Investigating the Impact of Antagonistic Selection in Vector-Transmitted Parasitic Diseases

by

Aiden Lewis

A Thesis
Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Arts

Approved October 2024 by the
Graduate Supervisory Committee

Jesse Taylor, Chair
Nicolas Lanchier
Sharon Crook

ARIZONA STATE UNIVERSITY
December 2024

ABSTRACT

A model of a vector-transmitted parasitic disease, based primarily on *Plasmodium*, was developed to investigate the evolutionary dynamics of mutations experiencing selection in both the host and the vector, with a specific focus on different fitnesses for parasite alleles in the host and vector. The main contexts simulations were performed for were (i) host advantage only, (ii) vector advantage only, and (iii) antagonistic advantages favoring one allele in the host and the other allele in the vector. A control simulation with no advantages was also performed. Contexts (i) and (ii) behaved as expected, with the advantaged allele demonstrating higher fitness overall in both cases, and it was found that vector advantages are much weaker than host advantages in the case of selection. Antagonistic selection saw neither allele achieve the same degree of dominance it would have done on its own; however, the host-favored allele was still generally more prevalent than the vector-favored allele in both the host and vector populations, albeit to a lesser degree in the vector relative to the host. The difference between the two populations was small but meaningful. Similar behavior was observed for selection and transmission advantages, but the significant disparity between hosts and vectors for selection was not observed for transmission.

TABLE OF CONTENTS

Chapter 1

INTRODUCTION

Pathogens can be transmitted in a variety of different ways. Some, like influenza and tuberculosis, are airborne; others, like cholera and polio, are waterborne; and others still, like dengue and Zika, are vector-borne (Ather, 2024; “Drinking-water”, 2023; “Vector-borne diseases”, 2024). The focus of this paper is that last category of pathogen – namely, eukaryotic protozoan parasites that are transmitted between vertebrate hosts and hematophagous arthropods via blood meals. Such parasites cause a wide variety of harmful diseases, including leishmaniasis (*Leishmania*), Chagas disease (*Trypanosoma cruzi*), and in particular malaria (*Plasmodium*). Among these, malaria is especially noteworthy: in 2022 alone, there were approximately 249 million malaria cases and 608 thousand deaths (“Malaria”, 2023) from the disease, rendering it – and vector-transmitted parasites as a whole – one of the most significant issues facing the world today.

The life cycle of eukaryotic protozoan parasites is complex, and the manner in which they develop depends largely on whether they are inhabiting an arthropod vector or a vertebrate host. In the case of the malaria-causing *Plasmodium*, the sporozoite parasites – following a transmission from an infected female *Anopheles* mosquito to a susceptible human – first migrate to the liver and begin replicating as merozoites, remaining latent for several days (Aron, 1988). During this period, the parasites multiply rapidly before spreading from the liver to the blood stream, at which point they begin to penetrate erythrocytes and consume their contents (White, 2017). It is this stage of the parasite’s life cycle that is responsible for the deleterious symptoms of malaria. While inside the blood stream, the parasites are asexual with haploid genotypes, some of which being the gametocytes they gradually produce over time. Though they comprise a relatively small fraction of the infected host's net parasite population, only the gametocytes may get transmitted to a susceptible mosquito when it takes a blood meal, not the more prevalent merozoites of the previous developmental stage. Upon a mosquito's infection by the host's gametocytes, the parasites develop into ookinetes and enter the midgut, developing again into oocysts before undergoing meiosis and transitioning finally to the sporozoite stage (Churcher, 2010). Unlike in the host, the parasite does not remain haploid throughout the process – rather, the ookinete and oocyst stages are diploid, having developed from the formation of a zygote by a pair of gametocytes (Bargués-Ribera, 2019). As sporozoites, the parasites reside within the mosquito's salivary glands; if it takes a blood meal from a susceptible host, the sporozoites may then be transmitted to them, thus renewing the cycle once more.

For our purposes, the most important parts of the parasite's life cycle are the haploid asexual stage within the host, the transmission of gametocytes from the host to the vector, the diploid oocyst stage within the vector, and the transmission of sporozoites from the vector to the host. The effectiveness with which a parasite strain succeeds during each stage of development – and, in particular, how different strains perform relative to one another – depends in large part on its specific traits. Certain traits might increase a strain's fitness within the host but decrease it within the vector, or vice versa; likewise, certain traits might increase the strain's transmission probability from an infected vector to a susceptible host, but decrease it from an infected host to a susceptible vector. This is termed antagonistic selection, and it has been the subject of some of the research into malaria – in 2003, Ferguson *et al.*

studied the positive correlation between parasite virulence (the harm it causes to its vertebrate host) and transmission, focusing specifically on what may be limiting the further evolution of increased virulence. Their hypothesis was that the more virulent strains might increase infected vector mortality; this would be a form of antagonism, as the parasite would be more infectious in the host at the cost of its fitness within the vector. They did not find evidence to support this conclusion, but the question nonetheless remains of interest.

The mechanisms by which these traits develop and spread through a population of parasites are fundamentally genetic in nature. Genes are composed of nucleotides, and during the cell replication process, the nucleotides of a cell's genetic code may not all be copied flawlessly to its offspring. One nucleotide might be switched out for another, which constitutes a point mutation; such is the kind of mutation this paper is primarily concerned with. The majority of point mutations do not have any effect, but they can sometimes meaningfully impact the expression of the gene that experienced the mutation. If the fitness of the mutated-allele strain within an individual is higher than that of the wild-type-allele strain, it may – by natural selection – increase in frequency in the overall population, possibly out-competing the wild-type-allele strain entirely if the disparity is severe enough. Quantifying fitnesses numerically, we can describe the change in frequency of an allele across successive generations with the following equation (Orr, 2009):

$$p' = \frac{pW_1}{pW_1 + qW_2}, \tag{1.1}$$

where p is the frequency of allele 1, W_1 is the relative fitness of allele 1, q is the frequency of allele 2, and W_2 is the relative fitness of allele 2. p' represents the frequency of allele 1 in the next generation. p and q must necessarily sum to 1, and it is helpful to consider the fitness of a mutated allele (allele 1) relative to its wild-type counterpart (allele 2), i.e., to set the wild-type allele's relative fitness to 1. Accordingly, Eq. 1.1 can be rewritten as:

$$p'_m = \frac{p_m(s+1)}{1+p_m s}, \tag{1.2}$$

where p_m is the mutated allele's frequency and s is the selection coefficient ($W_1 = 1+s$).

Evolution is not necessarily a deterministic process. Random chance can play a role in determining which individuals get to produce offspring; a mutation may be beneficial, but if the only individual with the mutation happens to be unable to pass it down to the next generation, it will disappear from the population (at least, for that particular generation). This process – by which allele frequencies change from generation to generation according to pure chance, not fitness advantages – is termed genetic drift. As suggested by the law of large numbers, however, genetic drift is expected to be weaker in larger populations – for an infected vertebrate host like a human, who can host on the order of a hundred million parasites within their body (White, 2017), the process can be treated as deterministic and modelled with Eq. 1.2 on its own. The parasite populations of infected arthropod vectors, meanwhile, are several orders of magnitude smaller; mosquitoes tend to host well under a hundred oocyst parasites (Churcher, 2010), where the effect of drift is much more significant. A common model for genetic drift is the Wright-Fisher model, which treats generations as distinct and uses them as the basis of a discrete-time Markov chain. The elements of the Markov chain's transition probability matrix are given by the binomial sampling probability (Messer, 2016):

$$P_{ij} = \binom{N}{j} \left(\frac{i}{N}\right)^{j} \left(1 - \frac{i}{N}\right)^{N-j}, \qquad i, j \in [0, N], \tag{1.3}$$

where i and j represent the initial and final allele counts for the generation respectively, N is the total size of the allele pool, and P_{ij} is the probability that the system will transition from state i to state j.

Selection within a host or vector is not the only way a mutation can impact the parasite's overall fitness in the population as a whole, however. When a mosquito takes a blood meal, the transmission of the infection in either direction is not guaranteed – the parasite will fail to make the transition more often than not, and even a slight increase or decrease to the probability with which this occurs may significantly affect how a particular strain performs, even if it has no selection advantages inside either population's individuals. As transmission probabilities are largely analogous to selection advantages, many of the same questions are worth asking in both contexts – namely, the relative strengths of transmission advantages in vectors & hosts and how antagonistic transmission advantages impact the disease's behavior. By modelling these genetic and epidemiological processes, this paper seeks to investigate how certain traits can develop in a population of vector-transmitted eukaryotic parasites and how different factors can influence that process, thus bettering our understanding of this incredibly important and consequential kind of pathogen.

Chapter 2

METHODOLOGY

2.1 The macroscopic model

We chose to represent the macroscopic side of the model – that is, the dynamics of the parasite in the host and vector populations as a whole – using a stochastic SIRS model. At their core, SIR models track three types of individuals: susceptible, infected, and recovered. Individuals are born susceptible, but they can become infected, at which point they will gradually recover and become immune. For an SIRS model like ours, immunity is temporary and will wane over time, rendering recovered individuals susceptible to the disease once more. For the sake of simplicity, certain aspects of the model are assumed to be negligible for each of the two population types: hosts neither die nor are born, keeping their population constant, and vectors' lifespans are treated as sufficiently short that death will typically precede recovery. Such models are often simulated using a continuous-time Markov chain, but ours takes a slightly different approach.

Continuous-time Markov chains find a set of transition rates R_n, each respectively corresponding to simulation events E_n, and choose a time between events according to an exponential random variable depending on the sum of those R_n. The higher the overall rate of events is, the lower the times between events will be, and so the simulation is effectively continuous with respect to time. However, random numbers are computationally expensive to generate, as is dynamically allocating memory to store the results of each successive event. (Preallocating memory can mitigate the latter issue, but that will almost always be either insufficient or excessive, so it is still an imperfect solution.) Moreover, the transition rates are sufficiently large that multiple events will generally happen each day, and they will not change enough in magnitude across the length of the simulation in order to warrant using a continuous-time Markov chain with a dynamic time step. Instead, we can achieve much the same thing by approximating it as a discrete-time Markov chain, with the number of times each event occurs per (uniform) time step being binomially distributed. The events' probabilities are determined according to the relative sizes of their corresponding transition rates.

Let the counts of susceptible, infected, and recovered individuals be denoted as S, I, and R respectively. The system may inhabit any state S, I, R in which its elements are positive integers, i.e. S, I, $R \in [0, 1, 2...)$ respectively. The events that may occur during the simulation and their transition rates are given in the following table:

Rate meaning	**Expression**	ΔS	ΔI	ΔR
Recovery	γI	0	-1	$+1$
Waning immunity	ωR	$+1$	0	-1
Infected contact	$\kappa S_v I_h / N_v$	-1^*	$+1^*$	0

(a) Host population

Rate meaning	**Expression**	ΔS	ΔI
Susceptible death	μS	-1	0
Infected death	μI	0	-1
Birth	μN	$+1$	0
Infected contact	$\kappa S_h I_v / N_h$	-1^*	$+1^*$

(b) Vector population

Table 2.1, Macroscopic events and transition rates. κ is the contact rate, μ is the birth/death rate, γ is the recovery rate, and ω is the waning immunity rate. S is the susceptible count, I is the weighted infected count, and R is the recovered count. N is the total population. Subscripts indicate the relevant population type to draw from where it is unclear. Δ denotes each compartment's state changes for each event. The stars on the state changes for infected contacts indicate that contacts do not necessarily result in infections.

It is worth noting that the individuals considered "susceptible" for infected contacts are both those who are uninfected and those who are infected with a different strain only. This allows for secondary infections to occur, though they are comparatively unlikely and constitute a relative minority of infection events. The variables are not all the same as in a basic SIR model (Drake, 2016) for two main reasons: first, the need to accommodate multiple strains of the infection, and second, complications stemming from the microscopic side of the model (discussed fully in the next section). The adjustments made and the reasoning behind them are as follows.

The contact rate, κ. The infection rate used in standard SIR models is, in reality, the product of the contact rate κ and the transmission probability of that particular strain. Transmission probabilities are handled on a microscopic basis, so the parameter pertinent to the macroscopic context is the contact rate.

The weighted infected count, *I*. I is not simply the number of infected individuals; rather, each individual's contribution to the total number is weighted according to the particular strain's prevalence within the gene pool of its internal parasite population. In other words, a host with 10% strain D and 90% strain d will contribute a value of 0.1 to D's I and 0.9 to d's I. Uniform populations of mixed individuals (i.e. allele frequencies as before) and mixed populations of uniform individuals (i.e. 10% uniformly D, 90% uniformly d) are distinguished from each other by retroactively choosing an individual to serve as the source of the infection when the events' state changes are applied.

2.2 Compartment diagram

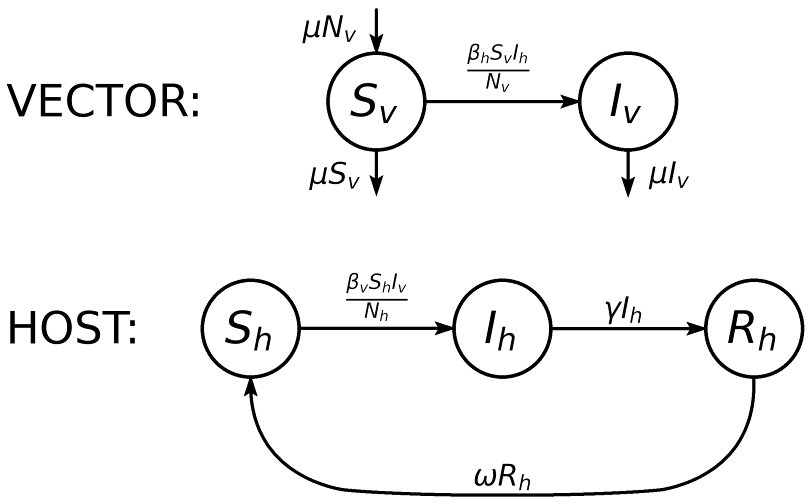


Figure 2.1, Compartment diagram. β denotes infection rate, i.e. the product of contact rate and transmission probability. Subscripts denote the relevant population type.

2.3 The microscopic model

Following each time step of the macroscopic model, the dynamics of infected individuals' internal parasites are simulated explicitly. Parasites in hosts are haploid and carry either the D (mutated) or d (wild-type) allele, while parasites in vectors are diploid and come in three genotypes (DD, Dd, dd). This makes use of two main algorithms: mutations and genetic drift. (Natural selection is integrated with genetic drift, as it directly affects the allele frequencies that are used in the binomial sampling of the Wright-Fisher model.) Mutations are assumed to only happen in hosts – they carry several orders of magnitude more parasites than vectors, so the overwhelming majority of mutations will happen there, allowing parasite mutations within vectors to be presumed negligible – and the Wright-Fisher model is only used in vectors. The number of parasite generations that get simulated per macroscopic time step is based on the parasites' lifespan and the length of the time step.

The mutation algorithm is quite simple: the number of mutations that occur per allele is determined according to a Poisson distribution, with the distribution's parameter being the product of the total number of parasites with that allele and the chance of mutation per parasite. If the parameter is sufficiently high (> 250), as it often is for the predominant allele inside the host, the Poisson distribution is instead approximated as being simply deterministic. This allows the algorithm to significantly cut down on computation time at no meaningful cost to the simulation's accuracy. The limit of 250 was chosen in order to strike the best balance between these two considerations – most parameters for the Poisson distribution are either very large or very small (for the more and less prevalent alleles within an infected host respectively), and for our mutation rate and host parasite population, the large parameters tend to be in the range of 280-300. Setting the threshold at 250 allows for all of these to be approximated without interfering with the smaller, more chance-dependent context.

The natural selection and genetic drift algorithm is somewhat more complex, though not overly so. As this is a single-locus model, only one allele's frequency needs to be tracked – we chose to explicitly track the mutated allele's frequency, though the wild-type allele would have been an equally valid choice. The algorithm's behavior differs between hosts and vectors. For vectors, the allele's frequency in the following generation is determined by a binomial distribution and Eq. 1.2, which biases the distribution's probability parameter p. (The count parameter N in the binomial sampling is the total number of alleles, i.e. parasite population times two due to their being diploid in that stage of development. With regards to Eq. 1.2, this essentially means that homozygous parasites are treated as normal, but heterozygous parasites – with only one copy of the mutated allele – have their selection coefficient halved. As dominance is not a feature of this model, this ensures both alleles are treated equally.) This is another, more efficient way of implementing the probabilities of the Wright-Fisher model as outlined in Eq. 1.3. The Wright-Fisher transition matrix could potentially have as many as 60 elements in each dimension; that would then require randomly choosing a result according to a list of 60 weights, which is needlessly time-consuming if the vast majority of those elements are inconsequentially small (as they would be, for any jumps of more than three or four alleles). Choosing a result according to a binomial distribution instead is much faster and more efficient from a computational perspective.

Once the final allele frequency has been decided, the alleles are distributed multinomially across the three genotypes according to the following probabilities, assuming Hardy-Weinberg equilibrium following sexual reproduction:

$$\begin{bmatrix} P_{dd}, & P_{Dd}, & P_{DD} \end{bmatrix} = \begin{bmatrix} q^2, & 2pq, & p^2 \end{bmatrix}, \tag{2.1}$$

where p is the frequency of the mutated allele D and q is the frequency of the wild-type allele d.

The microscopic side of the model interfaces with the macroscopic side in a few different ways: the adjusted infected counts, choosing individuals to perform infections, and translating contacts into infections. As the first two have already been discussed, we shall focus now on the third. When a contact occurs, the weighted average of the alleles' transmission probabilities is used to determine whether or not an infection was successful. If it was, then it transmits a certain number of parasites to the newly-infected individual, with the parasites' overall genetic composition determined by a multinomial distribution based on the alleles' relative transmission probabilities.

2.4 The code

The simulation was implemented in the coding language Python due to its object-oriented nature and the wide variety of packages that have been written to extend its functionality, many being intended specifically for scientific computing. As an object-oriented language, it is relatively easy to ensure that Python code remains readable throughout the development process: for instance, if one wants to use the genotype frequencies of an infected individual, they need only access the `genotype_freqs` property of the `individual` object. Similarly, if one wants to perform a mixed-strain infection on a population, they can call the `infectMix` method of the `population` object with the strain distribution as a parameter. In general, the classes and methods of the code were written in order to maintain readability – and semantic clarity – without unnecessarily sacrificing accuracy or efficiency.

Even still, there were certain tradeoffs that had to be made – namely, between generality and speed. The microscopic model originally allowed for multiple loci and alleles, with the intent of investigating the effect competing selection pressures between alleles would have on the overall genotype distribution; managing multiple loci was, however, highly impractical with regards to computational cost. In a broader sense, the biological processes the simulation is modelling are vastly more complex than the model itself would suggest – a model attempting to incorporate all of them would be prohibitively expensive to simulate, not to mention nigh impossible to formulate in the first place. Ultimately, we managed to strike a fair balance between the two considerations, producing accurate results in a reasonable amount of time.

