual(self.Step.level,           0)
508 
509         for index_i, thing in enumerate(actions.items()):
510             agent_key, action_keys = thing
511             self.assertIsInstance(self.Step[index_i], agent_actions.Actions)
512             self.assertEqual(self.Step[index_i].agent_key, agent_key)
513 
514             for index_j, action in enumerate(action_keys):
515                 self.assertIsInstance(self.Step[index_i][index_j],
516                                       agent_actions.Action)
517                 self.assertEqual(self.Step[index_i][index_j].action, action)
518             for index_j, action in enumerate(self.Step[index_i]):
519                 self.assertIsInstance(action, agent_actions.Action)
520                 self.assertEqual(action.action, action_keys[index_j])
521             self.assertEqual(len(self.Step[index_i]), 3)
522         self.assertEqual(len(self.Step), 3)
523 
524         # Test Not Default
525         self.Step = step.Step.setup(actions,
526                                     self.number,
527                                     self.shuffle_agents,
528                                     self.shuffle_actions,
529                                     self.parallel_reg,
530                                     False,
531                                     self.level)
532         self.assertIsInstance(self.Step, step.Step)
533         self.assertEqual(self.Step.number,          self.number)
534         self.assertEqual(self.Step.shuffle_agents,  self.shuffle_agents)
535         self.assertEqual(self.Step.shuffle_actions, self.shuffle_actions)
536         self.assertEqual(self.Step.parallel_reg,    self.parallel_reg)
537         self.assertEqual(self.Step.parallel_loc,    False)
538         self.assertEqual(self.Step.level,           self.level)
539 
540         for index_i, thing in enumerate(actions.items()):
541             agent_key, action_keys = thing
542             self.assertIsInstance(self.Step[index_i], agent_actions.Actions)
543             self.assertEqual(self.Step[index_i].agent_key, agent_key)
544 
```

```
            for index_j, action in enumerate(action_keys):
                self.assertIsInstance(self.Step[index_i][index_j],
                                      agent_actions.Action)
                self.assertEqual(self.Step[index_i][index_j].action, action)
            for index_j, action in enumerate(self.Step[index_i]):
                self.assertIsInstance(action, agent_actions.Action)
                self.assertEqual(action.action, action_keys[index_j])
            self.assertEqual(len(self.Step[index_i]), 3)
        self.assertEqual(len(self.Step), 3)

        # Test Not Default
        self.Step = step.Step.setup(actions,
                                    self.number,
                                    self.shuffle_agents,
                                    self.shuffle_actions,
                                    False,
                                    self.parallel_loc,
                                    self.level)
        self.assertIsInstance(self.Step, step.Step)
        self.assertEqual(self.Step.number,          self.number)
        self.assertEqual(self.Step.shuffle_agents,  self.shuffle_agents)
        self.assertEqual(self.Step.shuffle_actions, self.shuffle_actions)
        self.assertEqual(self.Step.parallel_reg,    False)
        self.assertEqual(self.Step.parallel_loc,    self.parallel_loc)
        self.assertEqual(self.Step.level,           self.level)

        for index_i, thing in enumerate(actions.items()):
            agent_key, action_keys = thing
            self.assertIsInstance(self.Step[index_i], agent_actions.Actions)
            self.assertEqual(self.Step[index_i].agent_key, agent_key)

            for index_j, action in enumerate(action_keys):
                self.assertIsInstance(self.Step[index_i][index_j],
                                      agent_actions.Action)
                self.assertEqual(self.Step[index_i][index_j].action, action)
            for index_j, action in enumerate(self.Step[index_i]):
                self.assertIsInstance(action, agent_actions.Action)
                self.assertEqual(action.action, action_keys[index_j])
            self.assertEqual(len(self.Step[index_i]), 3)
```

```
        self.assertEqual(len(self.Step), 3)

        # Test error
        with self.assertRaisesRegex(TypeError,
                                    'Cannot have both location and regular '
                                    'parallel'):
            self.Step = step.Step.setup(actions,
                                        self.number,
                                        self.shuffle_agents,
                                        self.shuffle_actions,
                                        True,
                                        True,
                                        self.level)
```

C.5.10 test_simulation

```
FallArmyworm
└── test
    └── test_simulation
        ├── test_behaviors.py
        ├── test_models.py
        └── test_simulation.py
```

C.5.10.1 test_behaviors.py

```
import unittest      as ut
import unittest.mock as mk

import dataclasses as dclass

import source.keyword as keyword

import source.biomass.gut  as agent_gut
import source.biomass.mass as agent_mass

import source.development.egg   as agent_develop_egg
import source.development.larva as agent_develop_larva
import source.development.pupa  as agent_develop_pupa

import source.forage.cannibalism as agent_cannibalism
import source.forage.egg         as agent_forage_egg
import source.forage.larva       as agent_forage_larva
import source.forage.plant       as agent_forage_plant
import source.forage.target      as agent_target

import source.movement.adult as agent_move_adult
import source.movement.larva as agent_move_larva

import source.reproduction.lay  as agent_lay
import source.reproduction.mate as agent_mate

import source.simulation.behaviors as behaviors

import source.survival.adult as agent_survive_adult
import source.survival.egg   as agent_survive_egg
import source.survival.larva as agent_survive_larva
import source.survival.pupa  as agent_survive_pupa


class TestBehaviors(ut.TestCase):
    """test the Behaviors class"""

```

```
38      def setUp(self):
39          """Setup the tests"""
40
41          self.gut  = mk.create_autospec(agent_gut.Gut, spec_set=True)
42          self.mass = mk.create_autospec(agent_mass.Mass, spec_set=True)
43
44          self.develop_egg   = mk.create_autospec(agent_develop_egg.Egg,
45                                                  spec_set=True)
46          self.develop_larva = mk.create_autospec(agent_develop_larva.Larva,
47                                                  spec_set=True)
48          self.develop_pupa  = mk.create_autospec(agent_develop_pupa.Pupa,
49                                                  spec_set=True)
50
51          self.cannibalism  = mk.create_autospec(agent_cannibalism.Cannibalism,
52                                                 spec_set=True)
53          self.forage_egg   = mk.create_autospec(agent_forage_egg.Egg,
54                                                 spec_set=True)
55          self.forage_larva = mk.create_autospec(agent_forage_larva.Larva,
56                                                 spec_set=True)
57          self.forage_plant = mk.create_autospec(agent_forage_plant.Plant,
58                                                 spec_set=True)
59          self.target       = mk.create_autospec(agent_target.Target,
60                                                 spec_set=True)
61
62          self.move_adult = mk.create_autospec(agent_move_adult.Adult,
63                                               spec_set=True)
64          self.move_larva = mk.create_autospec(agent_move_larva.Larva,
65                                               spec_set=True)
66
67          self.lay  = mk.create_autospec(agent_lay.Lay, spec_set=True)
68          self.mate = mk.create_autospec(agent_mate.Mate, spec_set=True)
69
70          self.survive_adult = mk.create_autospec(agent_survive_adult.Adult,
71                                                  spec_set=True)
72          self.survive_egg   = mk.create_autospec(agent_survive_egg.Egg,
73                                                  spec_set=True)
74          self.survive_larva = mk.create_autospec(agent_survive_larva.Larva,
75                                                  spec_set=True)
76          self.survive_pupa  = mk.create_autospec(agent_survive_pupa.Pupa,
```

```
                                            spec_set=True)

        self.Behavior = behaviors.Behaviors(self.gut,
                                            self.mass,
                                            self.develop_egg,
                                            self.develop_larva,
                                            self.develop_pupa,
                                            self.cannibalism,
                                            self.forage_egg,
                                            self.forage_larva,
                                            self.forage_plant,
                                            self.target,
                                            self.move_adult,
                                            self.move_larva,
                                            self.lay,
                                            self.mate,
                                            self.survive_adult,
                                            self.survive_egg,
                                            self.survive_larva,
                                            self.survive_pupa)

    def test___init__(self):
        """test __init__ for class"""

        self.assertIsInstance(self.Behavior, behaviors.Behaviors)

        self.assertEqual(self.Behavior.gut,  self.gut)
        self.assertEqual(self.Behavior.mass, self.mass)

        self.assertEqual(self.Behavior.develop_egg,   self.develop_egg)
        self.assertEqual(self.Behavior.develop_larva, self.develop_larva)
        self.assertEqual(self.Behavior.develop_pupa,  self.develop_pupa)

        self.assertEqual(self.Behavior.cannibalism,  self.cannibalism)
        self.assertEqual(self.Behavior.forage_egg,   self.forage_egg)
        self.assertEqual(self.Behavior.forage_larva, self.forage_larva)
        self.assertEqual(self.Behavior.forage_plant, self.forage_plant)
        self.assertEqual(self.Behavior.target,       self.target)

```

```
116         self.assertEqual(self.Behavior.move_adult, self.move_adult)
117         self.assertEqual(self.Behavior.move_larva, self.move_larva)
118 
119         self.assertEqual(self.Behavior.lay,  self.lay)
120         self.assertEqual(self.Behavior.mate, self.mate)
121 
122         self.assertEqual(self.Behavior.survive_adult, self.survive_adult)
123         self.assertEqual(self.Behavior.survive_egg,   self.survive_egg)
124         self.assertEqual(self.Behavior.survive_larva, self.survive_larva)
125         self.assertEqual(self.Behavior.survive_pupa,  self.survive_pupa)
126 
127         self.assertTrue(dclass.is_dataclass(self.Behavior))
128 
129     def test_make_biomass(self):
130         """test make the biomass behaviors"""
131 
132         kwargs = {keyword.max_gut: mk.MagicMock(spec=callable),
133                   keyword.growth:  mk.MagicMock(spec=callable)}
134 
135         # Already present
136         self.Behavior.make_biomass(**kwargs)
137         self.assertEqual(self.Behavior.gut,  self.gut)
138         self.assertEqual(self.Behavior.mass, self.mass)
139 
140         # Not present
141         self.Behavior.gut  = None
142         self.Behavior.mass = None
143         self.Behavior.make_biomass(**kwargs)
144 
145         self.assertIsInstance(self.Behavior.gut, agent_gut.Gut)
146         self.assertEqual(self.Behavior.gut.max_gut, kwargs[keyword.max_gut])
147 
148         self.assertIsInstance(self.Behavior.mass, agent_mass.Mass)
149         self.assertEqual(self.Behavior.mass.max_gut, kwargs[keyword.max_gut])
150         self.assertEqual(self.Behavior.mass.growth,  kwargs[keyword.growth])
151 
152     def test_make_development(self):
153         """test make the development behavior"""
154 
```

```
        kwargs = {keyword.egg_development:   mk.MagicMock(spec=callable),
                  keyword.larva_development: mk.MagicMock(spec=callable),
                  keyword.pupa_development:  mk.MagicMock(spec=callable)}

        # Already present
        self.Behavior.make_development(**kwargs)
        self.assertEqual(self.Behavior.develop_egg,   self.develop_egg)
        self.assertEqual(self.Behavior.develop_larva, self.develop_larva)
        self.assertEqual(self.Behavior.develop_pupa,  self.develop_pupa)

        # Not present
        self.Behavior.develop_egg   = None
        self.Behavior.develop_larva = None
        self.Behavior.develop_pupa  = None
        self.Behavior.make_development(**kwargs)

        self.assertIsInstance(self.Behavior.develop_egg,
                              agent_develop_egg.Egg)
        self.assertEqual(self.Behavior.develop_egg.development,
                         kwargs[keyword.egg_development])

        self.assertIsInstance(self.Behavior.develop_larva,
                              agent_develop_larva.Larva)
        self.assertEqual(self.Behavior.develop_larva.development,
                         kwargs[keyword.larva_development])

        self.assertIsInstance(self.Behavior.develop_pupa,
                              agent_develop_pupa.Pupa)
        self.assertEqual(self.Behavior.develop_pupa.development,
                         kwargs[keyword.pupa_development])

    def test_make_forage(self):
        """test make forage behaviors"""

        kwargs = {keyword.fight:        mk.MagicMock(spec=callable),
                  keyword.encounter:    mk.MagicMock(spec=callable),
                  keyword.radius:       mk.MagicMock(spec=callable),
                  keyword.egg_forage:   mk.MagicMock(spec=callable),
                  keyword.larva_forage: mk.MagicMock(spec=callable),
```

```
                keyword.plant_forage: mk.MagicMock(spec=callable),
                keyword.loss:         mk.MagicMock(spec=callable)}

        # Already present
        self.Behavior.make_forage(**kwargs)
        self.assertEqual(self.Behavior.cannibalism,  self.cannibalism)
        self.assertEqual(self.Behavior.forage_egg,   self.forage_egg)
        self.assertEqual(self.Behavior.forage_larva, self.forage_larva)
        self.assertEqual(self.Behavior.forage_plant, self.forage_plant)
        self.assertEqual(self.Behavior.target,       self.target)

        # Not present
        self.Behavior.cannibalism  = None
        self.Behavior.forage_egg   = None
        self.Behavior.forage_larva = None
        self.Behavior.forage_plant = None
        self.Behavior.target       = None
        self.Behavior.make_forage(**kwargs)

        self.assertIsInstance(self.Behavior.cannibalism,
                              agent_cannibalism.Cannibalism)
        self.assertEqual(self.Behavior.cannibalism.fight,
                         kwargs[keyword.fight])
        self.assertEqual(self.Behavior.cannibalism.encounter,
                         kwargs[keyword.encounter])
        self.assertEqual(self.Behavior.cannibalism.radius,
                         kwargs[keyword.radius])

        self.assertIsInstance(self.Behavior.forage_egg,
                              agent_forage_egg.Egg)
        self.assertEqual(self.Behavior.forage_egg.forage,
                         kwargs[keyword.egg_forage])

        self.assertIsInstance(self.Behavior.forage_larva,
                              agent_forage_larva.Larva)
        self.assertEqual(self.Behavior.forage_larva.forage,
                         kwargs[keyword.larva_forage])

        self.assertIsInstance(self.Behavior.forage_plant,
```

```
                            agent_forage_plant.Plant)
        self.assertEqual(self.Behavior.forage_plant.forage,
                         kwargs[keyword.plant_forage])

        self.assertIsInstance(self.Behavior.target,
                              agent_target.Target)
        self.assertEqual(self.Behavior.target.loss,
                         kwargs[keyword.loss])

    def test_make_movement(self):
        """test make movement behaviors"""

        kwargs = {keyword.adult_movement: mk.MagicMock(spec=callable),
                  keyword.larva_movement: mk.MagicMock(spec=callable)}

        # Already present
        self.Behavior.make_movement(**kwargs)
        self.assertEqual(self.Behavior.move_adult, self.move_adult)
        self.assertEqual(self.Behavior.move_larva, self.move_larva)

        # Not present
        self.Behavior.move_adult = None
        self.Behavior.move_larva = None
        self.Behavior.make_movement(**kwargs)

        self.assertIsInstance(self.Behavior.move_adult,
                              agent_move_adult.Adult)
        self.assertEqual(self.Behavior.move_adult.movement,
                         kwargs[keyword.adult_movement])

        self.assertIsInstance(self.Behavior.move_larva,
                              agent_move_larva.Larva)
        self.assertEqual(self.Behavior.move_larva.movement,
                         kwargs[keyword.larva_movement])

    def test_make_reproduction(self):
        """test make reproduction behaviors"""

        kwargs = {keyword.trials:      mk.MagicMock(spec=int),
```

```
                keyword.fecundity:   mk.MagicMock(spec=callable),
                keyword.density:     mk.MagicMock(spec=callable),
                keyword.mating:      mk.MagicMock(spec=callable),
                keyword.mate_radius: mk.MagicMock(spec=callable)}

        # Already present
        self.Behavior.make_reproduction(**kwargs)
        self.assertEqual(self.Behavior.lay,  self.lay)
        self.assertEqual(self.Behavior.mate, self.mate)

        # Not present
        self.Behavior.lay  = None
        self.Behavior.mate = None
        self.Behavior.make_reproduction(**kwargs)

        self.assertIsInstance(self.Behavior.lay, agent_lay.Lay)
        self.assertEqual(self.Behavior.lay.fecundity,
                         kwargs[keyword.fecundity])
        self.assertEqual(self.Behavior.lay.density,
                         kwargs[keyword.density])

        self.assertIsInstance(self.Behavior.mate, agent_mate.Mate)
        self.assertEqual(self.Behavior.mate.mating,
                         kwargs[keyword.mating])
        self.assertEqual(self.Behavior.mate.radius,
                         kwargs[keyword.mate_radius])

    def test_make_survival(self):
        """test make survival behaviors"""

        kwargs = {keyword.adult_survival: mk.MagicMock(spec=callable),
                  keyword.egg_survival:   mk.MagicMock(spec=callable),
                  keyword.larva_survival: mk.MagicMock(spec=callable),
                  keyword.pupa_survival:  mk.MagicMock(spec=callable)}

        # Already present
        self.Behavior.make_survival(**kwargs)
        self.assertEqual(self.Behavior.survive_adult, self.survive_adult)
        self.assertEqual(self.Behavior.survive_egg,   self.survive_egg)
```

```
311         self.assertEqual(self.Behavior.survive_larva, self.survive_larva)
312         self.assertEqual(self.Behavior.survive_pupa,  self.survive_pupa)
313 
314         # Not present
315         self.Behavior.survive_adult = None
316         self.Behavior.survive_egg   = None
317         self.Behavior.survive_larva = None
318         self.Behavior.survive_pupa  = None
319         self.Behavior.make_survival(**kwargs)
320 
321         self.assertIsInstance(self.Behavior.survive_adult,
322                               agent_survive_adult.Adult)
323         self.assertEqual(self.Behavior.survive_adult.survival,
324                          kwargs[keyword.adult_survival])
325 
326         self.assertIsInstance(self.Behavior.survive_egg,
327                               agent_survive_egg.Egg)
328         self.assertEqual(self.Behavior.survive_egg.survival,
329                          kwargs[keyword.egg_survival])
330 
331         self.assertIsInstance(self.Behavior.survive_larva,
332                               agent_survive_larva.Larva)
333         self.assertEqual(self.Behavior.survive_larva.survival,
334                          kwargs[keyword.larva_survival])
335 
336         self.assertIsInstance(self.Behavior.survive_pupa,
337                               agent_survive_pupa.Pupa)
338         self.assertEqual(self.Behavior.survive_pupa.survival,
339                          kwargs[keyword.pupa_survival])
340 
341     def test_setup(self):
342         """test setup the class"""
343 
344         kwargs = {'test': mk.MagicMock()}
345 
346         with mk.patch.object(behaviors.Behaviors, 'make_biomass',
347                              autospec=True) as mkBiomass:
348             with mk.patch.object(behaviors.Behaviors, 'make_development',
349                                  autospec=True) as mkDevelopment:
```

```
                with mk.patch.object(behaviors.Behaviors, 'make_forage',
                                     autospec=True) as mkForage:
                    with mk.patch.object(behaviors.Behaviors, 'make_movement',
                                         autospec=True) as mkMovement:
                        with mk.patch.object(behaviors.Behaviors,
                                             'make_reproduction',
                                             autospec=True) as mkReproduction:
                            with mk.patch.object(behaviors.Behaviors,
                                                 'make_survival',
                                                 autospec=True) as mkSurvival:
                                self.Behavior = behaviors.Behaviors.\
                                    setup(**kwargs)

                                self.assertIsInstance(self.Behavior,
                                                      behaviors.Behaviors)

                                self.assertEqual(self.Behavior.gut,  None)
                                self.assertEqual(self.Behavior.mass, None)

                                self.assertEqual(self.Behavior.develop_egg,
                                                 None)
                                self.assertEqual(self.Behavior.develop_larva,
                                                 None)
                                self.assertEqual(self.Behavior.develop_pupa,
                                                 None)

                                self.assertEqual(self.Behavior.cannibalism,
                                                 None)
                                self.assertEqual(self.Behavior.forage_egg,
                                                 None)
                                self.assertEqual(self.Behavior.forage_larva,
                                                 None)
                                self.assertEqual(self.Behavior.forage_plant,
                                                 None)

                                self.assertEqual(self.Behavior.move_adult, None)
                                self.assertEqual(self.Behavior.move_larva, None)

                                self.assertEqual(self.Behavior.lay,  None)
```

```
                                    self.assertEqual(self.Behavior.mate, None)

                                    self.assertEqual(self.Behavior.survive_adult,
                                                     None)
                                    self.assertEqual(self.Behavior.survive_egg,
                                                     None)
                                    self.assertEqual(self.Behavior.survive_larva,
                                                     None)
                                    self.assertEqual(self.Behavior.survive_pupa,
                                                     None)

                                    self.assertEqual(mkBiomass.call_args_list,
                                                     [mk.call(self.Behavior,
                                                              **kwargs)])
                                    self.assertEqual(mkDevelopment.call_args_list,
                                                     [mk.call(self.Behavior,
                                                              **kwargs)])
                                    self.assertEqual(mkForage.call_args_list,
                                                     [mk.call(self.Behavior,
                                                              **kwargs)])
                                    self.assertEqual(mkMovement.call_args_list,
                                                     [mk.call(self.Behavior,
                                                              **kwargs)])
                                    self.assertEqual(mkReproduction.call_args_list,
                                                     [mk.call(self.Behavior,
                                                              **kwargs)])
                                    self.assertEqual(mkSurvival.call_args_list,
                                                     [mk.call(self.Behavior,
                                                              **kwargs)])
```

C.5.10.2 test_models.py

```
import unittest      as ut
import unittest.mock as mk

import dataclasses as dclass
import collections as collect

import source.keyword as keyword

import source.simulation.models as models


class TestModel(ut.TestCase):
    """test base input Model"""

    def setUp(self):
        """Setup the tests"""

        self.Model = models.Model()

    def test___init__(self):
        """test __init__ for class"""

        self.assertIsInstance(self.Model, models.Model)

        self.assertEqual(self.Model.model_key, None)

        self.assertTrue(dclass.is_dataclass(self.Model))

    def test___call__(self):
        """test __call__ for model"""

        self.assertIsNone(self.Model(*(mk.MagicMock(), mk.MagicMock()),
                                     **{'test': mk.MagicMock()}))


class TestModels(ut.TestCase):
    """test the input Models handling system"""
```

```
38
39      def setUp(self):
40          """Setup the tests"""
41
42          self.models = {}
43          for input_key in keyword.required_inputs:
44              self.models[input_key] = mk.create_autospec(models.Model,
45                                                          spec_set=True)
46          for _ in range(3):
47              self.models[mk.MagicMock(spec=str)] = \
48                  mk.create_autospec(models.Model, spec_set=True)
49
50          self.Models = models.Models(self.models)
51
52      def test___init__(self):
53          """test __init__ for class"""
54
55          self.assertIsInstance(self.Models, collect.UserDict)
56          self.assertIsInstance(self.Models, models.Models)
57
58          self.assertEqual(self.Models,      self.models)
59          self.assertEqual(self.Models.data, self.Models)
60
61      def test_add_model(self):
62          """test add a model"""
63
64          model = mk.MagicMock()
65
66          # Test add new
67          self.assertNotIn(model.model_key, self.Models)
68          self.Models.add_model(model)
69          self.assertIn(model.model_key, self.Models)
70          self.assertEqual(self.Models[model.model_key], model)
71          self.assertNotEqual(self.Models, self.models)
72
73          # Test try overwrite
74          with self.assertRaisesRegex(TypeError,
75                                      'Input data clash: {}'.
76                                              format(model.model_key)):
```

```
77              self.Models.add_model(model)
78
79      def test_add_variable(self):
80          """test add a variable to the class"""
81
82          variable_key = mk.MagicMock(spec=str)
83          variable     = mk.MagicMock()
84
85          # Test add new
86          self.assertNotIn(variable_key, self.Models)
87          self.Models.add_variable(variable_key, variable)
88          self.assertIn(variable_key, self.Models)
89          self.assertEqual(self.Models[variable_key], variable)
90          self.assertNotEqual(self.Models, self.models)
91
92          # Test try overwrite
93          with self.assertRaisesRegex(TypeError,
94                                      'Input data clash: {}'.
95                                              format(variable_key)):
96              self.Models.add_variable(variable_key, variable)
97
98      def test_check_inputs(self):
99          """test check the inputs"""
100
101         self.Models.check_inputs()
102         for input_key in keyword.required_inputs:
103             del self.Models[input_key]
104             with self.assertRaisesRegex(TypeError,
105                                         'Required input, {}, not given'.
106                                                 format(input_key)):
107                 self.Models.check_inputs()
108             self.Models[input_key] = mk.MagicMock()
109             self.Models.check_inputs()
110
111     def test_setup(self):
112         """test setup the model inputs"""
113
114         args   = (mk.MagicMock() for _ in range(3))
115         kwargs = {'test{}'.format(index): mk.MagicMock() for index in range(3)}
```

```
116
117         with mk.patch.object(models.Models, 'check_inputs',
118                              autospec=True) as mkCheck:
119             self.Models = models.Models.setup(*args, **kwargs)
120             self.assertEqual(len(self.Models), 6)
121             for arg in args:
122                 self.assertIn(arg.model_key, self.Models)
123                 self.assertEqual(self.Models[arg.model_key], arg)
124             for variable_key, variable in kwargs.items():
125                 self.assertIn(variable_key, self.Models)
126                 self.assertEqual(self.Models[variable_key], variable)
127
128             self.assertEqual(mkCheck.call_args_list,
129                              [mk.call(self.Models)])
```

C.5.10.3 test_simulation.py

```
import unittest      as ut
import unittest.mock as mk

import dataclasses  as dclass
import itertools    as i_tools
import numpy.random as rnd
import pickle       as pk

import source.keyword as keyword

import source.schedule.schedule as schedule

import source.agents.adult    as adult
import source.agents.egg_mass as egg_mass
import source.agents.larva    as larva
import source.agents.pupa     as pupa

import source.data.database as database

import source.migration.emigration  as emigration
import source.migration.immigration as immigration

import source.simulation.behaviors  as behaviors
import source.simulation.models     as models
import source.simulation.simulation as simulation

import source.space.agents as agents
import source.space.graph  as main_graph
import source.space.space  as space


class GraphTest(main_graph.Graph):
    """Class to add dynamic values for tests"""

    adjacency = mk.create_autospec(main_graph.Adjacency, spec_seth=True)


```

```
38 class TestSimulation(ut.TestCase):
39     """test the model Simulation class"""
40
41     def setUp(self):
42         """Setup the tests"""
43
44         self.space     = mk.create_autospec(space.Space,           spec_set=True)
45         self.agents    = mk.create_autospec(agents.Agents,         spec_set=True)
46         self.schedule  = mk.create_autospec(schedule.Schedule,     spec_set=True)
47         self.models    = mk.create_autospec(models.Models,         spec_set=True)
48         self.behaviors = mk.create_autospec(behaviors.Behaviors,   spec_set=True)
49         self.database  = mk.create_autospec(database.Database,     spec_set=True)
50
51         self.emigration = mk.create_autospec(emigration.Emigrations,
52                                              spec_set=True)
53         self.immigration = mk.create_autospec(immigration.Immigrations,
54                                               spec_set=True)
55
56         self.timestep = mk.MagicMock(spec=int)
57
58         self.Simulation = simulation.Simulation(self.space,
59                                                 self.agents,
60                                                 self.schedule,
61                                                 self.models,
62                                                 self.behaviors,
63                                                 self.database,
64                                                 self.emigration,
65                                                 self.immigration,
66                                                 self.timestep)
67
68     def test___init__(self):
69         """test __init__ for class"""
70
71         self.assertIsInstance(self.Simulation, simulation.Simulation)
72
73         self.assertEqual(self.Simulation.space,       self.space)
74         self.assertEqual(self.Simulation.agents,      self.agents)
75         self.assertEqual(self.Simulation.schedule,    self.schedule)
76         self.assertEqual(self.Simulation.models,      self.models)
```

```
77          self.assertEqual(self.Simulation.behaviors,   self.behaviors)
78          self.assertEqual(self.Simulation.database,    self.database)
79          self.assertEqual(self.Simulation.emigration,  self.emigration)
80          self.assertEqual(self.Simulation.immigration, self.immigration)
81          self.assertEqual(self.Simulation.timestep,    self.timestep)
82
83          # noinspection PyTypeChecker
84          self.assertIsInstance(self.Simulation._id_count, i_tools.count)
85          # noinspection PyTypeChecker
86          self.assertEqual(next(self.Simulation._id_count),
87                           next(i_tools.count()))
88
89          self.assertTrue(dclass.is_dataclass(self.Simulation))
90
91      def test_count_step(self):
92          """test count a step"""
93
94          self.assertEqual(self.Simulation.count_step(),
95                           self.timestep.__add__.return_value)
96          self.assertEqual(self.Simulation.timestep,
97                           self.timestep.__add__.return_value)
98          self.assertEqual(self.timestep.__add__.call_args_list,
99                           [mk.call(1)])
100
101     def test_new_unique_id(self):
102         """test get a new unique_id"""
103
104         for index in range(10):
105             self.assertEqual(self.Simulation.new_unique_id(), index)
106
107         self.assertEqual(next(self.Simulation._id_count), 10)
108
109     def test_populate_egg_masses(self):
110         """test generate all new egg_masses"""
111
112         new = mk.create_autospec(egg_mass.EggMass, spec_set=True)
113
114         with mk.patch.object(egg_mass.EggMass, 'setup',
115                              autospec=True) as mkSetup:
```

```
            with mk.patch.object(simulation.Simulation, 'new_unique_id',
                                 autospec=True) as mkId:
                mkSetup.return_value = new

                self.Simulation.populate_egg_masses((3, 3, 3))

                for index, call in enumerate(mkSetup.call_args_list):
                    self.assertEqual(call,
                                     mk.call(mkId.return_value,
                                             keyword.init,
                                             self.Simulation,
                                             keyword.genotype_keys[index // 3]))
                self.assertEqual(len(mkSetup.call_args_list), 9)
                for call in new.activate.call_args_list:
                    self.assertEqual(call, mk.call())
                self.assertEqual(len(new.activate.call_args_list), 9)
                for call in mkId.call_args_list:
                    self.assertEqual(call, mk.call(self.Simulation))
                self.assertEqual(len(mkId.call_args_list), 9)

    def test_populate_larvae(self):
        """test generate all new larvae"""

        new = mk.create_autospec(larva.Larva, spec_set=True)

        with mk.patch.object(larva.Larva, 'setup',
                             autospec=True) as mkSetup:
            with mk.patch.object(simulation.Simulation, 'new_unique_id',
                                 autospec=True) as mkId:
                mkSetup.return_value = new

                self.Simulation.populate_larvae((3, 3, 3))

                for index, call in enumerate(mkSetup.call_args_list):
                    self.assertEqual(call,
                                     mk.call(mkId.return_value,
                                             keyword.init,
                                             self.Simulation,
                                             keyword.genotype_keys[index // 3]))
```