2.5 Numerical experiments

We performed multiple simulations per experiment in order to improve the quality of our results. The mutated and wild-type alleles were generally evenly distributed in the initial infected host population. The parameters that did not vary between the simulation and their values were:

Parameter	**Variable**	**Value**	**Units**
Birth/death rate	`bd`	0.071	births/deaths per day
Recovery rate	`rr`	2.19e-3	recoveries per day
Average immunity duration	`tau`	90	days
Contact rate	`ct_rate`	0.28	bites per day
Transmission probability (host)	`hst_base_transm_p`	0.11	probability
Transmission probability (vector)	`vec_base_transm_p`	0.021	probability
Initial population ratio	`p0`	1:1	humans:mosquitoes
Total population per type	—	25201	individuals
Host parasite population	`pc`	10^8	parasites
Vector parasite population	`pc`	20	parasites
Mutation chance	`mut_chance`	2.94e-6	probability/parasite
Parasite lifespan	`para_lifespan`	8	hours
Parasites transmitted (host)	`pc_to_transmit`	10	parasites
Parasites transmitted (vector)	`pc_to_transmit`	15	parasites

Table 2.2, Numerical parameters used by the simulation. The variable names are as in the Python code. `pc_to_transmit` and `pc` are population-specific, each being properties of the `hosts_1` and `vectors` objects. Rate parameter values are averages.

The waning immunity rate `wi` (ω) was calculated according to the following equation (Aron, 1988):

$$\omega = \frac{\beta e^{-\beta\tau}}{1 - e^{-\beta\tau}}, \tag{2.2}$$

where β is the infection rate and τ is the average immunity duration `tau`. For the parameters outlined above, γ is found to be 2.05e-3 immunity losses per day.

Control. In the control case, neither allele had any selection advantages in either the host or the vector. Five simulations were run over a period of 5000 days each.

Host selection advantage. In the host selection advantage case, the mutated allele had a relative fitness of 1.002 within the host. Four simulations were run over a period of 10000 days each.

Vector selection advantage. In the vector selection advantage case, the mutated allele had a relative fitness of 1.5 within the vector. Three simulations were run over a period of 30000 days each.

Antagonistic selection & selection parameter contour plots. In order to most effectively demonstrate the behavior of antagonistic selection, we generated contour plots comparing final allele frequencies for a grid of selection parameters. The mutated allele's relative fitness varied over 0.996-1.004 in the host (y dimension, 5 total points) and 0.5-1.5 in the vector (x dimension, 13 total points). Two simulations were performed over a period of 10000 days for each point.

Host transmission advantage. In the host transmission advantage case, the transmission probability of a parasite carrying a mutated allele was 0.12 (for a parasite carrying a wild-type allele, 0.11) from hosts to vectors. Three simulations were run over a period of 20000 days.

Vector transmission advantage. In the vector transmission advantage case, the transmission probability of a parasite carrying a mutated allele was 0.024 (for a parasite carrying a wild-type allele, 0.021) from vectors to hosts. Three simulations were run over a period of 20000 days.

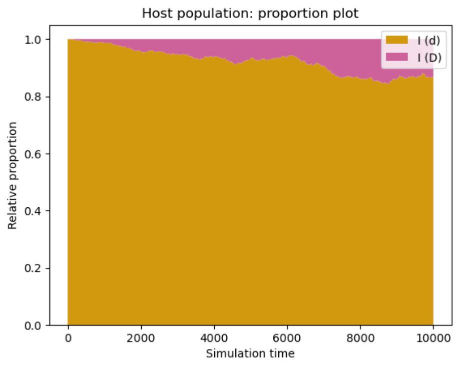
Antagonistic transmission & transmission parameter contour plots. As in the antagonistic selection case, we generated contour plots comparing allele frequencies for a grid of transmission parameters. The mutated allele's transmission probability varied over 0.10-0.12 in the host (y dimension, 9 total points) and 0.018-0.024 in the vector (x dimension, 9 total points). One simulation of 20000 days was performed for each point.

Chapter 3

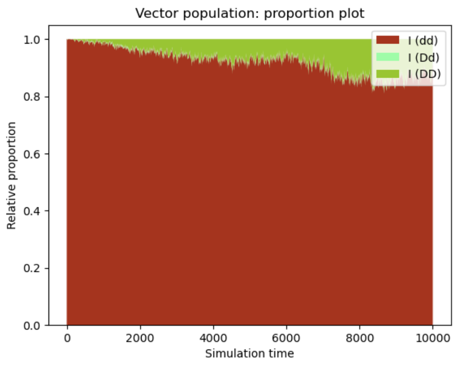
RESULTS

3.1 Control

For the control case, the expected long-term equilibrium frequencies of the two alleles are 0.5 each on average, as the two are equally fit in both populations. However, due to the relative rarity of mutations, the time the simulation would take to converge to its stationary distribution would be quite significant; the following graphs of the allele frequencies suggest as much.



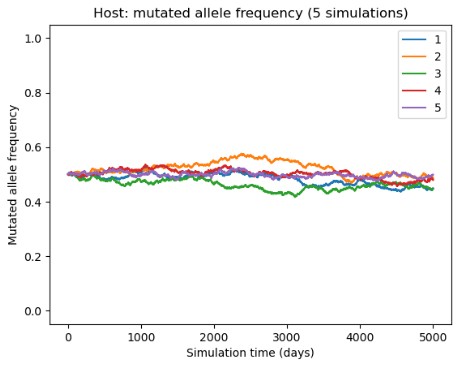
(a) Allele proportions, host parasite population.



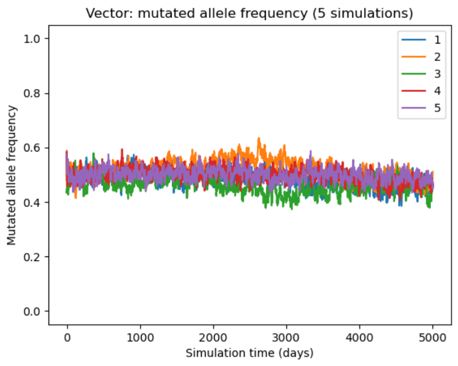
(b) Genotype proportions, vector parasite population.

Figure 3.1, Semantic example of the control case. Both alleles are equally fit in both populations. The x-axis represents simulation time in days, and the y-axis represents the proportions of each allele or genotype (distinguished by color) in the infected populations.

The mutated allele D's frequency grows incredibly slowly over time – the simulation length would likely need to be an order of magnitude or so larger to observe behavior characteristic of its stationary distribution, which is computationally impractical. Starting the simulation out with evenly-distributed alleles instead allows any deviations to be observed much more easily.



(a) Mutated allele frequencies, host parasite population.



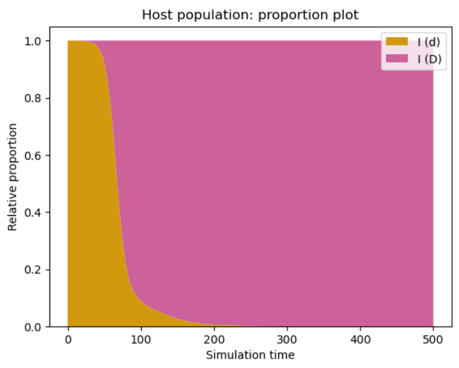
(b) Mutated allele frequencies, vector parasite population.

Figure 3.2, Control. The mutated allele has a relative fitness of 1.0 in both populations. Each line represents a different simulation trajectory. The y-axis represents the mutated allele's frequency, and the x-axis represents simulation time in days.

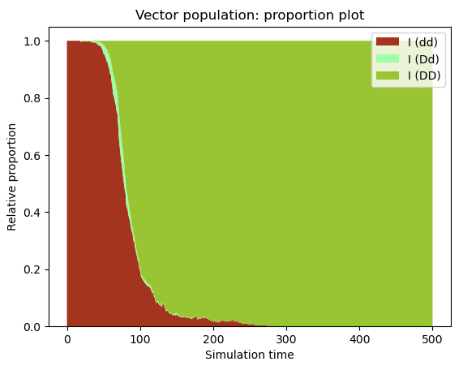
Overall, the equilibrium frequency of the mutated allele in the host population was found to be 0.524, with a standard deviation of 0.0167. As 0.5 is within two standard deviations of the mean, we can determine with reasonable confidence that our expectations have been confirmed.

3.2 Host selection advantage

The effect of even a small selection advantage for the mutated allele inside the host is significant, and it manifests in the results very rapidly. The graphs below were generated from a simulation using a relative fitness inside the host of 1.05 for the mutated allele over a period of 500 days:



(a) Allele proportions, host parasite population.

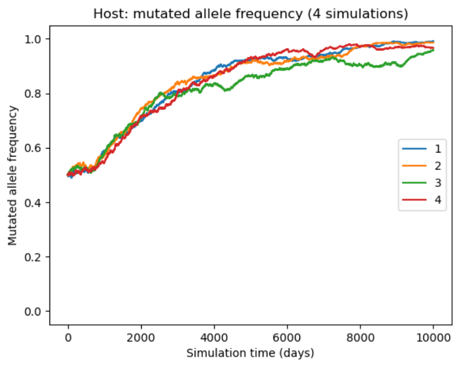


(b) Genotype proportions, vector parasite population.

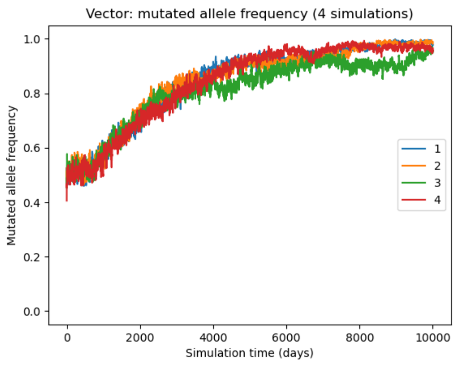
Figure 3.3, Invasion example of the host selection advantage case. The mutated allele D (relative fitness 1.05, host) invades and rapidly establishes dominance.

The allele frequencies in the host and vector populations are closely related to one another, and in only 250 days or so, the mutated allele has fully out-competed the wild-type allele despite having not been present in the initial infected population. Allowing for both alleles to coexist at non-trivial equilibrium frequencies requires that its selection advantage be reduced somewhat.

The following graphs were generated from multiple simulations using a relative fitness advantage for the mutated allele of 1.002 inside the host over a period of 10000 days:



(a) Mutated allele frequencies, host parasite population.



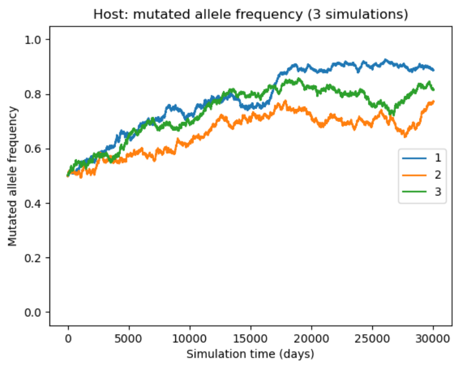
(b) Mutated allele frequencies, vector parasite population.

Figure 3.4, Host selection advantage. The mutated allele has a relative fitness of 1.002 in the host. Each line represents a different simulation trajectory. The y-axis represents the mutated allele's frequency, and the x-axis represents simulation time in days.

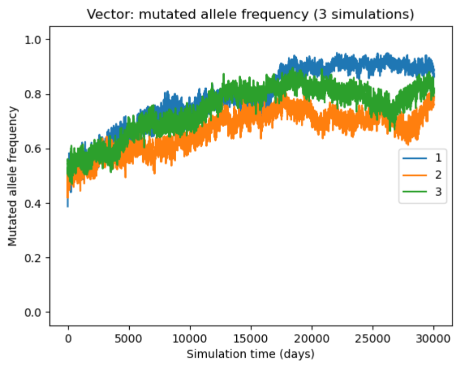
The mutated allele is still overwhelmingly predominant, but it does not manage to entirely out-compete the wild-type allele to the same degree as the previous case. The final frequency of the mutated allele in the host at the end of the simulation was found to be 0.963 overall, with a standard deviation of 0.027; considering how close this value is to 1, and how large the standard deviation is, it is likely that the true equilibrium frequency is higher still.

3.3 Vector selection advantage

The effect of a vector selection advantage is much smaller than the corresponding host selection advantage. The following graphs were generated using a relative fitness advantage for the mutated allele of 1.5 – ten times the relative fitness of the host selection advantage that proved so effective at dominating the gene pool – and simulated over a period of 30000 days:



(a) Mutated allele frequencies, host parasite population.



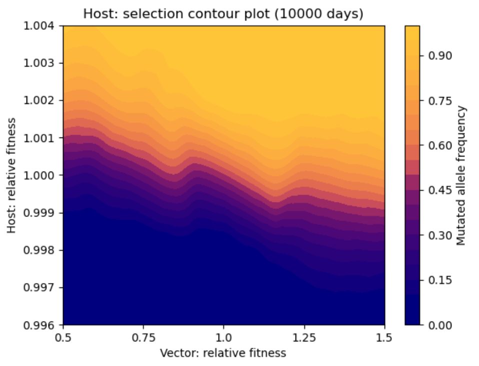
(b) Mutated allele frequencies, vector parasite population.

Figure 3.5, Vector selection advantage. The mutated allele has a relative fitness of 1.5 inside the vector. Each line represents a different simulation trajectory. The y-axis represents the mutated allele's frequency, and the x-axis represents simulation time in days.

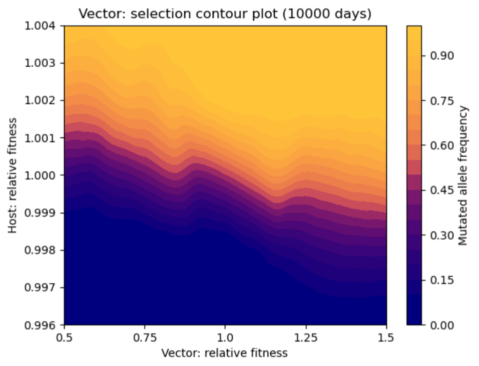
Averaging the final mutated allele frequencies of the simulations together, we find an overall frequency of 0.796 with a standard deviation of 0.084. Despite the relative fitness in this case being dramatically higher than its counterpart in the invasion example of section 3.2 (1.05), the mutated allele struggled to achieve a final frequency comparable to that of the coexistence example (1.002), even after 30000 days. As such, we can conclude that host selection advantages are significantly stronger than vector selection advantages.

3.4 Antagonistic selection

The case of antagonistic selection is especially interesting – with the prior three cases, it is possible to predict the broad strokes of the behavior, as the mutated allele universally performs either equivalently to or better than the wild-type allele. The same cannot be said of this context; if the two alleles are respectively favored by different populations, their pressures compete against one another, thus complicating the dynamics considerably. Accordingly, we must take a somewhat more complex and nuanced approach to data collection in this case. To that end, we generated a pair of contour plots showing the mutated allele's frequency in each population after a period of 10000 days for different sets of selection advantages. The range of values used for the mutated allele's fitness in the vector population was 0.5-1.5, and for the host population, it was 0.996-1.004.



(a) Mutated allele frequency contour plot, host parasite population.



(b) Mutated allele frequency contour plot, vector parasite population.

Figure 3.6, Selection parameter contour plots. The orange regions are those where the mutated allele is more prevalent, and the blue regions are those where the wild-type allele is more prevalent. The purple border between them represents those points with parameters of similar strength. The y-axis represents relative fitnesses in the host, and the x-axis represents relative fitnesses in the vector.

These graphs effectively demonstrate the severity of the discrepancy between the strengths of host and vector selection advantages – in most cases, even a slight relative fitness advantage for the mutated allele on the order of 10^{-3} in the host is enough to overcome a fitness disadvantage of order 10^{-1} in the vector, their differences in magnitude notwithstanding. The reasons for this behavior are related in large part to the relative parasite population sizes of hosts and vectors: as hosts carry several orders of magnitude more parasites than vectors do, the overwhelming majority of mutations happen there, enough so that approximating vector parasites as not mutating at all is reasonable. Consequently, a mutated allele will likely need to survive across multiple generations within the host before being transmitted to a vector – as such, the host-disfavored allele will struggle to succeed in the population as a whole, however strong its advantages within the vector may be. Moreover, the shortness of vectors' lifespans limits the number of parasite generations the vector-favored allele will be able to increase in frequency over, further inhibiting its performance.

Host selection advantages were not absolutely dominant over vector selection advantages, however; setting the two types of fitness parameters against one another did still sometimes produce non-trivial behavior where the allele performed better in one population than in the other. That this is the case is not easy to see when looking at the contour plots side-by-side, though – instead, a contour plot of the frequency differences is more appropriate.

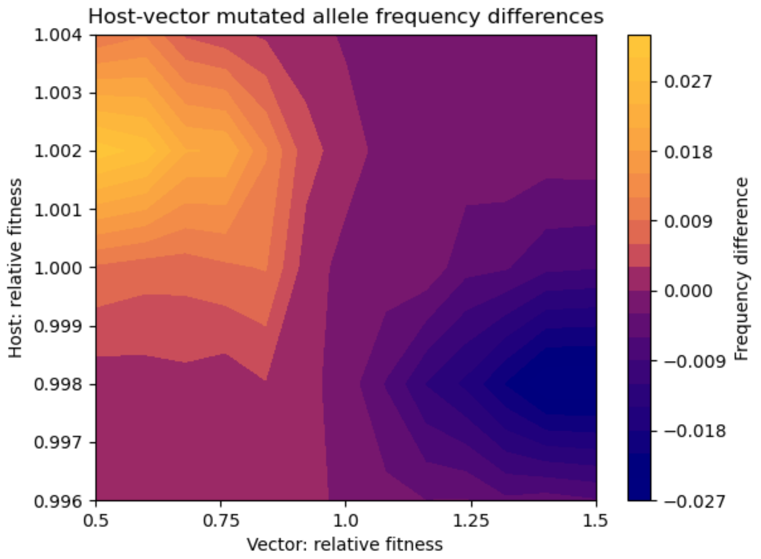
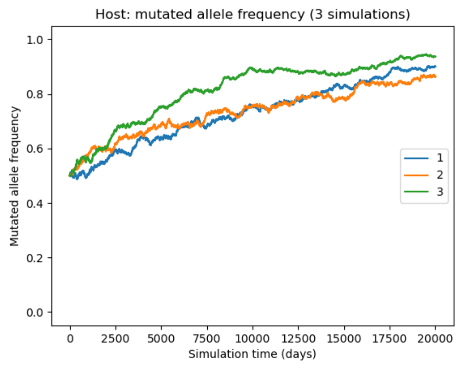


Figure 3.7, Selection parameter difference contour plot. The orange regions are those where the mutated allele is more prevalent in the host than in the vector, and the blue regions are those where it is more prevalent in the vector than in the host. The purple regions are those where there is no meaningful difference between the two populations. The y-axis represents relative fitnesses for the mutated allele in the host, and the x-axis represents relative fitnesses for the mutated allele in the vector.