```
155                 self.assertEqual(len(mkSetup.call_args_list), 9)
156                 for call in new.activate.call_args_list:
157                     self.assertEqual(call, mk.call())
158                 self.assertEqual(len(new.activate.call_args_list), 9)
159                 for call in mkId.call_args_list:
160                     self.assertEqual(call, mk.call(self.Simulation))
161                 self.assertEqual(len(mkId.call_args_list), 9)
162 
163     def test_populate_pupae(self):
164         """test generate all new pupae"""
165 
166         new = mk.create_autospec(pupa.Pupa, spec_set=True)
167 
168         with mk.patch.object(pupa.Pupa, 'setup',
169                              autospec=True) as mkSetup:
170             with mk.patch.object(simulation.Simulation, 'new_unique_id',
171                                  autospec=True) as mkId:
172                 mkSetup.return_value = new
173 
174                 self.Simulation.populate_pupae((3, 3, 3))
175 
176                 for index, call in enumerate(mkSetup.call_args_list):
177                     self.assertEqual(call,
178                                      mk.call(mkId.return_value,
179                                              keyword.init,
180                                              self.Simulation,
181                                              keyword.genotype_keys[index // 3]))
182                 self.assertEqual(len(mkSetup.call_args_list), 9)
183                 for call in new.activate.call_args_list:
184                     self.assertEqual(call, mk.call())
185                 self.assertEqual(len(new.activate.call_args_list), 9)
186                 for call in mkId.call_args_list:
187                     self.assertEqual(call, mk.call(self.Simulation))
188                 self.assertEqual(len(mkId.call_args_list), 9)
189 
190     def test_populate_adults(self):
191         """test generate all new adults"""
192 
193         new = mk.create_autospec(adult.Adult, spec_set=True)
```

```
194
195         with mk.patch.object(adult.Adult, 'setup',
196                              autospec=True) as mkSetup:
197             with mk.patch.object(simulation.Simulation, 'new_unique_id',
198                                  autospec=True) as mkId:
199                 mkSetup.return_value = new
200
201                 self.Simulation.populate_adults((3, 3, 3))
202
203                 for index, call in enumerate(mkSetup.call_args_list):
204                     self.assertEqual(call,
205                                      mk.call(mkId.return_value,
206                                              keyword.init,
207                                              self.Simulation,
208                                              keyword.genotype_keys[index // 3]))
209                 self.assertEqual(len(mkSetup.call_args_list), 9)
210                 for call in new.activate.call_args_list:
211                     self.assertEqual(call, mk.call())
212                 self.assertEqual(len(new.activate.call_args_list), 9)
213                 for call in mkId.call_args_list:
214                     self.assertEqual(call, mk.call(self.Simulation))
215                 self.assertEqual(len(mkId.call_args_list), 9)
216
217     def test_populate_pregnant(self):
218         """test generate all new pregnant adults"""
219
220         parents = [[keyword.homo_r, keyword.homo_r],
221                    [keyword.homo_r, keyword.homo_s],
222                    [keyword.homo_s, keyword.homo_s]]
223
224         new = mk.create_autospec(adult.Adult, spec_set=True)
225
226         with mk.patch.object(adult.Adult, 'setup',
227                              autospec=True) as mkSetup:
228             with mk.patch.object(simulation.Simulation, 'new_unique_id',
229                                  autospec=True) as mkId:
230                 with mk.patch.object(rnd, 'shuffle') as mkRND:
231                     mkSetup.return_value = new
232
```

```
                    self.Simulation.populate_pregnant((3, 3, 3))

                    for index, call in enumerate(mkSetup.call_args_list):
                        self.assertEqual(call,
                                         mk.call(mkId.return_value,
                                                 keyword.init,
                                                 self.Simulation,
                                                 parents[index // 3][0],
                                                 parents[index // 3][1]))
                    self.assertEqual(len(mkSetup.call_args_list), 9)
                    for call in new.activate.call_args_list:
                        self.assertEqual(call, mk.call())
                    self.assertEqual(len(new.activate.call_args_list), 9)
                    for call in mkId.call_args_list:
                        self.assertEqual(call, mk.call(self.Simulation))
                    self.assertEqual(len(mkId.call_args_list), 9)
                    for index, call in enumerate(mkRND.call_args_list):
                        self.assertEqual(call,
                                         mk.call(parents[index // 3]))
                    self.assertEqual(len(mkRND.call_args_list), 9)

    def test_populate(self):
        """test populate the simulation"""

        nums = [(mk.MagicMock(spec=int),
                 mk.MagicMock(spec=int),
                 mk.MagicMock(spec=int)) for _ in range(5)]
        nums = tuple(nums)

        with mk.patch.object(simulation.Simulation, 'populate_egg_masses',
                             autospec=True) as mkEggs:
            with mk.patch.object(simulation.Simulation, 'populate_larvae',
                                 autospec=True) as mkLarvae:
                with mk.patch.object(simulation.Simulation, 'populate_pupae',
                                     autospec=True) as mkPupae:
                    with mk.patch.object(simulation.Simulation,
                                         'populate_adults',
                                         autospec=True) as mkAdults:
                        with mk.patch.object(simulation.Simulation,
```

```
                                        'populate_pregnant',
                                        autospec=True) as mkPregnant:
                        # noinspection PyTypeChecker
                        self.Simulation.populate(nums)

                        self.assertEqual(mkEggs.call_args_list,
                                         [mk.call(self.Simulation,
                                                  nums[0])])
                        self.assertEqual(mkLarvae.call_args_list,
                                         [mk.call(self.Simulation,
                                                  nums[1])])
                        self.assertEqual(mkPupae.call_args_list,
                                         [mk.call(self.Simulation,
                                                  nums[2])])
                        self.assertEqual(mkAdults.call_args_list,
                                         [mk.call(self.Simulation,
                                                  nums[3])])
                        self.assertEqual(mkPregnant.call_args_list,
                                         [mk.call(self.Simulation,
                                                  nums[4])])

    def test_step(self):
        """test advance the model by one step"""

        master = mk.MagicMock()
        master.attach_mock(self.schedule.perform,        'perform')
        master.attach_mock(self.immigration.immigration, 'immigration')
        master.attach_mock(self.emigration.emigration,   'emigration')
        master.attach_mock(self.agents.record,           'record')
        master.attach_mock(self.database.save,           'save')

        self.Simulation.step()
        self.assertEqual(master.mock_calls,
                         [mk.call.perform(self.space, self.agents),
                          mk.call.immigration(self.Simulation),
                          mk.call.emigration(self.agents),
                          mk.call.record(),
                          mk.call.save(self.Simulation)])

```

```
    def test_save(self):
        """test save to file"""

        filename = mk.MagicMock(spec=str)

        with mk.patch('builtins.open', mk.mock_open()) as mkOpen:
            with mk.patch.object(pk, 'dump') as mkDump:
                self.Simulation.save(filename)
                self.assertEqual(mkDump.call_args_list,
                                 [mk.call(self.Simulation, mkOpen.return_value,
                                          protocol=pk.HIGHEST_PROTOCOL)])
                self.assertEqual(mkOpen.call_args_list,
                                 [mk.call(filename, 'wb')])

    def test_setup(self):
        """test setup the class"""

        graph = mk.create_autospec(GraphTest, spec_set=True)
        graph.adjacency = mk.create_autospec(main_graph.Adjacency,
                                             spec_set=True)
        self.space.__getitem__.return_value = graph

        nums = mk.MagicMock(spec=tuple)
        grid_generators = [mk.MagicMock(spec=tuple) for _ in range(3)]
        attrs = mk.MagicMock(spec=dict)
        data_tuple = mk.MagicMock(spec=tuple)
        bt_prop = mk.MagicMock(spec=float)
        step_tuples = [mk.MagicMock(spec=tuple) for _ in range(3)]
        emigration_tuples = [mk.MagicMock(spec=tuple) for _ in range(3)]
        immigration_tuples = [mk.MagicMock(spec=tuple) for _ in range(3)]
        args   = tuple([mk.MagicMock() for _ in range(3)])
        kwargs = {'test{}'.format(index): mk.MagicMock() for index in range(3)}

        cutoff = mk.MagicMock(spec=float)
        bt_prop.__mul__.return_value = cutoff

        init_plant  = mk.MagicMock(spec=callable)
        test_models = {keyword.init_plant: init_plant}

```

```
        with mk.patch.object(models.Models,
                             'setup') as mkModels:
            with mk.patch.object(behaviors.Behaviors,
                                 'setup') as mkBehaviors:
                with mk.patch.object(schedule.Schedule,
                                     'setup') as mkSchedule:
                    with mk.patch.object(database.Database,
                                         'setup') as mkDatabase:
                        with mk.patch.object(emigration.Emigrations,
                                             'setup') as mkEmigration:
                            with mk.patch.object(immigration.Immigrations,
                                                 'setup') as mkImmigration:
                                with mk.patch.object(space.Space,
                                                     'setup') as mkSpace:
                                    with mk.patch.object(agents.Agents,
                                                         'empty') as mkAgents:
                                        with mk.patch.object(simulation.
                                                             Simulation,
                                                             'populate') \
                                                as mkPop:
                                            mkModels.return_value = test_models
                                            mkBehaviors.return_value = \
                                                self.behaviors
                                            mkSchedule.return_value = \
                                                self.schedule
                                            mkDatabase.return_value = \
                                                self.database
                                            mkEmigration.return_value = \
                                                self.emigration
                                            mkImmigration.return_value = \
                                                self.immigration
                                            mkSpace.return_value = self.space
                                            mkAgents.return_value = self.agents

                                            sim = \
                                                simulation.Simulation.\
                                                    setup(nums,
                                                          grid_generators,
                                                          attrs,
```

```
                                                    data_tuple,
                                                    bt_prop,
                                                    step_tuples,
                                                    emigration_tuples,
                                                    immigration_tuples,
                                                    *args, **kwargs)

        self.assertEqual(mkModels.call_args_list,
                         [mk.call(*args, **kwargs)])
        self.assertEqual(mkBehaviors.call_args_list,
                         [mk.call(**test_models)])
        self.assertEqual(mkSchedule.call_args_list,
                         [mk.call(step_tuples)])
        self.assertEqual(mkDatabase.call_args_list,
                         [mk.call(data_tuple)])
        self.assertEqual(mkEmigration.call_args_list,
                         [mk.call(emigration_tuples)])
        self.assertEqual(mkImmigration.call_args_list,
                         [mk.call(immigration_tuples)])
        self.assertEqual(mkSpace.call_args_list,
                         [mk.call(grid_generators)])
        self.assertEqual(mkAgents.call_args_list,
                         [mk.call(self.space,
                                  keyword.agent_keys,
                                  attrs, (cutoff, init_plant))])
        self.assertEqual(bt_prop.__mul__.call_args_list,
                         [mk.call(self.space.
                                    __getitem__.return_value.adjacency.num)])
        self.assertEqual(self.space.__getitem__.call_args_list,
                         [mk.call(keyword.bt_level)])
        self.assertEqual(mkPop.call_args_list,
                         [mk.call(nums)])
        self.assertEqual(self.agents.record.call_args_list,
                         [mk.call()])

        self.assertIsInstance(sim, simulation.Simulation)

        self.assertEqual(sim.space,       self.space)
        self.assertEqual(sim.agents,      self.agents)
```

```
        self.assertEqual(sim.schedule,    self.schedule)
        self.assertEqual(sim.models,      test_models)
        self.assertEqual(sim.behaviors,   self.behaviors)
        self.assertEqual(sim.database,    self.database)
        self.assertEqual(sim.emigration,  self.emigration)
        self.assertEqual(sim.immigration, self.immigration)
        self.assertEqual(sim.timestep,    0)
```

C.5.11 test_space

```
FallArmyworm
└── test
    └── test_space
        ├── test_agents.py
        ├── test_environment.py
        ├── test_graph.py
        ├── test_grid.py
        ├── test_location.py
        └── test_space.py
```

C.5.11.1 test_agents.py

```
import unittest      as ut
import unittest.mock as mk

import collections as collect
import pandas      as pd

import source.keyword as keyword

import source.agents.agent as main_agent

import source.data.counter as counter

import source.space.agents      as agents
import source.space.environment as agent_environment
import source.space.location    as agent_location
import source.space.space       as agent_space


class AgentTest(main_agent.Agent):
    """Class to add dynamic values for tests"""

    agent_key = mk.MagicMock(spec=str)
    unique_id = mk.MagicMock(spec=str)
    location  = mk.create_autospec(agent_location.Location, spec_set=True)


class AgentBinTest(agents.AgentBin):
    """Class to add dynamic values for tests"""

    counts = mk.create_autospec(counter.Counts, spec_set=True)


class SpaceTest(agent_space.Space):
    """Class to add dynamic values for tests"""

    locations = [mk.create_autospec(agent_location.Location, spec_set=True)
                 for _ in range(3)]
```

```
38
39
40 class TestAgentBin(ut.TestCase):
41     """test AgentBin class"""
42
43     def setUp(self):
44         """Setup the tests"""
45
46         self.agents = {}
47         for _ in range(3):
48             unique_id = mk.MagicMock(spec=str)
49             agent     = mk.create_autospec(AgentTest, spec_set=True)
50             agent.unique_id = unique_id
51             self.agents[unique_id] = agent
52
53         self.counts    = mk.create_autospec(counter.Counts, spec_set=True)
54         self.agent_key = mk.MagicMock(spec=str)
55
56         self.AgentBin = agents.AgentBin(self.agents,
57                                         self.counts,
58                                         self.agent_key)
59
60     def test___init__(self):
61         """test __init__ for class"""
62
63         self.assertIsInstance(self.AgentBin, collect.UserDict)
64         self.assertIsInstance(self.AgentBin, agents.AgentBin)
65
66         self.assertEqual(self.AgentBin.counts,    self.counts)
67         self.assertEqual(self.AgentBin.agent_key, self.agent_key)
68
69         self.assertEqual(self.AgentBin,      self.agents)
70         self.assertEqual(self.AgentBin.data, self.agents)
71
72     def test_agents(self):
73         """test get the agents in system"""
74
75         # Test calls
76         with mk.patch.object(agents.AgentBin, 'values',
```

```
                            autospec=True) as mkValues:
            with mk.patch.object(agents, 'list') as mkList:
                self.assertEqual(self.AgentBin.agents,
                                 mkList.return_value)
                self.assertEqual(mkList.call_args_list,
                                 [mk.call(mkValues.return_value)])
                self.assertEqual(mkValues.call_args_list,
                                 [mk.call(self.AgentBin)])

        # Test practical
        self.assertEqual(self.AgentBin.agents, list(self.agents.values()))

    def test_activate(self):
        """test activate an agent"""


        agent     = mk.create_autospec(AgentTest, spec_set=True)
        unique_id = mk.MagicMock(spec=str)
        agent.unique_id = unique_id

        self.assertNotIn(unique_id, self.AgentBin)
        self.AgentBin.activate(agent)
        self.assertIn(unique_id, self.AgentBin)
        self.assertEqual(self.AgentBin[unique_id], agent)
        self.assertEqual(self.counts.add.call_args_list,
                         [mk.call(agent)])

        self.assertNotEqual(self.AgentBin, self.agents)

    def test_deactivate(self):
        """test deactivate"""

        self.assertEqual(len(self.AgentBin), 3)
        for unique_id, agent in self.agents.items():
            self.assertIn(unique_id, self.AgentBin)
            self.AgentBin.deactivate(agent)
            self.assertNotIn(unique_id, self.AgentBin)
            self.assertEqual(self.counts.sub.call_args_list,
                             [mk.call(agent)])
```

```
            self.counts.reset_mock()
        self.assertEqual(len(self.AgentBin), 0)

    def test_empty(self):
        """test create empty class"""

        attrs = {}
        for _ in range(3):
            attr   = mk.MagicMock(spec=str)
            values = [mk.MagicMock(spec=str) for _ in range(3)]
            removal = mk.MagicMock(spec=bool)
            attrs[mk.MagicMock(spec=str)] = (attr, values, removal)

        self.AgentBin = agents.AgentBin.empty(self.agent_key, attrs)
        self.assertIsInstance(self.AgentBin, agents.AgentBin)
        self.assertEqual(self.AgentBin, {})
        self.assertEqual(self.AgentBin.agent_key, self.agent_key)

        self.assertIsInstance(self.AgentBin.counts, counter.Counts)
        for attr_key, things in attrs.items():
            attr, values, removal = things
            self.assertIsInstance(self.AgentBin.counts[attr_key], counter.Count)
            self.assertEqual(self.AgentBin.counts[attr_key].attr,    attr)
            self.assertEqual(self.AgentBin.counts[attr_key].removal, removal)
            for key, value in self.AgentBin.counts[attr_key].items():
                self.assertIn(key, values)
                self.assertEqual(value, 0)
            for value in values:
                self.assertIn(value, self.AgentBin.counts[attr_key])
                self.assertEqual(self.AgentBin.counts[attr_key][value], 0)

            self.assertIsInstance(self.AgentBin.counts[attr_key].data_columns,
                                  counter.DataColumns)
            self.assertEqual(self.AgentBin.counts[attr_key].data_columns.attr,
                             attr)
            for key, column in self.AgentBin.counts[attr_key].\
                    data_columns.items():
                self.assertIn(key, values)
                self.assertIsInstance(column, counter.DataColumn)
```

```
                self.assertEqual(column.attr_value, key)
                self.assertEqual(column, [])
            for value in values:
                self.assertIn(value,
                              self.AgentBin.counts[attr_key].data_columns)
                self.assertIsInstance(self.AgentBin.counts[attr_key].
                                      data_columns[value],
                                    counter.DataColumn)
                self.assertEqual(self.AgentBin.counts[attr_key].
                                    data_columns[value].attr_value,
                                 value)
                self.assertEqual(self.AgentBin.counts[attr_key].
                                    data_columns[value],
                                 [])


class TestAgentsBin(ut.TestCase):
    """test AgentsBin class"""

    def setUp(self):
        """Setup the tests"""

        self.agents = {}
        self.agent_list = []
        for _ in range(3):
            agent = mk.create_autospec(AgentTest, spec_set=True)
            self.agents[agent.agent_key] = mk.create_autospec(AgentBinTest,
                                                              spec_set=True)
            self.agent_list.append(agent)

        self.location_key = mk.MagicMock(spec=tuple)
        self.environment  = mk.create_autospec(agent_environment.Environment,
                                               spec_set=True)

        self.AgentsBin = agents.AgentsBin(self.agents,
                                          self.location_key,
                                          self.environment)

    def test___init__(self):
```

```
        """test __init__ for class"""

        self.assertIsInstance(self.AgentsBin, collect.UserDict)
        self.assertIsInstance(self.AgentsBin, agents.AgentsBin)

        self.assertEqual(self.AgentsBin.location_key, self.location_key)
        self.assertEqual(self.AgentsBin.environment,  self.environment)

        self.assertEqual(self.AgentsBin,      self.agents)
        self.assertEqual(self.AgentsBin.data, self.agents)

    def test_activate(self):
        """test activate an agent"""

        for agent in self.agent_list:
            self.AgentsBin.activate(agent)
            self.assertEqual(self.agents[agent.agent_key].activate.
                                call_args_list,
                            [mk.call(agent)])
            self.agents[agent.agent_key].reset_mock()
            for agent_bin in self.agents.values():
                self.assertEqual(agent_bin.activate.call_args_list, [])
            self.assertEqual(len(self.agents), 3)
        self.assertEqual(len(self.agent_list), 3)

    def test_deactivate(self):
        """test deactivate an agent"""

        for agent in self.agent_list:
            self.AgentsBin.deactivate(agent)
            self.assertEqual(self.agents[agent.agent_key].deactivate.
                            call_args_list,
                            [mk.call(agent)])
            self.agents[agent.agent_key].reset_mock()
            for agent_bin in self.agents.values():
                self.assertEqual(agent_bin.deactivate.call_args_list, [])
            self.assertEqual(len(self.agents), 3)
        self.assertEqual(len(self.agent_list), 3)

```

```
233     def test_record(self):
234         """test record the counts"""
235 
236         for agent_bin in self.agents.values():
237             agent_bin.counts = mk.create_autospec(counter.Counts, spec_set=True)
238 
239         self.AgentsBin.record()
240         for agent_bin in self.AgentsBin.values():
241             self.assertEqual(agent_bin.counts.record.call_args_list,
242                              [mk.call()])
243 
244     def test_refresh(self):
245         """test refresh the stored data"""
246 
247         for agent_bin in self.agents.values():
248             agent_bin.counts = mk.create_autospec(counter.Counts, spec_set=True)
249 
250         self.AgentsBin.refresh()
251         for agent_bin in self.AgentsBin.values():
252             self.assertEqual(agent_bin.counts.refresh.call_args_list,
253                              [mk.call()])
254 
255     def test_dataframes(self):
256         """test create a dictionary of dataframes"""
257 
258         # Test no empty
259         dataframes = {}
260         for agent_key, agent_bin in self.agents.items():
261             key = '{}_{}'.format(self.location_key, agent_key)
262             agent_bin.counts = mk.create_autospec(counter.Counts, spec_set=True)
263             dataframe        = mk.create_autospec(pd.DataFrame, spec_set=True)
264             dataframe.empty  = False
265             agent_bin.counts.dataframe.return_value = dataframe
266             dataframes[key] = dataframe
267         self.assertEqual(len(dataframes), 3)
268 
269         self.assertEqual(self.AgentsBin.dataframes(), dataframes)
270         for agent_bin in self.AgentsBin.values():
271             self.assertEqual(agent_bin.counts.dataframe.call_args_list,
```

```
                            [mk.call()])

        # Test empty
        dataframes = {}
        for agent_key, agent_bin in self.agents.items():
            key = '{}_{}'.format(self.location_key, agent_key)
            agent_bin.counts = mk.create_autospec(counter.Counts, spec_set=True)
            dataframe        = mk.create_autospec(pd.DataFrame, spec_set=True)
            dataframe.empty  = True
            agent_bin.counts.dataframe.return_value = dataframe
            dataframes[key] = dataframe
        self.assertEqual(len(dataframes), 3)

        self.assertEqual(self.AgentsBin.dataframes(), {})
        for agent_bin in self.AgentsBin.values():
            self.assertEqual(agent_bin.counts.dataframe.call_args_list,
                             [mk.call()])

    def test_make_environment(self):
        """test make the environment"""

        location = mk.create_autospec(agent_location.Location, spec_set=True)
        location.depth.__eq__.side_effect = [False, True, True]

        location.__getitem__.return_value.__lt__.side_effect = [True, False]

        cutoff      = mk.MagicMock(spec=float)
        init_plant  = mk.MagicMock(spec=callable)
        environment = (cutoff, init_plant)

        # Test incorrect depth
        environ = self.AgentsBin.make_environment(location, environment)
        self.assertIsInstance(environ, agent_environment.Environment)
        self.assertEqual(environ.bt,    None)
        self.assertEqual(environ.plant, None)
        self.assertEqual(location.depth.__eq__.call_args_list,
                         [mk.call(keyword.bt_depth)])

        location.reset_mock()
```

```
311         # Test correct depth, bt
312         environ = self.AgentsBin.make_environment(location, environment)
313         self.assertIsInstance(environ, agent_environment.Environment)
314         self.assertEqual(environ.bt,    keyword.bt)
315         self.assertEqual(environ.plant,
316                          init_plant.return_value)
317         self.assertEqual(init_plant.call_args_list,
318                          [mk.call(keyword.bt)])
319         self.assertEqual(location.depth.__eq__.call_args_list,
320                          [mk.call(keyword.bt_depth)])
321         self.assertEqual(location.__getitem__.return_value.
322                             __lt__.call_args_list,
323                          [mk.call(cutoff)])
324         self.assertEqual(location.__getitem__.call_args_list,
325                          [mk.call(-1)])
326 
327         location.reset_mock()
328         init_plant.reset_mock()
329         # Test correct depth, not_bt
330         environ = self.AgentsBin.make_environment(location, environment)
331         self.assertIsInstance(environ, agent_environment.Environment)
332         self.assertEqual(environ.bt,    keyword.not_bt)
333         self.assertEqual(environ.plant,
334                          init_plant.return_value)
335         self.assertEqual(init_plant.call_args_list,
336                          [mk.call(keyword.not_bt)])
337         self.assertEqual(location.depth.__eq__.call_args_list,
338                          [mk.call(keyword.bt_depth)])
339         self.assertEqual(location.__getitem__.return_value.
340                          __lt__.call_args_list,
341                          [mk.call(cutoff)])
342         self.assertEqual(location.__getitem__.call_args_list,
343                          [mk.call(-1)])
344 
345     def test_make_bins(self):
346         """test make the bins"""
347 
348         agent_keys = []
349         attrs      = {}
```

```
        for _ in range(3):
            attr = {}
            for _ in range(3):
                attr_val = mk.MagicMock(spec=str)
                values = [mk.MagicMock(spec=str) for _ in range(3)]
                removal = mk.MagicMock(spec=bool)
                attr[mk.MagicMock(spec=str)] = (attr_val, values, removal)

            agent_key = mk.MagicMock(spec=str)
            agent_keys.append(agent_key)
            attrs[agent_key] = attr

        # with attrs
        # noinspection PyTypeChecker
        agent_bins = self.AgentsBin.make_bins(agent_keys, attrs)
        for agent_key in agent_keys:
            self.assertIn(agent_key, agent_bins)
            self.assertIsInstance(agent_bins[agent_key], agents.AgentBin)
            self.assertEqual(agent_bins[agent_key], {})
            self.assertEqual(agent_bins[agent_key].agent_key, agent_key)
            self.assertIsInstance(agent_bins[agent_key].counts,
                                  counter.Counts)
            for attr_key, things in attrs[agent_key].items():
                attr, values, removal = things
                self.assertIsInstance(agent_bins[agent_key].counts[attr_key],
                                      counter.Count)
                self.assertEqual(agent_bins[agent_key].counts[attr_key].attr,
                                 attr)
                self.assertEqual(agent_bins[agent_key].counts[attr_key].removal,
                                 removal)
                for key, value in \
                        agent_bins[agent_key].counts[attr_key].items():
                    self.assertIn(key, values)
                    self.assertEqual(value, 0)
                for value in values:
                    self.assertIn(value, agent_bins[agent_key].counts[attr_key])
                    self.assertEqual(agent_bins[agent_key].
                                     counts[attr_key][value],
                                     0)
```

```
                self.assertEqual(len(agent_bins[agent_key].counts[attr_key]), 3)
                self.assertIsInstance(agent_bins[agent_key].
                                      counts[attr_key].data_columns,
                                    counter.DataColumns)
                self.assertEqual(agent_bins[agent_key].
                                   counts[attr_key].data_columns.attr, attr)
                for key, column in agent_bins[agent_key].counts[attr_key].\
                        data_columns.items():
                    self.assertIn(key, values)
                    self.assertIsInstance(column, counter.DataColumn)
                    self.assertEqual(column.attr_value, key)
                    self.assertEqual(column, [])
                for value in values:
                    self.assertIn(value,
                                  agent_bins[agent_key].counts[attr_key].
                                    data_columns)
                    self.assertIsInstance(agent_bins[agent_key].
                                          counts[attr_key].
                                          data_columns[value],
                                        counter.DataColumn)
                    self.assertEqual(agent_bins[agent_key].counts[attr_key].
                                       data_columns[value].attr_value,
                                     value)
                    self.assertEqual(agent_bins[agent_key].counts[attr_key].
                                     data_columns[value], [])
                self.assertEqual(len(agent_bins[agent_key].counts[attr_key].
                                     data_columns), 3)
            self.assertEqual(len(agent_bins[agent_key].counts), 3)
        self.assertEqual(len(agent_bins), 3)

        # no attrs
        agent_bins = self.AgentsBin.make_bins(agent_keys, {})
        for agent_key in agent_keys:
            self.assertIsInstance(agent_bins[agent_key], agents.AgentBin)
            self.assertEqual(agent_bins[agent_key], {})
            self.assertEqual(agent_bins[agent_key].agent_key, agent_key)
            self.assertIsInstance(agent_bins[agent_key].counts, counter.Counts)
            self.assertEqual(agent_bins[agent_key].counts, {})
        self.assertEqual(len(agent_bins), 3)
```

```
428
429     def test_get_attrs(self):
430         """test get the correct attrs system"""
431
432         location = mk.create_autospec(agent_location.Location, spec_set=True)
433
434         attrs = mk.MagicMock(spec=dict)
435         attrs.__contains__.side_effect = [False, True]
436
437         # Test incorrect level
438         self.assertEqual(self.AgentsBin.get_attrs(location, attrs), {})
439         self.assertEqual(attrs.__contains__.call_args_list,
440                          [mk.call(location.level)])
441
442         attrs.reset_mock()
443         # Test incorrect level
444         self.assertEqual(self.AgentsBin.get_attrs(location, attrs),
445                          attrs.__getitem__.return_value)
446         self.assertEqual(attrs.__getitem__.call_args_list,
447                          [mk.call(location.level)])
448         self.assertEqual(attrs.__contains__.call_args_list,
449                          [mk.call(location.level)])
450
451     def test_empty(self):
452         """test create an empty class"""
453
454         agent_keys  = mk.MagicMock(spec=list)
455         location    = mk.create_autospec(agent_location.Location, spec_set=True)
456         attrs       = mk.MagicMock(spec=dict)
457         environment = mk.MagicMock(spec=tuple)
458
459         with mk.patch.object(agents.AgentsBin, 'get_attrs',
460                              autospec=True) as mkAttrs:
461             with mk.patch.object(agents.AgentsBin, 'make_bins',
462                                  autospec=True) as mkBins:
463                 with mk.patch.object(agents.AgentsBin, 'make_environment',
464                                      autospec=True) as mkEnvironment:
465                     mkBins.return_value        = self.agents
466                     mkEnvironment.return_value = self.environment
```

```
467
468                     self.AgentsBin = agents.AgentsBin.empty(agent_keys,
469                                                             location,
470                                                             attrs,
471                                                             environment)
472                     self.assertIsInstance(self.AgentsBin, agents.AgentsBin)
473                     self.assertEqual(self.AgentsBin.location_key,
474                                      location.location_key)
475                     self.assertEqual(self.AgentsBin.environment,
476                                      self.environment)
477                     self.assertEqual(self.AgentsBin,      self.agents)
478                     self.assertEqual(self.AgentsBin.data, self.agents)
479
480                     self.assertEqual(mkEnvironment.call_args_list,
481                                      [mk.call(location, environment)])
482                     self.assertEqual(mkBins.call_args_list,
483                                      [mk.call(agent_keys,
484                                               mkAttrs.return_value)])
485                     self.assertEqual(mkAttrs.call_args_list,
486                                      [mk.call(location, attrs)])
487
488
489 class TestAgents(ut.TestCase):
490     """test the Agents class"""
491
492     def setUp(self):
493         """Setup the tests"""
494
495         self.master = mk.create_autospec(agents.AgentsBin, spec_set=True)
496         self.agents = {mk.MagicMock(spec=tuple):
497                        mk.create_autospec(agents.AgentsBin, spec_set=True)
498                    for _ in range(3)}
499         self.agents[(0,)] = self.master
500
501         self.Agents = agents.Agents(self.agents)
502
503     def test___init__(self):
504         """test __init__ for class"""
505
```

```
        self.assertIsInstance(self.Agents, collect.UserDict)
        self.assertIsInstance(self.Agents, agents.Agents)

        self.assertEqual(self.Agents,      self.agents)
        self.assertEqual(self.Agents.data, self.agents)

        self.assertEqual(len(self.Agents), 4)

    def test_agents(self):
        """test get the master agents location"""

        agent_key = mk.MagicMock(spec=str)

        self.master.__getitem__.return_value = \
            mk.create_autospec(AgentBinTest, spec_set=True)

        self.assertEqual(self.Agents.agents(agent_key),
                         self.master.__getitem__.return_value.agents)
        self.assertEqual(self.master.__getitem__.call_args_list,
                         [mk.call(agent_key)])

    def test_activate(self):
        """test activate an agent"""

        location = agent_location.Location([0,
                                            mk.MagicMock(spec=int),
                                            mk.MagicMock(spec=int)])

        location_keys = [(0,)]
        for index in range(2, 4):
            key = location[:index].location_key
            self.Agents[key] = mk.create_autospec(agents.AgentsBin,
                                                  spec_set=True)
            location_keys.append(key)
        self.assertEqual(len(self.Agents), 6)

        agent = mk.create_autospec(AgentTest, spec_set=True)
        agent.location = location

```

```
        self.Agents.activate(agent)
        for location_key in self.Agents.keys():
            if location_key in location_keys:
                self.assertEqual(self.Agents[location_key].
                                 activate.call_args_list,
                                 [mk.call(agent)])
            else:
                self.assertEqual(self.Agents[location_key].
                                 activate.call_args_list,
                                 [])

    def test_deactivate(self):
        """test deactivate an agent"""

        location = agent_location.Location([0,
                                            mk.MagicMock(spec=int),
                                            mk.MagicMock(spec=int)])

        location_keys = [(0,)]
        for index in range(2, 4):
            key = location[:index].location_key
            self.Agents[key] = mk.create_autospec(agents.AgentsBin,
                                                  spec_set=True)
            location_keys.append(key)
        self.assertEqual(len(self.Agents), 6)

        agent = mk.create_autospec(AgentTest, spec_set=True)
        agent.location = location

        self.Agents.deactivate(agent)
        for location_key in self.Agents.keys():
            if location_key in location_keys:
                self.assertEqual(self.Agents[location_key].
                                 deactivate.call_args_list,
                                 [mk.call(agent)])
            else:
                self.assertEqual(self.Agents[location_key].
                                 deactivate.call_args_list,
                                 [])
```

```
584
585     def test_record(self):
586         """test record all the counts"""
587
588         self.Agents.record()
589         for agent_bin in self.agents.values():
590             self.assertEqual(agent_bin.record.call_args_list,
591                              [mk.call()])
592
593     def test_refresh(self):
594         """test refresh all the storage"""
595
596         self.Agents.refresh()
597         for agent_bin in self.agents.values():
598             self.assertEqual(agent_bin.refresh.call_args_list,
599                              [mk.call()])
600
601     def test_dataframes(self):
602         """test generate a dict of all dataframes"""
603
604         dataframes = {}
605         for agent_bin in self.agents.values():
606             data_dict = {mk.MagicMock(spec=str): mk.MagicMock()
607                          for _ in range(3)}
608             dataframes.update(data_dict)
609             agent_bin.dataframes.return_value = data_dict
610
611         self.assertEqual(len(dataframes), 12)
612
613         self.assertEqual(self.Agents.dataframes(), dataframes)
614         for agent_bin in self.agents.values():
615             self.assertEqual(agent_bin.dataframes.call_args_list,
616                              [mk.call()])
617
618     def test_empty(self):
619         """test create an empty agents system"""
620
621         locations  = [mk.create_autospec(agent_location.Location, spec_set=True)
622                       for _ in range(3)]
```

```
        space = mk.create_autospec(SpaceTest, spec_set=True)
        space.locations = locations

        agent_keys  = mk.MagicMock(spec=list)
        attrs       = mk.MagicMock(spec=dict)
        environment = mk.MagicMock(spec=tuple)

        agent_bins = [mk.create_autospec(agents.AgentsBin, spec_set=True)
                      for _ in range(3)]

        with mk.patch.object(agents.AgentsBin, 'empty',
                             autospec=True) as mkEmpty:
            mkEmpty.side_effect = agent_bins

            self.Agents = agents.Agents.empty(space, agent_keys,
                                              attrs, environment)
            self.assertIsInstance(self.Agents, agents.Agents)

            for index, location in enumerate(locations):
                location_key = location.location_key
                self.assertIn(location_key, self.Agents)
                self.assertEqual(self.Agents[location_key], agent_bins[index])
                self.assertEqual(mkEmpty.call_args_list[index],
                                 mk.call(agent_keys, location,
                                         attrs, environment))
            for index, things in enumerate(self.Agents.items()):
                location_key, agent_bin = things
                self.assertEqual(location_key, locations[index].location_key)
                self.assertEqual(agent_bin, agent_bins[index])
                self.assertEqual(mkEmpty.call_args_list[index],
                                 mk.call(agent_keys, locations[index],
                                         attrs, environment))
            self.assertEqual(len(self.Agents), 3)
```

C.5.11.2 test_environment.py

```
import unittest      as ut
import unittest.mock as mk

import dataclasses as dclass

import source.space.environment as environment


class TestEnvironment(ut.TestCase):
    """test the Environment class"""

    def setUp(self):
        """Setup the tests"""

        self.bt    = mk.MagicMock(spec=str)
        self.plant = mk.MagicMock(spec=str)

        self.Environment = environment.Environment(self.bt,
                                                   self.plant)

    def test___init__(self):
        """test __init__ for class"""

        self.assertIsInstance(self.Environment, environment.Environment)

        self.assertEqual(self.Environment.bt,    self.bt)
        self.assertEqual(self.Environment.plant, self.plant)

        self.assertTrue(dclass.is_dataclass(self.Environment))

    def test_setup(self):
        """test setup the class"""

        init_plant = mk.MagicMock(spec=callable)
        init_plant.return_value = self.plant

        self.Environment = environment.Environment.setup(self.bt, init_plant)
```

```
        self.assertIsInstance(self.Environment, environment.Environment)
        self.assertEqual(self.Environment.bt,    self.bt)
        self.assertEqual(self.Environment.plant, self.plant)

        self.assertEqual(init_plant.call_args_list,
                         [mk.call(self.bt)])
```

C.5.11.3 test_graph.py

```
import unittest      as ut
import unittest.mock as mk

import collections   as collect
import dataclasses   as dclass
import numpy         as np
import numpy.testing as utnp
import pickle        as pickle

import source.keyword as keyword

import source.space.graph as graph


class TestVertexNeighborhood(ut.TestCase):
    """test the VertexNeighborhood class"""

    def setUp(self):
        """Setup the tests"""

        self.vertex = mk.MagicMock(spec=int)

        self.neighborhood = {}
        for index in range(3):
            neighborhood = set()
            for _ in range(3):
                neighborhood.add(mk.MagicMock(spec=int))
            self.neighborhood[index] = neighborhood

        self.Neighborhood = graph.VertexNeighborhood(self.vertex,
                                                     self.neighborhood)

    def test___init__(self):
        """test __init__ for class"""

        self.assertIsInstance(self.Neighborhood, collect.UserDict)
        self.assertIsInstance(self.Neighborhood, graph.VertexNeighborhood)
```

```
38
39          self.assertEqual(self.Neighborhood.vertex, self.vertex)
40
41          for vertex, neighborhood in self.neighborhood.items():
42              self.assertIn(vertex, self.Neighborhood)
43              self.assertEqual(self.Neighborhood[vertex], neighborhood)
44          for vertex, neighborhood in self.Neighborhood.items():
45              self.assertIn(vertex, self.neighborhood)
46              self.assertEqual(self.neighborhood[vertex], neighborhood)
47          self.assertEqual(self.neighborhood, self.Neighborhood)
48
49      def test_minimum(self):
50          """test get minimum vertex distance"""
51
52          # Call tests
53          with mk.patch.object(graph.VertexNeighborhood, 'keys',
54                               autospec=True) as mkKeys:
55              with mk.patch('builtins.min') as mkMax:
56                  self.assertEqual(self.Neighborhood.minimum,
57                                   mkMax.return_value)
58                  self.assertEqual(mkMax.call_args_list,
59                                   [mk.call(mkKeys.return_value)])
60                  self.assertEqual(mkKeys.call_args_list,
61                                   [mk.call(self.neighborhood)])
62
63          # Practical test
64          self.assertEqual(self.Neighborhood.minimum, 0)
65
66      def test_maximum(self):
67          """test get maximum vertex distance"""
68
69          # Call tests
70          with mk.patch.object(graph.VertexNeighborhood, 'keys',
71                               autospec=True) as mkKeys:
72              with mk.patch('builtins.max') as mkMax:
73                  self.assertEqual(self.Neighborhood.maximum,
74                                   mkMax.return_value)
75                  self.assertEqual(mkMax.call_args_list,
76                                   [mk.call(mkKeys.return_value)])
```

```
                self.assertEqual(mkKeys.call_args_list,
                                 [mk.call(self.neighborhood)])

        # Practical test
        self.assertEqual(self.Neighborhood.maximum, 2)

    def test_add(self):
        """test set an vertex at a distance"""

        # Existing distance set
        vertices = {}
        for distance in self.neighborhood:
            vertices[distance] = mk.MagicMock(spec=int)

        for distance, vertex in vertices.items():
            self.assertNotIn(vertex, self.Neighborhood[distance])
            self.Neighborhood.add(distance, vertex)
            self.assertIn(vertex, self.Neighborhood[distance])
        for distance, neighborhood in self.Neighborhood.items():
            self.assertIn(vertices[distance], neighborhood)
            self.assertEqual(len(neighborhood), 4)
        self.assertEqual(len(self.Neighborhood), 3)

        # New distance set
        vertices = {}
        for _ in range(3):
            vertices[mk.MagicMock(spec=float)] = mk.MagicMock(spec=int)

        for distance, vertex in vertices.items():
            self.assertNotIn(distance, self.Neighborhood)
            self.Neighborhood.add(distance, vertex)
            self.assertIn(distance, self.Neighborhood)
            self.assertEqual(self.Neighborhood[distance], {vertex})
            self.assertEqual(len(self.Neighborhood[distance]), 1)
        self.assertEqual(len(self.Neighborhood), 6)

    def test__convert(self):
        """test convert a distance to those within the system"""

```

```
116         for dist in self.Neighborhood:
117             # Test distances above and below the distances in the class
118             self.assertEqual(dist, self.Neighborhood._convert(dist))
119             self.assertEqual(dist, self.Neighborhood._convert(dist + 0.1))
120             self.assertEqual(dist, self.Neighborhood._convert(dist + 0.2))
121             self.assertEqual(dist, self.Neighborhood._convert(dist + 0.3))
122             self.assertEqual(dist, self.Neighborhood._convert(dist + 0.4))
123             self.assertEqual(dist, self.Neighborhood._convert(dist + 0.5))
124             self.assertEqual(dist, self.Neighborhood._convert(dist - 0.1))
125             self.assertEqual(dist, self.Neighborhood._convert(dist - 0.2))
126             self.assertEqual(dist, self.Neighborhood._convert(dist - 0.3))
127 
128             # Test going above the distances in the class
129             self.assertEqual(self.Neighborhood.maximum, self.Neighborhood.
130                              _convert(dist + self.Neighborhood.maximum))
131 
132             # Test going below the distances in the class
133             self.assertEqual(self.Neighborhood.minimum, self.Neighborhood.
134                              _convert(dist - self.Neighborhood.maximum))
135 
136     def test__upper_lower(self):
137         """test get the upper and lower bounds for neighborhood"""
138 
139         upper = mk.MagicMock(spec=float)
140         lower = mk.MagicMock(spec=float)
141 
142         with mk.patch.object(graph.VertexNeighborhood, '_convert',
143                              autospec=True) as mkConvert:
144             # Pass in both
145             kwargs = {keyword.upper: upper,
146                       keyword.lower: lower}
147             convert = [mk.MagicMock(spec=float), mk.MagicMock(spec=float)]
148             mkConvert.side_effect = convert
149             self.assertEqual(self.Neighborhood._upper_lower(**kwargs),
150                              (convert[0], convert[1]))
151             self.assertEqual(mkConvert.call_args_list,
152                              [mk.call(self.Neighborhood, upper),
153                               mk.call(self.Neighborhood, lower)])
154 
```

```
            mkConvert.reset_mock()
            # Pass in only upper
            kwargs = {keyword.upper: upper}
            convert = [mk.MagicMock(spec=float)]
            mkConvert.side_effect = convert
            self.assertEqual(self.Neighborhood._upper_lower(**kwargs),
                             (convert[0], 0))
            self.assertEqual(mkConvert.call_args_list,
                             [mk.call(self.Neighborhood, upper)])

            mkConvert.reset_mock()
            # Pass in only lower
            kwargs = {keyword.lower: lower}
            convert = [mk.MagicMock(spec=float)]
            mkConvert.side_effect = convert
            self.assertEqual(self.Neighborhood._upper_lower(**kwargs),
                             (np.inf, convert[0]))
            self.assertEqual(mkConvert.call_args_list,
                             [mk.call(self.Neighborhood, lower)])

            mkConvert.reset_mock()
            # Pass in nothing
            self.assertEqual(self.Neighborhood._upper_lower(),
                             (np.inf, 0))
            self.assertEqual(mkConvert.call_args_list, [])

    def test__append_distance(self):
        """test append correct distances"""

        distance = mk.MagicMock(spec=float)
        upper    = mk.MagicMock(spec=float)
        lower    = mk.MagicMock(spec=float)
        union    = mk.MagicMock(spec=set)
        vertices = mk.MagicMock(spec=set)

        lower_test = [False, True,  True, True]
        upper_test = [       False, True, True]

        lower.   __le__.side_effect = lower_test
```

```
        distance.__le__.side_effect = upper_test

        # Less than lower
        self.assertEqual(self.Neighborhood.
                         _append_distance(distance, upper, lower,
                                          union, vertices),
                         vertices)
        self.assertEqual(vertices.union.call_args_list, [])
        self.assertEqual(lower.   __le__.call_args_list, [mk.call(distance)])
        self.assertEqual(distance.__le__.call_args_list, [])

        lower.__le__.reset_mock()
        # Greater than upper
        self.assertEqual(self.Neighborhood.
                         _append_distance(distance, upper, lower,
                                          union, vertices),
                         vertices)
        self.assertEqual(vertices.union.call_args_list, [])
        self.assertEqual(lower.   __le__.call_args_list, [mk.call(distance)])
        self.assertEqual(distance.__le__.call_args_list, [mk.call(upper)])

        lower.   __le__.reset_mock()
        distance.__le__.reset_mock()
        # Correct bounds
        self.assertEqual(self.Neighborhood.
                         _append_distance(distance, upper, lower,
                                          union, vertices),
                         vertices.union.return_value)
        self.assertEqual(vertices.union.call_args_list,
                         [mk.call(union)])
        self.assertEqual(lower.   __le__.call_args_list, [mk.call(distance)])
        self.assertEqual(distance.__le__.call_args_list, [mk.call(upper)])

        lower.   __le__.reset_mock()
        distance.__le__.reset_mock()
        vertices = {mk.MagicMock(), mk.MagicMock()}
        union    = {mk.MagicMock(), mk.MagicMock()}
        # Test union
        result = self.Neighborhood._append_distance(distance, upper, lower,
```

```
                                                union, vertices)
        self.assertNotEqual(result, vertices)
        self.assertNotEqual(result, union)
        self.assertEqual(len(result), 4)
        for vertex in vertices:
            self.assertIn(vertex, result)
        for vertex in union:
            self.assertIn(vertex, result)
        for vertex in vertices:
            self.assertTrue((vertex in vertices) or (vertex in union))

    def test_neighborhood(self):
        """test get the neighborhood"""

        kwargs = {'test': mk.MagicMock()}

        upper = mk.MagicMock(spec=float)
        lower = mk.MagicMock(spec=float)
        upper_lower = (upper, lower)

        vertices = []
        for _ in self.Neighborhood:
            vertices.append(mk.MagicMock(spec=set))
        self.assertEqual(len(vertices), 3)

        with mk.patch.object(graph.VertexNeighborhood, '_upper_lower',
                             autospec=True) as mkGet:
            with mk.patch.object(graph.VertexNeighborhood,
                                 '_append_distance') as mkAppend:
                mkGet.return_value    = upper_lower
                mkAppend.side_effect = vertices

                self.assertEqual(self.Neighborhood.neighborhood(**kwargs),
                                 vertices[-1])
                self.assertEqual(mkGet.call_args_list,
                                 [mk.call(self.Neighborhood, **kwargs)])

                keys   = list(self.Neighborhood.keys())
                values = list(self.Neighborhood.values())
```

```
                tmp = [set()]
                tmp.extend(vertices)
                vertices = tmp
                for index, call in enumerate(mkAppend.call_args_list):
                    self.assertEqual(call, mk.call(keys[index],
                                                   upper,
                                                   lower,
                                                   values[index],
                                                   vertices[index]))
                self.assertEqual(len(mkAppend.call_args_list), 3)

    def test_empty(self):
        """test generate empty Vertex neighborhood"""

        self.Neighborhood = graph.VertexNeighborhood.empty(self.vertex)
        self.assertIsInstance(self.Neighborhood, graph.VertexNeighborhood)
        self.assertEqual(self.Neighborhood.vertex, self.vertex)
        self.assertEqual(self.Neighborhood, {})


class TestVertexDistance(ut.TestCase):
    """test the VertexDistance class"""

    def setUp(self):
        """Setup the tests"""

        self.vertex = mk.MagicMock(spec=int)

        self.distances = {}
        for index in range(3):
            self.distances[index] = mk.MagicMock(spec=float)

        self.Distance = graph.VertexDistance(self.vertex, self.distances)

    def test___init__(self):
        """test __init__ for class"""

        self.assertIsInstance(self.Distance, collect.UserDict)
        self.assertIsInstance(self.Distance, graph.VertexDistance)
```

```
311
312         self.assertEqual(self.Distance.vertex, self.vertex)
313
314         for vertex, distance in self.distances.items():
315             self.assertIn(vertex, self.Distance)
316             self.assertEqual(self.Distance[vertex], distance)
317         for vertex, distance in self.Distance.items():
318             self.assertIn(vertex, self.distances)
319             self.assertEqual(self.distances[vertex], distance)
320         self.assertEqual(self.Distance, self.distances)
321
322     def test_minimum(self):
323         """test get the vertex with minimum distance"""
324
325         distances = {1: 9, 2: 1, 3: 7, 6: 4, 8: 2}
326         self.Distance = graph.VertexDistance(self.vertex, distances)
327         self.assertEqual(self.Distance.minimum, 2)
328
329         distances = {1: 9, 2: 4, 3: 7, 6: 4, 8: 2}
330         self.Distance = graph.VertexDistance(self.vertex, distances)
331         self.assertEqual(self.Distance.minimum, 8)
332
333         distances = {1: 9, 2: 4, 3: 7, 6: 4, 8: 12}
334         self.Distance = graph.VertexDistance(self.vertex, distances)
335         self.assertEqual(self.Distance.minimum, 2)
336
337     def test_radius(self):
338         """test get the minimum distance"""
339
340         with mk.patch('builtins.min') as mkMin:
341             with mk.patch.object(graph.VertexDistance, 'values',
342                                  autospec=True) as mkValues:
343                 self.assertEqual(self.Distance.radius,
344                                  mkMin.return_value)
345                 self.assertEqual(mkMin.call_args_list,
346                                  [mk.call(mkValues.return_value)])
347                 self.assertEqual(mkValues.call_args_list,
348                                  [mk.call(self.distances)])
349
```

```
    def test_add(self):
        """test set the distance"""

        # Does not exist
        vertex   = mk.MagicMock(spec=int)
        distance = mk.MagicMock(spec=float)
        distance.__lt__.return_value = True
        self.assertNotIn(vertex, self.Distance)
        self.Distance.add(vertex, distance)
        self.assertIn(vertex, self.Distance)
        self.assertEqual(self.Distance[vertex], distance)
        self.assertEqual(len(self.Distance), 4)

        # Change inf distance
        vertex   = mk.MagicMock(spec=int)
        distance = mk.MagicMock(spec=float)
        distance.__lt__.return_value = True
        self.assertNotIn(vertex, self.Distance)
        self.Distance[vertex] = np.inf
        self.assertIn(vertex, self.Distance)
        self.assertEqual(self.Distance[vertex], np.inf)
        self.Distance.add(vertex, distance)
        self.assertEqual(self.Distance[vertex], distance)
        self.assertEqual(len(self.Distance), 5)

        # Exists but is not smaller
        for vertex in self.Distance:
            distance = mk.MagicMock(spec=float)
            distance.__lt__.return_value = False
            self.assertIn(vertex, self.Distance)
            self.Distance.add(vertex, distance)
            self.assertIn(vertex, self.Distance)
            self.assertNotEqual(self.Distance[vertex], distance)
        self.assertEqual(len(self.Distance), 5)

    def test_convert(self):
        """test convert to neighborhood"""

        neighborhood = self.Distance.convert()
```

```
        self.assertIsInstance(neighborhood, graph.VertexNeighborhood)
        self.assertEqual(neighborhood.vertex, self.vertex)
        for distance, hood in neighborhood.items():
            self.assertIsInstance(hood, set)
            for vertex in hood:
                self.assertEqual(self.Distance[vertex], distance)
        for vertex, distance in self.Distance.items():
            self.assertIn(distance, neighborhood)
            self.assertEqual(neighborhood[distance], {vertex})
        self.assertEqual(len(neighborhood), len(self.Distance))

    def test_distances(self):
        """test get a subset of the distances"""

        all_vertices = list(self.Distance.keys())
        for index in range(3):
            vertices = set(all_vertices[:index + 1])
            distances = self.Distance.distances(vertices)
            self.assertIsInstance(distances, graph.VertexDistance)
            self.assertEqual(distances.vertex, self.vertex)
            for vertex, distance in distances.items():
                self.assertIn(vertex, self.Distance)
                self.assertEqual(self.Distance[vertex], distance)
            self.assertEqual(len(distances), index + 1)

    def test_empty(self):
        """test generate an empty class"""

        vertices = set()
        for _ in range(3):
            vertices.add(mk.MagicMock(spec=int))

        self.Distance = graph.VertexDistance.empty(self.vertex, vertices)
        self.assertIsInstance(self.Distance, graph.VertexDistance)
        self.assertEqual(self.Distance.vertex, self.vertex)
        for vertex in vertices:
            self.assertIn(vertex, self.Distance)
            self.assertEqual(self.Distance[vertex], np.inf)
        for vertex, distance in self.Distance.items():
```

```
            self.assertIn(vertex, vertices)
            self.assertEqual(distance, np.inf)
        self.assertEqual(len(self.Distance), len(vertices))

        # Test practical convert
        neighborhood = self.Distance.convert()
        self.assertIsInstance(neighborhood, graph.VertexNeighborhood)
        self.assertEqual(neighborhood.vertex, self.vertex)
        for distance, hood in neighborhood.items():
            self.assertEqual(distance, np.inf)
            self.assertEqual(hood, vertices)
        self.assertEqual(len(neighborhood), 1)


class TestGraphNeighborhood(ut.TestCase):
    """test the GraphNeighborhood class"""

    def setUp(self):
        """Setup the tests"""

        self.neighborhoods = {}
        for index in range(3):
            self.neighborhoods[index] = \
                mk.create_autospec(graph.VertexNeighborhood,
                                   spec_set=True)

        self.Neighborhood = graph.GraphNeighborhood(self.neighborhoods)

    def test___init__(self):
        """test __init__ for class"""

        self.assertIsInstance(self.Neighborhood, collect.UserDict)
        self.assertIsInstance(self.Neighborhood, graph.GraphNeighborhood)

        for vertex, neighborhood in self.neighborhoods.items():
            self.assertIn(vertex, self.Neighborhood)
            self.assertEqual(self.Neighborhood[vertex], neighborhood)
        for vertex, neighborhood in self.Neighborhood.items():
            self.assertIn(vertex, self.neighborhoods)
```

```
            self.assertEqual(self.neighborhoods[vertex], neighborhood)
        self.assertEqual(self.Neighborhood, self.neighborhoods)

    def test_minimum(self):
        """test get the minimum distance in the graph"""

        minimum = []
        for neighborhood in self.neighborhoods.values():
            this = mk.MagicMock(spec=float)
            neighborhood.minimum = this
            minimum.append(this)

        with mk.patch('builtins.min') as mkMin:
            self.assertEqual(self.Neighborhood.minimum,
                             mkMin.return_value)
            self.assertEqual(mkMin.call_args_list,
                             [mk.call(minimum)])

    def test_maximum(self):
        """test get the maximum distance in the graph"""

        maximum = []
        for neighborhood in self.neighborhoods.values():
            this = mk.MagicMock(spec=float)
            neighborhood.maximum = this
            maximum.append(this)

        with mk.patch('builtins.max') as mkMax:
            self.assertEqual(self.Neighborhood.maximum,
                             mkMax.return_value)
            self.assertEqual(mkMax.call_args_list,
                             [mk.call(maximum)])

    def test_add(self):
        """test add a distance between vertices"""

        # Existing distance set
        pairs = {}
        for vertex in self.neighborhoods:
```

```
            pairs[vertex] = (mk.MagicMock(spec=int), mk.MagicMock(spec=float))

        for vertex, pair in pairs.items():
            self.Neighborhood.add(vertex, pair)
            self.assertEqual(self.Neighborhood[vertex].
                             add.call_args_list,
                             [mk.call(pair[1], pair[0])])
        for vertex, neighborhood in self.Neighborhood.items():
            self.assertEqual(neighborhood.add.call_args_list,
                             [mk.call(pairs[vertex][1], pairs[vertex][0])])
            neighborhood.reset_mock()

        # New distance set
        with mk.patch.object(graph.VertexNeighborhood, 'empty') as mkEmpty:
            mkEmpty.return_value = mk.create_autospec(graph.GraphNeighborhood,
                                                      spec_set=True)
            vertex = mk.MagicMock(spec=int)
            pair = (mk.MagicMock(spec=int), mk.MagicMock(spec=float))

            self.assertNotIn(vertex, self.Neighborhood)
            self.Neighborhood.add(vertex, pair)
            self.assertIn(vertex, self.Neighborhood)
            self.assertEqual(self.Neighborhood[vertex],
                             mkEmpty.return_value)
            self.assertEqual(mkEmpty.call_args_list,
                             [mk.call(vertex)])
            self.assertEqual(mkEmpty.return_value.add.call_args_list,
                             [mk.call(pair[1], pair[0])])

    def test_neighborhood(self):
        """test get the neighborhood defined by the vertex and distances"""

        kwargs = {'test': mk.MagicMock()}

        for vertex in self.Neighborhood:
            self.assertEqual(self.Neighborhood.neighborhood(vertex, **kwargs),
                             self.neighborhoods[vertex].
                                  neighborhood.return_value)
            self.assertEqual(self.neighborhoods[vertex].
```

```
                                    neighborhood.call_args_list,
                                [mk.call(**kwargs)])


class TestGraphDistance(ut.TestCase):
    """test the GraphDistance class"""

    def setUp(self):
        """Setup the tests"""

        self.distances = {}
        for index in range(3):
            self.distances[index] = mk.create_autospec(graph.VertexDistance,
                                                       spec_set=True)

        self.Distance = graph.GraphDistance(self.distances)

    def test___init__(self):
        """test __init__ for class"""

        self.assertIsInstance(self.Distance, collect.UserDict)
        self.assertIsInstance(self.Distance, graph.GraphDistance)

        for vertex, distances in self.distances.items():
            self.assertIn(vertex, self.Distance)
            self.assertEqual(self.Distance[vertex], distances)
        for vertex, distances in self.Distance.items():
            self.assertIn(vertex, self.distances)
            self.assertEqual(self.distances[vertex], distances)
        self.assertEqual(self.Distance, self.distances)

    def test_add(self):
        """test add an item in class"""

        # Existing distance set
        pairs = {}
        for vertex in self.distances:
            pairs[vertex] = (mk.MagicMock(spec=int), mk.MagicMock(spec=float))

```

```
        for vertex, pair in pairs.items():
            self.Distance.add(vertex, pair)
            self.assertEqual(self.Distance[vertex].add.call_args_list,
                             [mk.call(pair[0], pair[1])])
        for vertex, distances in self.Distance.items():
            self.assertEqual(distances.add.call_args_list,
                             [mk.call(pairs[vertex][0], pairs[vertex][1])])
            distances.reset_mock()

        # New distance set
        with mk.patch.object(graph.VertexDistance, 'empty') as mkEmpty:
            mkEmpty.return_value = mk.create_autospec(graph.VertexDistance,
                                                      spec_set=True)
            vertex = mk.MagicMock(spec=int)
            pair   = (mk.MagicMock(spec=int), mk.MagicMock(spec=float))

            self.assertNotIn(vertex, self.Distance)
            self.Distance.add(vertex, pair)
            self.assertIn(vertex, self.Distance)
            self.assertEqual(self.Distance[vertex],
                             mkEmpty.return_value)
            self.assertEqual(mkEmpty.call_args_list,
                             [mk.call(vertex, set())])
            self.assertEqual(mkEmpty.return_value.add.call_args_list,
                             [mk.call(pair[0], pair[1])])

    def test_convert(self):
        """test convert to a graph neighborhood"""

        neighborhoods = self.Distance.convert()
        self.assertIsInstance(neighborhoods, graph.GraphNeighborhood)
        for vertex, neighborhood in neighborhoods.items():
            self.assertIn(vertex, self.Distance)
            self.assertEqual(neighborhood,
                             self.Distance[vertex].convert.return_value)
            self.assertEqual(self.Distance[vertex].convert.call_args_list,
                             [mk.call()])
        for vertex, distance in self.Distance.items():
            self.assertIn(vertex, neighborhoods)
```

```
            self.assertEqual(neighborhoods[vertex],
                             distance.convert.return_value)
            self.assertEqual(distance.convert.call_args_list,
                             [mk.call()])
        self.assertEqual(len(neighborhoods), len(self.Distance))

    def test_empty(self):
        """test create an empty class"""

        vertices = set()
        for _ in range(3):
            vertices.add(mk.MagicMock(spec=int))

        self.Distance = graph.GraphDistance.empty(vertices)
        self.assertIsInstance(self.Distance, graph.GraphDistance)
        for vertex in vertices:
            self.assertIn(vertex, self.Distance)
            self.assertIsInstance(self.Distance[vertex], graph.VertexDistance)
            self.assertEqual(self.Distance[vertex].vertex, vertex)
            for this, dist in self.Distance[vertex].items():
                self.assertIn(this, vertices)
                self.assertEqual(dist, np.inf)
            self.assertEqual(len(self.Distance[vertex]), len(vertices))
        self.assertEqual(len(vertices), 3)
        for vertex, distance in self.Distance.items():
            self.assertIn(vertex, vertices)
            self.assertIsInstance(distance, graph.VertexDistance)
            self.assertEqual(distance.vertex, vertex)
            for this, dist in distance.items():
                self.assertIn(this, vertices)
                self.assertEqual(dist, np.inf)
            self.assertEqual(len(distance), len(vertices))
        self.assertEqual(len(self.Distance), len(vertices))

        # Test practical convert
        neighborhoods = self.Distance.convert()
        self.assertIsInstance(neighborhoods, graph.GraphNeighborhood)
        for vertex, neighborhood in neighborhoods.items():
            self.assertIn(vertex, self.Distance)
```

```
            self.assertIsInstance(neighborhood, graph.VertexNeighborhood)
            self.assertEqual(neighborhood.vertex, vertex)
            for distance, hood in neighborhood.items():
                self.assertEqual(distance, np.inf)
                self.assertEqual(hood, vertices)
            self.assertEqual(len(neighborhood), 1)
        self.assertEqual(len(neighborhoods), len(vertices))


class VertexDistanceTest(graph.VertexDistance):
    """Class to add dynamic values for tests"""

    vertex = mk.MagicMock(spec=int)


class TestAdjacency(ut.TestCase):
    """test the Adjacency class"""

    def setUp(self):
        """Setup the tests"""

        self.matrix = []
        for _ in range(3):
            row = []
            for _ in range(3):
                row.append(mk.MagicMock(spec=float))
            self.matrix.append(row)

        self.Adjacency = graph.Adjacency(self.matrix)

    def test___init__(self):
        """test __init__ for class"""

        self.assertIsInstance(self.Adjacency, np.ndarray)
        self.assertIsInstance(self.Adjacency, graph.Adjacency)

        utnp.assert_array_equal(self.Adjacency, self.matrix)

    def test_num(self):
```

```
        """test get the number of vertices"""

        with mk.patch.object(graph.Adjacency, 'shape',
                             autospec=True) as mkShape:
            self.assertEqual(self.Adjacency.num,
                             mkShape.__getitem__.return_value)
            self.assertEqual(mkShape.__getitem__.call_args_list,
                             [mk.call(0)])