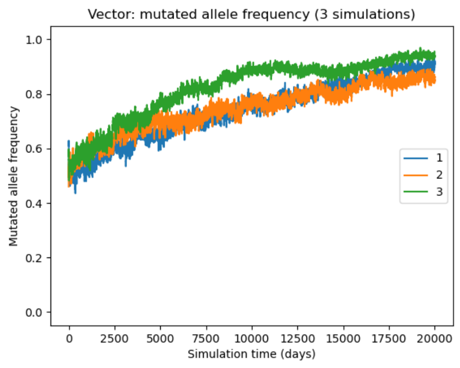
The differences may be comparatively small – no more than 3% or so in magnitude – but they exist nonetheless. The figure above depicts a clear and consistent pattern in the differences' behavior, and that is itself quite interesting: those parameters mutually favoring or disfavoring the mutated allele do not result in any meaningful differences in its performance between the host and vector populations, as expected, while the differences are strongest in the antagonistic quadrants of the plot. It is not determined solely by quadrant boundaries, however – in both the negative and positive regions of the plot, the zero-value contours 'curve' around them at the edges, indicating the point at which an advantage becomes strong enough to overwhelm the disadvantage in the other population opposing it.

3.5 Host transmission advantage

The similarities between selection and transmission advantages are apparent when observing their behavior within the simulation. The following graphs were generated using a host-vector transmission probability of 0.12 for the mutated allele (for the wild-type allele, 0.11) over a period of 20000 days, with three simulations being run in total:



(a) Mutated allele frequencies, host parasite population.



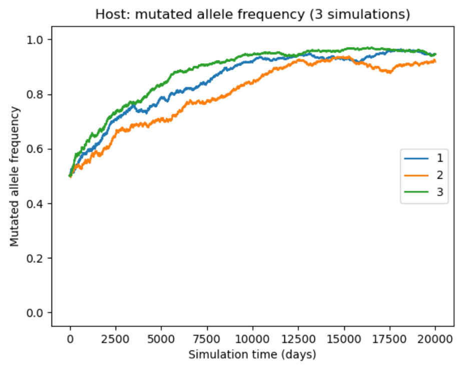
(b) Mutated allele frequencies, vector parasite population.

Figure 3.8, Host transmission advantage. The mutated and wild-type alleles had host-vector transmission probabilities of 0.12 and 0.11 respectively. Each line represents a different simulation trajectory. The y-axis represents the mutated allele's frequency, and the x-axis represents simulation time in days.

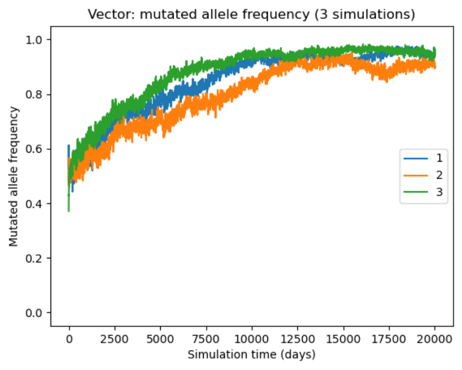
The final frequency of the mutated allele in the host was 0.882, with a standard deviation of 0.033. As conceptually similar as transmission advantages may be to selection advantages – both may be used to favor one particular allele, and both are population-specific – the behavior shown in the above graphs is visibly distinct from its counterpart in section 3.2, as is the final frequency value. This is due largely to the different manners in which the parameters act on the system: transmission probabilities are one of the components of the infection rate β (alongside contact rate), which is an element of the macroscopic part of the model, and so transmission advantages essentially render a mutated strain more infectious than its wild-type counterpart. In principle, this means that they have the potential to impact allele frequencies much more severely than the purely-microscopic selection advantages; however, the simulation's computational cost increases dramatically with the total number of infected individuals, rendering any especially substantial deviations from the base parameter values impractical to simulate. Accordingly, comparing the two contexts directly is infeasible, and we cannot make any meaningful conclusions about their relative strengths from the above graphs or statistics.

3.6 Vector transmission advantage

Interestingly, vector transmission advantages affect the allele frequencies notably differently to vector selection advantages – they are in fact fairly comparable to host transmission advantages in strength. The below graphs were generated from three simulations over 20000 days, where the mutated allele had a vector-host transmission probability of 0.024 (for the wild-type allele, 0.021):



(a) Mutated allele frequencies, host parasite population.



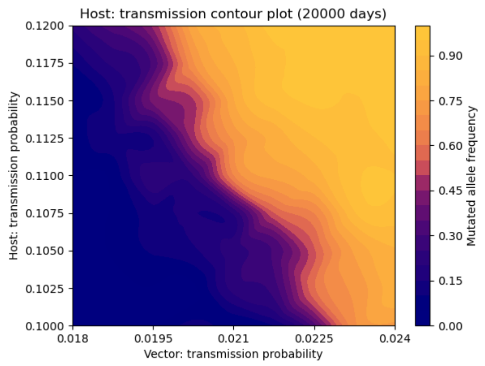
(b) Mutated allele frequencies, vector parasite population.

Figure 3.9, Vector transmission advantage. The mutated and wild-type alleles had vector-host transmission probabilities of 0.024 and 0.021 respectively. Each line represents a different simulation trajectory. The y-axis represents the mutated allele's frequency, and the x-axis represents simulation time in days.

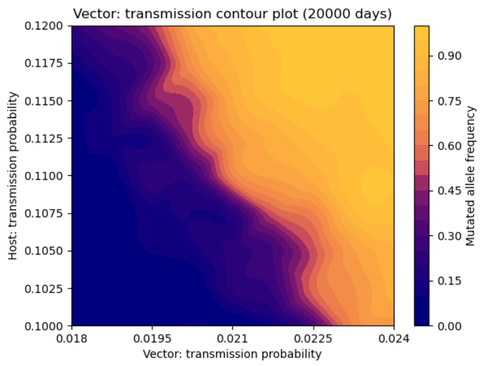
The final frequency of the mutated allele in the host was 0.938, with a standard deviation of 0.025. Though this is meaningfully higher than the value given in section 3.5, the relative advantage was also higher, being around 14% of the base value compared to around 9% previously. It is instead more useful to compare them with a contour plot, as in the following section – from a semantic perspective, there is reason to believe they likely will not differ much. The ultimate cause of the substantial difference in magnitudes between the effects of host and vector selection advantages was the disparity between their respective species' parasite population sizes; for transmission advantages, however, hosts and vectors are not treated remotely as differently. Transmission probabilities behave according to the same fundamental mechanism in the code between hosts and vectors, and if the infected individual's internal processes do not favor any one allele in particular, the nature of the transmission advantages' behavior is functionally no different from species to species.

3.7 Antagonistic transmission

Contour plots were generated for a 9-by-9 grid of points ranging over 0.10-0.12 in the y dimension (host) and 0.018-0.024 in the x dimension (vector), and each of the 81 simulations was run over a period of 20000 days:



(a) Mutated allele frequency contour plot, host parasite population.



(b) Mutated allele frequency contour plot, vector parasite population.

Figure 3.10, Transmission parameter contour plots. The orange regions are those where the mutated allele is more prevalent, and the blue regions are those where the wild-type allele is more prevalent. The purple border between them represents those points with parameters of similar strength. The y-axis represents host-vector transmission probabilities for the mutated allele, and the x-axis represents vector-host transmission probabilities for the mutated allele.

These graphs verify our earlier conclusion that transmission advantages are comparable in magnitude between hosts and vectors – the axes are scaled similarly, and slight advantages in one population tend to be balanced by slight disadvantages in the other. In order to investigate whether host or vector advantages are stronger, we can perform linear regression on those points of the contour plot whose frequency values are near 0.5 (i.e., in the range 0.45-0.55).

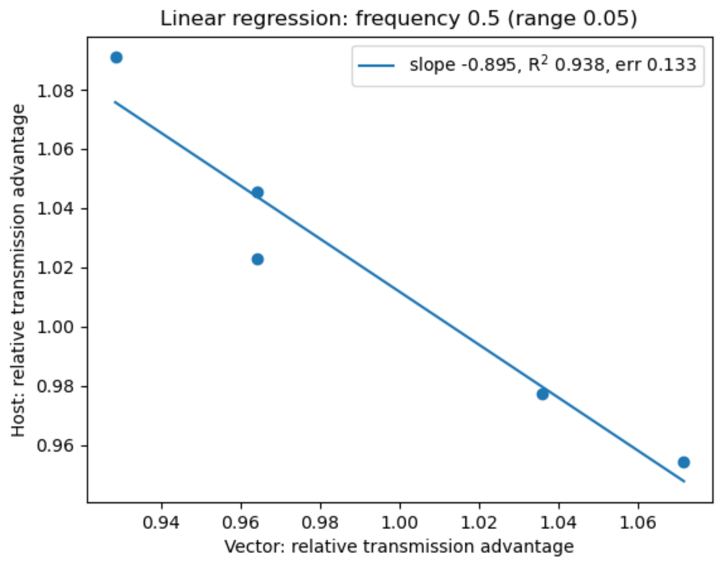


Figure 3.11, Transmission advantage linear regression. The axes are scaled according to the wild-type allele's transmission probability, rendering them relative transmission advantages. The y-axis represents the mutated allele's relative host-vector transmission advantage, and the x-axis represents the mutated allele's relative vector-host transmission advantage. The points represent parameters of roughly similar strength.

The line has a slope of -0.895, with an error of 0.133. A slope of -1 would indicate equal strength; -1 is within the range of the error bound, so we cannot conclude that either type of transmission advantage is meaningfully stronger than the other. As there are only five suitable data points, though, we also cannot necessarily say that they are equivalent outright; a higher contour resolution would be required to make a definitive statement on the matter.

To better observe antagonistic transmission behavior, we can graph the differences between the frequencies in the host and vector populations as a contour plot:

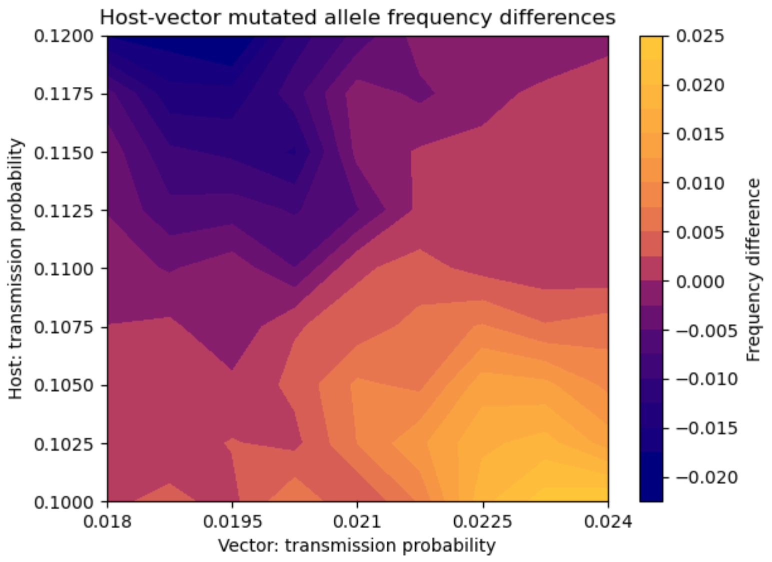


Figure 3.12, Transmission parameter difference contour plot. The orange regions are those where the mutated allele is more prevalent in the host than in the vector, and the blue regions are those where it is more prevalent in the vector than in the host. The purple regions are those where there is no meaningful difference between the two populations. The y-axis represents host-vector transmission probabilities for the mutated allele, and the x-axis represents vector-host transmission probabilities for the mutated allele.

It is quite similar in shape to its selection parameter counterpart, albeit rotated somewhat. The edge curving observed in the previous difference plot is present here as well, but the focus of the parameter grid appears to be comparatively constrained in this case, so it is less overt; the reduced magnitude of the frequency differences (in the vicinity of 2-2.5%) supports this interpretation. This suggests, though not necessarily demonstrates, that the antagonistic behavior for selection and transmission is consistent despite the fundamental processes driving them being largely unrelated to one another.

Chapter 4

CONCLUSIONS

Overall, we have found that selection advantages behave somewhat differently when applied on their own to parasites in host and vector populations respectively – host selection advantages are generally much stronger, while vector selection advantages are limited by the infected vectors' lack of parasite mutations. Even a small selection advantage for the mutated allele in the host allowed it to invade and establish complete dominance in only a few hundred days, whereas a selection advantage in the vector ten times larger was not remotely as successful, even after tens of thousands of days. When taken together and placed in opposition to one another, the results are quite interesting: the antagonistic quadrants of the parameter grid exhibit the strongest differences in allele frequency between the host and vector populations, but the curvature in the contours indicates that these differences are not monotonic with respect to any one axis. Transmission advantages behaved similarly, though the disparity between host and vector advantages seen for selection was absent.

The simulation was limited primarily by the significant computational costs associated with increases in its scope and complexity. The internal dynamics of the parasites' behavior within infected hosts and vectors occupy the bulk of its overall cost, rendering it highly dependent on total infected population size; the effective limit, with respect to reasonable computation times, is roughly a couple thousand infected individuals in total. This was especially significant for simulating transmission advantages, as they effectively increase macroscopic infection rate. Furthermore, the observation of non-trivial equilibrium frequencies would have required that selection be comparable in strength to mutation, which would have taken a substantial amount of time to observe in the simulations. Both limitations were particularly meaningful for the contour plots, due to the great number of separate simulations needed to create them – increasing the plots' resolution, lengths, and population sizes would be a natural extension of our work, as would an investigation of antagonism between selection and transmission themselves. Future research into this topic could benefit from using our code as a starting point, potentially optimising it and running it on a more powerful computer to conduct those expensive simulations, or expanding it to include multiple loci in order to investigate the many interesting questions that arise in such a context.

Chapter 5

REFERENCES

1. J. Aron, "Mathematical Modeling of Immunity to Malaria" (1988), *Mathematical Biosciences*, vol. 90, pp. 385-396.

2. B. Ather, T. Mirza, P. Edemekong, "Airborne Precautions" (Mar. 2023). Retrieved from `https://www.ncbi.nlm.nih.gov/books/NBK531468/`. Accessed October 2024.

3. M. Bargués-Ribera, R. Reeves, C. Gokhale, "Eco-evolutionary dynamics of *Plasmodium* genotypes under mass drug administration" (Oct. 2019). Retrieved from `https://www.researchgate.net/publication/336805141_Eco-evolutionary_dynamics_of_Plasmodium_genotypes_under_mass_drug_administration`. Accessed October 2024.

4. S. Bopp, M. Manary, A. Bright, G. Johnston, N. Dharia, F. Luna, S. McCormack, D. Plouffe, C. McNamara, J. Walker, D. Fidock, E. Denchi, E. Winzeler, "Mitotic Evolution of *Plasmodium falciparum* Shows a Stable Core Genome but Recombination in Antigen Families" (Feb. 2023), *PLOS Genetics*, vol. 9, iss. 2.

5. T. Churcher, E. Dawes, R. Sinden, G. Christophides, J. Koella, M. Basáñez, "Population biology of malaria within the mosquito: density-dependent processes and potential implications for transmission-blocking interventions" (Nov. 2010), *Malaria Journal*, vol. 9.

6. A. Drabo, F. Bere, S. Nitiema, "On a Stochastic Approach to Extensions of the Susceptible-Infected-Susceptible (SIS) Model Applied to Malaria" (Apr. 2024), *Journal of Applied Mathematics*, vol. 2024.

7. J. Drake, P. Rohani, "Stochastic Models" (Jul. 2016). Retrieved from `https://daphnia.ecology.uga.edu/drakelab/wp-content/uploads/2016/07/sismid-stochastic-lecture.pdf`. Accessed October 2024.

8. "Drinking-water," World Health Organization (Sept. 2023). Retrieved from `https://www.who.int/news-room/fact-sheets/detail/drinking-water`. Accessed October 2024.

9. H. Ferguson, M. Mackinnon, B. Chan, A. Read, "Mosquito mortality and the evolution of malaria virulence" (Dec. 2003), *Evolution*, vol. 57, iss. 12, pp. 2792-2804.

10. "Malaria," World Health Organization (Dec. 2023). Retrieved from `https://www.who.int/news-room/fact-sheets/detail/malaria`. Accessed October 2024.

11. S. Mandal, R. Sarkar, S. Sinha, "Mathematical models of malaria - a review" (2011), *Malaria Journal*, vol. 10.

12. P. Messer, "Neutral Models of Genetic Drift and Mutation" (2016), *Encyclopedia of Evolutionary Biology*, vol. 3, pp. 120-123.

13. "Mosquito Life Cycle," Environmental Protection Agency (Feb. 2024). Retrieved from `https://www.epa.gov/mosquitocontrol/mosquito-life-cycle`. Accessed October 2024.

14. H. Orr, "Fitness and its role in evolutionary genetics" (Jun. 2009), *Nature Reviews: Genetics*, vol. 10, pp. 531-539.

15. "Vector-borne diseases," World Health Organization (Sept. 2024). Retrieved from `https://www.who.int/news-room/fact-sheets/detail/vector-borne-diseases`. Accessed October 2024.

16. N. White, "Malaria parasite clearance" (2017), *Malaria Journal*, vol. 16.

APPENDIX A

PARAMETER SOURCES

The parameters outlined in table 2.2 and the sources of their values are as follows:

- Birth/death rate: “Mosquito Life Cycle”, 2024
- Recovery rate: Aron, 1988
- Contact rate, host & vector transmission probabilities: Drabo, 2024
- Initial population ratio: Mandal, 2011
- Host parasite population, vector parasites transmitted: White, 2017
- Vector parasite population, host parasites transmitted: Churcher, 2010
- Mutation chance (derived): Bopp, 2023

The average immunity duration and parasite lifespan were chosen so as to computationally expedite the collection of useful data. The number of parasites transmitted from host to vector was adjusted to align with the specific stage of the life cycle the simulation treats vector parasites as being in. The mutation chance was derived from a mutation rate, originally in units of mutations per site and translated into mutations per gene (i.e. per parasite, as this is a single-locus model).

All the code for the model can be found on Github (link: `https://github.com/aclewis242/paramodel.git`).