        # Practical
        self.assertEqual(self.Adjacency.num, 3)

    def test_vertices(self):
        """test the vertex set"""

        with mk.patch.object(graph.Adjacency, 'num', autospec=True) as mkNum:
            with mk.patch.object(graph, 'set') as mkSet:
                with mk.patch.object(graph, 'range') as mkRange:
                    self.assertEqual(self.Adjacency.vertices,
                                     mkSet.return_value)
                    self.assertEqual(mkSet.call_args_list,
                                     [mk.call(mkRange.return_value)])
                    self.assertEqual(mkRange.call_args_list,
                                     [mk.call(mkNum)])

        # Practical
        self.assertEqual(self.Adjacency.vertices, {0, 1, 2})

    def test__start_search(self):
        """test start a search"""

        distance = mk.create_autospec(VertexDistanceTest, spec_set=True)

        with mk.patch.object(graph.Adjacency, 'vertices',
                             autospec=True) as mkVertices:
            self.assertEqual(self.Adjacency._start_search(distance),
                             mkVertices)
            self.assertEqual(distance.add.call_args_list,
                             [mk.call(distance.vertex, 0)])
```

```
740
741     def test__minimum(self):
742         """test get the minimum distance within an unvisited set"""
743
744         unvisited = mk.MagicMock(spec=set)
745         distance  = mk.create_autospec(VertexDistanceTest, spec_set=True)
746         distance.distances.return_value = \
747             mk.create_autospec(VertexDistanceTest, spec_set=True)
748
749         self.assertEqual(self.Adjacency._minimum(unvisited, distance),
750                          distance.distances.return_value.radius)
751         self.assertEqual(distance.distances.call_args_list,
752                          [mk.call(unvisited)])
753
754     def test__terminate(self):
755         """test the terminate conditions check"""
756
757         unvisited = {mk.MagicMock(spec=int)}
758         distance  = mk.create_autospec(VertexDistanceTest, spec_set=True)
759
760         with mk.patch.object(graph.Adjacency, '_minimum') as mkMinimum:
761             mkMinimum.side_effect = [0, np.inf]
762
763             # No termination
764             self.assertFalse(self.Adjacency._terminate(unvisited, distance))
765             self.assertEqual(mkMinimum.call_args_list,
766                              [mk.call(unvisited, distance)])
767
768             mkMinimum.reset_mock()
769             # Minimum termination
770             self.assertTrue(self.Adjacency._terminate(unvisited, distance))
771             self.assertEqual(mkMinimum.call_args_list,
772                              [mk.call(unvisited, distance)])
773
774             mkMinimum.reset_mock()
775             # Empty unvisited termination
776             self.assertTrue(self.Adjacency._terminate(set(), distance))
777             self.assertEqual(mkMinimum.call_args_list, [])
778
```

```
779     def test__current(self):
780         """test choose the next vertex to search"""
781 
782         unvisited = mk.MagicMock(spec=set)
783         distance  = mk.create_autospec(VertexDistanceTest, spec_set=True)
784         distance.distances.return_value = \
785             mk.create_autospec(VertexDistanceTest, spec_set=True)
786 
787         self.assertEqual(self.Adjacency._current(unvisited, distance),
788                          distance.distances.return_value.minimum)
789         self.assertEqual(distance.distances.call_args_list,
790                          [mk.call(unvisited)])
791 
792     def test__adjacent(self):
793         """test get the adjacent vertices"""
794 
795         matrix = [[0, 1, 1, 1],
796                   [0, 0, 1, 1],
797                   [0, 0, 0, 1],
798                   [0, 0, 0, 0]]
799         self.Adjacency = graph.Adjacency(matrix)
800 
801         for vertex in self.Adjacency.vertices:
802             self.assertEqual(self.Adjacency._adjacent(vertex),
803                              set(range(vertex + 1, 4)))
804 
805     def test__visit(self):
806         """test get the vertices to visit"""
807 
808         current   = mk.MagicMock(spec=int)
809         unvisited = mk.MagicMock(spec=set)
810 
811         with mk.patch.object(graph.Adjacency, '_adjacent',
812                              autospec=True) as mkAdjacent:
813             self.assertEqual(self.Adjacency._visit(current, unvisited),
814                              unvisited.intersection.return_value)
815             self.assertEqual(unvisited.intersection.call_args_list,
816                              [mk.call(mkAdjacent.return_value)])
817             self.assertEqual(mkAdjacent.call_args_list,
```

```
                            [mk.call(self.Adjacency, current)])

        mkAdjacent.return_value = {1, 2, 4}
        unvisited               = {0, 1, 2, 3}
        self.assertEqual(self.Adjacency._visit(current, unvisited),
                         {1, 2})

    def test__distance(self):
        """test get the distance"""

        current  = mk.MagicMock(spec=int)
        vertex   = mk.MagicMock(spec=int)
        distance = mk.create_autospec(VertexDistanceTest, spec_set=True)

        with mk.patch.object(graph.Adjacency, '__getitem__',
                             autospec=True) as mkGet:
            mkGet.return_value = mk.MagicMock(spec=np.ndarray)
            self.assertEqual(self.Adjacency._distance(current, vertex,
                                                      distance),
                             distance.__getitem__.return_value.
                                      __add__.return_value)
            self.assertEqual(distance.__getitem__.return_value.
                                      __add__.call_args_list,
                             [mk.call(mkGet.return_value.
                                           __getitem__.return_value)])
            self.assertEqual(distance.__getitem__.call_args_list,
                             [mk.call(current)])
            self.assertEqual(mkGet.return_value.__getitem__.call_args_list,
                             [mk.call(vertex)])
            self.assertEqual(mkGet.call_args_list,
                             [mk.call(current)])
            distance.reset_mock()

        # Practical
        for current in range(3):
            for vertex in range(3):
                self.assertEqual(self.Adjacency._distance(current, vertex,
                                                          distance),
                                 distance.__getitem__.return_value.
```

```
                                        __add__.return_value)
            self.assertEqual(distance.__getitem__.return_value.
                                        __add__.call_args_list,
                             [mk.call(self.matrix[current][vertex])])
            self.assertEqual(distance.__getitem__.call_args_list,
                             [mk.call(current)])
            distance.reset_mock()

    def test__update(self):
        """test update the distances"""

        current  = mk.MagicMock(spec=int)
        distance = mk.create_autospec(VertexDistanceTest, spec_set=True)
        visit    = set()
        new      = []
        for _ in range(3):
            visit.add(mk.MagicMock(spec=int))
            new.append(mk.MagicMock(spec=float))

        with mk.patch.object(graph.Adjacency, '_distance',
                             autospec=True) as mkDistance:
            mkDistance.side_effect = new

            self.Adjacency._update(current, visit, distance)
            for index, vertex in enumerate(visit):
                self.assertEqual(distance.add.call_args_list[index],
                                 mk.call(vertex, new[index]))
                self.assertEqual(mkDistance.call_args_list[index],
                                 mk.call(self.Adjacency, current, vertex,
                                         distance))
            self.assertEqual(len(distance.add.call_args_list), 3)
            self.assertEqual(len(mkDistance.call_args_list), 3)

    def test__search(self):
        """test perform one search step"""

        unvisited = mk.MagicMock(spec=set)
        distance  = mk.create_autospec(VertexDistanceTest, spec_set=True)

```

```
        with mk.patch.object(graph.Adjacency, '_current') as mkCurrent:
            with mk.patch.object(graph.Adjacency, '_visit',
                                 autospec=True) as mkVisit:
                with mk.patch.object(graph.Adjacency, '_update',
                                     autospec=True) as mkUpdate:
                    self.Adjacency._search(unvisited, distance)
                    self.assertEqual(unvisited.remove.call_args_list,
                                     [mk.call(mkCurrent.return_value)])
                    self.assertEqual(mkUpdate.call_args_list,
                                     [mk.call(self.Adjacency,
                                              mkCurrent.return_value,
                                              mkVisit.return_value,
                                              distance)])
                    self.assertEqual(mkVisit.call_args_list,
                                     [mk.call(self.Adjacency,
                                              mkCurrent.return_value,
                                              unvisited)])
                    self.assertEqual(mkCurrent.call_args_list,
                                     [mk.call(unvisited, distance)])

    def test__dijkstra(self):
        """test run dijkstra on single vertex"""

        distance = mk.create_autospec(VertexDistanceTest, spec_set=True)

        with mk.patch.object(graph.Adjacency, '_start_search',
                             autospec=True) as mkStart:
            with mk.patch.object(graph.Adjacency, 'num',
                                 autospec=True) as mkNum:
                with mk.patch.object(graph.Adjacency, '_terminate',
                                     autospec=True) as mkTerminate:
                    with mk.patch.object(graph.Adjacency, '_search',
                                         autospec=True) as mkSearch:
                        mkNum.__get__ = mk.MagicMock(return_value=3)

                        # Error (too many searches)
                        mkTerminate.side_effect = [False, False, False, False]
                        with self.assertRaisesRegex(RuntimeError,
                                                    'Search of vertex {} has '
```

```
                                        'failed'.
                                         format(distance.vertex)):
                    self.Adjacency._dijkstra(distance)
                for call in mkTerminate.call_args_list:
                    self.assertEqual(call, mk.call(self.Adjacency,
                                                   mkStart.return_value,
                                                   distance))
                for call in mkSearch.call_args_list:
                    self.assertEqual(call, mk.call(self.Adjacency,
                                                   mkStart.return_value,
                                                   distance))
                self.assertEqual(len(mkTerminate.call_args_list), 4)
                self.assertEqual(len(mkSearch.call_args_list),    4)
                self.assertEqual(mkStart.call_args_list,
                                 [mk.call(self.Adjacency, distance)])

                mkStart.reset_mock()
                mkTerminate.reset_mock()
                mkSearch.reset_mock()
                # Maximum amount (3) of searches
                mkTerminate.side_effect = [False, False, False, True]
                self.Adjacency._dijkstra(distance)
                for call in mkTerminate.call_args_list:
                    self.assertEqual(call, mk.call(self.Adjacency,
                                                   mkStart.return_value,
                                                   distance))
                for call in mkSearch.call_args_list:
                    self.assertEqual(call, mk.call(self.Adjacency,
                                                   mkStart.return_value,
                                                   distance))
                self.assertEqual(len(mkTerminate.call_args_list), 4)
                self.assertEqual(len(mkSearch.call_args_list),    3)
                self.assertEqual(mkStart.call_args_list,
                                 [mk.call(self.Adjacency, distance)])

                mkStart.reset_mock()
                mkTerminate.reset_mock()
                mkSearch.reset_mock()
                # Amount (2) searches
```

```
                        mkTerminate.side_effect = [False, False, True]
                        self.Adjacency._dijkstra(distance)
                        for call in mkTerminate.call_args_list:
                            self.assertEqual(call, mk.call(self.Adjacency,
                                                           mkStart.return_value,
                                                           distance))
                        for call in mkSearch.call_args_list:
                            self.assertEqual(call, mk.call(self.Adjacency,
                                                           mkStart.return_value,
                                                           distance))
                        self.assertEqual(len(mkTerminate.call_args_list), 3)
                        self.assertEqual(len(mkSearch.call_args_list),    2)
                        self.assertEqual(mkStart.call_args_list,
                                         [mk.call(self.Adjacency, distance)])

                        mkStart.reset_mock()
                        mkTerminate.reset_mock()
                        mkSearch.reset_mock()
                        # Amount (1) searches
                        mkTerminate.side_effect = [False, True]
                        self.Adjacency._dijkstra(distance)
                        for call in mkTerminate.call_args_list:
                            self.assertEqual(call, mk.call(self.Adjacency,
                                                           mkStart.return_value,
                                                           distance))
                        for call in mkSearch.call_args_list:
                            self.assertEqual(call, mk.call(self.Adjacency,
                                                           mkStart.return_value,
                                                           distance))
                        self.assertEqual(len(mkTerminate.call_args_list), 2)
                        self.assertEqual(len(mkSearch.call_args_list),    1)
                        self.assertEqual(mkStart.call_args_list,
                                         [mk.call(self.Adjacency, distance)])

                        mkStart.reset_mock()
                        mkTerminate.reset_mock()
                        mkSearch.reset_mock()
                        # Amount (1) searches
                        mkTerminate.side_effect = [True]
```

```
                        self.Adjacency._dijkstra(distance)
                        for call in mkTerminate.call_args_list:
                            self.assertEqual(call, mk.call(self.Adjacency,
                                                           mkStart.return_value,
                                                           distance))
                        for call in mkSearch.call_args_list:
                            self.assertEqual(call, mk.call(self.Adjacency,
                                                           mkStart.return_value,
                                                           distance))
                        self.assertEqual(len(mkTerminate.call_args_list), 1)
                        self.assertEqual(len(mkSearch.call_args_list),    0)
                        self.assertEqual(mkStart.call_args_list,
                                         [mk.call(self.Adjacency, distance)])

    def test_dijkstra(self):
        """test run Dijkstra's for whole graph"""

        distance = {}
        for _ in range(3):
            distance[mk.MagicMock()] = mk.create_autospec(VertexDistanceTest,
                                                          spec_set=True)

        with mk.patch.object(graph.Adjacency, '_dijkstra',
                             autospec=True) as mkDijkstra:
            with mk.patch.object(graph.Adjacency, 'vertices',
                                 autospec=True) as mkVertices:
                with mk.patch.object(graph.GraphDistance, 'empty') as mkEmpty:
                    mkEmpty.return_value = mk.create_autospec(graph.
                                                              GraphDistance,
                                                              spec_set=True)
                    mkEmpty.return_value.values.return_value = distance.values()

                    self.assertEqual(self.Adjacency.dijkstra(),
                                     mkEmpty.return_value)
                    self.assertEqual(mkEmpty.call_args_list,
                                     [mk.call(mkVertices)])
                    distances = list(distance.values())
                    for index, call in enumerate(mkDijkstra.call_args_list):
                        self.assertEqual(call,
```

```
                                        mk.call(self.Adjacency,
                                                distances[index]))
                    self.assertEqual(len(mkDijkstra.call_args_list), 3)

        # Practical construction test

        # GRAPH
        # 0   1
        #     |
        #   .-2-.
        #   |   |
        #   3---4
        #   |   |
        #   5---6

        matrix = [[0, 0, 0, 0, 0, 0, 0],
                  [0, 0, 1, 0, 0, 0, 0],
                  [0, 1, 0, 1, 1, 0, 0],
                  [0, 0, 1, 0, 1, 1, 0],
                  [0, 0, 1, 1, 0, 0, 1],
                  [0, 0, 0, 1, 0, 0, 1],
                  [0, 0, 0, 0, 1, 1, 0]]
        self.Adjacency = graph.Adjacency(matrix)
        distances = self.Adjacency.dijkstra()

        distance_0 = {0: 0,
                      1: np.inf,
                      2: np.inf,
                      3: np.inf,
                      4: np.inf,
                      5: np.inf,
                      6: np.inf}
        self.assertEqual(distances[0], distance_0)
        distance_1 = {0: np.inf,
                      1: 0,
                      2: 1,
                      3: 2,
                      4: 2,
                      5: 3,
```

```
                        6: 3}
        self.assertEqual(distances[1], distance_1)
        distance_2 = {0: np.inf,
                      1: 1,
                      2: 0,
                      3: 1,
                      4: 1,
                      5: 2,
                      6: 2}
        self.assertEqual(distances[2], distance_2)
        distance_3 = {0: np.inf,
                      1: 2,
                      2: 1,
                      3: 0,
                      4: 1,
                      5: 1,
                      6: 2}
        self.assertEqual(distances[3], distance_3)
        distance_4 = {0: np.inf,
                      1: 2,
                      2: 1,
                      3: 1,
                      4: 0,
                      5: 2,
                      6: 1}
        self.assertEqual(distances[4], distance_4)
        distance_5 = {0: np.inf,
                      1: 3,
                      2: 2,
                      3: 1,
                      4: 2,
                      5: 0,
                      6: 1}
        self.assertEqual(distances[5], distance_5)
        distance_6 = {0: np.inf,
                      1: 3,
                      2: 2,
                      3: 2,
                      4: 1,
```

```
                    5: 1,
                    6: 0}
        self.assertEqual(distances[6], distance_6)

        true_distance = {0: distance_0,
                         1: distance_1,
                         2: distance_2,
                         3: distance_3,
                         4: distance_4,
                         5: distance_5,
                         6: distance_6}
        self.assertEqual(distances, true_distance)

        # Test convert practical
        neighborhood = distances.convert()

        neighborhood_0 = {np.inf: {1, 2, 3, 4, 5, 6},
                          0:      {0}}
        self.assertEqual(neighborhood[0], neighborhood_0)
        neighborhood_1 = {np.inf: {0},
                          0:      {1},
                          1:      {2},
                          2:      {3, 4},
                          3:      {5, 6}}
        self.assertEqual(neighborhood[1], neighborhood_1)
        neighborhood_2 = {np.inf: {0},
                          0:      {2},
                          1:      {1, 3, 4},
                          2:      {5, 6}}
        self.assertEqual(neighborhood[2], neighborhood_2)
        neighborhood_3 = {np.inf: {0},
                          0:      {3},
                          1:      {2, 4, 5},
                          2:      {1, 6}}
        self.assertEqual(neighborhood[3], neighborhood_3)
        neighborhood_4 = {np.inf: {0},
                          0:      {4},
                          1:      {2, 3, 6},
                          2:      {1, 5}}
```

```
        self.assertEqual(neighborhood[4], neighborhood_4)
        neighborhood_5 = {np.inf: {0},
                          0:      {5},
                          1:      {3, 6},
                          2:      {2, 4},
                          3:      {1}}
        self.assertEqual(neighborhood[5], neighborhood_5)
        neighborhood_6 = {np.inf: {0},
                          0:      {6},
                          1:      {4, 5},
                          2:      {2, 3},
                          3:      {1}}
        self.assertEqual(neighborhood[6], neighborhood_6)

        true_neighborhood = {0: neighborhood_0,
                             1: neighborhood_1,
                             2: neighborhood_2,
                             3: neighborhood_3,
                             4: neighborhood_4,
                             5: neighborhood_5,
                             6: neighborhood_6}
        self.assertEqual(neighborhood, true_neighborhood)


class TestGraph(ut.TestCase):
    """test the Graph class"""

    def setUp(self):
        """Setup the tests"""

        self.neighborhood = mk.create_autospec(graph.GraphNeighborhood,
                                               spec_set=True)
        self.distance     = mk.create_autospec(graph.GraphDistance,
                                               spec_set=True)
        self.adjacency    = mk.create_autospec(graph.Adjacency,
                                               spec_set=True)

        self.Graph = graph.Graph(self.neighborhood,
                                 self.distance,
```

```
                                self.adjacency)

    def test___init__(self):
        """test __init__ for class"""

        self.assertIsInstance(self.Graph, graph.Graph)

        self.assertEqual(self.Graph.neighborhood, self.neighborhood)
        self.assertEqual(self.Graph.distance,     self.distance)
        self.assertEqual(self.Graph.adjacency,    self.adjacency)

        self.assertTrue(dclass.is_dataclass(self.Graph))

    def test_setup(self):
        """test setup the class"""

        matrix = mk.MagicMock(spec=list)

        self.adjacency.dijkstra.return_value = self.distance
        self.distance. convert. return_value = self.neighborhood

        with mk.patch.object(graph, 'Adjacency') as mkAdjacency:
            mkAdjacency.return_value = self.adjacency

            self.Graph = graph.Graph.setup(matrix)
            self.assertIsInstance(self.Graph, graph.Graph)
            self.assertEqual(self.Graph.neighborhood, self.neighborhood)
            self.assertEqual(self.Graph.distance,     self.distance)
            self.assertEqual(self.Graph.adjacency,    self.adjacency)

            self.assertEqual(mkAdjacency.call_args_list,
                             [mk.call(matrix)])
            self.assertEqual(self.adjacency.dijkstra.call_args_list,
                             [mk.call(False)])
            self.assertEqual(self.distance.convert.call_args_list,
                             [mk.call()])

    def test_save(self):
        """test save to a file"""
```

```
        file_name = mk.MagicMock(spec=str)

        with mk.patch('builtins.open', mk.mock_open()) as mkOpen:
            with mk.patch.object(pickle, 'dump') as mkDump:
                self.Graph.save(file_name)
                self.assertEqual(mkDump.call_args_list,
                                 [mk.call(self.Graph, mkOpen.return_value,
                                          protocol=pickle.HIGHEST_PROTOCOL)])
                self.assertEqual(mkOpen.call_args_list,
                                 [mk.call(file_name, 'wb')])

    def test_empty(self):
        """test create a single vertex graph"""

        self.Graph = graph.Graph.empty()

        self.assertIsInstance(self.Graph, graph.Graph)

        self.assertIsInstance(self.Graph.neighborhood, graph.GraphNeighborhood)
        self.assertEqual(self.Graph.neighborhood, {0: {0: {0}}})

        self.assertIsInstance(self.Graph.distance,     graph.GraphDistance)
        self.assertEqual(self.Graph.distance, {0: {0: 0}})
        self.assertIsInstance(self.Graph.adjacency,    graph.Adjacency)
        utnp.assert_array_equal(self.Graph.adjacency, [[0]])
```

C.5.11.4 test_grid.py

```
import unittest      as ut
import unittest.mock as mk

import dataclasses   as dclass
import numpy         as np
import numpy.testing as utnp

import source.space.graph as graph
import source.space.grid  as grid


class TestGrid(ut.TestCase):
    """test the base Grid graph"""

    def setUp(self):
        """Setup the tests"""

        self.neighborhood = mk.create_autospec(graph.GraphNeighborhood,
                                               spec_set=True)
        self.distance     = mk.create_autospec(graph.GraphDistance,
                                               spec_set=True)
        self.adjacency    = mk.create_autospec(graph.Adjacency,
                                               spec_set=True)

        self.Grid = grid.Grid(self.neighborhood,
                              self.distance,
                              self.adjacency)

    def test___init__(self):
        """test __init__ for class"""

        self.assertIsInstance(self.Grid, graph.Graph)
        self.assertIsInstance(self.Grid, grid.Grid)

        self.assertEqual(self.Grid.neighborhood, self.neighborhood)
        self.assertEqual(self.Grid.distance,     self.distance)
        self.assertEqual(self.Grid.adjacency,    self.adjacency)
```

```
38
39          self.assertTrue(dclass.is_dataclass(self.Grid))
40
41      def test__boundary(self):
42          """test get the boundary vertices"""
43
44          bottom, top, left, right = self.Grid._boundary(4, 4)
45          self.assertEqual(bottom, [0,   1,  2,  3])
46          self.assertEqual(top,    [12, 13, 14, 15])
47          self.assertEqual(left,   [0,   4,  8, 12])
48          self.assertEqual(right,  [3,   7, 11, 15])
49
50          bottom, top, left, right = self.Grid._boundary(4, 6)
51          self.assertEqual(bottom, [0,   1,  2,  3])
52          self.assertEqual(top,    [20, 21, 22, 23])
53          self.assertEqual(left,   [0,   4,  8, 12, 16, 20])
54          self.assertEqual(right,  [3,   7, 11, 15, 19, 23])
55
56          bottom, top, left, right = self.Grid._boundary(6, 4)
57          self.assertEqual(bottom, [0,   1,  2,  3,  4,  5])
58          self.assertEqual(top,    [18, 19, 20, 21, 22, 23])
59          self.assertEqual(left,   [0,   6, 12, 18])
60          self.assertEqual(right,  [5,  11, 17, 23])
61
62      def test__upper_indicator(self):
63          """test the upper matrix 1 indicator"""
64
65          i = mk.MagicMock(spec=int)
66          j = mk.MagicMock(spec=int)
67          n = mk.MagicMock(spec=int)
68          m = mk.MagicMock(spec=int)
69
70          self.assertIsNone(self.Grid._upper_indicator(i, j, n, m))
71
72      def test__upper_generator(self):
73          """test the upper matrix generator"""
74
75          n = mk.MagicMock(spec=int)
76          m = mk.MagicMock(spec=int)
```

```
77
78          n.__mul__.return_value = 2
79
80          indicator = [True, False, False, True]
81          with mk.patch.object(grid.Grid, '_upper_indicator',
82                               autospec=True) as mkUpper:
83              mkUpper.side_effect = indicator
84
85              self.assertEqual(self.Grid._upper_generator(n, m),
86                               [[1, 0],
87                                [0, 1]])
88              self.assertEqual(mkUpper.call_args_list,
89                               [mk.call(self.Grid, 0, 0, n, m),
90                                mk.call(self.Grid, 0, 1, n, m),
91                                mk.call(self.Grid, 1, 0, n, m),
92                                mk.call(self.Grid, 1, 1, n, m)])
93              self.assertEqual(n.__mul__.call_args_list,
94                               [mk.call(m)])
95
96      def test__upper_torus(self):
97          """test adjust upper matrix to be a torus"""
98
99          upper = mk.MagicMock(spec=list)
100         n     = mk.MagicMock(spec=int)
101         m     = mk.MagicMock(spec=int)
102
103         self.assertIsNone(self.Grid._upper_torus(upper, n, m))
104
105     def test__full(self):
106         """test turn upper matrix into full adjacency matrix"""
107
108         upper = mk.MagicMock(spec=list)
109
110         upper_matrix = mk.create_autospec(graph.Adjacency,
111                                           spec_set=True)
112
113         with mk.patch.object(graph, 'Adjacency') as mkAdjacency:
114             mkAdjacency.return_value = upper_matrix
115
```

```
            self.assertEqual(self.Grid._full(upper),
                             upper_matrix.__add__.return_value)
            self.assertEqual(upper_matrix.__add__.call_args_list,
                             [mk.call(upper_matrix.transpose.return_value)])
            self.assertEqual(mkAdjacency.call_args_list,
                             [mk.call(upper)])
            self.assertEqual(upper_matrix.transpose.call_args_list,
                             [mk.call()])

        # Practical test
        upper = [[0, 1, 0, 1],
                 [0, 0, 1, 0],
                 [0, 0, 0, 1],
                 [0, 0, 0, 0]]
        true_full = [[0, 1, 0, 1],
                     [1, 0, 1, 0],
                     [0, 1, 0, 1],
                     [1, 0, 1, 0]]
        full = self.Grid._full(upper)
        self.assertIsInstance(full, graph.Adjacency)
        utnp.assert_array_equal(full, true_full)

    def test__generator(self):
        """test the grid generator"""

        n = mk.MagicMock(spec=int)
        m = mk.MagicMock(spec=int)

        # test method calls
        with mk.patch.object(grid.Grid, '_upper_generator',
                             autospec=True) as mkUpper:
            with mk.patch.object(grid.Grid, '_upper_torus',
                                 autospec=True) as mkTorus:
                with mk.patch.object(grid.Grid, '_full') as mkFull:
                    # Torus is False
                    self.assertEqual(self.Grid._generator(n, m, False),
                                     mkFull.return_value)
                    self.assertEqual(mkFull.call_args_list,
                                     [mk.call(mkUpper.return_value)])
```

```
                    self.assertEqual(mkUpper.call_args_list,
                                     [mk.call(self.Grid, n, m)])
                    self.assertEqual(mkTorus.call_args_list, [])

                    mkUpper.reset_mock()
                    mkFull.reset_mock()
                    # Torus is True
                    self.assertEqual(self.Grid._generator(n, m, True),
                                     mkFull.return_value)
                    self.assertEqual(mkFull.call_args_list,
                                     [mk.call(mkUpper.return_value)])
                    self.assertEqual(mkUpper.call_args_list,
                                     [mk.call(self.Grid, n, m)])
                    self.assertEqual(mkTorus.call_args_list,
                                     [mk.call(self.Grid,
                                              mkUpper.return_value, n, m)])

        # test method call order
        with mk.patch.object(grid.Grid, '_upper_generator') as mkUpper:
            with mk.patch.object(grid.Grid, '_upper_torus') as mkTorus:
                with mk.patch.object(grid.Grid, '_full') as mkFull:
                    master = mk.MagicMock()
                    master.attach_mock(mkUpper, 'upper')
                    master.attach_mock(mkTorus, 'torus')
                    master.attach_mock(mkFull,  'full')

                    # Torus is False
                    adjacency = self.Grid._generator(n, m, False)
                    self.assertEqual(master.mock_calls,
                                     [mk.call.upper(n, m),
                                      mk.call.full(mkUpper.return_value)])
                    self.assertEqual(adjacency, mkFull.return_value)

                    master.reset_mock()
                    # Torus is True
                    adjacency = self.Grid._generator(n, m, True)
                    self.assertEqual(master.mock_calls,
                                     [mk.call.upper(n, m),
                                      mk.call.torus(mkUpper.return_value, n, m),
```

```
                                    mk.call.full(mkUpper.return_value)])
                self.assertEqual(adjacency, mkFull.return_value)

        # test practical
        indicator = [False, True, False, False]
        with mk.patch.object(grid.Grid, '_upper_indicator',
                             autospec=True) as mkUpper:
            mkUpper.side_effect = indicator

            adjacency = self.Grid._generator(2, 1, False)
            self.assertIsInstance(adjacency, graph.Adjacency)
            utnp.assert_array_equal(adjacency, [[0, 1],
                                                [1, 0]])

    def test_grid(self):
        """test generate a grid graph"""

        n     = mk.MagicMock(spec=int)
        m     = mk.MagicMock(spec=int)
        torus = mk.MagicMock(spec=int)

        self.adjacency.dijkstra.return_value = self.distance
        self.distance. convert. return_value = self.neighborhood

        with mk.patch.object(grid.Grid, '_generator') as mkGenerator:
            mkGenerator.return_value = self.adjacency

            self.Grid = grid.Grid.grid(n, m, torus)
            self.assertIsInstance(self.Grid, grid.Grid)
            self.assertEqual(self.Grid.neighborhood, self.neighborhood)
            self.assertEqual(self.Grid.distance,     self.distance)
            self.assertEqual(self.Grid.adjacency,    self.adjacency)

            self.assertEqual(mkGenerator.call_args_list,
                             [mk.call(n, m, torus)])
            self.assertEqual(self.adjacency.dijkstra.call_args_list,
                             [mk.call(False)])
            self.assertEqual(self.distance.convert.call_args_list,
                             [mk.call()])
```

```
233
234         # test practical
235         indicator = [False, True, False, False]
236         with mk.patch.object(grid.Grid, '_upper_indicator',
237                              autospec=True) as mkUpper:
238             mkUpper.side_effect = indicator
239
240             self.Grid = grid.Grid.grid(2, 1, False)
241             self.assertIsInstance(self.Grid, grid.Grid)
242             self.assertIsInstance(self.Grid.neighborhood,
243                                   graph.GraphNeighborhood)
244             self.assertEqual(self.Grid.neighborhood,
245                              {0: {0: {0},
246                                   1: {1}},
247                               1: {0: {1},
248                                   1: {0}}})
249             self.assertIsInstance(self.Grid.distance, graph.GraphDistance)
250             self.assertEqual(self.Grid.distance,
251                              {0: {0: 0,
252                                   1: 1},
253                               1: {0: 1,
254                                   1: 0}})
255             self.assertIsInstance(self.Grid.adjacency, graph.Adjacency)
256             utnp.assert_array_equal(self.Grid.adjacency,
257                                     [[0, 1],
258                                      [1, 0]])
259
260
261 class TestHexagon(ut.TestCase):
262     """test the Hexagon tile grid"""
263
264     def setUp(self):
265         """Setup the tests"""
266
267         self.neighborhood = mk.create_autospec(graph.GraphNeighborhood,
268                                                spec_set=True)
269         self.distance     = mk.create_autospec(graph.GraphDistance,
270                                                spec_set=True)
271         self.adjacency    = mk.create_autospec(graph.Adjacency,
```