APPENDIX B

CODE FILES

```
'''
Main file for the host-vector parasite & SIRS model.
'''

from sim_lib import *
from allele import *
from color import *
from data_lib import *
from time import time
from scipy.interpolate import griddata
from scipy.stats import linregress
from PIL import Image, ImageDraw
from io import BytesIO
import matplotlib.pyplot as plt
import pandas as pd
import os
import cProfile
import pstats
import colorist

H1 = 'h1'
VEC = 'vec'
FULL_NMS = {'Vector': VEC, 'Host': H1}
FULL_NMS_R = {VEC: 'Vector', H1: 'Host'}

# Rates are in terms of '# events expected per day'
VCT = {         # Parameters for transmission vector behavior (
   mosquito)
  'bd': 0.071,  # Birth/death rate
  'ir': 0.,     # Infection rate between individuals. Largely
      obsolete for a vector-based model
  'rr': 0,     # Recovery rate
  'wi': 0.,     # Waning immunity rate
  'pn': VEC,         # Short population name, used in console
      output & within the code
  'pn_full': 'Vector',  # Full population name, used in graph
      titles
  'is_vector': True,    # Whether or not the population is a
      disease vector
}

hst_transm_p_wi = 0.11    # The base transmission probability for
   hosts, only used here to come up with wi_rate
ct_rate = 0.28         # Contact rate (average bites per day)
tau = 90           # Average duration of immunity (days)
h_ir = ct_rate*hst_transm_p_wi                # Infection rate (
   only used in the context of wi_rate)
wi_rate = h_ir*np.exp(-h_ir*tau)/(1 - np.exp(-h_ir*tau))  # Waning
     immunity rate

HST1 = {
  'bd': 0.,
  'ir': 0.,
  'rr': 2.19e-3,
  'wi': wi_rate,
  'pn': H1,
  'pn_full': 'Host',
}

para_lifespan = 8.         # Parasite lifespan (hours)
INDV_VEC = {
  'pc': 20,            # Parasite count
  'para_lsp': para_lifespan,
  'is_hap': False,       # Whether or not this individual's
     parasites are haploid
  'do_mutation': False,     # Whether or not this individual's
     parasites can mutate
  'pc_to_transmit': 15,     # The number of parasites transmitted
     per infection
}
INDV_HST = {
  'pc': int(1e8),
  'mut_chance': 2.94e-6,     # Chance of mutation per parasite per
     generation
  'para_lsp': para_lifespan,
  'is_hap': True,
  'do_mutation': True,
  'pc_to_transmit': 10,
}
INDVS = [INDV_VEC, INDV_HST]

D = allele(char='D')

####################################
### ----- BEGIN USER INPUT ----- ###
####################################

# Selection biases for mutated allele, relative to the wild allele
    (always 1.0)
control_sa = {H1: 1.0, VEC: 1.0}     # Used for the control case
hst_sa_inv = {H1: 1.05, VEC: 1.0}     # Used for the invasion
   example of the host selection advantage case
hst_sa_exp = {H1: 1.002, VEC: 1.0}     # Used for the host
   selection advantage case (quantitative)
vec_sa_exp = {H1: 1.0, VEC: 1.5}     # Used for the vector
   selection advantage case

asel_dims = {H1: 0.004, VEC: 0.5}     # Range to use in each
   direction around the base value for antagonistic

D.sel_advs = hst_sa_exp     # Choose the parameter set to use here
   (except antagonistic, which is further below)

# For transmission probabilities: the pop ID is the source -- i.e
   ., vec: 0.021 means 2.1% transmission chance from vector to
   host

hst_base_transm_p = 0.11   # Probability of contact resulting in a
   host-vector transmission
hst_adv_transm_p = 0.12     # Advantageous transmission probability
    for hosts
```

```
vec_base_transm_p = 0.021  # Probability of contact resulting in a
     vector-host transmission
vec_adv_transm_p = 0.024  # Advantageous transmission probability
    for vectors

D.base_transm_probs = {H1: hst_base_transm_p, VEC:
    vec_base_transm_p} # for wild-type allele

base_ta = D.base_transm_probs.copy()          # No advantages
hst_ta = {H1: hst_adv_transm_p, VEC: vec_base_transm_p} # Host
    transmission advantage
vec_ta = {H1: hst_base_transm_p, VEC: vec_adv_transm_p} # Vector
    transmission advantage
atrans = {H1: 0.10, VEC: vec_adv_transm_p}         # Antagonistic
    transmission

atrans_dims = {H1: 0.01, VEC: 0.003}  # Range in each direction to
     use around the base value

D.transm_probs = base_ta   # Choose the parameter set to use here
    (except antagonistic, which is below)

antag_types = {'sel': asel_dims, 'trans': atrans_dims}

NUM_RUNS = 1         # Number of simulations to run
FILE_DIR = 'control_new'   # Directory to save files under, if
    multiple simulations are being run (irrelevant for
    antagonistic)
INIT_MUT_PROP = 0.5       # Initial proportion of mutated alleles
SIM_LENGTH = 10000       # The length of the simulation (days)
SHOW_RES = False        # Whether or not to show the results
PROP_FOR_STATS = 0.2     # The proportion (from the end) of the
    results to use for statistics
NUM_HISTS = 0         # Set to 0 for no histograms

DO_ANTAG = True        # Whether or not to do antagonistic
    parameters
ANTAG_TYPE = 'sel'       # 'sel' | 'trans' (for antagonistic
    selection and transmission respectively)
CONTOUR_DENSITY = 5      # Performs (num)^2 rounds
DO_RETRO_CONTOUR = False    # Whether or not to generate the
    contour plot from (all) existing data. Will not generate
    anything new
DO_EXTEND = True       # Whether or not to extend preexisting data.
     Will either generate or read data, depending on what's
    available
COLOR_SCALE_TYPE = 'edge'  # 'lin' | 'mid' | 'edge', how to scale
    the colors. 'mid' flattens the middle and 'edge' flattens the
    edges

#################################
### ----- END USER INPUT ----- ###
#################################

TO_RUN = 'primary'  # 'cost' | 'primary' | 'overall' | 'diagram'
```

```
ANTAG_NMS = {'sel': 'selection', 'trans': 'transmission'}

USER_INPTS = {'Simulation time': SIM_LENGTH,
        'Total number of runs': NUM_RUNS,
        'Selection advantages': D.sel_advs,
        'Transmission advantages': D.transm_probs}

SHOW_CONTOUR = SHOW_RES
SHOW_MULTI = SHOW_RES and not DO_ANTAG
if DO_ANTAG or NUM_RUNS > 1: SHOW_RES = False

ALLELES = [D] # Do NOT have more than one allele here -- the
   simulation has been optimised for the single-locus case.
        # Adding more WILL break it!

PARAMS_1 = HST1
PARAMS_2 = VCT

def run(p0: np.ndarray=np.array([[20, 1, 0], [21, 0, 0]], dtype='
   float64'), p_fac: float=1200., nt: float=1., num_hist: int=
   NUM_HISTS,
    plot_res: bool=SHOW_RES, t_scale: float=SIM_LENGTH,
        init_mut_prop: float=INIT_MUT_PROP, fdir: str=''):
  '''
  Run the simulation. Keyword arguments should be changed in the "
      user input" section of the main file (above this method).

  ### Parameters
  - `p0`: The initial population ratios (S, I, R) as a NumPy array
       of 3-element NumPy arrays.
  - `p_fac`: The scale factor on population.
  - `nt`: Time steps per day. Alternatively, 24/`nt` is hours per
      time step.
  - `num_hist`: The number of histograms to generate (varying over
       time).
  - `plot_res`: Whether or not to display the results' graph, as a
       bool.
  - `t_scale`: The number of days to run the simulation for.
  - `init_mut_prop`: The initial fraction of mutated (uppercase)
      alleles. Must be between 0 and 1.
  - `fdir`: The directory to save the results under.

  ### Returns
  - `data_to_return`: dict of infected population name to a two-
      element list containing the mean (0) and standard deviation
      (1) of the data.
  '''
  exts_to_rm = ['dat', 'csv', 'txt']
  [[os.remove(file) for file in os.listdir() if file.endswith(f'.{
      ext}')] for ext in exts_to_rm]
  mkDir('hists', 'old images')
  if fdir: os.makedirs(fdir, exist_ok=True)
  mkFilePath('inf_events_raw.dat', 'last_event.dat',)
  alleles = ALLELES
  p0 = p_fac*p0
  t_max = t_scale
  for i in range(len(INDVS)):
```

```
  i_params = INDVS[i]
  para_gens = int((24/nt)/i_params['para_lsp'])
  i_params['para_gens'] = para_gens
  i_params['alleles'] = alleles
nt = float(int(nt*t_scale))

hosts_1 = population(p0[0], **INDV_HST)
vectors = population(p0[1], **INDV_VEC)

m1 = SIR(hosts_1, **PARAMS_1)
m2 = SIR(vectors, **PARAMS_2)

m1.itr = {vectors: ct_rate}
m1.other_pop = vectors
m2.itr = {hosts_1: ct_rate}
m2.other_pop = hosts_1

mdls = [m1, m2]

t0 = time()
ts, ps, pops, hists_v, hists_h, hist_tms, mdls_fin = simShell(
  t_max, mdls, nt=nt, alleles=alleles, init_mut_prop=
     init_mut_prop, num_hist=num_hist)
ex_tm = time() - t0
print(f'\nExecution time: {roundNum(ex_tm, prec=3)}') # consider
     colored console output for readability

[p.printDat() for p in pops]
[mdl.save(f'{fdir}models') for mdl in mdls_fin]
# dimensions of ps: layer 1 is times, layer 2 is models at that
    time, layer 3 is pop #s for that model

f = open('inf_events.dat', 'x')
f_raw = open('inf_events_raw.dat', 'r')
inf_events = {}
for line in f_raw.readlines():
  line = line[:-1]
  if line in inf_events: inf_events[line] += 1
  else: inf_events[line] = 1
[f.write(f'{i_e}: {inf_events[i_e]}\n') for i_e in inf_events]
f.close()
f_raw.close()

def getDims(lst: list, tab: str=''): # Displays the dimensions
    of the given list. Useful when handling complex/unorthodox
    structures
  if type(lst) == list:
    print(f'{tab}list of dim {len(lst)} containing:')
    getDims(lst[0], f'{tab}\t')
    if type(lst[0]) == list:
      if len(lst[1]) != len(lst[0]):
        print(f'{tab}additionally:')
        getDims(lst[1], f'{tab}\t')
    return
  else:
    print(f'{tab}{type(lst).__name__}')
    return
```

```
output_fn = f'{fdir}net_output.opt' # '.opt' is a plain text
    file. Only marked that way to keep the file clearing from
    catching it
f = mkFile(output_fn)
data_to_return: dict[str, list[float]] = {}
for i in range(len(mdls)):
  ns = [''.join(n.split('.')) for n in pops[i].getAllPopNms()]
  f.write(f'\t{mdls[i].pn_full}:\n')
  gens = []
  for n in ns:
    if '(' in n and ')' in n: gens += [n[n.index('(')+1:n.index(
        ')')]]
    else: gens += [n]
  ps_i = np.array([k[i] for k in ps])
  csv_data = {'times': ts}
  plt_datas = []
  stplt_labels = []
  stplt_colors = []
  for j in range(len(ns)):
    if (ns[j][0] != 'R') and (ns[j][0] != 'S'):
      plt_data = ps_i[:,j]
      plt_lst = list(plt_data)
      plt_datas += [plt_lst]
      stplt_labels += [ns[j]]
      plt_color = str2Color(gens[j])
      stplt_colors += [plt_color]
      csv_data[ns[j]] = plt_lst
      mean_stdev = saveStats(plt_lst, frac_to_take=
          PROP_FOR_STATS)
      data_to_return[ns[j]] = list(mean_stdev)
      writeOptLine(f, ns[j], *mean_stdev)
      plt.plot(ts, plt_data, label=ns[j], color=plt_color, alpha
          =pop2Alpha(ns[j]))
  f.write('\n')

  # Plot frequencies
  net_i = [1. for t in ts]
  plt.plot(ts, net_i, label='I (total)')
  plt.plot(ts, 0*ts, alpha=0.)
  plt.title(f'{mdls[i].pn_full} population (infected)')
  plt.legend()
  plt.xlabel('Simulation time')
  plt.ylabel('Population')
  file_nm = f'{fdir}{fn(mdls[i].pn)}'
  plt.close()

  pd.DataFrame(csv_data).to_csv(f'{file_nm}.csv')

  # Plot proportions
  plt.stackplot(ts, *plt_datas, labels=stplt_labels, colors=
      stplt_colors)
  plt.title(f'{mdls[i].pn_full} population: proportion plot')
  plt.legend()
  plt.ylabel('Relative proportion')
  plt.xlabel('Simulation time')
  plt.savefig(f'{file_nm}.png')
```

```
      if plot_res: plt.show()
      plt.close()

    # Save histograms
    hists_with_pn = {VEC: hists_v, H1: hists_h}
    hist_max = p0[0][1]
    for hpn in hists_with_pn:
      hists_2_p = hists_with_pn[hpn]
      for i in range(num_hist):
        tm = hist_tms[i]
        hist_2_p = [0.0] + hists_2_p[i] + [1.0]
        plt.hist(hist_2_p, bins=100)
        plt.title(f'Population {hpn}, time {roundNum(tm)}')
        plt.xlabel('Mutated (capital) allele frequencies')
        plt.ylabel('Individual count')
        if hpn == H1: plt.ylim(top=hist_max)
        plt.savefig(f'hists/{hpn}_{i}.png')
        plt.close()

    writeInputs(f)
    f.close()
    return data_to_return

  def doTimeBreakdown(): # Runs the simulation with a profiler &
      processes the output accordingly
    time_output_fn = 'time_breakdown.dat'
    cProfile.run('run()', time_output_fn)
    time_output_txt = 'time_breakdown.txt'
    time_output_file = open(time_output_txt, 'x')
    p_stats = pstats.Stats(time_output_fn, stream=time_output_file)
    p_stats.strip_dirs()
    p_stats.sort_stats('cumtime')
    p_stats.print_stats()

  def doOverallData(all_data: dict[str, list[list[float]]], full_dir
      : str=''):
    '''
    Takes the results of multiple simulations, writes them to a file
       , and returns them.

    ### Parameters
    - `all_data`: dict of strain (I (D), I (Dd), I (d), ...):list of
        mean-standard deviation lists (at indices 0 and 1
       respectively).
       Each element should be the result of a different simulation.
    - `full_dir`: The directory to save the output file under (full
       filepath).

    ### Returns
    - `f`: The file object the overall data was saved to.
    - `stat_lsts`: The data saved in `f` (dict strain:[mean,
       standard deviation]).
    '''
    all_data: dict[str, list[list[float]]] = {k: transpose(all_data[
       k]) for k in all_data}
    lengths = {'Vector': 6, 'Host': 5} # lengths of row names (ref.
       net_output.opt et al for examples)
    f = mkFile(f'{full_dir}net_output_overall.opt')
    f.write(f'\t----- OVERALL OUTPUT: {full_dir.split("/")[-2]}
       -----\n')
    stat_lsts: dict[str, list[float]] = {k: [np.mean(v[0]), propUnc(
       np.std(v[0]), propUnc(*v[1]), do_div=False)] for k, v in
       all_data.items()}
    for pn, lng in lengths.items():
      keys = [k for k in all_data if len(k) == lng]
      keys.sort(reverse=True)
      f.write(f'\t{pn}:\n')
      [writeOptLine(f, k, *stat_lsts[k]) for k in keys]
      f.write('\n')
    writeInputs(f)
    return f, stat_lsts

  def writeInputs(f: TextIO): # Writes the user's input data to a
     file
    [f.write(f'{k}: {v}\n') for k, v in USER_INPTS.items()]

  def doMultipleRuns(n: int=3, fdir: str='', force_multi: bool=False
     , do_qc: bool=False) -> dict[str, list[float]]:
    '''
    Performs multiple simulations, saves the data, and returns the
       results (dict strain:[mean, standard deviation]).

    ### Parameters
    - `n`: The number of simulations to perform.
    - `fdir`: The directory (under full_outputs) to save the output
       files to.
    - `force_multi`: Whether or not to force the multiple-run
       structure to be used. If `False` and `n` is 1, it will
       simply perform a single
       run as normal.
    - `do_qc`: Whether or not to allow "useless" runs to be skipped
       (if a preexisting run's mutated frequency is below 0.1 or
       above 0.9).
    '''
    if n == 1 and not force_multi: run(); return {}
    full_dir = f'full_outputs/{fdir}/'
    all_data: dict[str, list[list[float]]] = {}
    num_runs_done = numSubDirs(full_dir)
    num_tot = n + num_runs_done
    ignore_amt = 0.1
    for i in range(1,num_tot+1):
      if num_tot != 1: colorist.red(10*'-' + f' RUN {i} ' + 10*'-')
      ret_data = {}
      if i <= num_runs_done:
        colorist.red(f'\tRun {i} already complete. Skipping.')
        ret_data = doRetroStats(fdir)
      else: ret_data = run(fdir=f'{full_dir}{i}/')
      for k in ret_data:
        if k in all_data: all_data[k] += [ret_data[k]]
        else: all_data[k] = [ret_data[k]]
      if do_qc:
        final_freq = getNetAllFreq(ret_data)
        if not isInRange(final_freq, [ignore_amt, 1-ignore_amt],
           do_trunc_check=False):
```

```
        colorist.red(f'\tResult insufficiently ambiguous to
           warrant further simulations. Breaking.')
        break
  f, stat_lsts = doOverallData(all_data, full_dir)
  f.close()
  compilePlots(full_dir[:-1], do_show=SHOW_MULTI)
  return stat_lsts

def doRetroStats(fdir: str):
  '''
  Retroactively calculates overall stats from the given directory
     (under full_outputs). Useful if data has been generated, but
      the
  `stat_lsts` objects from those generations are not available for
      whatever reason (this method rebuilds them).
  '''
  full_dir = f'full_outputs/{fdir}/'
  csv_names = FULL_NMS
  bad_cols = ['Unnamed', 'times']
  all_data: dict[str, list[list[float]]] = {}
  for r_dir in os.listdir(full_dir):
    if '.' in r_dir: continue
    full_r_dir = f'{full_dir}{r_dir}/'
    if not os.listdir(full_r_dir): continue
    for csv_nm in csv_names.values():
      csv_data: dict[str, list[float]] = readCSVData(f'{full_r_dir
         }{csv_nm}.csv')
      real_bad_cols: list[str] = []
      for k in csv_data:
        for bad_col in bad_cols:
          if bad_col in k: real_bad_cols += [k]
      for rbc in real_bad_cols: del csv_data[rbc]