```
                                            spec_set=True)

        self.Grid = grid.Hexagon(self.neighborhood,
                                 self.distance,
                                 self.adjacency)

    def test___init__(self):
        """test __init__ for class"""

        self.assertIsInstance(self.Grid, grid.Grid)
        self.assertIsInstance(self.Grid, grid.Hexagon)

        self.assertEqual(self.Grid.neighborhood, self.neighborhood)
        self.assertEqual(self.Grid.distance,     self.distance)
        self.assertEqual(self.Grid.adjacency,    self.adjacency)

        self.assertTrue(dclass.is_dataclass(self.Grid))

    def test__upper_indicator(self):
        """test the upper matrix 1 indicator"""

        self.assertFalse(self.Grid._upper_indicator(0, 0, 4, 4))
        self.assertTrue( self.Grid._upper_indicator(0, 1, 4, 4))
        self.assertFalse(self.Grid._upper_indicator(0, 2, 4, 4))
        self.assertFalse(self.Grid._upper_indicator(0, 3, 4, 4))
        self.assertTrue( self.Grid._upper_indicator(0, 4, 4, 4))
        self.assertFalse(self.Grid._upper_indicator(0, 5, 4, 4))
        self.assertFalse(self.Grid._upper_indicator(0, 6, 4, 4))
        self.assertFalse(self.Grid._upper_indicator(0, 7, 4, 4))

        self.assertFalse(self.Grid._upper_indicator(1, 0, 4, 4))
        self.assertFalse(self.Grid._upper_indicator(1, 1, 4, 4))
        self.assertTrue( self.Grid._upper_indicator(1, 2, 4, 4))
        self.assertFalse(self.Grid._upper_indicator(1, 3, 4, 4))
        self.assertTrue( self.Grid._upper_indicator(1, 4, 4, 4))
        self.assertTrue( self.Grid._upper_indicator(1, 5, 4, 4))
        self.assertFalse(self.Grid._upper_indicator(1, 6, 4, 4))
        self.assertFalse(self.Grid._upper_indicator(1, 7, 4, 4))

```

```
        self.assertFalse(self.Grid._upper_indicator(2, 0, 4, 4))
        self.assertFalse(self.Grid._upper_indicator(2, 1, 4, 4))
        self.assertFalse(self.Grid._upper_indicator(2, 2, 4, 4))
        self.assertTrue( self.Grid._upper_indicator(2, 3, 4, 4))
        self.assertFalse(self.Grid._upper_indicator(2, 4, 4, 4))
        self.assertTrue( self.Grid._upper_indicator(2, 5, 4, 4))
        self.assertTrue( self.Grid._upper_indicator(2, 6, 4, 4))
        self.assertFalse(self.Grid._upper_indicator(2, 7, 4, 4))

        self.assertFalse(self.Grid._upper_indicator(3, 0, 4, 4))
        self.assertFalse(self.Grid._upper_indicator(3, 1, 4, 4))
        self.assertFalse(self.Grid._upper_indicator(3, 2, 4, 4))
        self.assertFalse(self.Grid._upper_indicator(3, 3, 4, 4))
        self.assertFalse(self.Grid._upper_indicator(3, 4, 4, 4))
        self.assertFalse(self.Grid._upper_indicator(3, 5, 4, 4))
        self.assertTrue( self.Grid._upper_indicator(3, 6, 4, 4))
        self.assertTrue( self.Grid._upper_indicator(3, 7, 4, 4))

        self.assertFalse(self.Grid._upper_indicator(4, 0, 4, 4))
        self.assertFalse(self.Grid._upper_indicator(4, 1, 4, 4))
        self.assertFalse(self.Grid._upper_indicator(4, 2, 4, 4))
        self.assertFalse(self.Grid._upper_indicator(4, 3, 4, 4))
        self.assertFalse(self.Grid._upper_indicator(4, 4, 4, 4))
        self.assertTrue( self.Grid._upper_indicator(4, 5, 4, 4))
        self.assertFalse(self.Grid._upper_indicator(4, 6, 4, 4))
        self.assertFalse(self.Grid._upper_indicator(4, 7, 4, 4))

        self.assertFalse(self.Grid._upper_indicator(5, 0, 4, 4))
        self.assertFalse(self.Grid._upper_indicator(5, 1, 4, 4))
        self.assertFalse(self.Grid._upper_indicator(5, 2, 4, 4))
        self.assertFalse(self.Grid._upper_indicator(5, 3, 4, 4))
        self.assertFalse(self.Grid._upper_indicator(5, 4, 4, 4))
        self.assertFalse(self.Grid._upper_indicator(5, 5, 4, 4))
        self.assertTrue( self.Grid._upper_indicator(5, 6, 4, 4))
        self.assertFalse(self.Grid._upper_indicator(5, 7, 4, 4))

        self.assertFalse(self.Grid._upper_indicator(6, 0, 4, 4))
        self.assertFalse(self.Grid._upper_indicator(6, 1, 4, 4))
        self.assertFalse(self.Grid._upper_indicator(6, 2, 4, 4))
```

```
350         self.assertFalse(self.Grid._upper_indicator(6, 3, 4, 4))
351         self.assertFalse(self.Grid._upper_indicator(6, 4, 4, 4))
352         self.assertFalse(self.Grid._upper_indicator(6, 5, 4, 4))
353         self.assertFalse(self.Grid._upper_indicator(6, 6, 4, 4))
354         self.assertTrue( self.Grid._upper_indicator(6, 7, 4, 4))
355 
356         self.assertFalse(self.Grid._upper_indicator(7, 0, 4, 4))
357         self.assertFalse(self.Grid._upper_indicator(7, 1, 4, 4))
358         self.assertFalse(self.Grid._upper_indicator(7, 2, 4, 4))
359         self.assertFalse(self.Grid._upper_indicator(7, 3, 4, 4))
360         self.assertFalse(self.Grid._upper_indicator(7, 4, 4, 4))
361         self.assertFalse(self.Grid._upper_indicator(7, 5, 4, 4))
362         self.assertFalse(self.Grid._upper_indicator(7, 6, 4, 4))
363         self.assertFalse(self.Grid._upper_indicator(7, 7, 4, 4))
364 
365     def test__upper_generator(self):
366         """test the upper matrix generator as a practical test"""
367 
368         matrix = self.Grid._upper_generator(4, 4)
369         correct = [[0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
370                    [0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
371                    [0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0],
372                    [0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0],
373                    [0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0],
374                    [0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0],
375                    [0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0],
376                    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0],
377                    [0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0],
378                    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0],
379                    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0],
380                    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1],
381                    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0],
382                    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0],
383                    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
384                    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]
385         self.assertEqual(matrix, correct)
386 
387         matrix = self.Grid._upper_generator(4, 2)
388         correct = [[0, 1, 0, 0, 1, 0, 0, 0],
```

```
                    [0, 0, 1, 0, 1, 1, 0, 0],
                    [0, 0, 0, 1, 0, 1, 1, 0],
                    [0, 0, 0, 0, 0, 0, 1, 1],
                    [0, 0, 0, 0, 0, 1, 0, 0],
                    [0, 0, 0, 0, 0, 0, 1, 0],
                    [0, 0, 0, 0, 0, 0, 0, 1],
                    [0, 0, 0, 0, 0, 0, 0, 0]]
        self.assertEqual(matrix, correct)

        matrix = self.Grid._upper_generator(2, 4)
        correct = [[0, 1, 1, 0, 0, 0, 0, 0],
                   [0, 0, 1, 1, 0, 0, 0, 0],
                   [0, 0, 0, 1, 1, 0, 0, 0],
                   [0, 0, 0, 0, 1, 1, 0, 0],
                   [0, 0, 0, 0, 0, 1, 1, 0],
                   [0, 0, 0, 0, 0, 0, 1, 1],
                   [0, 0, 0, 0, 0, 0, 0, 1],
                   [0, 0, 0, 0, 0, 0, 0, 0]]
        self.assertEqual(matrix, correct)

        matrix = self.Grid._upper_generator(3, 4)
        correct = [[0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0],
                   [0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0],
                   [0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0],
                   [0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0],
                   [0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0],
                   [0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0],
                   [0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0],
                   [0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0],
                   [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1],
                   [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0],
                   [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
                   [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]
        self.assertEqual(matrix, correct)

    def test__upper_torus(self):
        """test adjust upper matrix to be a torus"""

        zeros = np.zeros((16, 16), dtype=np.int64)
```

```
        upper = list(zeros.tolist())
        self.Grid._upper_torus(upper, 4, 4)
        # top-bottom verticals
        self.assertEqual(upper[0][12], 1)
        self.assertEqual(upper[1][13], 1)
        self.assertEqual(upper[2][14], 1)
        self.assertEqual(upper[3][15], 1)
        # left-right horizontals
        self.assertEqual(upper[ 0][ 3], 1)
        self.assertEqual(upper[ 4][ 7], 1)
        self.assertEqual(upper[ 8][11], 1)
        self.assertEqual(upper[12][15], 1)
        # top-bottom angles
        self.assertEqual(upper[0][13], 1)
        self.assertEqual(upper[1][14], 1)
        self.assertEqual(upper[2][15], 1)
        self.assertEqual(upper[3][12], 1)
        # left-right angles
        self.assertEqual(upper[0][7],  1)
        self.assertEqual(upper[4][11], 1)
        self.assertEqual(upper[8][15], 1)
        upper = np.array(upper)
        self.assertEqual(len(upper.nonzero()[0]), 15)
        self.assertEqual(len(upper.nonzero()[1]), 15)

        zeros = np.zeros((12, 12), dtype=np.int64)
        upper = list(zeros.tolist())
        self.Grid._upper_torus(upper, 3, 4)
        # top-bottom verticals
        self.assertEqual(upper[0][ 9], 1)
        self.assertEqual(upper[1][10], 1)
        self.assertEqual(upper[2][11], 1)
        # top-bottom angles
        self.assertEqual(upper[0][10], 1)
        self.assertEqual(upper[1][11], 1)
        self.assertEqual(upper[2][ 9], 1)
        # left-right horizontals
        self.assertEqual(upper[0][ 2], 1)
        self.assertEqual(upper[3][ 5], 1)
```

```
        self.assertEqual(upper[6][ 8], 1)
        self.assertEqual(upper[9][11], 1)
        # left-right angles
        self.assertEqual(upper[0][ 5], 1)
        self.assertEqual(upper[3][ 8], 1)
        self.assertEqual(upper[6][11], 1)
        upper = np.array(upper)
        self.assertEqual(len(upper.nonzero()[0]), 13)
        self.assertEqual(len(upper.nonzero()[1]), 13)

        zeros = np.zeros((12, 12), dtype=np.int64)
        upper = list(zeros.tolist())
        self.Grid._upper_torus(upper, 4, 3)
        # top-bottom verticals
        self.assertEqual(upper[0][ 8], 1)
        self.assertEqual(upper[1][ 9], 1)
        self.assertEqual(upper[2][10], 1)
        self.assertEqual(upper[3][11], 1)
        # top-bottom angles
        self.assertEqual(upper[0][ 9], 1)
        self.assertEqual(upper[1][10], 1)
        self.assertEqual(upper[2][11], 1)
        self.assertEqual(upper[3][ 8], 1)
        # left-right horizontals
        self.assertEqual(upper[0][ 3], 1)
        self.assertEqual(upper[4][ 7], 1)
        self.assertEqual(upper[8][11], 1)
        # left-right angles
        self.assertEqual(upper[0][ 7], 1)
        self.assertEqual(upper[4][11], 1)
        upper = np.array(upper)
        self.assertEqual(len(upper.nonzero()[0]), 13)
        self.assertEqual(len(upper.nonzero()[1]), 13)

    def test__generator(self):
        """test the grid generator as a practical test"""

        matrix = self.Grid._generator(4, 4, False)
        correct = np.array([[0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
```

```
                            [0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                            [0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0],
                            [0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0],
                            [0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0],
                            [0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0],
                            [0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0],
                            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0],
                            [0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0],
                            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0],
                            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1],
                            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0],
                            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
                            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]])
        correct += correct.transpose()
        utnp.assert_array_equal(matrix, correct)

        matrix = self.Grid._generator(4, 4, True)
        correct = np.array([[0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0],
                            [0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0],
                            [0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1],
                            [0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 0, 1],
                            [0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0],
                            [0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0],
                            [0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0],
                            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0],
                            [0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0, 1],
                            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0],
                            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0],
                            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1],
                            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1],
                            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0],
                            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
                            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]])
        correct += correct.transpose()
        utnp.assert_array_equal(matrix, correct)

        matrix = self.Grid._generator(4, 2, False)
```

```
        correct = np.array([[0, 1, 0, 0, 1, 0, 0, 0],
                            [0, 0, 1, 0, 1, 1, 0, 0],
                            [0, 0, 0, 1, 0, 1, 1, 0],
                            [0, 0, 0, 0, 0, 0, 1, 1],
                            [0, 0, 0, 0, 0, 1, 0, 0],
                            [0, 0, 0, 0, 0, 0, 1, 0],
                            [0, 0, 0, 0, 0, 0, 0, 1],
                            [0, 0, 0, 0, 0, 0, 0, 0]])
        correct += correct.transpose()
        utnp.assert_array_equal(matrix, correct)

        matrix = self.Grid._generator(4, 2, True)
        correct = np.array([[0, 1, 0, 1, 1, 1, 0, 1],
                            [0, 0, 1, 0, 1, 1, 1, 0],
                            [0, 0, 0, 1, 0, 1, 1, 1],
                            [0, 0, 0, 0, 1, 0, 1, 1],
                            [0, 0, 0, 0, 0, 1, 0, 1],
                            [0, 0, 0, 0, 0, 0, 1, 0],
                            [0, 0, 0, 0, 0, 0, 0, 1],
                            [0, 0, 0, 0, 0, 0, 0, 0]])
        correct += correct.transpose()
        utnp.assert_array_equal(matrix, correct)

        matrix = self.Grid._generator(2, 4, False)
        correct = np.array([[0, 1, 1, 0, 0, 0, 0, 0],
                            [0, 0, 1, 1, 0, 0, 0, 0],
                            [0, 0, 0, 1, 1, 0, 0, 0],
                            [0, 0, 0, 0, 1, 1, 0, 0],
                            [0, 0, 0, 0, 0, 1, 1, 0],
                            [0, 0, 0, 0, 0, 0, 1, 1],
                            [0, 0, 0, 0, 0, 0, 0, 1],
                            [0, 0, 0, 0, 0, 0, 0, 0]])
        correct += correct.transpose()
        utnp.assert_array_equal(matrix, correct)

        matrix = self.Grid._generator(2, 4, True)
        correct = np.array([[0, 1, 1, 1, 0, 0, 1, 1],
                            [0, 0, 1, 1, 0, 0, 1, 1],
                            [0, 0, 0, 1, 1, 1, 0, 0],
```

```
                            [0, 0, 0, 0, 1, 1, 0, 0],
                            [0, 0, 0, 0, 0, 1, 1, 1],
                            [0, 0, 0, 0, 0, 0, 1, 1],
                            [0, 0, 0, 0, 0, 0, 0, 1],
                            [0, 0, 0, 0, 0, 0, 0, 0]])
        correct += correct.transpose()
        utnp.assert_array_equal(matrix, correct)


class TestSquare(ut.TestCase):
    """test the Square tile grid"""

    def setUp(self):
        """Setup the tests"""

        self.neighborhood = mk.create_autospec(graph.GraphNeighborhood,
                                               spec_set=True)
        self.distance     = mk.create_autospec(graph.GraphDistance,
                                               spec_set=True)
        self.adjacency    = mk.create_autospec(graph.Adjacency,
                                               spec_set=True)

        self.Grid = grid.Square(self.neighborhood,
                                self.distance,
                                self.adjacency)

    def test___init__(self):
        """test __init__ for class"""

        self.assertIsInstance(self.Grid, grid.Grid)
        self.assertIsInstance(self.Grid, grid.Square)

        self.assertEqual(self.Grid.neighborhood, self.neighborhood)
        self.assertEqual(self.Grid.distance,     self.distance)
        self.assertEqual(self.Grid.adjacency,    self.adjacency)

        self.assertTrue(dclass.is_dataclass(self.Grid))

    def test__upper_indicator(self):
```

```
623         """test the upper matrix 1 indicator"""
624 
625         self.assertFalse(self.Grid._upper_indicator(0, 0, 4, 4))
626         self.assertTrue( self.Grid._upper_indicator(0, 1, 4, 4))
627         self.assertFalse(self.Grid._upper_indicator(0, 2, 4, 4))
628         self.assertFalse(self.Grid._upper_indicator(0, 3, 4, 4))
629         self.assertTrue( self.Grid._upper_indicator(0, 4, 4, 4))
630         self.assertFalse(self.Grid._upper_indicator(0, 5, 4, 4))
631         self.assertFalse(self.Grid._upper_indicator(0, 6, 4, 4))
632         self.assertFalse(self.Grid._upper_indicator(0, 7, 4, 4))
633 
634         self.assertFalse(self.Grid._upper_indicator(1, 0, 4, 4))
635         self.assertFalse(self.Grid._upper_indicator(1, 1, 4, 4))
636         self.assertTrue( self.Grid._upper_indicator(1, 2, 4, 4))
637         self.assertFalse(self.Grid._upper_indicator(1, 3, 4, 4))
638         self.assertFalse(self.Grid._upper_indicator(1, 4, 4, 4))
639         self.assertTrue( self.Grid._upper_indicator(1, 5, 4, 4))
640         self.assertFalse(self.Grid._upper_indicator(1, 6, 4, 4))
641         self.assertFalse(self.Grid._upper_indicator(1, 7, 4, 4))
642 
643         self.assertFalse(self.Grid._upper_indicator(2, 0, 4, 4))
644         self.assertFalse(self.Grid._upper_indicator(2, 1, 4, 4))
645         self.assertFalse(self.Grid._upper_indicator(2, 2, 4, 4))
646         self.assertTrue( self.Grid._upper_indicator(2, 3, 4, 4))
647         self.assertFalse(self.Grid._upper_indicator(2, 4, 4, 4))
648         self.assertFalse(self.Grid._upper_indicator(2, 5, 4, 4))
649         self.assertTrue( self.Grid._upper_indicator(2, 6, 4, 4))
650         self.assertFalse(self.Grid._upper_indicator(2, 7, 4, 4))
651 
652         self.assertFalse(self.Grid._upper_indicator(3, 0, 4, 4))
653         self.assertFalse(self.Grid._upper_indicator(3, 1, 4, 4))
654         self.assertFalse(self.Grid._upper_indicator(3, 2, 4, 4))
655         self.assertFalse(self.Grid._upper_indicator(3, 3, 4, 4))
656         self.assertFalse(self.Grid._upper_indicator(3, 4, 4, 4))
657         self.assertFalse(self.Grid._upper_indicator(3, 5, 4, 4))
658         self.assertFalse(self.Grid._upper_indicator(3, 6, 4, 4))
659         self.assertTrue( self.Grid._upper_indicator(3, 7, 4, 4))
660 
661     def test__upper_torus(self):
```

```
        """test adjust upper matrix to be a torus"""

        zeros = np.zeros((16, 16), dtype=np.int64)
        upper = list(zeros.tolist())
        self.Grid._upper_torus(upper, 4, 4)
        self.assertEqual(upper[ 0][12], 1)
        self.assertEqual(upper[ 1][13], 1)
        self.assertEqual(upper[ 2][14], 1)
        self.assertEqual(upper[ 3][15], 1)
        self.assertEqual(upper[ 0][ 3], 1)
        self.assertEqual(upper[ 4][ 7], 1)
        self.assertEqual(upper[ 8][11], 1)
        self.assertEqual(upper[12][15], 1)
        upper = np.array(upper)
        self.assertEqual(len(upper.nonzero()[0]), 8)
        self.assertEqual(len(upper.nonzero()[1]), 8)

        zeros = np.zeros((12, 12), dtype=np.int64)
        upper = list(zeros.tolist())
        self.Grid._upper_torus(upper, 3, 4)
        self.assertEqual(upper[ 0][ 9], 1)
        self.assertEqual(upper[ 1][10], 1)
        self.assertEqual(upper[ 2][11], 1)
        self.assertEqual(upper[ 0][ 2], 1)
        self.assertEqual(upper[ 3][ 5], 1)
        self.assertEqual(upper[ 6][ 8], 1)
        self.assertEqual(upper[ 9][11], 1)
        upper = np.array(upper)
        self.assertEqual(len(upper.nonzero()[0]), 7)
        self.assertEqual(len(upper.nonzero()[1]), 7)

        zeros = np.zeros((12, 12), dtype=np.int64)
        upper = list(zeros.tolist())
        self.Grid._upper_torus(upper, 4, 3)
        self.assertEqual(upper[ 0][ 8], 1)
        self.assertEqual(upper[ 1][ 9], 1)
        self.assertEqual(upper[ 2][10], 1)
        self.assertEqual(upper[ 3][11], 1)
        self.assertEqual(upper[ 0][ 3], 1)
```

```
        self.assertEqual(upper[ 4][ 7], 1)
        self.assertEqual(upper[ 8][11], 1)
        upper = np.array(upper)
        self.assertEqual(len(upper.nonzero()[0]), 7)
        self.assertEqual(len(upper.nonzero()[1]), 7)

    def test__generator(self):
        """test the grid generator as a practical test"""

        matrix = self.Grid._generator(4, 4, False)
        correct = np.array([[0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                            [0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                            [0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                            [0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0],
                            [0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0],
                            [0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0],
                            [0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0],
                            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0],
                            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0],
                            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0],
                            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0],
                            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
                            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0],
                            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0],
                            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
                            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]])
        correct += correct.transpose()
        utnp.assert_array_equal(matrix, correct)

        matrix = self.Grid._generator(4, 4, True)
        correct = np.array([[0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0],
                            [0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0],
                            [0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0],
                            [0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1],
                            [0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0],
                            [0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0],
                            [0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0],
                            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0],
                            [0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0],
```

```
                            [0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0],
                            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0],
                            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
                            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1],
                            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0],
                            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
                            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]])
        correct += correct.transpose()
        utnp.assert_array_equal(matrix, correct)


class TestMoore(ut.TestCase):
    """test the Moore tile grid"""

    def setUp(self):
        """Setup the tests"""

        self.neighborhood = mk.create_autospec(graph.GraphNeighborhood,
                                               spec_set=True)
        self.distance     = mk.create_autospec(graph.GraphDistance,
                                               spec_set=True)
        self.adjacency    = mk.create_autospec(graph.Adjacency,
                                               spec_set=True)

        self.Grid = grid.Moore(self.neighborhood,
                               self.distance,
                               self.adjacency)

    def test___init__(self):
        """test __init__ for class"""

        self.assertIsInstance(self.Grid, grid.Grid)
        self.assertIsInstance(self.Grid, grid.Moore)

        self.assertEqual(self.Grid.neighborhood, self.neighborhood)
        self.assertEqual(self.Grid.distance,     self.distance)
        self.assertEqual(self.Grid.adjacency,    self.adjacency)

        self.assertTrue(dclass.is_dataclass(self.Grid))
```

```
779
780     def test__upper_indicator(self):
781         """test the upper matrix 1 indicator"""
782
783         self.assertFalse(self.Grid._upper_indicator(0, 0, 4, 4))
784         self.assertTrue( self.Grid._upper_indicator(0, 1, 4, 4))
785         self.assertFalse(self.Grid._upper_indicator(0, 2, 4, 4))
786         self.assertFalse(self.Grid._upper_indicator(0, 3, 4, 4))
787         self.assertTrue( self.Grid._upper_indicator(0, 4, 4, 4))
788         self.assertTrue( self.Grid._upper_indicator(0, 5, 4, 4))
789         self.assertFalse(self.Grid._upper_indicator(0, 6, 4, 4))
790         self.assertFalse(self.Grid._upper_indicator(0, 7, 4, 4))
791
792         self.assertFalse(self.Grid._upper_indicator(1, 0, 4, 4))
793         self.assertFalse(self.Grid._upper_indicator(1, 1, 4, 4))
794         self.assertTrue( self.Grid._upper_indicator(1, 2, 4, 4))
795         self.assertFalse(self.Grid._upper_indicator(1, 3, 4, 4))
796         self.assertTrue( self.Grid._upper_indicator(1, 4, 4, 4))
797         self.assertTrue( self.Grid._upper_indicator(1, 5, 4, 4))
798         self.assertTrue( self.Grid._upper_indicator(1, 6, 4, 4))
799         self.assertFalse(self.Grid._upper_indicator(1, 7, 4, 4))
800
801         self.assertFalse(self.Grid._upper_indicator(2, 0, 4, 4))
802         self.assertFalse(self.Grid._upper_indicator(2, 1, 4, 4))
803         self.assertFalse(self.Grid._upper_indicator(2, 2, 4, 4))
804         self.assertTrue( self.Grid._upper_indicator(2, 3, 4, 4))
805         self.assertFalse(self.Grid._upper_indicator(2, 4, 4, 4))
806         self.assertTrue( self.Grid._upper_indicator(2, 5, 4, 4))
807         self.assertTrue( self.Grid._upper_indicator(2, 6, 4, 4))
808         self.assertTrue( self.Grid._upper_indicator(2, 7, 4, 4))
809
810         self.assertFalse(self.Grid._upper_indicator(3, 0, 4, 4))
811         self.assertFalse(self.Grid._upper_indicator(3, 1, 4, 4))
812         self.assertFalse(self.Grid._upper_indicator(3, 2, 4, 4))
813         self.assertFalse(self.Grid._upper_indicator(3, 3, 4, 4))
814         self.assertFalse(self.Grid._upper_indicator(3, 4, 4, 4))
815         self.assertFalse(self.Grid._upper_indicator(3, 5, 4, 4))
816         self.assertTrue( self.Grid._upper_indicator(3, 6, 4, 4))
817         self.assertTrue( self.Grid._upper_indicator(3, 7, 4, 4))
```

```
818
819     def test__upper_torus(self):
820         """test adjust upper matrix to be a torus"""
821
822         zeros = np.zeros((16, 16), dtype=np.int64)
823         upper = list(zeros.tolist())
824         self.Grid._upper_torus(upper, 4, 4)
825         # Vertical
826         self.assertEqual(upper[ 0][12], 1)
827         self.assertEqual(upper[ 1][13], 1)
828         self.assertEqual(upper[ 2][14], 1)
829         self.assertEqual(upper[ 3][15], 1)
830         # Vertical left
831         self.assertEqual(upper[ 0][13], 1)
832         self.assertEqual(upper[ 1][14], 1)
833         self.assertEqual(upper[ 2][15], 1)
834         self.assertEqual(upper[ 3][12], 1)
835         # Vertical right
836         self.assertEqual(upper[ 0][15], 1)
837         self.assertEqual(upper[ 1][12], 1)
838         self.assertEqual(upper[ 2][13], 1)
839         self.assertEqual(upper[ 3][14], 1)
840         # Horizontal
841         self.assertEqual(upper[ 0][ 3], 1)
842         self.assertEqual(upper[ 4][ 7], 1)
843         self.assertEqual(upper[ 8][11], 1)
844         self.assertEqual(upper[12][15], 1)
845         # Horizontal left
846         self.assertEqual(upper[ 0][ 7], 1)
847         self.assertEqual(upper[ 4][11], 1)
848         self.assertEqual(upper[ 8][15], 1)
849         # Horizontal left
850         self.assertEqual(upper[ 0][ 3], 1)
851         self.assertEqual(upper[ 4][ 7], 1)
852         self.assertEqual(upper[ 8][11], 1)
853         upper = np.array(upper)
854         self.assertEqual(len(upper.nonzero()[0]), 22)
855         self.assertEqual(len(upper.nonzero()[1]), 22)
856
```

```
        zeros = np.zeros((12, 12), dtype=np.int64)
        upper = list(zeros.tolist())
        self.Grid._upper_torus(upper, 3, 4)
        # Vertical
        self.assertEqual(upper[ 0][ 9], 1)
        self.assertEqual(upper[ 1][10], 1)
        self.assertEqual(upper[ 2][11], 1)
        # Vertical left
        self.assertEqual(upper[ 0][10], 1)
        self.assertEqual(upper[ 1][11], 1)
        self.assertEqual(upper[ 2][ 9], 1)
        # Vertical right
        self.assertEqual(upper[ 0][11], 1)
        self.assertEqual(upper[ 1][ 9], 1)
        self.assertEqual(upper[ 2][10], 1)
        # Horizontal
        self.assertEqual(upper[ 0][ 2], 1)
        self.assertEqual(upper[ 3][ 5], 1)
        self.assertEqual(upper[ 6][ 8], 1)
        self.assertEqual(upper[ 9][11], 1)
        # Horizontal left
        self.assertEqual(upper[ 0][ 5], 1)
        self.assertEqual(upper[ 3][ 8], 1)
        self.assertEqual(upper[ 6][11], 1)
        # Horizontal right
        self.assertEqual(upper[ 0][11], 1)
        self.assertEqual(upper[ 3][ 2], 1)
        self.assertEqual(upper[ 6][ 8], 1)
        upper = np.array(upper)
        self.assertEqual(len(upper.nonzero()[0]), 19)
        self.assertEqual(len(upper.nonzero()[1]), 19)

        zeros = np.zeros((12, 12), dtype=np.int64)
        upper = list(zeros.tolist())
        self.Grid._upper_torus(upper, 4, 3)
        # Vertical
        self.assertEqual(upper[ 0][ 8], 1)
        self.assertEqual(upper[ 1][ 9], 1)
        self.assertEqual(upper[ 2][10], 1)
```

```
        self.assertEqual(upper[ 3][11], 1)
        # Vertical left
        self.assertEqual(upper[ 0][ 9], 1)
        self.assertEqual(upper[ 1][10], 1)
        self.assertEqual(upper[ 2][11], 1)
        self.assertEqual(upper[ 3][ 8], 1)
        # Vertical right
        self.assertEqual(upper[ 0][11], 1)
        self.assertEqual(upper[ 1][ 8], 1)
        self.assertEqual(upper[ 2][ 9], 1)
        self.assertEqual(upper[ 3][10], 1)
        # Horizontal
        self.assertEqual(upper[ 0][ 3], 1)
        self.assertEqual(upper[ 4][ 7], 1)
        self.assertEqual(upper[ 8][11], 1)
        # Horizontal left
        self.assertEqual(upper[ 0][ 7], 1)
        self.assertEqual(upper[ 4][11], 1)
        # Horizontal right
        self.assertEqual(upper[ 0][11], 1)
        self.assertEqual(upper[ 4][ 7], 1)
        upper = np.array(upper)
        self.assertEqual(len(upper.nonzero()[0]), 19)
        self.assertEqual(len(upper.nonzero()[1]), 19)

    def test__generator(self):
        """test the grid generator as a practical test"""

        matrix = self.Grid._generator(4, 4, False)
        correct = np.array([[0, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                            [0, 0, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                            [0, 0, 0, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0],
                            [0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0],
                            [0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0],
                            [0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0],
                            [0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 1, 0, 0, 0, 0],
                            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0],
                            [0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0],
                            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 1, 0],
```

```
                            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 1],
                            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1],
                            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0],
                            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0],
                            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
                            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]])
        correct += correct.transpose()
        utnp.assert_array_equal(matrix, correct)

        matrix = self.Grid._generator(4, 4, True)
        correct = np.array([[0, 1, 0, 1, 1, 1, 0, 1, 0, 0, 0, 0, 1, 1, 0, 1],
                            [0, 0, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 0],
                            [0, 0, 0, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1],
                            [0, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 1, 0, 1, 1],
                            [0, 0, 0, 0, 0, 1, 0, 1, 1, 1, 0, 1, 0, 0, 0, 0],
                            [0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0],
                            [0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 1, 0, 0, 0, 0],
                            [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0],
                            [0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 1, 0, 1],
                            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 1, 0],
                            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 1],
                            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1],
                            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1],
                            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0],
                            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
                            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]])
        correct += correct.transpose()
        utnp.assert_array_equal(matrix, correct)


class TestTriangle(ut.TestCase):
    """test the Triangle tile grid"""

    def setUp(self):
        """Setup the tests"""

        self.neighborhood = mk.create_autospec(graph.GraphNeighborhood,
                                               spec_set=True)
        self.distance     = mk.create_autospec(graph.GraphDistance,
```

```
                                            spec_set=True)
        self.adjacency    = mk.create_autospec(graph.Adjacency,
                                            spec_set=True)

        self.Grid = grid.Triangle(self.neighborhood,
                                  self.distance,
                                  self.adjacency)

    def test___init__(self):
        """test __init__ for class"""

        self.assertIsInstance(self.Grid, grid.Grid)
        self.assertIsInstance(self.Grid, grid.Triangle)

        self.assertEqual(self.Grid.neighborhood, self.neighborhood)
        self.assertEqual(self.Grid.distance,     self.distance)
        self.assertEqual(self.Grid.adjacency,    self.adjacency)

        self.assertTrue(dclass.is_dataclass(self.Grid))

    def test__upper_indicator(self):
        """test the upper matrix 1 indicator"""

        self.assertFalse(self.Grid._upper_indicator(0, 0, 4, 4))
        self.assertTrue( self.Grid._upper_indicator(0, 1, 4, 4))
        self.assertFalse(self.Grid._upper_indicator(0, 2, 4, 4))
        self.assertFalse(self.Grid._upper_indicator(0, 3, 4, 4))
        self.assertTrue( self.Grid._upper_indicator(0, 4, 4, 4))
        self.assertFalse(self.Grid._upper_indicator(0, 5, 4, 4))
        self.assertFalse(self.Grid._upper_indicator(0, 6, 4, 4))
        self.assertFalse(self.Grid._upper_indicator(0, 7, 4, 4))

        self.assertFalse(self.Grid._upper_indicator(1, 0, 4, 4))
        self.assertFalse(self.Grid._upper_indicator(1, 1, 4, 4))
        self.assertTrue( self.Grid._upper_indicator(1, 2, 4, 4))
        self.assertFalse(self.Grid._upper_indicator(1, 3, 4, 4))
        self.assertFalse(self.Grid._upper_indicator(1, 4, 4, 4))
        self.assertFalse(self.Grid._upper_indicator(1, 5, 4, 4))
        self.assertFalse(self.Grid._upper_indicator(1, 6, 4, 4))
```

```
        self.assertFalse(self.Grid._upper_indicator(1, 7, 4, 4))

        self.assertFalse(self.Grid._upper_indicator(2, 0, 4, 4))
        self.assertFalse(self.Grid._upper_indicator(2, 1, 4, 4))
        self.assertFalse(self.Grid._upper_indicator(2, 2, 4, 4))
        self.assertTrue( self.Grid._upper_indicator(2, 3, 4, 4))
        self.assertFalse(self.Grid._upper_indicator(2, 4, 4, 4))
        self.assertFalse(self.Grid._upper_indicator(2, 5, 4, 4))
        self.assertTrue( self.Grid._upper_indicator(2, 6, 4, 4))
        self.assertFalse(self.Grid._upper_indicator(2, 7, 4, 4))

        self.assertFalse(self.Grid._upper_indicator(3, 0, 4, 4))
        self.assertFalse(self.Grid._upper_indicator(3, 1, 4, 4))
        self.assertFalse(self.Grid._upper_indicator(3, 2, 4, 4))
        self.assertFalse(self.Grid._upper_indicator(3, 3, 4, 4))
        self.assertFalse(self.Grid._upper_indicator(3, 4, 4, 4))
        self.assertFalse(self.Grid._upper_indicator(3, 5, 4, 4))
        self.assertFalse(self.Grid._upper_indicator(3, 6, 4, 4))
        self.assertFalse(self.Grid._upper_indicator(3, 7, 4, 4))

        self.assertFalse(self.Grid._upper_indicator(4,  0, 4, 4))
        self.assertFalse(self.Grid._upper_indicator(4,  1, 4, 4))
        self.assertFalse(self.Grid._upper_indicator(4,  2, 4, 4))
        self.assertFalse(self.Grid._upper_indicator(4,  3, 4, 4))
        self.assertFalse(self.Grid._upper_indicator(4,  4, 4, 4))
        self.assertTrue( self.Grid._upper_indicator(4,  5, 4, 4))
        self.assertFalse(self.Grid._upper_indicator(4,  6, 4, 4))
        self.assertFalse(self.Grid._upper_indicator(4,  7, 4, 4))
        self.assertFalse(self.Grid._upper_indicator(4,  8, 4, 4))
        self.assertFalse(self.Grid._upper_indicator(4,  9, 4, 4))
        self.assertFalse(self.Grid._upper_indicator(4, 10, 4, 4))
        self.assertFalse(self.Grid._upper_indicator(4, 11, 4, 4))

        self.assertFalse(self.Grid._upper_indicator(5,  0, 4, 4))
        self.assertFalse(self.Grid._upper_indicator(5,  1, 4, 4))
        self.assertFalse(self.Grid._upper_indicator(5,  2, 4, 4))
        self.assertFalse(self.Grid._upper_indicator(5,  3, 4, 4))
        self.assertFalse(self.Grid._upper_indicator(5,  4, 4, 4))
        self.assertFalse(self.Grid._upper_indicator(5,  5, 4, 4))
```

```
        self.assertTrue( self.Grid._upper_indicator(5,  6, 4, 4))
        self.assertFalse(self.Grid._upper_indicator(5,  7, 4, 4))
        self.assertFalse(self.Grid._upper_indicator(5,  8, 4, 4))
        self.assertTrue( self.Grid._upper_indicator(5,  9, 4, 4))
        self.assertFalse(self.Grid._upper_indicator(5, 10, 4, 4))
        self.assertFalse(self.Grid._upper_indicator(5, 11, 4, 4))

        self.assertFalse(self.Grid._upper_indicator(6,  0, 4, 4))
        self.assertFalse(self.Grid._upper_indicator(6,  1, 4, 4))
        self.assertFalse(self.Grid._upper_indicator(6,  2, 4, 4))
        self.assertFalse(self.Grid._upper_indicator(6,  3, 4, 4))
        self.assertFalse(self.Grid._upper_indicator(6,  4, 4, 4))
        self.assertFalse(self.Grid._upper_indicator(6,  5, 4, 4))
        self.assertFalse(self.Grid._upper_indicator(6,  6, 4, 4))
        self.assertTrue( self.Grid._upper_indicator(6,  7, 4, 4))
        self.assertFalse(self.Grid._upper_indicator(6,  8, 4, 4))
        self.assertFalse(self.Grid._upper_indicator(6,  9, 4, 4))
        self.assertFalse(self.Grid._upper_indicator(6, 10, 4, 4))
        self.assertFalse(self.Grid._upper_indicator(6, 11, 4, 4))

        self.assertFalse(self.Grid._upper_indicator(7,  0, 4, 4))
        self.assertFalse(self.Grid._upper_indicator(7,  1, 4, 4))
        self.assertFalse(self.Grid._upper_indicator(7,  2, 4, 4))
        self.assertFalse(self.Grid._upper_indicator(7,  3, 4, 4))
        self.assertFalse(self.Grid._upper_indicator(7,  4, 4, 4))
        self.assertFalse(self.Grid._upper_indicator(7,  5, 4, 4))
        self.assertFalse(self.Grid._upper_indicator(7,  6, 4, 4))
        self.assertFalse(self.Grid._upper_indicator(7,  7, 4, 4))
        self.assertFalse(self.Grid._upper_indicator(7,  8, 4, 4))
        self.assertFalse(self.Grid._upper_indicator(7,  9, 4, 4))
        self.assertFalse(self.Grid._upper_indicator(7, 10, 4, 4))
        self.assertTrue( self.Grid._upper_indicator(7, 11, 4, 4))

    def test__upper_torus(self):
        """test adjust upper matrix to be a torus"""

        zeros = np.zeros((16, 16), dtype=np.int64)
        upper = list(zeros.tolist())
        self.Grid._upper_torus(upper, 4, 4)
```

```
        # Vertical
        self.assertEqual(upper[ 1][13], 1)
        self.assertEqual(upper[ 3][15], 1)
        # Horizontal
        self.assertEqual(upper[ 0][ 3], 1)
        self.assertEqual(upper[ 4][ 7], 1)
        self.assertEqual(upper[ 8][11], 1)
        self.assertEqual(upper[12][15], 1)
        upper = np.array(upper)
        self.assertEqual(len(upper.nonzero()[0]), 6)
        self.assertEqual(len(upper.nonzero()[1]), 6)

        zeros = np.zeros((12, 12), dtype=np.int64)
        upper = list(zeros.tolist())
        with self.assertRaisesRegex(ValueError, 'n must be even'):
            self.Grid._upper_torus(upper, 3, 4)

        zeros = np.zeros((12, 12), dtype=np.int64)
        upper = list(zeros.tolist())
        self.Grid._upper_torus(upper, 4, 3)
        # Vertical
        self.assertEqual(upper[ 0][ 8], 1)
        self.assertEqual(upper[ 2][10], 1)
        # Horizontal
        self.assertEqual(upper[ 0][ 3], 1)
        self.assertEqual(upper[ 4][ 7], 1)
        self.assertEqual(upper[ 8][11], 1)
        upper = np.array(upper)
        self.assertEqual(len(upper.nonzero()[0]), 5)
        self.assertEqual(len(upper.nonzero()[1]), 5)

    def test__generator(self):
        """test the grid generator as a practical test"""

        matrix = self.Grid._generator(4, 4, False)
        correct = np.array([[0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                            [0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                            [0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
```

```
                            [0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                            [0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0],
                            [0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0],
                            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0],
                            [0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0],
                            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0],
                            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1],
                            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0],
                            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0],
                            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
                            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]])
        correct += correct.transpose()
        utnp.assert_array_equal(matrix, correct)

        matrix = self.Grid._generator(4, 4, True)
        correct = np.array([[0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                            [0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0],
                            [0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0],
                            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
                            [0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0],
                            [0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0],
                            [0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0],
                            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0],
                            [0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0],
                            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0],
                            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0],
                            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1],
                            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0],
                            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
                            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]])
        correct += correct.transpose()
        utnp.assert_array_equal(matrix, correct)

        matrix = self.Grid._generator(4, 3, False)
        correct = np.array([[0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0],
                            [0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                            [0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0],
```

```
                            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                            [0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
                            [0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0],
                            [0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0],
                            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
                            [0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0],
                            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
                            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
                            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]])
        correct += correct.transpose()
        utnp.assert_array_equal(matrix, correct)

        matrix = self.Grid._generator(4, 3, True)
        correct = np.array([[0, 1, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0],
                            [0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                            [0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0],
                            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                            [0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0],
                            [0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0],
                            [0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0],
                            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
                            [0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1],
                            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0],
                            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
                            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]])
        correct += correct.transpose()
        utnp.assert_array_equal(matrix, correct)
```

C.5.11.5 test_location.py

```
import unittest      as ut
import unittest.mock as mk

import collections as collect

import source.space.location as location


class TestLocation(ut.TestCase):
    """test the Location class"""

    def setUp(self):
        """Setup the tests"""

        self.locs = [mk.MagicMock(spec=int) for _ in range(3)]

        self.Location = location.Location(self.locs)

    def test___init__(self):
        """test __init__ for class"""

        self.assertIsInstance(self.Location, collect.UserList)
        self.assertIsInstance(self.Location, location.Location)

        self.assertEqual(self.Location,      self.locs)
        self.assertEqual(self.Location.data, self.locs)

    def test_location_key(self):
        """test create a location key"""

        self.assertEqual(self.Location.location_key, tuple(self.locs))

    def test_depth(self):
        """test get the depth of the location"""

        with mk.patch.object(location, 'len') as mkLen:
            self.assertEqual(self.Location.depth,
```

```
                            mkLen.return_value)
            self.assertEqual(mkLen.call_args_list,
                            [mk.call(self.locs)])

        self.assertEqual(self.Location.depth, 3)

    def test_level(self):
        """test get operation level of location"""

        with mk.patch.object(location, 'len') as mkLen:
            self.assertEqual(self.Location.level,
                            mkLen.return_value.__sub__.return_value)
            self.assertEqual(mkLen.return_value.__sub__.call_args_list,
                            [mk.call(1)])
            self.assertEqual(mkLen.call_args_list,
                            [mk.call(self.locs)])

        self.assertEqual(self.Location.level, 2)
```

C.5.11.6 test_space.py

```
import unittest      as ut
import unittest.mock as mk

import collections  as collect
import numpy.random as rnd

import source.keyword as keyword

import source.data.counter as data_counter

import source.space.agents      as main_agents
import source.space.environment as environ
import source.space.graph       as main_graph
import source.space.grid        as grid
import source.space.location    as agent_location
import source.space.space       as space


class GraphTest(main_graph.Graph):
    """Class to add dynamic values for tests"""

    neighborhood = mk.create_autospec(main_graph.GraphNeighborhood,
                                      spec_set=True)
    adjacency    = mk.create_autospec(main_graph.Adjacency,
                                      spec_set=True)


class TestSpace(ut.TestCase):
    """test Space class"""

    def setUp(self):
        """Setup the tests"""

        self.graphs    = [mk.create_autospec(GraphTest, spec_set=True)
                          for _ in range(3)]
        self.locations = [mk.create_autospec(agent_location, spec_set=True)
                          for _ in range(3)]
```

```
38
39          self.location_keys = {mk.MagicMock(spec=int):
40                                    [mk.MagicMock(spec=tuple) for _ in range(3)]
41                                for _ in range(3)}
42
43          self.Space = space.Space(self.graphs,
44                                   self.locations,
45                                   self.location_keys)
46
47      def test___init__(self):
48          """test __init__ for class"""
49
50          self.assertIsInstance(self.Space, collect.UserList)
51          self.assertIsInstance(self.Space, space.Space)
52
53          self.assertEqual(self.Space.locations,     self.locations)
54          self.assertEqual(self.Space.location_keys, self.location_keys)
55
56          self.assertEqual(self.Space,      self.graphs)
57          self.assertEqual(self.Space.data, self.graphs)
58
59      def test_depth(self):
60          """test get the depth of locations"""
61
62          with mk.patch.object(space, 'len') as mkLen:
63              self.assertEqual(self.Space.depth,
64                               mkLen.return_value)
65              self.assertEqual(mkLen.call_args_list,
66                               [mk.call(self.Space)])
67
68          self.assertEqual(self.Space.depth, 3)
69
70      def test_neighborhood(self):
71          """test get the neighborhood of a location"""
72
73          for graph in self.graphs:
74              graph.neighborhood = mk.create_autospec(main_graph.
75                                                      GraphNeighborhood,
76                                                  spec_set=True)
```

```
77
78          location = mk.create_autospec(agent_location.Location, spec_set=True)
79          kwargs   = {'test': mk.MagicMock()}
80
81          for level in range(len(self.graphs)):
82              location.level = level
83              self.assertEqual(self.Space.neighborhood(location, **kwargs),
84                               self.graphs[level].neighborhood.
85                                  neighborhood.return_value)
86              self.assertEqual(self.graphs[level].neighborhood.
87                                  neighborhood.call_args_list,
88                               [mk.call(location.__getitem__.return_value,
89                                        **kwargs)])
90              self.assertEqual(location.__getitem__.call_args_list,
91                               [mk.call(level)])
92              location.reset_mock()
93
94              self.graphs[level].reset_mock()
95              for graph in self.graphs:
96                  self.assertEqual(graph.neighborhood.neighborhood.call_args_list,
97                                   [])
98
99          self.assertEqual(len(self.graphs), 3)
100
101     def test_extend_location(self):
102         """test extend a location"""
103
104         for graph in self.graphs:
105             graph.adjacency = mk.create_autospec(main_graph.Adjacency,
106                                                  spec_set=True)
107
108         location = mk.create_autospec(agent_location.Location, spec_set=True)
109         location.copy.return_value = \
110             mk.create_autospec(agent_location.Location, spec_set=True)
111
112         with mk.patch.object(rnd, 'choice') as mkRND:
113             with mk.patch.object(space, 'list') as mkList:
114                 for depth in range(len(self.graphs)):
115                     location.depth = depth
```

```
                    self.assertEqual(self.Space.extend_location(location),
                                     location.copy.return_value)
                    self.assertEqual(location.copy.call_args_list,
                                     [mk.call()])
                    self.assertEqual(location.copy.return_value.append.
                                        call_args_list,
                                     [mk.call(mkRND.return_value)])
                    location.reset_mock()
                    self.assertEqual(mkRND.call_args_list,
                                     [mk.call(mkList.return_value)])
                    mkRND.reset_mock()
                    self.assertEqual(mkList.call_args_list,
                                     [mk.call(self.graphs[depth].
                                              adjacency.vertices)])
                    mkList.reset_mock()

                self.assertEqual(len(self.graphs), 3)

    def test_new_location(self):
        """test generate a new location"""

        for graph in self.graphs:
            graph.adjacency = mk.create_autospec(main_graph.Adjacency,
                                                 spec_set=True)

        with mk.patch.object(rnd, 'choice') as mkRND:
            with mk.patch.object(space, 'list') as mkList:
                for depth in range(1, len(self.graphs) + 1):
                    locs     = [mk.MagicMock(spec=int) for _ in range(depth)]
                    vertices = [mk.MagicMock(spec=list) for _ in range(depth)]
                    mkRND. side_effect = locs
                    mkList.side_effect = vertices

                    location = self.Space.new_location(depth)
                    self.assertIsInstance(location, agent_location.Location)
                    self.assertEqual(len(location), depth)
                    for level in range(depth):
                        self.assertEqual(location[level],
                                         locs[level])
```

```
                        self.assertEqual(mkRND.call_args_list[level],
                                         mk.call(vertices[level]))
                        self.assertEqual(mkList.call_args_list[level],
                                         mk.call(self.graphs[level].
                                                 adjacency.vertices))
                    self.assertEqual(len(mkList.call_args_list), depth)
                    self.assertEqual(len(mkRND.call_args_list),  depth)
                    self.assertEqual(len(location),              depth)
                    mkList.reset_mock()
                    mkRND.reset_mock()

                self.assertEqual(len(self.graphs), 3)

    def test__make_locations(self):
        """test generate location data for a specific level"""

        vertices = []
        for graph in self.graphs:
            vertex          = {mk.MagicMock(spec=int) for _ in range(3)}
            graph.adjacency = mk.create_autospec(main_graph.Adjacency,
                                                 spec_set=True)
            graph.adjacency.vertices = vertex
            vertices.append(vertex)

        for level in range(len(self.graphs)):
            vertex = vertices[level]
            locations = []
            new       = []
            keys      = []
            locs      = []
            for _ in range(3):
                location = mk.create_autospec(agent_location.Location,
                                              spec_set=True)
                copies = []
                for _ in vertex:
                    copy = mk.create_autospec(agent_location.Location,
                                                  spec_set=True)
                    copies.append(copy)
                    new.append(copy)
```

```
                    keys.append(copy.location_key)
                location.copy.side_effect = copies
                locations.append(location)
                locs.append(copies)

            self.assertEqual(self.Space._make_locations(locations, level),
                             (new, keys))
            for index_i, location in enumerate(locations):
                for index_j, vert in enumerate(vertex):
                    self.assertEqual(location.copy.call_args_list[index_j],
                                     mk.call())
                    self.assertEqual(locs[index_i][index_j].
                                        append.call_args_list,
                                     [mk.call(vert)])
                self.assertEqual(len(location.copy.call_args_list),
                                 len(vertex))
                self.assertEqual(len(vertex), 3)

        self.assertEqual(len(self.graphs), 3)

    def test_get_locations(self):
        """test get the locations from graphs"""

        make               = []
        true_locations     = []
        true_location_keys = {}
        for index in range(3):
            locs = [mk.create_autospec(agent_location.Location,
                                       spec_set=True)
                    for _ in range(3)]
            keys = [mk.MagicMock(spec=tuple) for _ in range(3)]
            make.append((locs, keys))
            true_locations.extend(locs)
            true_location_keys[index + 1] = keys

        with mk.patch.object(space.Space, 'depth', autospec=True) as mkDepth:
            with mk.patch.object(space.Space, '_make_locations',
                                 autospec=True) as mkMake:
                mkDepth.__get__ = mk.MagicMock(return_value=4)
```

```
                mkMake.side_effect = make

                locations, location_keys = self.Space.get_locations()

                self.assertIsInstance(locations, list)
                self.assertEqual(len(locations), 10)
                first = locations.pop(0)
                self.assertIsInstance(first, agent_location.Location)
                self.assertEqual(first, [0])
                self.assertEqual(locations, true_locations)

                self.assertIsInstance(location_keys, dict)
                self.assertEqual(len(location_keys), 4)
                self.assertIn(0, location_keys)
                self.assertIsInstance(location_keys[0], list)
                self.assertEqual(location_keys[0][0], (0,))
                self.assertEqual(len(location_keys[0]), 1)
                del location_keys[0]
                self.assertEqual(location_keys, true_location_keys)

                self.assertEqual(len(mkMake.call_args_list), 3)
                call = mkMake.call_args_list.pop(0)
                self.assertEqual(call,
                                 mk.call(self.Space, [first], 1))
                for index, call in enumerate(mkMake.call_args_list):
                    self.assertEqual(call,
                                     mk.call(self.Space,
                                             make[index][0], index + 2))
                self.assertEqual(len(mkMake.call_args_list), 2)

    # noinspection PyTypeChecker
    def test_create_grid(self):
        """test create a grid graph"""

        grid_args = (mk.MagicMock(), mk.MagicMock())

        with mk.patch.object(grid.Hexagon, 'grid') as mkHexagon:
            with mk.patch.object(grid.Square, 'grid') as mkSquare:
                with mk.patch.object(grid.Moore, 'grid') as mkMoore:
```

```
                    with mk.patch.object(grid.Triangle, 'grid') as mkTriangle:
                        # hexagon
                        grid_generator = (keyword.hexagon, *grid_args)
                        self.assertEqual(self.Space.create_grid(grid_generator),
                                         mkHexagon.return_value)
                        self.assertEqual(mkHexagon.call_args_list,
                                         [mk.call(*grid_args)])
                        # square
                        grid_generator = (keyword.square, *grid_args)
                        self.assertEqual(self.Space.create_grid(grid_generator),
                                         mkSquare.return_value)
                        self.assertEqual(mkSquare.call_args_list,
                                         [mk.call(*grid_args)])
                        # moore
                        grid_generator = (keyword.moore, *grid_args)
                        self.assertEqual(self.Space.create_grid(grid_generator),
                                         mkMoore.return_value)
                        self.assertEqual(mkMoore.call_args_list,
                                         [mk.call(*grid_args)])
                        # triangle
                        grid_generator = (keyword.triangle, *grid_args)
                        self.assertEqual(self.Space.create_grid(grid_generator),
                                         mkTriangle.return_value)
                        self.assertEqual(mkTriangle.call_args_list,
                                         [mk.call(*grid_args)])

    def test_setup(self):
        """test setup space"""

        # test calls
        with mk.patch.object(space.Space, 'get_locations',
                             autospec=True) as mkGet:
            with mk.patch.object(space.Space, 'create_grid') as mkGrid:
                # test use generators
                grid_generators = [mk.MagicMock(spec=tuple) for _ in range(3)]
                graphs = [mk.create_autospec(GraphTest, spec_set=True)
                          for _ in range(3)]

                mkGet.return_value = (self.locations, self.location_keys)
```

```
                mkGrid.side_effect = graphs

                self.Space = space.Space.setup(grid_generators)
                self.assertIsInstance(self.Space, space.Space)

                self.assertEqual(self.Space.locations,     self.locations)
                self.assertEqual(self.Space.location_keys, self.location_keys)
                self.assertEqual(mkGet.call_args_list,
                                 [mk.call(self.Space)])

                self.assertEqual(len(self.Space), 4)
                graph = self.Space.pop(0)
                self.assertIsInstance(graph, main_graph.Graph)
                self.assertIsInstance(graph.neighborhood,
                                      main_graph.GraphNeighborhood)
                self.assertEqual(len(graph.neighborhood), 1)
                self.assertIn(0, graph.neighborhood)
                self.assertIsInstance(graph.neighborhood[0],
                                      main_graph.VertexNeighborhood)
                self.assertEqual(graph.neighborhood[0].vertex, 0)
                self.assertEqual(len(graph.neighborhood[0]), 1)
                self.assertIn(0, graph.neighborhood[0])
                self.assertEqual(graph.neighborhood[0][0], {0})
                self.assertEqual(graph.neighborhood, {0: {0: {0}}})
                self.assertIsInstance(graph.distance,
                                      main_graph.GraphDistance)
                self.assertEqual(len(graph.distance), 1)
                self.assertIn(0, graph.distance)
                self.assertIsInstance(graph.distance[0],
                                      main_graph.VertexDistance)
                self.assertEqual(graph.distance[0].vertex, 0)
                self.assertEqual(len(graph.distance[0]), 1)
                self.assertIn(0, graph.distance[0])
                self.assertEqual(graph.distance[0][0], 0)
                self.assertEqual(graph.distance, {0: {0: 0}})
                self.assertIsInstance(graph.adjacency,
                                      main_graph.Adjacency)
                self.assertEqual(graph.adjacency.tolist(), [[0]])

```

```
                self.assertEqual(len(self.Space), 3)
                self.assertEqual(self.Space, graphs)
                for index, call in enumerate(mkGrid.call_args_list):
                    self.assertEqual(call,
                                     mk.call(grid_generators[index]))
                for index, grid_generator in enumerate(grid_generators):
                    self.assertEqual(mkGrid.call_args_list[index],
                                     mk.call(grid_generator))
                self.assertEqual(len(mkGrid.call_args_list),
                                 len(grid_generators))

                mkGrid.reset_mock()
                mkGet.reset_mock()
                # Pass in graphs
                self.Space = space.Space.setup(graphs)
                self.assertIsInstance(self.Space, space.Space)

                self.assertEqual(self.Space.locations,     self.locations)
                self.assertEqual(self.Space.location_keys, self.location_keys)
                self.assertEqual(mkGet.call_args_list,
                                 [mk.call(self.Space)])

                self.assertEqual(len(self.Space), 4)
                graph = self.Space.pop(0)
                self.assertIsInstance(graph, main_graph.Graph)
                self.assertIsInstance(graph.neighborhood,
                                      main_graph.GraphNeighborhood)
                self.assertEqual(len(graph.neighborhood), 1)
                self.assertIn(0, graph.neighborhood)
                self.assertIsInstance(graph.neighborhood[0],
                                      main_graph.VertexNeighborhood)
                self.assertEqual(graph.neighborhood[0].vertex, 0)
                self.assertEqual(len(graph.neighborhood[0]), 1)
                self.assertIn(0, graph.neighborhood[0])
                self.assertEqual(graph.neighborhood[0][0], {0})
                self.assertEqual(graph.neighborhood, {0: {0: {0}}})
                self.assertIsInstance(graph.distance,
                                      main_graph.GraphDistance)
                self.assertEqual(len(graph.distance), 1)
```

```
                self.assertIn(0, graph.distance)
                self.assertIsInstance(graph.distance[0],
                                      main_graph.VertexDistance)
                self.assertEqual(graph.distance[0].vertex, 0)
                self.assertEqual(len(graph.distance[0]), 1)
                self.assertIn(0, graph.distance[0])
                self.assertEqual(graph.distance[0][0], 0)
                self.assertEqual(graph.distance, {0: {0: 0}})
                self.assertIsInstance(graph.adjacency,
                                      main_graph.Adjacency)
                self.assertEqual(graph.adjacency.tolist(), [[0]])

                self.assertEqual(len(self.Space), 3)
                self.assertEqual(self.Space, graphs)

                self.assertEqual(mkGrid.call_args_list, [])

        # Test practical
        grid_generators = [(keyword.square, 3, 3, False),
                           (keyword.square, 2, 2, False)]
        self.Space = space.Space.setup(grid_generators)
        self.assertIsInstance(self.Space, space.Space)

        self.assertIsInstance(self.Space.locations, list)
        self.assertEqual(len(self.Space.locations), 46)

        location = self.Space.locations[0]
        self.assertIsInstance(location, agent_location.Location)
        self.assertEqual(location, [0])

        locations_1 = self.Space.locations[1:10]
        self.assertEqual(len(locations_1), 9)
        locations_2 = self.Space.locations[10:]
        self.assertEqual(len(locations_2), 36)

        counter = 0
        for vertex_1 in range(9):
            self.assertIsInstance(locations_1[vertex_1],
                                  agent_location.Location)
```

```
            self.assertEqual(locations_1[vertex_1], [0, vertex_1])
            for vertex_2 in range(4):
                self.assertIsInstance(locations_2[counter],
                                      agent_location.Location)
                self.assertEqual(locations_2[counter], [0, vertex_1, vertex_2])
                counter += 1
        self.assertEqual(counter, 36)

        self.assertEqual(len(self.Space), 3)
        graph = self.Space.pop(0)
        self.assertIsInstance(graph, main_graph.Graph)
        self.assertIsInstance(graph.neighborhood,
                              main_graph.GraphNeighborhood)
        self.assertEqual(len(graph.neighborhood), 1)
        self.assertIn(0, graph.neighborhood)
        self.assertIsInstance(graph.neighborhood[0],
                              main_graph.VertexNeighborhood)
        self.assertEqual(graph.neighborhood[0].vertex, 0)
        self.assertEqual(len(graph.neighborhood[0]), 1)
        self.assertIn(0, graph.neighborhood[0])
        self.assertEqual(graph.neighborhood[0][0], {0})
        self.assertEqual(graph.neighborhood, {0: {0: {0}}})
        self.assertIsInstance(graph.distance,
                              main_graph.GraphDistance)
        self.assertEqual(len(graph.distance), 1)
        self.assertIn(0, graph.distance)
        self.assertIsInstance(graph.distance[0],
                              main_graph.VertexDistance)
        self.assertEqual(graph.distance[0].vertex, 0)
        self.assertEqual(len(graph.distance[0]), 1)
        self.assertIn(0, graph.distance[0])
        self.assertEqual(graph.distance[0][0], 0)
        self.assertEqual(graph.distance, {0: {0: 0}})
        self.assertIsInstance(graph.adjacency,
                              main_graph.Adjacency)
        self.assertEqual(graph.adjacency.tolist(), [[0]])

        space_1_neighborhood = {0: {0: {0},
                                    1: {1, 3},
```