      for k, v in csv_data.items():
        stats_to_add = list(saveStats(v, frac_to_take=
           PROP_FOR_STATS))
        if k in all_data: all_data[k] += [stats_to_add]
        else: all_data[k] = [stats_to_add]
  f, stat_lsts = doOverallData(all_data, full_dir)
  f.close()
  return stat_lsts

def getRetroContourData(fn_save: str):
  '''
  Reads data from the given file name and returns it in the format
      required of the contour plotting methods.

  ### Parameters
  - `fn_save`: The full path to the desired file, not including
     the .csv extension.

  ### Returns
  - `xs_vec`: The column names of the csv file. Corresponds to
     vector parameters, which are on the x-axis.
  - `ys_h1`: The row names of the csv file. Corresponds to host
     parameters, which are on the y-axis.
  - `zs_freq`: A two-dimensional list of the values in the csv
     file, structured similarly.
  '''
  contour_df = pd.read_csv(f'{fn_save}.csv')
  xs_vec = [float(x) for x in contour_df.columns if x != 'y']
  ys_h1 = [float(y) for y in contour_df.loc[:,'y']]
  zs_freq: list[list[float]] = [[ys[i_x] for i_x in range(len(ys))
      if i_x] for ys in contour_df.to_numpy()]
  return xs_vec, ys_h1, zs_freq

def getGenContourData(pop_data: dict[tuple[float,float], float],
     fn_save: str):
  '''
  Saves the contour plot data to a particular file and returns it
     in the format required of the plotting methods.

  ### Parameters
  - `pop_data`: A dict of coordinates to frequencies (x,y):z used
     to generate a contour plot.
  - `fn_save`: The full path to the desired file, not including
     the .csv extension.

  ### Returns
  - `xs_vec`: The column names of the csv file. Corresponds to
     vector parameters, which are on the x-axis.
  - `ys_h1`: The row names of the csv file. Corresponds to host
     parameters, which are on the y-axis.
  - `zs_freq`: A two-dimensional NumPy array of the values in the
     csv file, structured similarly.
  '''
  contour_df = makeCoordDF(pop_data)
  contour_df.to_csv(f'{fn_save}.csv')
  xs_vec = [float(x) for x in contour_df.columns]
  ys_h1 = [float(y) for y in contour_df.index]
  zs_freq = contour_df.to_numpy()
  return xs_vec, ys_h1, zs_freq

def getNetAllFreq(stat_lsts: dict[str, list[float]], pop_len:
     Literal[5,6]=5):
  '''
  Takes a `stat_lsts` object (dict strain:[mean, standard
     deviation]), filters out the relevant entries (`pop_len` 5
     for hosts, 6 for
  vectors), and returns the net frequency of the mutated allele
     across all strains. Does not consider standard deviation.
  '''
  keys = [k for k in stat_lsts if len(k) == pop_len]
  keys.sort(reverse=True)
  net_all_freq = 0.
  for k in keys: net_all_freq += k.count(D.char)*stat_lsts[k][0]
  net_all_freq /= (len(keys) - 1)
  return net_all_freq

def doContourPlots(retro: bool=False):
  '''
  The primary method for producing contour plots. `retro` should
     in general be `DO_RETRO_CONTOUR` (it is only a parameter due
      to scoping
  issues).
```

```
'''
D.sel_advs = control_sa
D.transm_probs = base_ta
if ANTAG_TYPE not in antag_types: print('bad antagonistic type')
    ; exit()
antag_dims = antag_types[ANTAG_TYPE]
fdir = f'a{ANTAG_TYPE}_contour'
fulldir = f'full_outputs/{fdir}'
param_to_change = D.sel_advs if ANTAG_TYPE == 'sel' else D.
    transm_probs
antag_name = ANTAG_NMS[ANTAG_TYPE]
h1_rng, vec_rng = [getRange(param_to_change[pop_nm], antag_dims[
    pop_nm], num_els=CONTOUR_DENSITY) for pop_nm in [H1, VEC]]
lengths = {'Vector': 6, 'Host': 5}
contour_data = {pop_nm: {(vec_idx, h1_idx): -1. for vec_idx,
    h1_idx in zip(vec_rng, h1_rng)} for pop_nm in lengths}
round_idx = 0
h1_bounds = [h1_rng[0], h1_rng[-1]]
vec_bounds = [vec_rng[0], vec_rng[-1]]
h1_pre_bounds = h1_bounds
vec_pre_bounds = vec_bounds
if DO_EXTEND:
  if not os.path.exists(fulldir): os.makedirs(fulldir)
  existing_dirs = [hv_dir for hv_dir in os.listdir(fulldir) if
      not ('png' in hv_dir or 'csv' in hv_dir) and '_' in hv_dir
      ]
  h1s = []
  vecs = []
  for e_ds in existing_dirs:
    h1v, vecv = [float(hvv[1:]) for hvv in e_ds.split('_')]
    h1s += [h1v]
    vecs += [vecv]
  if existing_dirs:
    h1_pre_bounds = [min(h1s), max(h1s)]
    vec_pre_bounds = [min(vecs), max(vecs)]
  else:
    h1_pre_bounds = [0, 0]
    vec_pre_bounds = [0, 0]
if not retro: # generate data if it's not already there
  for i in range(CONTOUR_DENSITY):
    for j in range(CONTOUR_DENSITY):
      round_idx += 1
      param_to_change[H1] = h1_rng[i]
      param_to_change[VEC] = vec_rng[j]
      fdir_ij = f'{fdir}/h{trunc(h1_rng[i])}_v{trunc(vec_rng[j])
          }'
      fulldir_ij = f'full_outputs/{fdir_ij}'
      num_inc_runs = NUM_RUNS - numSubDirs(fulldir_ij) if os.
          path.exists(fulldir_ij) else 0
      if num_inc_runs < 0: num_inc_runs = 0
      colorist.red(f'Beginning round {round_idx}/{
          CONTOUR_DENSITY**2} for antagonistic {antag_name}, {
          param_to_change}')
      if DO_EXTEND and ((isInRange(h1_rng[i], h1_pre_bounds) and
           isInRange(vec_rng[j], vec_pre_bounds)) and not
          num_inc_runs):
        colorist.red('\tSkipping round on account of extension')
```

```
          stat_lsts = {}
          if fdir_ij in os.listdir(fulldir):
            f = open(f'{fulldir_ij}/net_output_overall.opt', 'r')
            for opt_line in f.readlines():
              if 'I (' in opt_line:
                key, val = opt_line.split(':\t')
                stat_lsts[key] = [float(msv) for msv in val.split(
                   '\t+- ')]
        else: stat_lsts = doMultipleRuns(n=num_inc_runs, fdir=
           fdir_ij, force_multi=True, do_qc=True)
        for pop_nm, pop_len in lengths.items():
          if not stat_lsts: continue
          contour_data[pop_nm][(vec_rng[j], h1_rng[i])] =
             getNetAllFreq(stat_lsts, pop_len)
  for pop_nm, pop_data in contour_data.items():
    to_pop = [coord for coord, val in pop_data.items() if val < 0]
    for coord in to_pop: pop_data.pop(coord)
  colorist.cyan('Data generation complete. Reading data from
     full_outputs...')
  for opt_dir in os.listdir(fulldir):
    if ('png' in opt_dir or 'csv' in opt_dir) or '_' not in
       opt_dir: continue
    h1_c, vec_c = [float(hvc[1:]) for hvc in opt_dir.split('_')]
    if (h1_c, vec_c) in contour_data['Vector']: continue
    for pop_nm, pop_len in lengths.items():
      stat_lsts = doRetroStats(f'{fdir}/{opt_dir}')
      contour_data[pop_nm][(vec_c, h1_c)] = getNetAllFreq(
         stat_lsts, pop_len)
  colorist.cyan('Done.')
  for pop_nm, pop_data in contour_data.items():
    to_pop = [coord for coord in pop_data if not isInRange(coord
       [0], vec_bounds) or not isInRange(coord[1], h1_bounds)]
    [pop_data.pop(coord) for coord in to_pop]
  xtick_lbls = roundTrailing(*vec_rng)
  len_lbls = sum([len(str(xtl)) for xtl in xtick_lbls])
  if len_lbls > 50: xtick_lbls = [v for i, v in enumerate(
     xtick_lbls) if not i%2]
  xs_vec = []
  ys_h1 = []
  zs_all: dict[str, dict[tuple[float, float], float]] = {pop_nm:
     {} for pop_nm in FULL_NMS}

  def addLabels(colorbar_label: str='Mutated allele frequency'): #
      axes, color bar for contour plots
    param_type = ''
    if antag_name == 'selection': param_type = 'relative fitness'
    elif antag_name == 'transmission': param_type = 'transmission
       probability'
    plt.colorbar(label=colorbar_label)
    plt.xlabel(f'Vector: {param_type}')
    plt.xticks(xtick_lbls, labels=xtick_lbls)
    plt.ylabel(f'Host: {param_type}')

  def plotScatter(pop_data: dict[tuple[float, float], float],
     save_fig: str, l_type: Literal['dec', 'pct']='dec', **kwargs
     ):
    scatter_dir = f'{fulldir}/scatters'
    if not os.path.exists(scatter_dir): os.makedirs(scatter_dir)
    label_method = trunc
    def round_pct(f: float): return f'{roundNum(100*f)}%'
    if l_type == 'pct': label_method = round_pct
    plt.scatter(*transpose(pop_data.keys()), c=list(pop_data.
       values()), cmap=getColorMap(scale_type=COLOR_SCALE_TYPE))
    [plt.annotate(label_method(pop_data[coords]), coords, rotation
       =45) for coords in pop_data]
    addLabels(**kwargs)
    plt.savefig(f'{scatter_dir}/{save_fig}.png')
    plt.close()

  def getContourLine(pop_data: dict[tuple[float, float], float],
     mid_val: float=0.5, rng: float=0.05):
    ok_rng = getRange(mid_val, rng, num_els=2)
    coords_lst = [coords for coords, val in pop_data.items() if
       isInRange(val, ok_rng)]
    return coords_lst

  contour_kwargs = {'levels': 20, 'cmap': getColorMap(scale_type=
     COLOR_SCALE_TYPE)}
  lr_done = False
  for pop_nm in contour_data:
    fn_save = f'{fulldir}/{FULL_NMS[pop_nm]}'
    xs_vec, ys_h1, zs_freq = getGenContourData(contour_data[pop_nm
       ], fn_save)
    if DO_EXTEND:
      nan_mask = np.isnan(zs_freq)
      xs_vec_mg, ys_h1_mg = np.meshgrid(xs_vec, ys_h1)
      xs_vec, ys_h1 = np.mgrid[min(xs_vec):max(xs_vec):100j, min(
         ys_h1):max(ys_h1):100j]
      zs_freq = np.array(zs_freq)
      zs_freq = griddata((xs_vec_mg[~nan_mask].flatten(), ys_h1_mg
         [~nan_mask].flatten()), zs_freq[~nan_mask].flatten(),
            (xs_vec, ys_h1), method='cubic')
      for i in range(len(zs_freq)):
        for j in range(len(zs_freq[i])):
          if zs_freq[i,j] > 1: zs_freq[i,j] = 1
          elif zs_freq[i,j] < 0: zs_freq[i,j] = 0
    plt.contourf(xs_vec, ys_h1, zs_freq, **contour_kwargs)
    plt.title(f'{pop_nm}: {antag_name} contour plot ({SIM_LENGTH}
       days)')
    addLabels()
    plt.savefig(f'{fn_save}.png')
    if SHOW_CONTOUR: plt.show()
    plt.close()
    pop_data = contour_data[pop_nm]
    plotScatter(pop_data, save_fig=FULL_NMS[pop_nm])
    zs_all[pop_nm] = pop_data
    if ANTAG_TYPE == 'trans' and not lr_done: # linear regression
      lr_base, lr_rng = 0.5, 0.05
      lr_coords_raw = getContourLine(pop_data, mid_val=lr_base,
         rng=lr_rng)
      lr_coords = transpose([(xc/vec_base_transm_p, yc/
         hst_base_transm_p) for (xc, yc) in lr_coords_raw])
      slope, intercept, r, p, stderr = linregress(*lr_coords)
      plt.scatter(*lr_coords)
```

```
        lr_xs = lr_coords[0]
        lr_ys = [slope*lrx + intercept for lrx in lr_xs]
        plt.plot(lr_xs, lr_ys, label=rf'slope -{roundAndSN(-slope)},
            R$^2$ {roundAndSN(r**2)}, err {roundAndSN(stderr)}')
        plt.title(f'Linear regression: frequency {lr_base} (range {
           lr_rng})')
        plt.ylabel('Host: relative transmission advantage')
        plt.xlabel('Vector: relative transmission advantage')
        plt.legend()
        plt.savefig(f'{fn_save}_linreg.png')
        plt.show()
        plt.close()
        lr_done = True
    diff_data = {coord: zs_all['Host'][coord] - zs_all['Vector'][
       coord] for coord in zs_all['Host']}
    xv_diff, yh_diff, zs_diff = getGenContourData(diff_data, f'{
       fulldir}/diff')
    plt.contourf(xv_diff, yh_diff, zs_diff, **contour_kwargs)
    plt.title('Host-vector mutated allele frequency differences')
    addLabels(colorbar_label='Frequency difference')
    plt.savefig(f'{fulldir}/diff.png')
    plt.close()
    plotScatter(diff_data, save_fig='diff', l_type='pct')

def fixAntagDirs():
    '''
    Ensures the directories created by antagonistic parameters &
       associated contour plots are of the correct naming scheme.
       Originally
    used to switch from an older naming scheme to the current one;
       should not see much use overall.
    '''
    full_dir = f'full_outputs/a{ANTAG_TYPE}_contour'
    new_dir_names = {}
    test_str = f'{ANTAG_TYPE[0].upper()}{ANTAG_TYPE[1:]}'
    for hv_dir in os.listdir(full_dir):
      if '_' in hv_dir:
        opt_file = open(f'{full_dir}/{hv_dir}/net_output_overall.opt
            ', 'r')
        for opt_line in opt_file.readlines():
          if test_str in opt_line:
            hv_nums = opt_line.split('{')[1][:-1]
            strs = hv_nums.split(', ')
            ndn = ''
            for str_spt in strs:
              pop_nm, pop_val = str_spt.split(': ')
              ndn += f'{pop_nm[1]}{trunc(pop_val)}_'
            new_dir_names[hv_dir] = ndn[:-2]
        opt_file.close()
    [os.rename(f'{full_dir}/{k}', f'{full_dir}/{v}') for k, v in
       new_dir_names.items()]

def getAllFreq(csv_data: dict[str, list[float]]) -> list[float]:
    div_fac = len(csv_data) - 3
    return list(np.sum(transpose([hdr.count(D.char)*np.array(vals)/
       div_fac for hdr, vals in csv_data.items() if 'I (' in hdr]),
        axis=1))
```

```
def compilePlots(output_dir: str, do_show: bool=True):
  data_dirs = getSubDirs(output_dir)
  times: list[float] = []
  freqs: dict[str, list[list[float]]] = {k: [] for k in FULL_NMS_R
     }
  for d_dir in data_dirs:
    full_dir = f'{output_dir}/{d_dir}'
    for pop_type in freqs:
      pop_data = readCSVData(f'{full_dir}/{pop_type}.csv')
      if not times: times = pop_data['times']
      freqs[pop_type] += [getAllFreq(pop_data)]
  for pop_type, pop_freqs in freqs.items():
    [plt.plot(times, run_freqs, label=i+1) for i, run_freqs in
        enumerate(pop_freqs)]
    plt.plot(times, 0*np.array(times), alpha=0.)
    plt.plot(times, [1 for t in times], alpha=0)
    plt.title(f'{FULL_NMS_R[pop_type]}: mutated allele frequency
        ({len(pop_freqs)} simulations)')
    plt.xlabel('Simulation time (days)')
    plt.ylabel('Mutated allele frequency')
    plt.legend()
    plt.savefig(f'{output_dir}/{pop_type}.png')
    if do_show: plt.show()
    plt.close()

def render_latex(formula: str, fontsize: int=12, dpi: int=300,
    path: str='tex.png'):
  if '.' not in path: path += '.png'
  fig = plt.figure()
  text = fig.text(0, 0, rf'{formula}', fontsize=fontsize)

  fig.savefig(BytesIO(), dpi=dpi)

  bbox = text.get_window_extent()
  width, height = bbox.size/float(dpi) + 0.05
  fig.set_size_inches((width, height))

  dy = (bbox.ymin/float(dpi))/height
  text.set_position((0, -dy))

  fig.savefig(f'tex_imgs/{path}', dpi=dpi, transparent=True)
  plt.close(fig)

def drawDiagram(): # for drawing the compartment diagram
  WHITE = '#FFFFFFFF'
  BLACK = '#000000FF'
  TRANS = '#FFFFFF00'
  raw_dims = (2820, 1272)
  img_raw = Image.new('RGBA', size=raw_dims, color=WHITE)
  img = Image.new('RGBA', size=img_raw.size, color=TRANS)
  draw = ImageDraw.Draw(img)
  img_width, img_height = img.size
  cpmts = {'sv': (3*img_width/8, img_height/4),
       'iv': (5*img_width/8, img_height/4),
       'sh': (img_width/4, 3*img_height/4),
       'ih': (img_width/2, 3*img_height/4),
       'rh': (3*img_width/4, 3*img_height/4)}
  rad = 130
  font_sz = 30
  lwdth = 10

  mkDir('tex_imgs')

  def paste_img(name: str, c_coords: tuple[int,int], do_center:
       bool=True):
    if '.png' not in name: name += '.png'
    tex_im = Image.open(f'tex_imgs/{name}').convert('RGBA')
    txi_sz = tex_im.size
    tex_cs = tuple([int(cc-tc/2) for cc, tc in zip(*[c_coords,
        txi_sz])])
    if not do_center:
      tcl = list(tex_cs)
      tcl[0] = int(c_coords[0])
      tex_cs = tuple(tcl)
    img.paste(tex_im, box=tex_cs)
    tex_im.close()
    os.remove(f'tex_imgs/{name}')

  for name, coords in cpmts.items():
    draw.circle(coords, rad+lwdth, fill=BLACK)
    draw.circle(coords, rad, fill=WHITE)
    tex_nm = rf'${name[0].upper()}_{name[1]}$'
    render_latex(tex_nm, fontsize=font_sz, path=name)
    paste_img(name, coords)

  render_latex(r'VECTOR:', fontsize=font_sz, path='vec.png')
  paste_img('vec.png', (20, img_height/4), do_center=False)
  render_latex(r'HOST:', fontsize=font_sz, path='hst.png')
  paste_img('hst.png', (20, 3*img_height/4), do_center=False)

  latex_strs = {
    r'$\mu S_v$': 1.5,
    r'$\mu N_v$': 1.5,
    r'$\frac{\beta_hS_vI_h}{N_v}$': 1.5,
    r'$\mu I_v$': 1.5,
    r'$\frac{\beta_vS_hI_v}{N_h}$': 1.5,
    r'$\gamma I_h$': 1.5,
    r'$\omega R_h$': 1.5
  }

  [render_latex(tex_s, fontsize=int(font_sz/latex_strs[tex_s]),
      path=str(i)) for i, tex_s in enumerate(latex_strs.keys())]

  out = Image.alpha_composite(img_raw, img)
  out.show()
  out.save('model_out.png')

if __name__ == '__main__':
  match TO_RUN:
    ### For computational cost tracking
    case 'cost':
      doTimeBreakdown()
    ### Primary
    case 'primary':
      if DO_ANTAG: doContourPlots(retro=DO_RETRO_CONTOUR); exit()
      fdir = 'temp_dir'
      if FILE_DIR: fdir = FILE_DIR
      num_comp_runs = numSubDirs(f'full_outputs/{fdir}')
      doMultipleRuns(n=NUM_RUNS-num_comp_runs, fdir=fdir,
          force_multi=(NUM_RUNS != 1))

    ### To get overall data from the given pre-generated directory
    case 'overall':
      doRetroStats(FILE_DIR)
      compilePlots(f'full_outputs/{FILE_DIR}')

    ### Draw compartment diagram (partial; some assembly required)
    case 'diagram':
      drawDiagram()
```