```
                                    2: {2, 4, 6},
                                    3: {5, 7},
                                    4: {8}},
                                1: {0: {1},
                                    1: {0, 2, 4},
                                    2: {3, 5, 7},
                                    3: {6, 8}},
                                2: {0: {2},
                                    1: {1, 5},
                                    2: {0, 4, 8},
                                    3: {3, 7},
                                    4: {6}},
                                3: {0: {3},
                                    1: {0, 4, 6},
                                    2: {1, 5, 7},
                                    3: {2, 8}},
                                4: {0: {4},
                                    1: {1, 3, 5, 7},
                                    2: {0, 2, 6, 8}},
                                5: {0: {5},
                                    1: {2, 4, 8},
                                    2: {1, 3, 7},
                                    3: {0, 6}},
                                6: {0: {6},
                                    1: {3, 7},
                                    2: {0, 4, 8},
                                    3: {1, 5},
                                    4: {2}},
                                7: {0: {7},
                                    1: {4, 6, 8},
                                    2: {1, 3, 5},
                                    3: {0, 2}},
                                8: {0: {8},
                                    1: {5, 7},
                                    2: {2, 4, 6},
                                    3: {1, 3},
                                    4: {0}}}
    space_2_neighborhood = {0: {0: {0},
                                    1: {1, 2},
```

```
                                    2: {3}},
                                1: {0: {1},
                                    1: {0, 3},
                                    2: {2}},
                                2: {0: {2},
                                    1: {0, 3},
                                    2: {1}},
                                3: {0: {3},
                                    1: {1, 2},
                                    2: {0}}}
        space_1_distance = {0: {0: 0,
                                1: 1,
                                2: 2,
                                3: 1,
                                4: 2,
                                5: 3,
                                6: 2,
                                7: 3,
                                8: 4},
                            1: {0: 1,
                                1: 0,
                                2: 1,
                                3: 2,
                                4: 1,
                                5: 2,
                                6: 3,
                                7: 2,
                                8: 3},
                            2: {0: 2,
                                1: 1,
                                2: 0,
                                3: 3,
                                4: 2,
                                5: 1,
                                6: 4,
                                7: 3,
                                8: 2},
                            3: {0: 1,
                                1: 2,
```

```
                                    2: 3,
                                    3: 0,
                                    4: 1,
                                    5: 2,
                                    6: 1,
                                    7: 2,
                                    8: 3},
                                4: {0: 2,
                                    1: 1,
                                    2: 2,
                                    3: 1,
                                    4: 0,
                                    5: 1,
                                    6: 2,
                                    7: 1,
                                    8: 2},
                                5: {0: 3,
                                    1: 2,
                                    2: 1,
                                    3: 2,
                                    4: 1,
                                    5: 0,
                                    6: 3,
                                    7: 2,
                                    8: 1},
                                6: {0: 2,
                                    1: 3,
                                    2: 4,
                                    3: 1,
                                    4: 2,
                                    5: 3,
                                    6: 0,
                                    7: 1,
                                    8: 2},
                                7: {0: 3,
                                    1: 2,
                                    2: 3,
                                    3: 2,
                                    4: 1,
```

```
                                5: 2,
                                6: 1,
                                7: 0,
                                8: 1},
                            8: {0: 4,
                                1: 3,
                                2: 2,
                                3: 3,
                                4: 2,
                                5: 1,
                                6: 2,
                                7: 1,
                                8: 0}}
        space_2_distance = {0: {0: 0,
                                1: 1,
                                2: 1,
                                3: 2},
                            1: {0: 1,
                                1: 0,
                                2: 2,
                                3: 1},
                            2: {0: 1,
                                1: 2,
                                2: 0,
                                3: 1},
                            3: {0: 2,
                                1: 1,
                                2: 1,
                                3: 0}}
        space_1_adj = [[0, 1, 0, 1, 0, 0, 0, 0, 0],
                       [1, 0, 1, 0, 1, 0, 0, 0, 0],
                       [0, 1, 0, 0, 0, 1, 0, 0, 0],
                       [1, 0, 0, 0, 1, 0, 1, 0, 0],
                       [0, 1, 0, 1, 0, 1, 0, 1, 0],
                       [0, 0, 1, 0, 1, 0, 0, 0, 1],
                       [0, 0, 0, 1, 0, 0, 0, 1, 0],
                       [0, 0, 0, 0, 1, 0, 1, 0, 1],
                       [0, 0, 0, 0, 0, 1, 0, 1, 0]]
        space_2_adj = [[0, 1, 1, 0],
```

```
                        [1, 0, 0, 1],
                        [1, 0, 0, 1],
                        [0, 1, 1, 0]]

        self.assertEqual(len(self.Space), 2)
        graph = self.Space.pop(0)
        self.assertIsInstance(graph, main_graph.Graph)
        self.assertIsInstance(graph.neighborhood,
                              main_graph.GraphNeighborhood)
        self.assertEqual(graph.neighborhood, space_1_neighborhood)
        self.assertIsInstance(graph.distance,
                              main_graph.GraphDistance)
        self.assertEqual(graph.distance, space_1_distance)
        self.assertIsInstance(graph.adjacency,
                              main_graph.Adjacency)
        self.assertEqual(graph.adjacency.tolist(), space_1_adj)

        self.assertEqual(len(self.Space), 1)
        graph = self.Space.pop(0)
        self.assertIsInstance(graph, main_graph.Graph)
        self.assertIsInstance(graph.neighborhood,
                              main_graph.GraphNeighborhood)
        self.assertEqual(graph.neighborhood, space_2_neighborhood)
        self.assertIsInstance(graph.distance,
                              main_graph.GraphDistance)
        self.assertEqual(graph.distance, space_2_distance)
        self.assertIsInstance(graph.adjacency,
                              main_graph.Adjacency)
        self.assertEqual(graph.adjacency.tolist(), space_2_adj)

        # Test construct the agents system from a real space

        agent_keys = [mk.MagicMock(spec=str) for _ in range(3)]
        attrs_dict = {}
        for index in range(3):
            attrs      = {}
            for agent_key in agent_keys:
                attr = {}
                for _ in range(3):
```

```
                    attr_val = mk.MagicMock(spec=str)
                    values = [mk.MagicMock(spec=str) for _ in range(3)]
                    removal = mk.MagicMock(spec=bool)
                    attr[mk.MagicMock(spec=str)] = (attr_val, values, removal)
                attrs[agent_key] = attr
            attrs_dict[index] = attrs

        cutoff     = 1
        init_plant = mk.MagicMock(spec=callable)
        environment = (cutoff, init_plant)

        # noinspection PyTypeChecker
        agents = main_agents.Agents.empty(self.Space,
                                          agent_keys,
                                          attrs_dict,
                                          environment)
        self.assertIsInstance(agents, main_agents.Agents)

        plant_count = 0
        for location in self.Space.locations:
            level        = location.level
            location_key = location.location_key
            self.assertIn(location_key, agents)
            agents_bin = agents[location_key]
            self.assertIsInstance(agents_bin, main_agents.AgentsBin)
            self.assertEqual(agents_bin.location_key, location_key)

            self.assertIsInstance(agents_bin.environment,
                                  environ.Environment)
            if location.depth == keyword.plant_depth:
                self.assertEqual(agents_bin.environment.plant,
                                 init_plant.return_value)
                if location[-1] == 0:
                    self.assertEqual(agents_bin.environment.bt,
                                     keyword.bt)
                    self.assertEqual(init_plant.call_args_list[plant_count],
                                     mk.call(keyword.bt))
                else:
                    self.assertEqual(agents_bin.environment.bt,
```

```
                                    keyword.not_bt)
                self.assertEqual(init_plant.call_args_list[plant_count],
                                    mk.call(keyword.not_bt))
            plant_count += 1
        else:
            self.assertEqual(agents_bin.environment.bt,    None)
            self.assertEqual(agents_bin.environment.plant, None)

        for agent_key in agent_keys:
            self.assertIn(agent_key, agents_bin)
            agent_bin = agents_bin[agent_key]
            self.assertIsInstance(agent_bin, main_agents.AgentBin)
            self.assertEqual(agent_bin.agent_key, agent_key)

            counts = agent_bin.counts
            self.assertIsInstance(counts, data_counter.Counts)
            for attr_key, attr_tuple in \
                    attrs_dict[level][agent_key].items():
                attr, values, removal = attr_tuple

                self.assertIn(attr_key, counts)
                count = counts[attr_key]
                self.assertIsInstance(count, data_counter.Count)
                self.assertEqual(count.attr,    attr)
                self.assertEqual(count.removal, removal)
                for value in values:
                    self.assertIn(value, count)
                    self.assertEqual(count[value], 0)
                self.assertEqual(len(count), 3)
                data_columns = count.data_columns
                self.assertIsInstance(data_columns,
                                      data_counter.DataColumns)
                self.assertEqual(data_columns.attr, attr)
                for value in values:
                    self.assertIn(value, data_columns)
                    data_column = data_columns[value]
                    self.assertIsInstance(data_column,
                                          data_counter.DataColumn)
                    self.assertEqual(data_column.attr_value, value)
```

```
                        self.assertEqual(data_column, [])
                    self.assertEqual(len(data_columns), 3)
                self.assertEqual(len(counts), 3)
            self.assertEqual(len(agents_bin), 3)
        self.assertEqual(len(agents), 46)

        self.assertEqual(plant_count, 9)
```

C.5.12 test_survival

```
FallArmyworm
└── test
    └── test_survival
        ├── test_adult.py
        ├── test_egg.py
        ├── test_larva.py
        ├── test_models.py
        └── test_pupa.py
```

C.5.12.1 test_adult.py

```
import unittest      as ut
import unittest.mock as mk

import dataclasses as dclass

import source.keyword as keyword

import source.agents.adult as agent_adult

import source.survival.adult as survival


class AdultTest(agent_adult.Adult):
    """Class to add dynamic values for tests"""

    mass     = mk.MagicMock(spec=float)
    genotype = mk.MagicMock(spec=str)


class TestAdult(ut.TestCase):
    """test the Adult survival class"""

    def setUp(self):
        """Setup the tests"""

        self.survival = mk.MagicMock(spec=callable)

        self.Adult = survival.Adult(self.survival)

    def test___init__(self):
        """test __init__ for class"""

        self.assertIsInstance(self.Adult, survival.Adult)

        self.assertEqual(self.Adult.survival, self.survival)

        self.assertTrue(dclass.is_dataclass(self.Adult))
```

```
38
39      def test__use_survival(self):
40          """test if we use the survival system"""
41
42          self.assertTrue(self.Adult._use_survival)
43
44          self.Adult.survival = None
45          self.assertFalse(self.Adult._use_survival)
46
47      def test__survive(self):
48          """test determine if adult survives"""
49
50          adult = mk.create_autospec(AdultTest, spec_set=True)
51
52          with mk.patch.object(survival.Adult, '_use_survival',
53                               autospec=True) as mkUse:
54              mkUse.__get__ = mk.MagicMock(side_effect=[False, True])
55
56              # Test when we don't have a model
57              self.assertTrue(self.Adult._survive(adult))
58              self.assertEqual(self.survival.call_args_list, [])
59
60              # Test when have a model
61              self.assertEqual(self.Adult._survive(adult),
62                               self.survival.return_value)
63              self.assertEqual(self.survival.call_args_list,
64                               [mk.call(adult.mass, adult.genotype)])
65
66      def test_survive(self):
67          """test run the behavior"""
68
69          adult = mk.create_autospec(AdultTest, spec_set=True)
70
71          with mk.patch.object(survival.Adult, '_survive',
72                               autospec=True) as mkSurvive:
73              mkSurvive.side_effect = [True, False]
74
75              # Test if survives
76              self.Adult.survive(adult)
```

```
            self.assertEqual(adult.die.call_args_list, [])
            self.assertEqual(mkSurvive.call_args_list,
                             [mk.call(self.Adult, adult)])

            mkSurvive.reset_mock()
            # Test if not survives
            self.Adult.survive(adult)
            self.assertEqual(adult.die.call_args_list,
                             [mk.call(keyword.survival)])
            self.assertEqual(mkSurvive.call_args_list,
                             [mk.call(self.Adult, adult)])

    def test_setup(self):
        """test setup the class"""

        # Test if have the model
        kwargs = {keyword.adult_survival: self.survival}
        self.Adult = survival.Adult.setup(**kwargs)
        self.assertIsInstance(self.Adult, survival.Adult)
        self.assertEqual(self.Adult.survival, self.survival)

        # Test if have the model
        kwargs = {}
        self.Adult = survival.Adult.setup(**kwargs)
        self.assertIsInstance(self.Adult, survival.Adult)
        self.assertEqual(self.Adult.survival, None)
```

C.5.12.2 test_egg.py

```
import unittest      as ut
import unittest.mock as mk

import dataclasses as dclass

import source.keyword as keyword

import source.agents.egg as agent_egg

import source.survival.egg as survival


class EggTest(agent_egg.Egg):
    """Class to add dynamic values for tests"""

    mass     = mk.MagicMock(spec=float)
    genotype = mk.MagicMock(spec=str)


class TestEgg(ut.TestCase):
    """test the Egg survival class"""

    def setUp(self):
        """Setup the tests"""

        self.survival = mk.MagicMock(spec=callable)

        self.Egg = survival.Egg(self.survival)

    def test___init__(self):
        """test __init__ for class"""

        self.assertIsInstance(self.Egg, survival.Egg)

        self.assertEqual(self.Egg.survival, self.survival)

        self.assertTrue(dclass.is_dataclass(self.Egg))
```

```
38
39      def test__use_survival(self):
40          """test if we use the survival system"""
41
42          self.assertTrue(self.Egg._use_survival)
43
44          self.Egg.survival = None
45          self.assertFalse(self.Egg._use_survival)
46
47      def test__survive(self):
48          """test determine if egg survives"""
49
50          egg = mk.create_autospec(EggTest, spec_set=True)
51
52          with mk.patch.object(survival.Egg, '_use_survival',
53                               autospec=True) as mkUse:
54              mkUse.__get__ = mk.MagicMock(side_effect=[False, True])
55
56              # Test when we don't have a model
57              self.assertTrue(self.Egg._survive(egg))
58              self.assertEqual(self.survival.call_args_list, [])
59
60              # Test when have a model
61              self.assertEqual(self.Egg._survive(egg),
62                               self.survival.return_value)
63              self.assertEqual(self.survival.call_args_list,
64                               [mk.call(egg.mass, egg.genotype, egg.bt)])
65
66      def test_survive(self):
67          """test run the behavior"""
68
69          egg = mk.create_autospec(EggTest, spec_set=True)
70
71          with mk.patch.object(survival.Egg, '_survive',
72                               autospec=True) as mkSurvive:
73              mkSurvive.side_effect = [True, False]
74
75              # Test if survives
76              self.Egg.survive(egg)
```

```
            self.assertEqual(egg.die.call_args_list, [])
            self.assertEqual(mkSurvive.call_args_list,
                             [mk.call(self.Egg, egg)])

            mkSurvive.reset_mock()
            # Test if not survives
            self.Egg.survive(egg)
            self.assertEqual(egg.die.call_args_list,
                             [mk.call(keyword.survival)])
            self.assertEqual(mkSurvive.call_args_list,
                             [mk.call(self.Egg, egg)])

    def test_setup(self):
        """test setup the class"""

        # Test if have the model
        kwargs = {keyword.egg_survival: self.survival}
        self.Egg = survival.Egg.setup(**kwargs)
        self.assertIsInstance(self.Egg, survival.Egg)
        self.assertEqual(self.Egg.survival, self.survival)

        # Test if have the model
        kwargs = {}
        self.Egg = survival.Egg.setup(**kwargs)
        self.assertIsInstance(self.Egg, survival.Egg)
        self.assertEqual(self.Egg.survival, None)
```

C.5.12.3 test_larva.py

```
import unittest      as ut
import unittest.mock as mk

import dataclasses as dclass

import source.keyword as keyword

import source.agents.larva as agent_larva

import source.survival.larva as survival


class LarvaTest(agent_larva.Larva):
    """Class to add dynamic values for tests"""

    alive    = mk.MagicMock(spec=bool)
    mass     = mk.MagicMock(spec=float)
    genotype = mk.MagicMock(spec=str)
    starve   = mk.MagicMock(spec=bool)


class TestLarva(ut.TestCase):
    """test the Larva survival class"""

    def setUp(self):
        """Setup the tests"""

        self.survival = mk.MagicMock(spec=callable)

        self.Larva = survival.Larva(self.survival)

    def test___init__(self):
        """test __init__ for class"""

        self.assertIsInstance(self.Larva, survival.Larva)

        self.assertEqual(self.Larva.survival, self.survival)
```

```
38
39          self.assertTrue(dclass.is_dataclass(self.Larva))
40
41      def test__use_survival(self):
42          """test if we use the survival system"""
43
44          self.assertTrue(self.Larva._use_survival)
45
46          self.Larva.survival = None
47          self.assertFalse(self.Larva._use_survival)
48
49      def test_starve(self):
50          """test if the larva starves"""
51
52          larva = mk.create_autospec(LarvaTest, spec_set=True)
53
54          # Not starve
55          larva.starve = False
56          self.Larva._starve(larva)
57          self.assertEqual(larva.die.call_args_list, [])
58
59          # starve
60          larva.starve = True
61          self.Larva._starve(larva)
62          self.assertEqual(larva.die.call_args_list,
63                           [mk.call(keyword.starve)])
64
65      def test__survive(self):
66          """test determine if larva survives"""
67
68          larva = mk.create_autospec(LarvaTest, spec_set=True)
69
70          with mk.patch.object(survival.Larva, '_use_survival',
71                               autospec=True) as mkUse:
72              mkUse.__get__ = mk.MagicMock(side_effect=[False, True, True])
73
74              # Test when we don't have a model
75              self.assertTrue(self.Larva._survive(larva))
76              self.assertEqual(self.survival.call_args_list, [])
```

```
77
78              # Test when have a model and are dead
79              larva.alive = False
80              self.assertTrue(self.Larva._survive(larva))
81              self.assertEqual(self.survival.call_args_list, [])
82
83              # Test when have a model and are alive
84              larva.alive = True
85              self.assertEqual(self.Larva._survive(larva),
86                               self.survival.return_value)
87              self.assertEqual(self.survival.call_args_list,
88                               [mk.call(larva.mass, larva.genotype, larva.bt)])
89
90      def test_survive(self):
91          """test run the behavior"""
92
93          larva = mk.create_autospec(LarvaTest, spec_set=True)
94
95          with mk.patch.object(survival.Larva, '_survive',
96                               autospec=True) as mkSurvive:
97              with mk.patch.object(survival.Larva, '_starve',
98                                   autospec=True) as mkStarve:
99                  mkSurvive.side_effect = [True, False]
100
101                 # Test if survives
102                 self.Larva.survive(larva)
103                 self.assertEqual(larva.die.call_args_list, [])
104                 self.assertEqual(mkSurvive.call_args_list,
105                                  [mk.call(self.Larva, larva)])
106                 self.assertEqual(mkStarve.call_args_list,
107                                  [mk.call(larva)])
108
109                 mkSurvive.reset_mock()
110                 mkStarve.reset_mock()
111                 # Test if not survives
112                 self.Larva.survive(larva)
113                 self.assertEqual(larva.die.call_args_list,
114                                  [mk.call(keyword.survival)])
115                 self.assertEqual(mkSurvive.call_args_list,
```

```
                                    [mk.call(self.Larva, larva)])
                self.assertEqual(mkStarve.call_args_list,
                                    [mk.call(larva)])

    def test_setup(self):
        """test setup the class"""

        # Test if have the model
        kwargs = {keyword.larva_survival: self.survival}
        self.Larva = survival.Larva.setup(**kwargs)
        self.assertIsInstance(self.Larva, survival.Larva)
        self.assertEqual(self.Larva.survival, self.survival)

        # Test if have the model
        kwargs = {}
        self.Larva = survival.Larva.setup(**kwargs)
        self.assertIsInstance(self.Larva, survival.Larva)
        self.assertEqual(self.Larva.survival, None)
```

C.5.12.4 test_models.py

```
import unittest      as ut
import unittest.mock as mk

import dataclasses   as dclass
import numpy         as np
import numpy.random  as rnd

import source.keyword as keyword

import source.simulation.models as models

import source.survival.models as model


class TestLarva(ut.TestCase):
    """test the Larva survival mathematical model class"""

    def setUp(self):
        """Setup the tests"""

        self.minimum = mk.MagicMock(spec=dict)
        self.maximum = mk.MagicMock(spec=dict)

        self.inflection = mk.MagicMock(spec=dict)
        self.steepness  = mk.MagicMock(spec=dict)

        self.Larva = model.Larva(self.minimum,
                                 self.maximum,
                                 self.inflection,
                                 self.steepness)

    def test___init__(self):
        """test __init__ for class"""

        self.assertIsInstance(self.Larva, models.Model)
        self.assertIsInstance(self.Larva, model.Larva)

```

```
        self.assertEqual(self.Larva.minimum,    self.minimum)
        self.assertEqual(self.Larva.maximum,    self.maximum)
        self.assertEqual(self.Larva.inflection, self.inflection)
        self.assertEqual(self.Larva.steepness,  self.steepness)

        self.assertEqual(self.Larva.model_key, keyword.larva_survival)

        self.assertTrue(dclass.is_dataclass(self.Larva))

    def test__logistic(self):
        """test evaluate the logistic probability"""

        mass     = mk.MagicMock(spec=float)
        genotype = mk.MagicMock(spec=str)
        bt       = mk.MagicMock(spec=str)

        lower = mk.MagicMock(spec=float)
        upper = mk.MagicMock(spec=float)
        m0    = mk.MagicMock(spec=float)
        k     = mk.MagicMock(spec=float)

        self.minimum.__getitem__.return_value = mk.MagicMock(spec=dict)
        self.maximum.__getitem__.return_value = mk.MagicMock(spec=dict)
        self.minimum.__getitem__.return_value.__getitem__.return_value = lower
        self.maximum.__getitem__.return_value.__getitem__.return_value = upper

        self.inflection.__getitem__.return_value = m0
        self.steepness. __getitem__.return_value = k

        with mk.patch.object(np, 'exp', autospec=True) as mkExp:
            self.assertEqual(self.Larva._logistic(mass, genotype, bt),
                             lower.__add__.return_value)
            self.assertEqual(lower.__add__.call_args_list,
                             [mk.call(upper.__sub__.return_value.
                                      __truediv__.return_value)])
            self.assertEqual(upper.__sub__.return_value.
                               __truediv__.call_args_list,
                             [mk.call(mkExp.return_value.__add__.return_value)])
            self.assertEqual(upper.__sub__.call_args_list,
```

```
                            [mk.call(lower)])
        self.assertEqual(mkExp.return_value.__add__.call_args_list,
                            [mk.call(1)])
        self.assertEqual(mkExp.call_args_list,
                            [mk.call(k.__neg__.return_value.
                                        __mul__.return_value)])
        self.assertEqual(k.__neg__.return_value.__mul__.call_args_list,
                            [mk.call(mass.__sub__.return_value)])
        self.assertEqual(k.__neg__.call_args_list,
                            [mk.call()])
        self.assertEqual(mass.__sub__.call_args_list,
                            [mk.call(m0)])

        self.assertEqual(self.minimum.__getitem__.return_value.
                            __getitem__.call_args_list,
                            [mk.call(genotype)])
        self.assertEqual(self.minimum.__getitem__.call_args_list,
                            [mk.call(bt)])
        self.assertEqual(self.maximum.__getitem__.return_value.
                            __getitem__.call_args_list,
                            [mk.call(genotype)])
        self.assertEqual(self.maximum.__getitem__.call_args_list,
                            [mk.call(bt)])
        self.assertEqual(self.inflection.__getitem__.call_args_list,
                            [mk.call(genotype)])
        self.assertEqual(self.steepness.__getitem__.call_args_list,
                            [mk.call(genotype)])

    def test___call__(self):
        """test call the model"""

        mass     = mk.MagicMock(spec=float)
        genotype = mk.MagicMock(spec=str)
        bt       = mk.MagicMock(spec=str)

        with mk.patch.object(model.Larva, '_logistic',
                            autospec=True) as mkLogistic:
            with mk.patch.object(rnd, 'random') as mkRND:
                mkRND.return_value.__le__.side_effect = [True, False]
```

```
116
117                 # Test is true
118                 self.assertTrue(self.Larva(mass, genotype, bt))
119                 self.assertEqual(mkRND.return_value.__le__.call_args_list,
120                                  [mk.call(mkLogistic.return_value)])
121                 self.assertEqual(mkRND.call_args_list,
122                                  [mk.call()])
123                 self.assertEqual(mkLogistic.call_args_list,
124                                  [mk.call(self.Larva, mass, genotype, bt)])
125
126                 mkLogistic.reset_mock()
127                 mkRND.reset_mock()
128                 # Test is false
129                 self.assertFalse(self.Larva(mass, genotype, bt))
130                 self.assertEqual(mkRND.return_value.__le__.call_args_list,
131                                  [mk.call(mkLogistic.return_value)])
132                 self.assertEqual(mkRND.call_args_list,
133                                  [mk.call()])
134                 self.assertEqual(mkLogistic.call_args_list,
135                                  [mk.call(self.Larva, mass, genotype, bt)])
136
137
138 class TestLarvaFixed(ut.TestCase):
139     """test the fixed value survival for larvae"""
140
141     def setUp(self):
142         """Setup the tests"""
143
144         self.prob = mk.MagicMock(spec=dict)
145         self.prob.__getitem__.return_value = mk.MagicMock(spec=dict)
146
147         self.Larva = model.LarvaFixed(self.prob)
148
149     def test___init__(self):
150         """test __init__ for class"""
151
152         self.assertIsInstance(self.Larva, models.Model)
153         self.assertIsInstance(self.Larva, model.LarvaFixed)
154
```

```
        self.assertEqual(self.Larva.prob, self.prob)

        self.assertEqual(self.Larva.model_key, keyword.larva_survival)

        self.assertTrue(dclass.is_dataclass(self.Larva))

    def test___call__(self):
        """test call the model"""

        mass     = mk.MagicMock(spec=float)
        genotype = mk.MagicMock(spec=str)
        bt       = mk.MagicMock(spec=str)

        with mk.patch.object(rnd, 'random') as mkRND:
            mkRND.return_value.__le__.side_effect = [True, False]

            self.assertTrue(self.Larva(mass, genotype, bt))
            self.assertEqual(mkRND.return_value.__le__.call_args_list,
                             [mk.call(self.prob.__getitem__.return_value.
                                      __getitem__.return_value)])
            self.assertEqual(self.prob.__getitem__.return_value.
                                 __getitem__.call_args_list,
                             [mk.call(genotype)])
            self.assertEqual(self.prob.__getitem__.call_args_list,
                             [mk.call(bt)])

            mkRND.reset_mock()
            self.prob.reset_mock()
            self.assertFalse(self.Larva(mass, genotype, bt))
            self.assertEqual(mkRND.return_value.__le__.call_args_list,
                             [mk.call(self.prob.__getitem__.return_value.
                                      __getitem__.return_value)])
            self.assertEqual(self.prob.__getitem__.return_value.
                             __getitem__.call_args_list,
                             [mk.call(genotype)])
            self.assertEqual(self.prob.__getitem__.call_args_list,
                             [mk.call(bt)])

```

```
194 class TestFixed(ut.TestCase):
195     """test the Fixed survival mathematical model class"""
196 
197     def setUp(self):
198         """Setup the tests"""
199 
200         self.prob = mk.MagicMock(spec=float)
201 
202         self.Fixed = model.Fixed(self.prob)
203 
204     def test___init__(self):
205         """test __init__ for class"""
206 
207         self.assertIsInstance(self.Fixed, models.Model)
208         self.assertIsInstance(self.Fixed, model.Fixed)
209 
210         self.assertEqual(self.Fixed.prob, self.prob)
211 
212         self.assertEqual(self.Fixed.model_key, None)
213 
214         self.assertTrue(dclass.is_dataclass(self.Fixed))
215 
216     def test___call__(self):
217         """test call the model"""
218 
219         mass = mk.MagicMock(spec=float)
220         args = (mk.MagicMock(), mk.MagicMock())
221 
222         with mk.patch.object(rnd, 'random') as mkRND:
223             mkRND.return_value.__le__.side_effect = [True, False]
224 
225             # Test is true
226             self.assertTrue(self.Fixed(mass, *args))
227             self.assertEqual(mkRND.return_value.__le__.call_args_list,
228                              [mk.call(self.prob)])
229             self.assertEqual(mkRND.call_args_list,
230                              [mk.call()])
231 
232             mkRND.reset_mock()
```

```
            # Test is false
            self.assertFalse(self.Fixed(mass, *args))
            self.assertEqual(mkRND.return_value.__le__.call_args_list,
                             [mk.call(self.prob)])
            self.assertEqual(mkRND.call_args_list,
                             [mk.call()])


class TestEgg(ut.TestCase):
    """test the Egg survival mathematical model class"""

    def setUp(self):
        """Setup the tests"""

        self.prob = mk.MagicMock(spec=float)

        self.Egg = model.Egg(self.prob)

    def test___init__(self):
        """test __init__ for class"""

        self.assertIsInstance(self.Egg, models.Model)
        self.assertIsInstance(self.Egg, model.Fixed)
        self.assertIsInstance(self.Egg, model.Egg)

        self.assertEqual(self.Egg.prob, self.prob)

        self.assertEqual(self.Egg.model_key, keyword.egg_survival)

        self.assertTrue(dclass.is_dataclass(self.Egg))


class TestPupa(ut.TestCase):
    """test the Pupa survival mathematical model class"""

    def setUp(self):
        """Setup the tests"""

        self.prob = mk.MagicMock(spec=float)
```

```
272
273         self.Pupa = model.Pupa(self.prob)
274
275     def test___init__(self):
276         """test __init__ for class"""
277
278         self.assertIsInstance(self.Pupa, models.Model)
279         self.assertIsInstance(self.Pupa, model.Fixed)
280         self.assertIsInstance(self.Pupa, model.Pupa)
281
282         self.assertEqual(self.Pupa.prob, self.prob)
283
284         self.assertEqual(self.Pupa.model_key, keyword.pupa_survival)
285
286         self.assertTrue(dclass.is_dataclass(self.Pupa))
287
288
289 class TestAdult(ut.TestCase):
290     """test the Adult survival mathematical model class"""
291
292     def setUp(self):
293         """Setup the tests"""
294
295         self.prob = mk.MagicMock(spec=float)
296
297         self.Adult = model.Adult(self.prob)
298
299     def test___init__(self):
300         """test __init__ for class"""
301
302         self.assertIsInstance(self.Adult, models.Model)
303         self.assertIsInstance(self.Adult, model.Fixed)
304         self.assertIsInstance(self.Adult, model.Adult)
305
306         self.assertEqual(self.Adult.prob, self.prob)
307
308         self.assertEqual(self.Adult.model_key, keyword.adult_survival)
309
310         self.assertTrue(dclass.is_dataclass(self.Adult))
```

C.5.12.5 test_pupa.py