allele.py

```
'''
The file containing the class for alleles.
'''

class allele:
  '''
  The class for alleles. Describes the phenotype observed from its
      expression and contains the character representing it in
     text.
  '''
  char = ''
  fav_pop = ''
  unf_pop = ''
  param = ''
  fac = 0.0
  sel_advs: dict[str, float] = {}        # specifically for mutated
      allele (capital)
  transm_probs: dict[str, float] = {}      # also specifically for
     mutated allele
  base_transm_probs: dict[str, float] = {}    # specifically for
     wild allele (lowercase)

  def __init__(self, **kwargs):
    '''
    Initialises the allele.

    ### Parameters
    - `char`: The character representing the allele in text
    - `fav_pop`: The name of the population it's well-adapted to
        *(obsolete)*
    - `unf_pop`: The name of the population it's poorly-adapted to
        *(obsolete)*
    - `param`: The name of the trait it affects *(obsolete)*
    - `fac`: The numerical factor by which the trait is affected.
        'Good' traits should be positive, 'bad' traits should be
        negative
      (its value should be between -1 and 1, not inclusive) *(
          obsolete)*
    '''
    self.__dict__.update(kwargs)
    self.char = self.char.upper()
    self.sel_advs: dict[str, float] = {}
    self.transm_probs: dict[str, float] = {}
    self.base_transm_probs: dict[str, float] = {}

  @property
  def locus(self):
    '''
    The 'ID' of the locus this allele is at. (This is the
        lowercase of the allele's character.)
    '''
    return self.char.lower()

  def __str__(self):
    return self.char

  def __repr__(self):
    return self.__str__()
```

color.py

```
'''
Library of miscellaneous functions related to color handling.
'''

from numpy import cos, pi, sqrt, log10
from matplotlib.colors import LinearSegmentedColormap
from func_lib import *
from colorist import ColorHex
from typing import Literal
import matplotlib.pyplot as plt

def procHex(*args):
  '''
  Converts the given values (0-255) into a hex color code.
  '''
  return '#'+''.join([f'0{hex(s)[2:]}'[-2:] for s in args])

def str2Color(s: str):
  '''
  Converts the given string into an arbitrary (but consistent)
      color code.
  '''
  tot_num = 3*2551*sum([ord(c)**5 for c in s])
  R = int(tot_num%255)
  G = int((tot_num/1739)%255)
  B = int((tot_num*3717)%255)
  if R + G + B > 2.5*255: [R, G, B] = [int(c/2) for c in [R, G, B
      ]]
  return procHex(R, G, B)

def pop2Alpha(p: str):
    '''
    Returns an alpha value corresponding with the kind of population
        (S, I, R) submitted. (R is lighter.)

    *Largely obsolete.*
    '''
    p = p[0]
    if p == 'S' or p == 'I': return 1.
    else: return 0.7

# Basic functions for color mapping. Determines how each channel's
    value changes over a 0-1 interval
def r(i): return sqrt(int(-127*cos(pi*i) + 127)/255)
def g(i): return int(-200*cos(0.5*pi*i) + 200)/255
def b(i): return int(70*cos(0.75*pi*(0.5*i + 0.25)) + 70)/255

def getColor(rv: float, gv: float, bv: float):
    '''
    Turns the given RGB values (floats, range 0-1) into a
        hexadecimal color code.
    '''
    return procHex(*[int(rgb*255) for rgb in [r(rv), g(gv), b(bv)]])

def scale_mid(lsp: list[float]):
    '''
    Scales the given list (should be linear, bounded between 0 and
        1) such that values near the middle (0.5) are "flattened."
        Useful
    for color maps intending to exaggerate differences near the
        edges (0, 1).
    '''
    return [((2*i-1)**3 + 1)/2 for i in lsp]

def scale_edg(lsp: list[float]) -> list[float]:
    '''
    Scales the given list (should be linear, bounded between 0 and
        1) such that values near the edges (0, 1) are "flattened."
        Useful
    for color maps intending to exaggerate differences near the
        middle (0.5).
    '''
    return [(np.sign(2*i-1)*log10(1 + 9*abs(2*i-1)) + 1)/2 for i in
        lsp]

def scale_lin(lsp: list[float]):
    '''
    "Scales" the given list (should be linear, bounded between 0 and
        1) such that its values are linearly separated, i.e. leaves
        them
    unchanged. Mainly a function wrapper indicating that no action
        should be taken.
    '''
    return lsp

def getColorMap(nm: str='cmap', res: int=256, scale_type: Literal[
    'mid','edge','lin'] = 'lin'):
    '''
```

```
  Creates a ‘mpl.LinearSegmentedColormap‘ object based on the
     gradient defined by ‘r‘, ‘g‘, and ‘b‘ (dark blue-orange-gold
     ). ‘res‘ is
  its resolution, ‘nm‘ is its name, and ‘scale_type‘ defines the
     method used to scale the color map’s values.
  - ‘mid‘: Flattens the middle (0.5), exaggerates the edges (0, 1)
     .
  - ‘edge‘: Flattens the edges (0, 1), exaggerates the middle
     (0.5).
  - ‘lin‘: Leaves the scaling unchanged (default).
  '''
  scale_fn = scale_mid
  if scale_type is 'edge': scale_fn = scale_edg
  elif scale_type is 'lin': scale_fn = scale_lin
  elif scale_type is not 'mid': print('invalid scale type
     submitted; defaulting to mid')
  def makeGrad(fn):
    idxs = linspace(res=res)
    idxs_unsc = scale_fn(idxs)
    r_t = ()
    for i in range(res):
      idx = idxs[i]
      idx_u = idxs_unsc[i]
      y1 = fn(idxs_unsc[i-1])
      if not i: y1 = fn(idx_u)
      r_t += ((idx, fn(idx_u), y1),)
    return r_t

  return LinearSegmentedColormap(nm, {'red': makeGrad(r), 'green':
      makeGrad(g), 'blue': makeGrad(b)})

def plotGradient():
  '''
  Displays the gradient defined by ‘r‘, ‘g‘, and ‘b‘ (dark blue-
     orange-gold).
  '''
  idxs = linspace()
  test_spc = linspace(res=5)
  hexes = [getColor(i, i, i) for i in test_spc]
  print(', '.join(f'{ColorHex(hex_col)}{hex_col[1:]}{ColorHex(
     hex_col).OFF}' for hex_col in hexes))
  plt.imshow((idxs,idxs), aspect='auto', cmap=getColorMap())
  plt.show()
```

data_lib.py

```
'''
Library of miscellaneous functions related to data analysis and
   recording.
'''

from func_lib import *
from typing import TextIO
import pandas as pd

def propUnc(*args, do_div: bool=True) -> float:
  '''
  Propagates uncertainty for mean calculations. Takes an arbitrary
      number of standard deviation arguments. `do_div` denotes
      whether or not
  this is for a mean's direct uncertainty (`True`) or a sum of a
      standard deviation with an uncertainty (`False`).
  '''
  div_fac = 1
  if do_div: div_fac = len(args)
  return np.sqrt(sum([(std/div_fac)**2 for std in args]))

def writeOptLine(f: TextIO, name: str, mean_num: float, std_num:
   float):
   '''
  Writes the given data (`mean_num`, `std_num`) to file `f` with
      row name `name`.
  '''
  f.write(f'{name}:\t{roundAndSN(mean_num)}\t+- {roundAndSN(
      std_num)}\n')

def readCSVData(csv_path: str) -> dict[str, list[float]]:
  '''
  Returns data from a CSV file as a dict of column header:value
      list.
  '''
  return {col_header: list(col_data) for col_header, col_data in
     pd.read_csv(csv_path).items()}

def saveStats(lst: list[float], frac_to_take: float=0.2) -> tuple[
   float, float]:
   '''
  Gets the mean & standard deviation of the last `frac_to_take`
     proportion of its elements. Returned in that order.
  '''
  data_to_keep = lst[int(-frac_to_take*len(lst)):]
  return np.mean(data_to_keep), np.std(data_to_keep)

def getRange(base_val: float, dim_val: float, num_els: int=5):
  '''
  Gets a range around `base_val` with dimensions `dim_val` in each
     direction & total element count `num_els`.
  '''
  u_lim = base_val + dim_val
  l_lim = base_val - dim_val
  step_size = (u_lim - l_lim)/(num_els-1)
  return [l_lim + i*step_size for i in range(num_els)]

def makeCoordDF(c_dict: dict[tuple[float, float], float]):
  '''
  Turns the given dictionary (structure - coordinate tuple:value,
     i.e. (x, y):z) into a `pandas.DataFrame` object.
  '''
  df_data = [(y, x, z) for (x, y), z in c_dict.items()]
  return pd.DataFrame(df_data, columns=['y', 'x', 'z']).pivot(
     index='y', columns='x', values='z')
```

func_lib.py

```
'''
Library of miscellaneous functions, most of which being fairly
    general and not semantically tied to a specific project.
'''

from typing import Any
import numpy as np
import os

def normalise(l: list[float]):
  '''
  Normalises the given list such that its sum is 1.
  '''
  l_sum = sum(l)
  if not l_sum: return l
  return [l_el/l_sum for l_el in l]

def normalise_np(l: np.ndarray[float]):
  '''
  Normalises the given NumPy array.
  '''
  return l/sum(l)

def normalise_dict(d: dict[str, float]) -> dict[str, float]:
  '''
  Normalises the given dictionary (as per its values).
  '''
  return dictify(d.keys(), normalise(d.values()))

def fn(f_n: str):
  '''
  Converts the given string into a proper file name.
  '''
  return f_n.lower().replace(' ', '_')

def float2SN(f: float, p: int=2, do_sci: bool=False):
  '''
  Converts the given number to scientific notation with the given
     precision.

  ### Parameters
  - `f`: The number in question.
  - `p`: The number (int) of digits of precision (past the decimal
      point). For example: f=1302, p=2 -> 1.30e3.
  - `do_sci`: Whether or not to force scientific notation (e.g. 13
      -> 1.3e1).
  '''
  if ((f < 10**(p+1) and f > 10**(-p)) or not f) and not do_sci:
     return f'{f}'
  else:
    pwr = np.floor(np.log10(f))
    return f'{int(f/(10**(pwr-p)))/(10**p)}e{int(pwr)}'

def roundNum(f: float, prec: int=2) -> float:
  '''
  Rounds the given number to the given number of decimal points.

  ### Parameters
  - `f`: The number in question.
  - `prec`: The number of decimal points to round it to.
  '''
  return round(f*(10**prec))/(10**prec)

def roundAndSN(f: float, u_lim: int=4, l_lim: int=2, prec: int=3):
  '''
  General-purpose number processing method. Will return a number
      with the given precision if it falls outside the given
      limits.

  ### Parameters
  - `f`: The number in question.
  - `u_lim`: The power of 10 to use as an upper limit for
      scientific notation.
  - `l_lim`: The power of 10 to use as a lower limit for
      scientific notation. Positive means negative (i.e. l_lim = 2
       means a limit of 0.01.)
  '''
  if (f < 10**u_lim and f > 10**(-l_lim+1)) or not f: return f'{
     roundNum(f, prec=prec)}'
  else: return float2SN(f, p=prec, do_sci=True)

def roundTrailing(*args, max_prec: int=6) -> list[float]:
  '''
  Rounds trailing 0s/9s that may arise from tiny floating-point
     errors elsewhere.
  '''
  rv = []
  for arg in args:
    arg_str = str(arg)
    if len(arg_str) > max_prec + 2:
      if not int(arg_str[-2]): rv += [float(arg_str[:-1])]
      else: rv += [arg + 10**(2-len(arg_str))]
    else: rv += [arg]
  return rv

def normPercentList(l: list[float]) -> list[float]:
  '''
  Turns the given list into a normalised list of percentages.
  '''
  return list(100*normalise_np(np.array(l)))

def printFloatList(l: list[float]):
  '''
  Prints the items of a list of floats (rounded to 2 decimal
     places), as well as their indices.
  '''
  [print(f'{i}:\t{roundNum(l[i])}') for i in range(len(l))]

def printMat(m: list[list]):
  '''
  Prints the given 2d matrix (usually of transition probabilities,
      though not necessarily) in an easier-to-read fashion.
  '''
  rv = '\n'.join(['\t'.join([str(int(100*i)) for i in j]) for j in
      m])
  print(rv)
  return rv

def hapify(g: str):
  '''
  Turns the given genotype into its haploid equivalent (e.g., AA.
      Bb.cc becomes A.B.c). Haploid genotypes are unaffected.

  *Note that heterozygous genes are not preserved, and are instead
      assumed effectively mutated/capital.*
  *As heterozygotes comprise a minuscule fraction of the overall
      population, the effect of this is negligible.*
  '''
  return '.'.join([s[0] for s in g.split('.')])

def dipify(g: str):
  '''
  Turns the given genotype into its diploid equivalent (e.g., A.B.
      c becomes AA.BB.cc). Diploid genotypes are unaffected.
  '''
  return '.'.join([''.join(2*[s])[:2] for s in g.split('.')])

def listify(a: list | np.ndarray | Any) -> list | Any:
  '''
  Produces a deep copy of type list from a given multi-dimensional
      list (or NumPy array). If the elements of the list are
      complex
  types, then these objects will still be the same in memory (the
      deep copy only extends to the lists themselves).
  '''
  if (type(a) is np.ndarray) or (type(a) is list): return [listify
      (el) for el in a]
  else: return a

def dictify(ks: list, vs: list):
  '''
  Turns the given lists of keys and values into a dict. These
      should be the same length!
  '''
  return {k: v for k, v in zip(ks, vs)}

def mkDir(*args):
  '''
  Makes a new directory with the given name, if it does not exist
      already. Takes an arbitrary number of arguments.
  '''
  [os.mkdir(arg) for arg in args if arg not in os.listdir()]

def mkFilePath(*args):
  '''
  Makes a new file at the given path, if it does not exist already
     . Takes an arbitrary number of arguments.
  '''
```

```
  for a in args:
    if a not in os.listdir(): f = open(a, 'x'); f.close()

def mkFile(f_path: str):
  '''
  Makes and returns a new file at the given path, clearing it if
      it already exists.
  '''
  if os.path.exists(f_path): os.remove(f_path)
  return open(f_path, 'x')

def getSubDirs(f_path: str) -> list[str]:
  '''
  Returns a list of all directories within the given directory (
      does not include full path or their own sub-directories).
  '''
  if f_path[-1] != '/': f_path += '/'
  if not os.path.exists(f_path): return []
  return [path_loc for path_loc in os.listdir(f_path) if os.path.
      isdir(f'{f_path}{path_loc}')]

def numSubDirs(f_path: str):
  return len(getSubDirs(f_path)) if os.path.exists(f_path) else 0

def transpose(l: list[list]):
  '''
  Transposes the given two-dimensional list. Must be rectangular;
      that is, all the second-order lists must be equal in length.
  '''
  return [list(l2) for l2 in zip(*l)]

def linspace(l_lim: float=0., u_lim: float=1., res: int=256):
  '''
  Generates an evenly-spaced list of `res` points between `l_lim`
      and `u_lim`.
  '''
  return [l_lim + u_lim*i/(res-1) for i in range(res)]

def isInRange(num: float, rng: list[float]=[0.,1.], do_trunc_check
    : bool=True):
  '''
  Checks whether or not the given number is in the given range (
      default 0-1).
  '''
  if do_trunc_check: num = float(trunc(num)); rng = [float(trunc(r
      )) for r in rng]
  return num >= rng[0] and num <= rng[1]

def trunc(pop_val: str | float, trunc_len: int=7, **kwargs):
  '''
  Truncates the given string/float (cast to string) such that it
      has the specified length (default 7).
  '''
  pop_val = str(pop_val)
  return pop_val if len(pop_val) <= trunc_len else pop_val[:
      trunc_len]
```

```
def list_str(lst: list, limit: int=40, shoulder: int=5):
  '''
  Writes the given list as a string, if it is too long to
     reasonably print to the console.
  '''
  if len(lst) <= limit: return str(lst)
  else: return f'{str(lst[:shoulder])[:-1]} ... {str(lst[-shoulder
     :])[1:]}'
```

gen_funcs.py

```
'''
A library of miscellaneous functions related to the handling of
   alleles and genotypes.
'''

from allele import *
import math

def wf(pc: int):
  '''
  Produces the Wright-Fisher genetic drift transition matrix for
     the given # of alleles.

  *As genetic drift now uses binomial sampling directly instead of
      a transition matrix, this method is now largely obsolete.*
  '''
  tp_dim = pc + 1
  trans_ps = []
  for i in range(tp_dim):
    trans_ps_row = []
    for j in range(pc+1): trans_ps_row += [0]
    trans_ps += [trans_ps_row]
  for i in range(tp_dim):
    for j in range(tp_dim):
      trans_ps[i][j] = math.comb(pc, j)*((i/pc)**j)*(1 - i/pc)**(
         pc-j)
  return trans_ps

def genGenotypes(alleles: list[allele], is_haploid: bool):
  '''
  Generates all possible genotypes.

  ### Parameters
  - `alleles`: The list of allele objects to generate the
     genotypes from.
  - `is_haploid`: Whether or not the genotypes should be haploid.
  '''
  loci = ''
  for a in alleles:
    if a.locus not in loci: loci += a.locus
  num_combs = 3 - is_haploid
  gt = ['' for i in range(num_combs**len(loci))]
  for l in loci:
    for i in range(len(gt)):
      if is_haploid:
        if i%2 == 0: gt[i] += l.upper()
        if i%2 == 1: gt[i] += l.lower()
      else:
        if i%3 == 0: gt[i] += (l.upper() + l.lower())
        if i%3 == 1: gt[i] += 2*l.upper()
        if i%3 == 2: gt[i] += 2*l.lower()
      gt[i] += '.'
    gt.sort()
  gt = [g[:-1] for g in gt]
  return gt

def genAlleles(al: str):
  '''
  Generates genotypes from a single genotype (either haploid or
      diploid). Analogous to `genGenotypes`, but simpler and
      faster.
  '''
  is_haploid = len(al) == 1
  al = al[0]
  if is_haploid: return [al.lower(), al.upper()]
  else: return [2*al.lower(), (al.upper() + al.lower()), 2*al.
     upper()]
```

individual.py

```
'''
File for the class for explicitly-modelled infected individuals.
'''

from gen_funcs import *
from func_lib import *
from random import random

class individual:
    '''
    The class for explicitly-modelled infected individuals.
    '''
    pc = 0                    # Parasite population within the
        individual
    genotype_freqs: dict[str, int] = {}     # Dict of genotypes to '
        frequencies' (absolute #s)
    is_hap = False               # Whether or not this individual's
        parasites are haploid
    mut_chance = 0.0             # Chance of mutation per parasite
        per generation
    para_gens = 1               # Number of parasite generations per
        time step
    do_mutation = False             # Whether or not this individual'
        s parasites are allowed to mutate
    rng: np.random.Generator = None       # NumPy random number
        generator object
    pc_to_transmit = 0             # The number of parasites that get
         transmitted during a successful infection
    marked_for_death = False          # Marked true to indicate that
        it has died/recovered and needs to be removed
    all_sel_bias: dict[str, float] = {}     # Allele selection biases
```

```
      (for genetic drift)
  all_transm_probs: dict[str, float] = {} # Allele transmission
     probabilities
  num_genes: int = 0              # The total number of alleles
     throughout the individual's parasite population
  pc_flt: float = 0.0             # Parasite population as a float
  main_all_char: str = ''           # Mutated (uppercase) allele
     character
  scnd_all_char: str = ''           # Wild (lowercase) allele
     character
  is_mixed_vec: bool = False          # Whether or not this
     individual started out as a mixed-strain vector

  def __init__(self, gnts: list[str]=[], gnt: str='', rng: np.
     random.Generator=None, **kwargs):
    '''
    Initialises the individual.

    ### Parameters
    - `gnts`: The list of genotypes that are allowed to be present
        within the individual
    - `gnt`: The starting genotype of the parasites
    - `rng`: The NumPy random number generator object to use. Not
       meant to vary, but making a new one for every individual
      would be unnecessary & time-consuming
    - `pc`: The number of parasites present in the individual
    - `is_hap`: Whether or not the individual's parasites are
       haploid
    - `mut_chance`: The chance that a parasite will experience a
       mutation in an individual generation
    - `para_gens`: The number of generations the parasites go
       through per time step
    - `do_mutation`: Whether or not mutations can occur inside the
        individual
    - `pc_to_transmit`: The number of parasites to transmit during
        a mixed infection
    '''
    self.__dict__.update(kwargs)
    self.num_genes = self.pc*self.ploidy
    self.genotype_freqs = dict.fromkeys(gnts, 0)
    if gnt: self.genotype_freqs[gnt] = self.pc
    if rng is None: self.rng = np.random.default_rng()
    else: self.rng = rng
    if self.pc_to_transmit > self.pc: self.pc_to_transmit = self.
       pc

  def simPara(self):
    '''
    Simulates the parasites' behavior.
    '''
    for i in range(self.para_gens):
      if self.do_mutation: self.mutate()
      self.genDrift()

  def genDrift(self):
    '''
    Simulates genetic drift & selection.
    '''
    if self.is_dip and not self.is_mixed_vec: return
    if not self.is_mixed: return
    all_freq = self.getAlleleFreqs()
    a = self.main_all_char
    b = self.scnd_all_char
    all_prop = all_freq/self.num_genes
    asb = self.all_sel_bias[a]
    w_avg = asb*all_prop + (1. - all_prop)
    all_prop *= asb/w_avg
    if self.is_dip: all_prop = self.rng.binomial(self.num_genes,
       all_prop)/self.num_genes
    if not all_prop or all_prop == 1:
      all_prop_bool = bool(all_prop)
      self.genotype_freqs[self.ploidy*a] = all_prop_bool*self.pc
      self.genotype_freqs[self.ploidy*b] = (not all_prop_bool)*
         self.pc
      if self.is_dip: self.genotype_freqs[a + b] = 0
      return
    if self.is_hap:
      gtfs_big = round(all_prop*self.pc_flt)
      gtfs_sml = round((1.-all_prop)*self.pc_flt)
      self.genotype_freqs[a] = gtfs_big
      self.genotype_freqs[b] = gtfs_sml
    else:
      inv_prop = 1. - all_prop
      probs = [inv_prop**2, 2*all_prop*inv_prop, all_prop**2]
      all_dist = self.rng.multinomial(n=self.pc, pvals=probs)
      self.genotype_freqs[b + b] = all_dist[0]
      self.genotype_freqs[a + b] = all_dist[1]
      self.genotype_freqs[a + a] = all_dist[2]

  def mutate(self):
    '''
    Effects the mutations observed over a single generation.
    '''
    if self.is_dip: return
    pre_gtfs = self.genotype_freqs.copy()
    num_muts = 0
    for gt in self.genotype_freqs:
      if not pre_gtfs[gt]: continue
      mut_param = self.mut_chance*pre_gtfs[gt]
      if mut_param < 0.05: continue
      if mut_param > 250: num_muts = int(mut_param)
      else: num_muts = self.rng.poisson(mut_param)
      if not num_muts: continue
      self.genotype_freqs[gt] -= num_muts
      self.genotype_freqs[gt.swapcase()] += num_muts

  def getAlleleFreqs(self):
    '''
    Gets the frequency of the mutated allele in the individual's
       parasite population. Used primarily in `genDrift`.
    '''
    if self.is_hap: return self.genotype_freqs[self.main_all_char]
    else:
      mac = self.main_all_char
      return 2*self.genotype_freqs[mac+mac] + self.genotype_freqs[
          mac+self.scnd_all_char]

  def getGenotypes(self):
    '''
    Returns all present genotypes as a list.
    '''
    return [gt for gt in self.genotype_freqs if self.
        genotype_freqs[gt]]

  def getGenotypeTransWeights_unwgt(self, do_weights: bool=False):
    '''
    Returns the genotypes' transmission weights, either unweighted
        (default) or weighted (`do_weights=True`) with respect to
    transmission probabilities.
    '''
    if do_weights: return normalise([self.genotype_freqs[gt]*self.
        all_transm_probs[gt[0]] for gt in self.genotype_freqs])
    else: return [self.genotype_freqs[gt]/self.pc_flt for gt in
        self.genotype_freqs]

  def doesContactTransmit(self):
    '''
    Whether or not a contact between this individual and a
        susceptible individual results in a transmission.
    '''
    gtf_wgt_sum = 0
    for gt in self.genotype_freqs: gtf_wgt_sum += self.
        all_transm_probs[gt[0]]*self.genotype_freqs[gt]/self.
        pc_flt
    return random() < gtf_wgt_sum

  def infectMix(self, pc_num: int=1, do_test_contact: bool=True)
     -> dict[str, float]:
    '''
    Performs a mixed infection (dict of parasite genotype to count
        ). Can also be used to simply get a distribution of
        parasites if desired.

    ### Parameters
    - `pc_num`: The number of parasites to use in the distribution
        .
    - `do_test_contact`: Whether or not to allow the transmission
        to fail, depending on the alleles' transmission
        probabilities.
    '''
    if do_test_contact:
      if not self.doesContactTransmit(): return dict.fromkeys(self
          .genotype_freqs, 0)
    return dictify(self.genotype_freqs.keys(), self.rng.
        multinomial(pc_num, self.getGenotypeTransWeights_unwgt(
        do_weights=True)))

  def infectSelf(self, pc_num: int, strn: str) -> list[str]:
    '''
    Infects the individual itself.
    ### Parameters
    - 'pc_num': The number of parasites in the transmission.
    - 'strn': The strain to infect it with.
    '''
    if pc_num > self.pc: pc_num = self.pc
    to_replace = self.infectMix(pc_num, do_test_contact=False)
    strn_match = self.match(strn)
    for stn in to_replace:
      self.genotype_freqs[stn] -= to_replace[stn]
      if self.genotype_freqs[stn] < 0: to_replace[stn] += self.
          genotype_freqs[stn]; self.genotype_freqs[stn] = 0
      self.genotype_freqs[strn_match] += to_replace[stn]

  def infectSelfMult(self, mix: dict[str, int]):
    '''
    Infects the individual with the given strain distribution.
    '''
    mix_sum = sum(mix.values())
    if mix_sum >= self.pc:
      self.setToMix(mix, mix_sum=mix_sum)
      return
    for strn in mix: self.infectSelf(mix[strn], strn)

  def setToMix(self, mix: dict[str, int], mix_sum: int=0):
    '''
    Sets the individual's parasite distribution to the given
        strain distribution. (If the sum of the distribution is
        already known,
    it can be provided as a keyword argument.)
    '''
    pc_transmitted = 0
    if not mix_sum: pc_transmitted = sum(mix.values())
    else: pc_transmitted = mix_sum
    rem = 0.
    matched = ''
    if self.is_dip:
      if not(not mix[self.main_all_char] or mix[self.main_all_char
          ] == pc_transmitted): self.is_mixed_vec = True
    self.genotype_freqs = dict.fromkeys(self.genotype_freqs, 0)
    for strn in mix:
      matched = self.match(strn)
      amt_raw = self.pc_flt*(mix[strn]/pc_transmitted) + rem
      amt = int(amt_raw)
      rem = amt_raw - amt
      self.genotype_freqs[matched] += amt
    if rem > 0.999: self.genotype_freqs[matched] += 1

  def match(self, s2m: str):
    '''
    Matches the given strain to the format required of this
        individual type (i.e., haploid or diploid).
    '''
    m2u = dipify
    if self.is_hap: m2u = hapify
    return m2u(s2m)

  def matchDist(self, sd2m: dict[str, int]):
    '''
    Takes a given strain distribution and matches it to the format
        required of this individual (i.e. hap/diploid).
    '''
    new_dist = dict.fromkeys(self.genotype_freqs, 0)
    for s in sd2m: new_dist[self.match(s)] += sd2m[s]
    return new_dist

  def correction_det(self, sn: str=''):
    '''
    'Corrects' for multi-counting by weighting the individual
       according to either the number of strains present within
       it (no parameter),
    or how prevalent the given strain is.
    '''
    if sn: return self.genotype_freqs[sn]/self.pc_flt
    else: return 1/self.num_gnts_present

  def correction(self, sn: str=''):
    '''
    Returns either true or false with a probability of 1/(num.
       strains present) for no parameter, or the frequency of the
        given strain.
    '''
    return random() < self.correction_det(sn)

  def checkGtfs(self, loc: str=''):
    '''
    Makes sure the genotype freq. values align with the total
       parasite #. Takes some identifying string (e.g. the source
        of the call)
    as an argument.
    '''
    if sum(self.genotype_freqs.values()) != self.pc:
      print(f'({loc}) pc {self.pc}, gtfs {self.genotype_freqs} (
         sum {sum(self.genotype_freqs.values())})')
      exit()

  @property
  def is_dip(self):
    return not self.is_hap

  @is_dip.setter
  def is_dip(self, value: bool):
    self.is_hap = not value

  @property
  def ploidy(self):
    return self.is_dip + 1

  @property
  def is_mixed(self):
    if self.is_hap:
      if not self.genotype_freqs[self.main_all_char]: return False
      if self.genotype_freqs[self.main_all_char] == self.pc:
         return False
      return True
    if not self.is_mixed_vec: return False
    been_a_something_already = False
    for gt in self.genotype_freqs:
      if been_a_something_already and self.genotype_freqs[gt]:
          return True
      elif self.genotype_freqs[gt]: been_a_something_already =
          True; continue
    return False

  @property
  def num_gnts_present(self):
    if self.is_hap:
      if not self.genotype_freqs[self.main_all_char]: return 1
      if self.genotype_freqs[self.main_all_char] == self.pc:
          return 1
      return 2
    else:
      been_a_nothing_already = False
      for gt in self.genotype_freqs:
        if been_a_nothing_already and not self.genotype_freqs[gt]:
            return 1
        elif not self.genotype_freqs[gt]: been_a_nothing_already =
            True; continue
      if been_a_nothing_already: return 2
      return 3

  def __str__(self):
    return f'{self.genotype_freqs}'

  def __repr__(self):
    return self.__str__()
```

model.py

```
'''
The file for the stochastic SIRS model class.
'''

from func_lib import *
from population import *
from allele import *
from random import shuffle
import pickle

class SIR:
  '''
  The stochastic SIRS model class.
  '''
  pn = ''       # Short population name
  pn_full = ''  # Full population name
  sn = 'init'    # Strain name (i.e. genotype)
  alleles = []  # List of allele objects
  bd = -1.0     # Birth/death rate
  ir = -1.0     # Infection rate (obsolete)
  rr = -1.0     # Recovery rate
  wi = -1.0     # Waning immunity rate
```

```
itr: dict[population, float] = {}  # Dict of population objects
   to contact rates (usually only one entry)
bds = 0.0     # Birth/death rate (for susceptibles specifically;
   value assigned in setRs)
Rs = []       # Transition rate matrix
Es = []       # List of events (3-element arrays that change S, I
   , R respectively)
num_Es = -1    # Length of Es
is_vector = False         # Whether or not this model describes a
    vector
other_pop: population = None  # The other population this model
   interacts with

def __init__(self, p0: population, **kwargs):
  '''
  Initialises the model with the given parameters.

  ### Parameters
  - `p0`: Initial population, as a 3-element list (S, I, R)
  - `pn`: The name (short) of the population this model belongs
     to
  - `pn_full`: The full name of the population this model
     belongs to
  - `sn`: The name of the strain this model belongs to (i.e.
     genotype, in most cases)
  - `bd`: Birth/death rate (only relevant to vectors)
  - `ir`: Infection rate (obsolete)
  - `rr`: Recovery rate (only relevant to hosts)
  - `wi`: Waning immunity rate (only relevant to hosts)
  - `itr`: Interspecific transmission rates from this model to
     other populations (dict pop:float)
  - `is_vector`: Whether or not this model describes a vector
     population (bool)
  '''
  self.pop = p0
  self.__dict__.update(kwargs)
  self.pop.is_vector = self.is_vector
  self.pop.pn = self.pn
  self.pop.addStrain(self.sn)
  E1 = [1, 0, 0]  # Birth (used 3 times, one for each pop type)
  E2 = [-1, 1, 0] # Infection
  E3 = [0, -1, 1] # Recovery
  E4 = [-1, 0, 0] # Death of susceptible
  E5 = [0, -1, 0] # Death of infected
  E6 = [0, 0, -1] # Death of recovered
  E7 = [1, 0, -1] # Waning immunity
  self.Es = [E1, E2, E3, E4, E5, E6, E7, E1, E1, E1] # first E1,
      E2 are deprecated
  self.num_Es = len(self.Es)

def setRs(self):
  '''
  Generates the different transition rates based on the model's
     parameters and population.
  '''
  # consider making this return the result instead of store it
  S = self.pop.sus
```

```
    R = self.pop.rec[self.sn]
    is_hap = self.pop.is_hap
    p2 = self.other_pop
    I_UW = 0
    I_WS = 0
    for ind in self.pop.individuals:
      if ind.genotype_freqs[self.sn]:
        I_WS += ind.correction_det(sn=self.sn)  # weighted
            according to how prevalent the strain is inside the
            indv
        if is_hap: continue
        I_UW += ind.correction_det()      # unweighted infections
            (simple 'yes/no' on strain presence)
    self.Rs = [ 0,        # deprecated (formerly births)
          0,        # deprecated (formerly intra-population
             infections)
          self.rr*I_WS,  # recoveries
          self.bds*S,    # susceptible deaths
          self.bd*I_UW,  # infected deaths
          self.bd*R,    # recovered deaths
          self.wi*R,    # waning immunity
          self.bds*S,    # susceptible births
          self.bd*I_UW,  # infected births (as in births from
             infecteds, not newly-born infecteds)
          self.bd*R,    # recovered births (again as in births
             from recovereds)
          self.itr[p2]*I_WS*(p2.sus+p2.getSusInfNum(self.sn))/p2.
             tot_pop]
                  # interspecific contacts (cross-pop infections)

  def trans(self, idx: int, rpt: int=1):
    '''
    Effects the changes in the population dictated by the
        simulation.

    ### Parameters
    - 'idx': The index of the desired event, corresponding with
        the order of 'Rs'.
    - 'rpt': The number of times to repeat said event.
    '''
    pop = self.pop
    self_sn = self.sn
    pc_2_trans = self.pop.pc_to_transmit
    if idx >= self.num_Es:
      pop = self.other_pop
      idx = 1
      num_inf = 0
      num_mixes = 0
      num_loops = 0
      num_failed_trans = 0
      max_loops = 10000
      indvs_lst = self.pop.individuals
      while num_inf < rpt: # loops until the required number of
          events have occurred, or until the hard limit has been
          reached (unusual)
        for indv in indvs_lst:
          if not indv.genotype_freqs[self_sn]: continue
          elif indv.correction(sn=self_sn):
            num_inf += 1
            if indv.is_hap: # weird conditional structure is to
                save on time
              pop.infectMix(indv.infectMix(pc_2_trans))
              num_mixes += 1
            elif indv.is_mixed_vec:
              if indv.is_mixed:
                pop.infectMix(indv.infectMix(pc_2_trans))
                num_mixes += 1
              elif not indv.doesContactTransmit():
                 num_failed_trans += 1
            elif not indv.doesContactTransmit(): num_failed_trans
                += 1
          if num_inf >= rpt: break
        num_loops += 1
        if num_loops >= max_loops: break
        shuffle(indvs_lst) # slow but necessary to ensure fairness
      rpt -= (num_mixes + num_failed_trans)
    if rpt: pop.addPop(list(np.multiply(self.Es[idx], rpt)),
        self_sn, pc_2_trans)

  def newStrain(self, nsn='new'):
    '''
    Generates a copy of this model with the given strain name.
    '''
    new_mdl = SIR(self.pop, sn=nsn, **self.__dict__)
    new_mdl.itr = dict(new_mdl.itr)
    return new_mdl

  def mutate(self, param: str, fac: float, vec: 'SIR'=None):
    '''
    Effects the given parameter change.

    *Now obsolete. Largely a holdover from when alleles had
        macroscopic (population-level), not microscopic (
        individual-level),
    effects on the simulation.*

    ### Parameters
    - `param`: The parameter of the model to change.
    - `fac`: The numerical factor to change it by. This value is
        used directly.
    - `vec`: The corresponding (same strain) model for the vector
        population. Only necessary if `param` is `itr`.
    '''
    if type(self.__dict__[param]) is dict:
      for k in self.__dict__[param]: self.__dict__[param][k] *=
          fac
      if vec is not None: vec.__dict__[param][self.pop] *= fac
    else: self.__dict__[param] *= fac

  def mutateMult(self, params: list[str], fac: float, vec: 'SIR'=
      None):
    '''
    Effects the given parameter changes.
    *Now obsolete. Largely a holdover from when alleles had
       macroscopic (population-level), not microscopic (
       individual-level),
    effects on the simulation.*

    ### Parameters
    - `params`: The parameters of the model to change.
    - `fac`: The numerical factor to change them by (all the same)
       . This value is used directly.
    - `vec`: The corresponding (same strain) model for the vector
       population. Only necessary if `params` includes `itr`.
    '''
    for p in params: self.mutate(p, fac, vec)

  def updateGenotype(self, g: str, alleles: list[allele], vec: '
     SIR'=None):
    '''
    Generates a new model based on the given genotype.

    ### Parameters
    - `g`: The genotype, as a string of characters corresponding
       to alleles.
    - `alleles`: The list of all possible alleles, as allele
       objects.
    - `vec`: The corresponding (same strain) model for the vector
       population. Only necessary if one of the alleles affects `
       itr`.
    '''
    new_model = self.newStrain(g)
    for a in alleles:
      if a.char in g:
        if a.fav_pop == new_model.pn: new_model.mutate(a.param, 1+
           a.fac, vec)
        if a.unf_pop == new_model.pn: new_model.mutate(a.param,
           1/(1+a.fac), vec)
    return new_model

  def r0(self, vec_mdl: 'SIR') -> float:
    '''
    Estimates R0 for the given model. Meant to be more a vague
       guideline than a hard and fast rule. Takes the
       corresponding (same strain)
    vector model as an input.
    '''
    return (vec_mdl.pop.tot_pop/self.pop.tot_pop)*self.itr[vec_mdl
       .pop]*vec_mdl.itr[self.pop]/(self.rr*vec_mdl.bd)

  def printParams(self):
    '''
    Prints each of the model's parameters and their values to the
       console.
    '''
    print(f'\n{self.__str__()} params:')
    [print(f'{k}: {self.__dict__[k]}') for k in self.__dict__]

  def save(self, s_dir: str=''):
    s_fn = f'{s_dir}/{self.fn}'
    os.makedirs(s_dir, exist_ok=True)
    f = open(s_fn, 'wb')
    pickle.dump(self, f)
    f.close()

  def load(path: str) -> 'SIR':
    if not os.path.exists(path): print(f'failed to load model from
        path {path}'); return
    f = open(path, 'rb')
    mdl = pickle.load(f)
    f.close()
    return mdl

  @property
  def fn(self): # strange naming scheme is due to pickle pathing
     being case-insensitive (alleles don't work)
    disc = 'et'
    if self.sn == self.sn.upper(): disc = 'm'
    elif self.sn == self.sn.lower(): disc = 'w'
    if len(self.sn) == 2: disc = f'h{disc}'
    return f'{self.pn}_{disc}.sir'

  def __str__(self):
    return f'population {self.pn}, strain {self.sn}'

  def __repr__(self):
    return self.__str__()
```