```
import unittest      as ut
import unittest.mock as mk

import dataclasses as dclass

import source.keyword as keyword

import source.agents.pupa as agent_pupa

import source.survival.pupa as survival


class PupaTest(agent_pupa.Pupa):
    """Class to add dynamic values for tests"""

    mass     = mk.MagicMock(spec=float)
    genotype = mk.MagicMock(spec=str)


class TestPupa(ut.TestCase):
    """test the Pupa survival class"""

    def setUp(self):
        """Setup the tests"""

        self.survival = mk.MagicMock(spec=callable)

        self.Pupa = survival.Pupa(self.survival)

    def test___init__(self):
        """test __init__ for class"""

        self.assertIsInstance(self.Pupa, survival.Pupa)

        self.assertEqual(self.Pupa.survival, self.survival)

        self.assertTrue(dclass.is_dataclass(self.Pupa))
```

```
38
39      def test__use_survival(self):
40          """test if we use the survival system"""
41
42          self.assertTrue(self.Pupa._use_survival)
43
44          self.Pupa.survival = None
45          self.assertFalse(self.Pupa._use_survival)
46
47      def test__survive(self):
48          """test determine if pupa survives"""
49
50          pupa = mk.create_autospec(PupaTest, spec_set=True)
51
52          with mk.patch.object(survival.Pupa, '_use_survival',
53                               autospec=True) as mkUse:
54              mkUse.__get__ = mk.MagicMock(side_effect=[False, True])
55
56              # Test when we don't have a model
57              self.assertTrue(self.Pupa._survive(pupa))
58              self.assertEqual(self.survival.call_args_list, [])
59
60              # Test when have a model
61              self.assertEqual(self.Pupa._survive(pupa),
62                               self.survival.return_value)
63              self.assertEqual(self.survival.call_args_list,
64                               [mk.call(pupa.mass, pupa.genotype)])
65
66      def test_survive(self):
67          """test run the behavior"""
68
69          pupa = mk.create_autospec(PupaTest, spec_set=True)
70
71          with mk.patch.object(survival.Pupa, '_survive',
72                               autospec=True) as mkSurvive:
73              mkSurvive.side_effect = [True, False]
74
75              # Test if survives
76              self.Pupa.survive(pupa)
```

```
            self.assertEqual(pupa.die.call_args_list, [])
            self.assertEqual(mkSurvive.call_args_list,
                             [mk.call(self.Pupa, pupa)])

            mkSurvive.reset_mock()
            # Test if not survives
            self.Pupa.survive(pupa)
            self.assertEqual(pupa.die.call_args_list,
                             [mk.call(keyword.survival)])
            self.assertEqual(mkSurvive.call_args_list,
                             [mk.call(self.Pupa, pupa)])

    def test_setup(self):
        """test setup the class"""

        # Test if have the model
        kwargs = {keyword.pupa_survival: self.survival}
        self.Pupa = survival.Pupa.setup(**kwargs)
        self.assertIsInstance(self.Pupa, survival.Pupa)
        self.assertEqual(self.Pupa.survival, self.survival)

        # Test if have the model
        kwargs = {}
        self.Pupa = survival.Pupa.setup(**kwargs)
        self.assertIsInstance(self.Pupa, survival.Pupa)
        self.assertEqual(self.Pupa.survival, None)
```

Bibliography

[1] J. J. Adamczyk, D. D. Hardee, L. C. Adams, and D. V. Sumerford. Correlating Differences in Larval Survival and Development of Bollworm (Lepidoptera: Noctuidae) and Fall Armyworm (Lepidoptera: Noctuidae) to Differential Expression of Cry1A(c) δ-Endotoxin in Various Plant Parts Among Commercial Cultivars of Transgenic B. *Journal of Economic Entomology*, 94(1):284–290, 2001.

[2] J. J. Adamczyk, J. W. Holloway, G. E. Church, B. R. Leonard, and J. B. Graves. Larval Survival and Development of the Fall Armyworm (Lepidoptera: Noctuidae) on Normal and Transgenic Cotton Expressing the Bacillus thuringiensis CryIA(c) δ-Endotoxin. *Journal of Economic Entomology*, 91(2):539–545, 1998.

[3] G. Alder and D. Benson. JGraph draw.io, 2019.

[4] S. Allesina and M. Wilmes. *Computing Skills for Biologists A Toolbox*. Princeton University Press, Princeton, 2019.

[5] Anaconda Software Distribution. Anaconda 3, 2019.

[6] S. Barbosa, K. Kay, N. Chitnis, and I. M. Hastings. Modelling the impact of insecticide-based control interventions on the evolution of insecticide resistance and disease transmission. *Parasites & Vectors*, 11(1):482, 2018.

[7] C. S. Barfield, J. L. Stimac, and M. A. Keller. State-of-the-Art for Predicting Damaging Infestations of Fall Armyworm. *Florida Entomologist*, 63(4):364–375, 1980.

[8] C. Bass and L. M. Field. Gene amplification and insecticide resistance. *Pest Management Science*, 67(8):886–890, 2011.

[9] C. Bass, A. M. Puinean, C. T. Zimmer, I. Denholm, L. M. Field, S. P. Foster, O. Gutbrod, R. Nauen, R. Slater, and M. S. Williamson. The evolution of insecticide resistance in the peach potato aphid, Myzus persicae. *Insect Biochemistry and Molecular Biology*, 51:41–51, 2014.

[10] M. Bayer. SQLAlchemy. In A. Brown and G. Wilson, editors, *The Architecture of Open Source Applications Volume II: Structure, Scale, and a Few More Fearless Hacks*. aosabook.org, 2012.

[11] K. Beck. *Test-Driven Development*. Pearson, Boston, 2003.

[12] J. P. F. Bentivenha, E. L. L. Baldin, D. G. Montezano, T. E. Hunt, and S. V. Paula-Moraes. Attack and defense movements involved in the interaction of Spodoptera frugiperda and Helicoverpa zea (Lepidoptera: Noctuidae). *Journal of Pest Science*, 90(2):433–445, 2017.

[13] J. P. F. Bentivenha, D. G. Montezano, T. E. Hunt, E. Ll, J. A. Peterson, V. S. Victor, E. R. Pannuti, M. Vélez, and S. V. Paula-moraes. Intraguild interactions and behavior of Spodoptera frugiperda and Helicoverpa spp. on maize. *Pest Management Science*, 73:2244–2251, 2017.

[14] D. Bernardi, E. Salmeron, R. J. Horikoshi, O. Bernardi, P. M. Dourado, R. A. Carvalho, S. Martinelli, G. P. Head, and C. Omoto. Cross-resistance between

Cry1 Proteins in Fall Armyworm (Spodoptera frugiperda) May Affect the Durability of Current Pyramided Bt Maize Hybrids in Brazil. *PLoS ONE*, 10(10):1–15, 2015.

[15] T. Billeisen and R. Brandenburg. Fall Armyworms in Turf, 2017.

[16] C. P. D. Birch, S. P. Oom, J. A. Beecham, B. V. Dhv, P. O. Box, and B. C. Amersfoort. Rectangular and hexagonal grids used for observation, experiment and simulation in ecology. *Ecological Modelling*, 206(3-4):347–359, 2007.

[17] K. L. Blaxter. Bioenergetics and growth: the whole and the parts. *Journal of Animal Science*, 63(Suppl. 2):1–10, 1986.

[18] T. Boivin, J. Chadœuf, J.-C. Bouvier, D. Beslay, and B. Sauphanor. Modelling the interactions between phenology and insecticide resistance genes in the codling moth Cydia pomonella. *Pest Management Science*, 61(1):53–67, 2005.

[19] Bokeh Development Team. *Bokeh: Python library for interactive visualization*, 2019.

[20] D. Bourguet, A. Genissel, and M. Raymond. Insecticide Resistance and Dominance Levels. *Journal of Economic Entomology*, 93(6):1588–1595, 2000.

[21] L. B. Brattsten, C. W. Holyoke Jr, J. R. Leeper, and K. F. Raffa. Insecticide Resistance: Challenge to Pest Management and Basic Research. *Science*, 231(4743):1255–1260, 1986.

[22] J. L. Capinera. *Fall Armyworm , Spodoptera frugiperda (J. E. Smith) (Insecta: Lepidoptera: Noctuidae)*. University of Florida Cooperative Extension Service, Institute of Food and Agricultural Sciences EDS, 2017.

[23] M. A. Caprio and B. E. Tabashnik. Gene Flow Accelerates Local Adaptation Among Finite Populations: Simulating the Evolution of Insecticide Resistance. *Journal of Economic Entomology*, 85(3):611–620, 1992.

[24] H. Caswell. *Matrix Population Models Construction, Analysis, and Interpretation*. Sinauer, second edition, 2001.

[25] J. W. Chapman, T. Williams, A. N. A. Escribano, P. Caballero, and R. D. Cave. Age-related cannibalism and horizontal transmission of a nuclear polyhedrosis virus in larval Spodoptera frugiperda. *Ecological Entomology*, 24:268–275, 1999.

[26] J. W. Chapman, T. Williams, A. M. Martínez, and J. Cisneros. Does cannibalism in Spodoptera frugiperda (Lepidoptera: Noctuidae) reduce the risk of predation? *Behavioral Ecology and Sociobiology*, 48(2):321–327, 2000.

[27] A. Chormule, N. Shejawal, S. Deshmukh, C. M. Kalleshwaraswamy, R. Asokan, and H. M. Swamy. First report of the fall Armyworm , Spodoptera frugiperda (J. E. Smith) (Lepidoptera, Noctuidae) on sugarcane and other crops from Maharashtra, India. *Journal of Entomology and Zoology Studies*, 7(1):114–117, 2019.

[28] M. J. W. Cock, P. K. Beseh, A. G. Buddie, G. Cafá, and J. Crozier. Molecular methods to detect Spodoptera frugiperda in Ghana, and implications for monitoring the spread of invasive species in developing countries. *Scientific Reports*, 7(4103):1–10, 2017.

[29] M. Coleman, J. Hemingway, K. A. Gleave, A. Wiebe, P. W. Gething, and C. L. Moyes. Developing global maps of insecticide resistance risk to improve vector control. *Malaria Journal*, 16(86):1–9, 2017.

[30] R. Consortium. Structure of the Scientific Community Modelling the Evolution of Resistance. *PLoS ONE*, 2(12):1–9, 2007.

[31] N. R. Council. *Pesticide Resistance: Strategies and Tactics for Management*. The National Academies Press, Washington, DC, 1986.

[32] J. D. Cryer and K.-S. Chan. *Time Series Analysis With Applications in R*. Springer, New York, 2008.

[33] C. F. Curtis. Theoretical models of the use of insecticide mixtures for the management of resistance. *Bulletin of Entomological Research*, 75(2):259–266, 1985.

[34] V. Dangal and F. Huang. Fitness costs of Cry1F resistance in two populations of fall armyworm, Spodoptera frugiperda (J. E. Smith), collected from Puerto Rico and Florida. *Journal of Invertebrate Pathology*, 127:81–86, 2015.

[35] A. M. De Roos and L. Persson. *Population and Community Ecology of Ontogenetic Development*. Princeton University Press, 2013.

[36] D. H. Dean. Biochemical Genetics of the Bacterial Insect-Control Agent Bacillus thuringiensis: Basic Principles and Prospects for Genetic Engineering. *Biotechnology and Genetic Engineering Reviews*, 2(1):341–363, 1984.

[37] I. Denholm, J. A. Pickett, A. L. Devonshire, J. A. Mckenzie, and P. Batterham. Predicting insecticide resistance: mutagenesis, selection and response. *Philosophical Transactions of the Royal Society of London. Series B: Biological Sciences*, 353(1376):1729–1734, 1998.

[38] S. Deshmukh, C. Kalleshwaraswamy, R. Asokan, M. H.M., M. Maruthi, B. Pavithra, K. Hegde, S. Navi, S. Prabhu, and G. Goergen. First report of the

Fall armyworm, Spodoptera frugiperda (J E Smith) (Lepidoptera: Noctuidae), an alien invasive pest on maize in India. *Pest Management in Horticultural Ecosytems*, 24(1):23–29, 2018.

[39] L. Despres, J.-p. David, and C. Gallet. The evolutionary ecology of insect resistance to plant chemicals. *Trends in Ecology and Evolution*, 22(6):298–307, 2007.

[40] A. L. Devonshire and L. M. Field. Gene Amplification and Insecticide Resistance. *Annual Review of Entomology*, 36(1):1–21, 1991.

[41] E. W. Dijkstra. A note on two problems with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[42] R. Early, P. González-Moreno, S. T. Murphy, and R. Day. Forecasting the global extent of invasion of the cereal pest Spodoptera frugiperda, the fall armyworm. *NeoBiota*, 40, 2018.

[43] D. S. Falconer. *Introduction to Quantitative Genetics*. Longmand Scientific & Technical, fourth edition, 1989.

[44] J. R. Farias, D. A. Andow, R. J. Horikoshi, R. J. Sorgatto, A. C. dos Santos, and C. Omoto. Dominance of Cry1F resistance in Spodoptera frugiperda (Lepidoptera: Noctuidae) on TC1507 Bt maize in Brazil. *Pest Management Science*, 72:974–979, 2016.

[45] J. C. Fatoretto, A. P. Michel, M. C. S. Filho, and N. Silva. Adaptive Potential of Fall Armyworm (Lepidoptera: Noctuidae) Limits Bt Trait Durability in Brazil. *Journal of Integrated Pest Management*, 8(1), 2017.

[46] D. N. Ferro and N. Ferro. Potential for Resistance to Bacillus thuringiensis: Colorado Potato Beetle (Coleoptera: Chrysomelidae)A Model System. *American Entomologist*, 39(1):38–44, 1993.

[47] L. Flagel, Y. W. Lee, H. Wanjugi, S. Swarup, A. Brown, E. Kraft, J. Greenplate, J. Simmons, N. Adams, Y. Wang, S. Martinelli, J. A. Haas, A. Gowda, and G. Head. Mutational disruption of the ABCC2 gene in fall armyworm, Spodoptera frugiperda, confers resistance to the Cry1Fa and Cry1A.105 insecticidal proteins. *Scientific Reports*, 8(7255):1–11, 2018.

[48] K. L. Flanders, D. M. Ball, and P. P. Cobb. Management of Fall Armyworm in Pastures and Hayfields. *Alabama Cooperative Extension System*, ANR-1019:1–4, 2011.

[49] H. C. Giacomini, D. L. DeAngelis, J. C. Trexler, and M. J. Petrere. Trait contributions to fish community assembly emerge from trophic interactions in an individual-based model. *Ecological Modelling*, 251:32–43, 2013.

[50] D. S. Glazier. Beyond the '3/4-power law': Variation in the intra- and interspecific scaling of metabolic rate in animals. *Biological Reviews*, 80(4):611–662, 2005.

[51] G. Goergen, P. L. Kumar, S. B. Sankung, A. Togola, and M. Tam. First Report of Outbreaks of the Fall Armyworm Spodoptera frugiperda (J E Smith) (Lepidoptera , Noctuidae), a New Alien Invasive Pest in West and Central Africa. *PLoS ONE*, 11:1–9, 2016.

[52] D. F. Griffiths and D. J. Higham. *Numerical Methods for Ordinary Differential Equations Initial Value Problems*. Springer, New York, 2010.

[53] V. Grimm and S. F. Railsback. *Individual-based Modeling and Ecology*. Princeton University Press, 2004.

[54] V. Grimm, E. Revilla, U. Berger, F. Jeltsch, W. M. Mooij, S. F. Railsback, H.-H. Thulke, J. Weiner, T. Wiegand, and D. L. DeAngelis. Pattern-Oriented Modeling of Agent-Based Complex Systems: Lessons from Ecology. *Science*, 310(5750):987–991, 2005.

[55] A. T. Groot, M. Marr, D. Heckel, and G. Schofl. The roles and interactions of reproductive isolation mechanisms in fall armyworm (Lepidoptera: Noctuidae). *Ecological Entomology*, 35(1):105–118, 2010.

[56] J. T. Hardke, B. R. Leonard, F. Huang, and R. E. Jackson. Damage and survivorship of fall armyworm (Lepidoptera: Noctuidae) on transgenic field corn expressing Bacillus thuringiensis Cry proteins. *Crop Protection*, 30(2):168–172, 2011.

[57] N. J. Hawkins, C. Bass, A. Dixon, and P. Neve. The evolutionary origins of pesticide resistance. *Biological Reviews*, 94:135–155, 2019.

[58] J. M. C. Hutchinson and P. M. Waser. Use, misuse and extensions of "ideal gas" models of animal encounter. *Biological Reviews*, 82:335–359, 2007.

[59] D. A. Ingber, C. E. Mason, and L. Flexner. Cry1 Bt Susceptibilities of Fall Armyworm (Lepidoptera: Noctuidae) Host Strains. *Journal of Economic Entomology*, 111(1):361–368, 2018.

[60] JetBrains. PyCharm, 2019.

[61] D. Johnson. A Note on Dijkstra' s Shortest Path Algorithm. *Journal of the ACM*, 20(3):385–388, 1973.

[62] S. J. Johnson. Migration and the life history strategy of the fall armyworm, Spodoptera frugiperda in the western hemisphere. *International Journal of Tropical Insect Science*, 8(4-5-6):543–549, 1987.

[63] A. Kliot and M. Ghanim. Fitness costs associated with insecticide resistance. *Pest Management Science*, 68:1431–1437, 2012.

[64] H. Kokko. *Modelling for Field Biologists and Other Interesting People*. Cambridge University Press, 2007.

[65] S. A. Kooijman. *Dynamic energy budget theory for metabolic organisation*. Cambridge University Press, third edition, 2010.

[66] P. G. Lemaux. Genetically Engineered Plants and Foods: A Scientist's Analysis of the Issues (Part I). *Annual Review of Plant Biology*, 59(1):771–812, 2008.

[67] X.-j. Li, M.-f. Wu, J. Ma, B.-y. Gao, Q.-l. Wu, A. Chen, J. Liu, Y.-y. Jiang, B.-p. Zhai, R. Early, W. Jason, and G. Hu. Prediction of migratory routes of the invasive fall armyworm in eastern China using a trajectory analytical approach. *Pest Management Science*, 2019.

[68] S. M. Louda, T. A. Rand, A. A. R. Kula, A. E. Arnett, N. M. West, B. Tenhumberg, and B. Tenhumberg. Priority resource access mediates competitive intensity between an invasive weevil and native floral herbivores. *Biological Invasions*, 12:2233–2248, 2011.

[69] P. Luginbill. *The Fall Armyworm*. USDA Technical Bulletin, 1928.

[70] R. E. Lynch. Effects of 'Coastal' Bermudagrass Fertilization Levels and Age of Regrowth on Fall Armyworm (Lepidoptera: Noctuidae): Larval Biology and Adult Fecundity. *Journal of Economic Entomology*, 77(4):948–953, 1984.

[71] J. Mallet. The Evolution of Insecticide Resistance: Have the Insects Won? *Trends in Ecology and Evolution*, 4(11):336–340, 1989.

[72] D. Masad and J. Kazil. Mesa : An Agent-Based Modeling Framework. *Proceeings of the Python in Science Conference*, 14:51–58, 2015.

[73] W. Mckinney. Data Structures for Statistical Computing in Python. *Proceeings of the Python in Science Conference*, 9:51–56, 2010.

[74] A. Meurer, C. P. Smith, M. Paprocki, O. Čertík, S. B. Kirpichev, M. Rocklin, A. Kumar, S. Ivanov, J. K. Moore, S. Singh, T. Rathnayake, S. Vig, B. E. Granger, R. P. Muller, F. Bonazzi, H. Gupta, S. Vats, F. Johansson, F. Pedregosa, M. J. Curry, A. R. Terrel, Š. Roučka, A. Saboo, I. Fernando, S. Kulal, R. Cimrman, and A. Scopatz. SymPy: symbolic computing in Python. *PeerJ Computer Science*, 3:e103, jan 2017.

[75] C. A. O. Midega, J. O. Pittchar, J. A. Pickett, G. W. Hailu, and Z. R. Khan. A climate-adapted push-pull system effectively controls fall armyworm , Spodoptera frugiperda (J E Smith), in maize in East Africa. *Crop Protection*, 105(August 2017):10–15, 2018.

[76] A. D. G. Montezano, A. Specht, D. Sosa-Gómez, V. Roque-Specht, J. Sousa-Silva, S. Paula-Moraes, J. Peterson, and T. Hunt. Host Plants of Spodoptera frugiperda (Lepidoptera: Noctuidae) in the Americas. *African Entomology*, 26(2):286–300, 2018.

[77] A. R. N. Nagoshi and R. L. Meagher. Review of Fall Armyworm (Lepidoptera: Noctuidae) Gene Complexity and Migration. *The Florida Entomologist*, 91(4):546–554, 2008.

[78] R. N. Nagoshi, G. Goergen, K. A. Tounou, K. Agboka, D. Koffi, and R. L. Meagher. Analysis of strain distribution , migratory potential , and invasion history of fall armyworm populations in northern Sub- Saharan Africa. *Scientific Reports*, 8(3710):1–10, 2018.

[79] R. N. Nagoshi and R. L. Meagher. Behavior and Disribution of the Two Fall Armyworm Host Strains in Florida. *Florida Entomologist*, 87(4):440–449, 2004.

[80] Y. Niu, R. L. . M. Jr, F. Yang, and F. Huang. Susceptibility of field populations of the fall armyworm (Lepidoptera: Noctuidae) from Florida and Puerto Rico to purified Cry1f protein and corn leaf tissue containing single and pyramided bt genes. *The Florida Entomologist*, 96(3):701–713, 2013.

[81] B. C. Nolting, T. M. Hinkelman, C. E. Brassil, and B. Tenhumberg. Composite random search strategies based on non-directional sensory cues. *Ecological Complexity*, 22:126–138, 2015.

[82] S. P. Otto and T. Day. *A Biologist's Guide to Mathematical Modeling in Ecology and Evolution*. Princeton University Press, 2007.

[83] L. E. Pannuti, E. L. Baldin, T. E. Hunt, and S. V. Paula-Moraes. On-plant larval movement and feeding behavior of fall armyworm (Lepidoptera: Noctuidae) on reproductive corn stages. *Environmental Entomology*, 45(1):192–200, 2016.

[84] S. L. Peck, F. Gould, and S. P. Ellner. Spread of Resistance in Spatially Extended Regions of Transgenic Cotton: Implications for Management of Heliothis virescens (Lepidoptera: Noctuidae). *Journal of Economic Entomology*, 92(1):1–16, 1999.

[85] N. L. Pencoe and P. B. Martin. Fall Armyworm (Lepidoptera: Noctuidae) Larval Development and Adult Fecundity on Five Grass Hosts. *Environmental Entomology*, 11(3):720–723, 1982.

[86] F. Pérez and B. E. Granger. IPython: A System for Interactive Scientific Computing. *Computing in Science & Engineering*, 9:21–29, 2007.

[87] C. Piou, U. Berger, and V. Grimm. Proposing an information criterion for individual-based models developed in a pattern-oriented modelling framework. *Ecological Modelling*, 220(17):1957–1967, 2009.

[88] H. N. Pitre and D. B. Hogg. Development of the fall armyworm on cotton, soybean and corn. *Journal of the Georgia Entomological Society*, 18(2):182–187, 1983.

[89] H. N. Pitre, J. E. Mulrooney, and D. B. Hogg. Fall Armyworm (Lepidoptera: Noctuidae) Oviposition: Crop Preferences and Egg Distribution on Plants. *Journal of Economic Entomology*, 76(3):463–466, 1983.

[90] M. J. Plank, M. Auger-Méthé, and E. A. Codling. *Lévy or Not? Analysing Positional Data from Animal Movement Paths*, pages 33–52. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[91] F. W. Plapp, C. R. Browning, and P. J. H. Sharpe. Analysis of Rate of Development of Insecticide Resistance Based on Simulation of a Genetic Model. *Environmental Entomology*, 8(3):494–500, 1979.

[92] J. G. Polhill, D. C. Parker, D. G. Brown, and V. Grimm. Using the ODD protocol for comparing three agent-based social simulation models of land use

change. *JASSS - The Journal of Artificial Societies and Social Simulation*, 11(2):Article Number 3, 2008.

[93] J. R. Prasifka, J. D. Bradshaw, R. L. Meagher, R. N. Nagoshi, K. L. Steffey, and M. E. Gray. Development and Feeding of Fall Armyworm on Miscanthus x giganteus and Switchgrass. *Journal of Economic Entomology*, 102(6):2154–2159, 2009.

[94] Python Software Foundation. Python 3, 2019.

[95] K. F. Raffa. Effect of Host Plant on Cannibalism Rates by Fall Armyworm (Lepidoptera: Noctuidae) Larvae. *Environmental Entomology*, 16(3):672–675, 1987.

[96] S. F. Railsback and V. Grimm. *Agent-Based and Individual-Based Modeling A practical Introduction*. Princeton University Press, 2012.

[97] A. M. Reynolds. Bridging the gulf between correlated random walks and Lévy walks: autocorrelation as a source of Lévy walk movement patterns. *Journal of the Royal Society Interface*, 7(July):1753–1758, 2010.

[98] A. M. Reynolds. Current status and future directions of Lévy walk research. *Biology Open*, 2018(7):1–6, 2018.

[99] R. T. Roush and J. A. McKenzie. Ecological Genetics of Insecticide and Acaricide Resistance. *Annual Review of Entomology*, 32(1):361–380, 1987.

[100] V. M. Savage, J. F. Gillooly, W. H. Woodruff, G. B. West, A. P. Allen, B. J. Enquist, and J. H. Brown. The predominance of quarter-power scaling in biology. *Functional Ecology*, 18(2):257–282, 2004.

[101] E. Schnepf, N. Crickmore, J. Van Rie, D. Lereclus, J. Baum, J. Feitelson, D. R. Zeigler, and D. H. Dean. Bacillus thuringiensis and Its Pesticidal Crystal Proteins. *Mircorbiology and Molecular Biology Reviews*, 62(3):775–806, 1998.

[102] S. Seabold and J. Perktold. Statsmodels: Econometric and Statistical Modeling with Python. *Proceeings of the Python in Science Conference*, 9:57–61, 2010.

[103] E. Seier and K. H. Joblin. *Introduction to STATISTICS in a biological context*. CreateSpace, 2011.

[104] A. M. Shelton, J. D. Tang, R. T. Roush, T. D. Metz, and E. D. Earle. Field tests on managing resistance to Bt-engineered plants. *Nature Biotechnology*, 18:339–342, 2000.

[105] M. W. Siebert, J. M. Babock, S. Nolting, and A. C. Santos. Efficacy of Cry1F Insecticidal Protein in Maize and Cotton for Control of Fall Armyworm (Lepidoptera: Noctuidae). *Florida Entomologist*, 91(4):555–565, 2014.

[106] A. Sparks. A Review of the Biology of the Fall Armyworm. *The Florida Entomologist*, 62(2):82–87, 1979.

[107] T. C. Sparks and R. Nauen. IRAC: Mode of action classification and insecticide resistance management. *Pesticide Biochemistry and Physiology*, 121:122–128, 2015.

[108] B. F. Stone. A formula for determining degree of dominance in cases of monofactorial inheritance of resistance to chemicals. *Bulletin of the World Health Organization*, 38(2):325–326, 1968.

[109] N. P. Storer, J. M. Babcock, M. Schlenz, T. Meade, G. D. Thompson, J. W. Bing, and R. M. Huckaba. Discovery and Characterization of Field Resistance

to Bt Maize: Spodoptera frugiperda (Lepidoptera: Noctuidae) in Puerto Rico. *Journal of Economic Entomology*, 103(4):1031–1038, 2010.

[110] P. Stratonovitch, J. Elias, I. Denholm, R. Slater, M. A. Semenov, and R. N. C. Guedes. An individual-based model of the evolution of pesticide resistance in heterogeneous environments: Control of meligethes aeneus population in oilseed rape Crops. *PLoS ONE*, 9(12):1–24, 2014.

[111] S. Tavares, H. M. Monteiro, and T. P. M. Teixeira. Genetic basis of Cry1F resistance in two Brazilian populations of fall armyworm, Spodoptera frugiperda. *Crop Protection*, 81:154–162, 2016.

[112] C. E. Taylor and G. P. Georghiou. Suppression of Insecticide Resistance by Alteration of Gene Dominance and Migration. *Journal of Economic Entomology*, 72(1):105–109, 1979.

[113] S. van der Walt, S. C. Colbert, and G. Varoquaux. The NumPy Array: A Structure for Efficient Numerical Computation. *Computing in Science & Engineering*, 13:22–30, 2011.

[114] K. H. Veenstra, D. P. Pashley, and J. A. Ottea. Host-Plant Adaptation in Fall Armyworm Host Strains: Comparison of Food Consumption, Utilization, and Detoxication Enzyme Activities. *Annals of the Entomological Society of America*, 88(1):80–91, 1995.

[115] A. M. Velez, A. P. Analiza, E. E. Blankenship, and B. D. Siegfried. Effect of Cry1F maize on the behavior of susceptible and resistant Spodoptera frugiperda and Ostrinia nubilalis. *Entomologia Experimentalis et Applicata*, 159(1):37–45, 2016.

[116] A. M. Vélez, T. A. Spencer, A. P. Alves, A. L. Crespo, and B. D. Siegfried. Fitness costs of Cry1F resistance in fall armyworm, Spodoptera frugiperda. *Journal of Applied Entomology*, 138(5):315–325, 2014.

[117] A. M. Vélez, T. A. Spencer, A. P. Alves, D. Moellenbeck, R. L. Meagher, H. Chirakkal, and B. D. Siegfried. Inheritance of Cry1F resistance, cross-resistance and frequency of resistant alleles in Spodoptera frugiperda (Lepidoptera: Noctuidae). *Bulletin of Entomological Research*, 103(6):700–713, 2013.

[118] A. M. Vélez, N. N. Vellichirammal, J. L. Jurat-fuentes, and B. D. Siegfried. Cry1F resistance among lepidopteran pests : A model for improved resistance management? *Current Opinion in Insect Science*, 15, 2016.

[119] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. Jarrod Millman, N. Mayorov, A. R. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, b. Polat, Y. Feng, E. W. Moore, J. Vand erPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and S. . . Contributors. SciPy 1.0–Fundamental Algorithms for Scientific Computing in Python. *arXiv e-prints*, page arXiv:1907.10121, jul 2019.

[120] G. B. West, J. H. Brown, and B. J. Enquist. A general model for ontogenetic growth. *Nature*, 413:628, 2001.

[121] G. B. West, B. J. Enquist, and J. H. Brown. Modelling universality and scaling. *Nature*, 420(6916):626–627, 2002.

[122] J. K. Westbrook, R. N. Nagoshi, R. L. Meagher, S. J. Fleischer, and S. Jairam. Modeling seasonal migration of fall armyworm moths. *International Journal of Biometeorolgy*, 60(2):255–267, 2016.

[123] F. Whitford, S. S. Quisenberry, T. J. Riley, and J. W. Lee. Oviposition Preference, Mating Compatibility, and Development of Two Fall Armyworm Strains. *The Florida Entomologist*, 71(3):234–243, 1988.

[124] J. R. Wood, S. L. Poe, and N. C. Leppla. Winter Survival of Fall Armyworm Pupae in Florida. *Environmental Entomology*, 8(2):249–252, 1979.

[125] Q.-l. Wu, L.-m. He, X.-j. Shen, Y.-y. Jiang, J. Liu, G. Hu, and K.-M. Wu. Estimation of the Potential Infestation Area of Newly-invaded Fall Armyworm Spodoptera Frugiperda in the Yangtze River Valley of China. *Insects*, 10(9):298–313, 2019.

[126] G. Yang, K. E. Espelie, B. R. Wiseman, and D. J. Isenhour. Effect of Corn Foliar Cuticular Lipids on the Movement of Fall Armyworm (Lepidoptera: Noctuidae) Neonate Larvae. *The Florida Entomologist*, 76(2):302–316, 1993.

[127] G. Yang, B. R. Wiseman, and K. E. Espelie. Movement of Neonate Fall Armyworm (Lepidoptera: Noctuidae) Larvae on Resistant and Susceptible Genotypes of Corn. *Environmental Entomology*, 22(3):547–553, 1993.

[128] M. P. Zalucki and M. J. Furlong. ScienceDirect Behavior as a mechanism of insecticide resistance: evaluation of the evidence. *Current Opinion in Insect Science*, 21:19–25, 2017.