population.py

```
'''
The file for the class for a population.
'''

from func_lib import *
from individual import *
from random import choice

class population:
  '''
  The class for a population.
  '''
  sus = -1
  inf: dict[str, int] = {}
  rec: dict[str, int] = {}
  pn = ''
  is_vector = False
  indvs: list[individual] = []
  is_hap = False
  indv_params = {}
  rng: np.random.Generator = None
  pc_to_transmit = 0
  all_sel_bias: dict[str, float] = {}
  all_transm_probs: dict[str, float] = {}
  init_pop = 0
  gnts: list[str] = []
```

```
num_gnts = 0
main_all_char: str = ''

def __init__(self, p0: list[int], pn: str='', isn: str='init',
    **kwargs):
   '''
   Initialises the population.

   ### Parameters
   - `p0`: A 3-element integer list of the starting population
      amounts (S, I, R)
   - `pn`: The population's name
   - `isn`: The name of the initial strain
   - `is_vector`: Whether or not this population is a vector
   - `is_hap`: Whether or not this population is haploid
   '''
   self.sus = p0[0] + 1 # adding 1 to ensure the population is
      always non-zero (irrelevant in the grand scheme of things)
   self.inf: dict[str, int] = {}
   self.rec: dict[str, int] = {}
   self.indvs: list[individual] = []
   self.inf[isn] = p0[1]
   self.rec[isn] = p0[2]
   self.pn = pn
   self.__dict__.update(kwargs)
   self.indv_params = kwargs
   self.rng = np.random.default_rng()
   self.indv_params['rng'] = self.rng
   self.indv_params['pc_flt'] = float(self.indv_params['pc'])
   self.all_sel_bias: dict[str, float] = {}
   self.all_transm_probs: dict[str, float] = {}
   self.gnts: list[str] = []

def getPop(self, sn: str='init') -> list[int]:
   '''
   Returns a 3-element S, I, R list for the given strain.
   '''
   return [self.sus, self.inf[sn], self.rec[sn]]

def getAllPop(self) -> list[float]:
   '''
   Returns all population elements as a list. S is first, then
      all Is (frequencies), then all Rs.
   '''
   infs = normalise(list(self.getInfWgt().values()))
   return [self.sus] + infs + list(self.rec.values())

def getAllPopNms(self):
   '''
   Returns the names of all population elements. S is first, then
       all Is, then all Rs.
   '''
   return ['S'] + [f'I ({sn})' for sn in self.inf.keys()] + [f'R
      ({sn})' for sn in self.rec.keys()]

def getInfWgt(self):
   '''
```

```
    Get a dict of strain to # infected, weighted according to the
      strain's presence within the individual.
    '''
    infs = dict.fromkeys(self.inf, 0)
    for ind in self.indvs:
      for gt in ind.genotype_freqs: infs[gt] += ind.genotype_freqs
        [gt]/ind.pc_flt
    return infs

  def addPop(self, p: list[int], sn: str='init', pc_trans: int=0):
    '''
    Adds the given population quantities.

    ### Parameters
    - `p`: The quantities to add, as a 3-element list (S, I, R).
    - `sn`: The strain to add them to.
    - `pc_trans`: The number of parasites a mixed infection uses.
      Note that this is based on the *source,* not the
      destination!
    '''
    sn = self.match(sn)
    test_add = [self.sus, self.inf[sn], self.rec[sn]]
    neg = -1
    for i in range(3):
      if test_add[i] < 0:
        print(test_add)
        print(f'strain {sn}')
        self.printDat()
        exit()
      if test_add[i] + p[i] < 0: neg = test_add[i]/abs(p[i])
    if neg >= 0:
      for i in range(3):
        p[i] *= neg
        if test_add[i] + p[i] < 0:
          print(test_add)
          print(p)
          print(neg)
          print(f'strain {sn}')
          self.printDat()
          exit()
    S = int(p[0])
    I = int(p[1])
    R = int(p[2])
    sus_new = self.sus
    inf_new = self.inf.copy()
    rec_new = self.rec.copy()

    sus_new += S
    inf_new[sn] += I
    inf_indvs = []
    i_change = 0
    to_change = {}
    tba: list[individual] = []
    if I > 0:
      sus_indvs = self.getSusInf(sn)
      strain_indvs = {sn_i: self.getSusInf(sn_i, is_present=True,
        indvs_lst=sus_indvs) for sn_i in self.inf}
      sus_pop_nums = [self.sus] + [len(strain_indvs[sn_i]) if sn
         != sn_i else 0 for sn_i in self.inf]
      sus_pop_wgts = normalise(sus_pop_nums)
      i_change, to_change = self.getChanges(I, sus_pop_wgts)
      tba = self.makeIndvs(sn, i_change)
      for strn in to_change:
        sus_new += to_change[strn]
        indvs_to_infect = strain_indvs[strn]
        for ind in indvs_to_infect[:to_change[strn]]: ind.
           infectSelf(pc_trans, sn)
      self.indvs += tba
    elif I < 0:
      inf_indvs = self.getSusInf(sn, is_present=True)
      for ind in inf_indvs[:-I]:
        ind.marked_for_death = True
        for gt in ind.genotype_freqs:
          if ind.genotype_freqs[gt]:
            if gt != sn: inf_new[gt] -= 1
      self.refresh()
    rec_new[sn] += R

    self.sus = sus_new
    self.inf = inf_new
    self.rec = rec_new

  def getChanges(self, pop_num: int, weights: list[float]) ->
     tuple[int, dict[str, int]]:
    '''
    Distributes infections between full-susceptibles and infected-
       susceptibles according to the given weights.
    '''
    to_change_lst = self.rng.multinomial(pop_num, weights)
    return to_change_lst[0], dictify(self.inf.keys(),
       to_change_lst[1:])

  def makeIndvs(self, sn: str, num_indvs: int):
    '''
    Make the given number of individuals with the given strain.
    '''
    return [individual(gnts=self.gnts, gnt=sn, **self.indv_params)
        for i in range(int(num_indvs))]

  def addStrain(self, nsn: str):
    '''
    Creates a new strain with the given name.
    '''
    if nsn in self.inf.keys(): return
    self.inf[nsn] = 0
    self.rec[nsn] = 0

  def match(self, s2m: str):
    '''
    Matches the given strain to the format required of this
       population (i.e., haploid or diploid).
    '''
    m2u = dipify
    if self.is_hap: m2u = hapify
    return m2u(s2m)

  def getSusInf(self, sn: str, is_present: bool=False, indvs_lst:
     list[individual]=None):
    '''
    Gets list of infected individuals either with strain `sn` (`
       is_present`) or susceptible to infection by the strain
    (`not is_present`). A list to draw from can be provided, if
       desired (otherwise, it'll draw from `self.indvs`).
    '''
    is_present -= 1
    if indvs_lst is None: indvs_lst = self.indvs
    return [ind for ind in indvs_lst if bool(ind.genotype_freqs[sn
       ])+is_present]

  def getSusInfNum(self, sn: str, is_present: bool=False,
     indvs_lst: list[individual]=[]):
    '''
    Gets the number of infected individuals either with strain `sn
       ` (`is_present`) or susceptible to infection by the strain
    (`not is_present`). A list to draw from can be provided, if
       desired (otherwise, it'll draw from `self.indvs`).
    '''
    rv = 0
    sn = self.match(sn)
    if not indvs_lst: indvs_lst = self.indvs
    is_present -= 1
    for ind in indvs_lst:
      if bool(ind.genotype_freqs[sn])+is_present: rv += 1
    return rv

  def infectMix(self, mix: dict[str, int]):
    '''
    Performs a mixed infection using the given strain distribution
       .
    '''
    is_empty = True
    for gt in mix:
      if mix[gt]: is_empty = False
    if is_empty: return
    is_a_sus = random() < self.sus/(self.sus + len(self.indvs))
    if is_a_sus:
      new_indv = individual(gnts=self.gnts, **self.indv_params)
      new_indv.setToMix(mix)
      self.indvs += [new_indv]
      for gnt in new_indv.genotype_freqs:
        if new_indv.genotype_freqs[gnt]: self.inf[gnt] += 1
      self.sus -= 1
    else:
      indv_to_infect = choice(self.indvs)
      init_gts: list[str] = []
      init_is_full = indv_to_infect.num_gnts_present-1 ==
         indv_to_infect.ploidy
      if not init_is_full: init_gts = indv_to_infect.getGenotypes
         ()
      indv_to_infect.infectSelfMult(mix)
      fin_is_full = indv_to_infect.num_gnts_present-1 ==
          indv_to_infect.ploidy
      if init_is_full and fin_is_full: return
      fin_gts: list[str] = []
      if not fin_is_full: fin_gts = indv_to_infect.getGenotypes()
      if fin_gts != init_gts: # can likely be made meaningfully
          faster
        gts_rmv = list(set(init_gts) - set(fin_gts))
        gts_add = list(set(fin_gts) - set(init_gts))
        for gt in gts_rmv: self.inf[gt] -= 1
        for gt in gts_add: self.inf[gt] += 1

  def updateSelBiases(self, alleles: list[allele]):
    '''
    Updates the population's selection & transmission biases based
        on the properties of the given alleles.
    '''
    self.all_sel_bias: dict[str, float] = {}
    self.all_transm_probs: dict[str, float] = {}
    for a in alleles:
      self.all_sel_bias[a.char] = a.sel_advs[self.pn]
      self.all_transm_probs[a.char] = a.transm_probs[self.pn]
      self.all_transm_probs[a.char.lower()] = a.base_transm_probs[
          self.pn]
      self.main_all_char = a.char.upper()
    for ind in self.indvs:
      ind.all_sel_bias = self.all_sel_bias.copy()
      ind.all_transm_probs = self.all_transm_probs.copy()
      ind.main_all_char = self.main_all_char
      ind.scnd_all_char = self.main_all_char.lower()
    self.indv_params['all_sel_bias'] = self.all_sel_bias
    self.indv_params['all_transm_probs'] = self.all_transm_probs
    self.indv_params['main_all_char'] = self.main_all_char
    self.indv_params['scnd_all_char'] = self.main_all_char.lower()

  def getGntDist(self):
    '''
    Gets all the individuals' (capital/mutated) allele frequencies
        .
    '''
    return [ind.getAlleleFreqs()/ind.num_genes for ind in self.
        indvs]

  def refresh(self, update: bool=True):
    '''
    Filters out individuals that have been 'marked for death,' and
        (optionally) updates the macroscopic infection tracking
        to make sure
    it aligns with the microscopic state of the simulation.
    '''
    self.indvs = [ind for ind in self.indvs if not ind.
        marked_for_death] # consider trying to fold the two for
        loops here together
    if update: self.update()

  def update(self):
    '''
    Update the (macroscopic) strain to infected dictionary to make
```

```
        sure it aligns with the microscopic state of the
       simulation.
    '''
    # presumably unimportant, given that the macroscopic inf dict
       is no longer really used - consider eliminating entirely
       if it
    # starts to incur a meaningfully large computational cost
    self.inf = dict.fromkeys(self.inf, 0)
    for ind in self.indvs:
      gtfs = ind.genotype_freqs
      for gt in gtfs:
        if gtfs[gt]: self.inf[gt] += 1

  def printDatStr(self):
    '''
    Returns the population's information as a string.
    '''
    return ''.join([f'Population {self.pn} (total {self.tot_pop})\
       n',
           f'S:\t{float2SN(self.sus, p=3)}\n',
           f'\tI\n',
           '\n'.join([f'{k}:\t{float2SN(self.inf[k])}' for k in
              self.inf]),
           f'\nTotal:\t{self.num_inf}',
           f'\n\tR\n',
           '\n'.join([f'{k}:\t{float2SN(self.rec[k])}' for k in
              self.rec])])

  def printDat(self):
    '''
    Prints the population's information to the console.
    '''
    print(self.printDatStr())

  @property
  def is_dip(self):
    return not self.is_hap

  @is_dip.setter
  def is_dip(self, value: bool):
    self.is_hap = not value

  @property
  def individuals(self) -> list[individual]:
    return self.indvs

  @individuals.setter
  def individuals(self, value: list[individual]):
    self.indvs = value

  @property
  def tot_pop(self):
    if self.is_hap and self.init_pop: return self.init_pop
    return int(self.sus + self.num_inf + sum(self.rec.values()))

  @property
  def num_inf(self):
    return len(self.individuals)

  def __str__(self):
    return self.pn

  def __repr__(self):
    return self.__str__()
```

sim_lib.py

```
'''
The file containing the core shell of the simulation.
'''

from model import *
from allele import *
from random import shuffle
import numpy as np

def simShell(tmax: float, mdls: list[SIR], nt: float=2e5, alleles:
     list[allele]=[], init_mut_prop: float=0.,
          num_hist: int=5):
  '''
  Manages the time iterations of the simulation.

  ### Parameters
  - `tmax`: The maximum amount of time to run the simulation for.
  - `mdls`: A list containing the models governing each population
      (initialised with parameters & populations).
  - `nt`: The number of time steps to use, as a 'float' (e.g. 2e5
    - integer in floating-point form).
  - `alleles`: The list of all possible alleles (allele objects).
    Irrelevant if not using the allele model.
  - `init_mut_prop`: The initial proportion of mutated alleles to
    use. Must be between 0 and 1.
  - `num_hist`: The number of histograms to generate (varying over
      time).

  ### Returns
  - `ts`: A NumPy array of the times visited by the simulation (
    neither equispaced nor integer).
  - `ps`: A NumPy array that contains the populations (flattened)
    at each time. Rows index population, columns index time.
  - `pops`: A list of all population objects.
  - `hists_v`: Histogram data for vectors.
  - `hists_h`: Histogram data for hosts.
  - `hist_tms`: The times at which histogram data were taken.
  '''
  dt = tmax/(nt - 1)
  nt = int(nt)
  if num_hist < 2 and num_hist != 0: num_hist = 2
  hist_fac = ((num_hist - 1)/nt)*bool(num_hist)

  vec_strain_2_mdl = {}
  is_haploid = False
  for m in mdls:
    if m.pop.is_hap: is_haploid = True
  is_hyb = False
  for m in mdls:
    if m.pop.is_dip and is_haploid: is_hyb = True
  if len(alleles):
    genotypes = genGenotypes(alleles, is_haploid)
    if is_hyb: genotypes_dip = genGenotypes(alleles, False)
    else: genotypes_dip = []
    new_models = []
    mdls.sort(key=lambda x: x.is_vector, reverse=True)
    for m in mdls: # vec mdl is assumed to be first
      gt = genotypes
      if m.is_vector and is_hyb: gt = genotypes_dip
      m.pop.gnts = gt.copy()
      gt.reverse() # to put 'recessive' (small) first, as it's
          assumed to be the 'wild' type
      for g in gt:
        vec_mdls = [None]
        new_models_temp = []
        if not m.is_vector: vec_mdls = vec_strain_2_mdl[g]
        new_models_temp = [m.updateGenotype(g, alleles, vec_mdl)
           for vec_mdl in vec_mdls]
        new_models_temp = new_models_temp[:1]
        for new_model in new_models_temp:
          if 'init' in new_model.pop.inf.keys():
            new_model.pop.addPop([0] + new_model.pop.getPop()[1:],
                g)
            del new_model.pop.inf['init']
            del new_model.pop.rec['init']
          new_models += [new_model]
          if m.is_vector:
            g2 = g
            if is_hyb: g2 = hapify(g)
            if g2 in vec_strain_2_mdl.keys(): vec_strain_2_mdl[g2]
                += [new_model]
            else: vec_strain_2_mdl[g2] = [new_model]
    mdls = new_models

  [m.setRs() for m in mdls]
  strain_2_mdl: dict[str, list[SIR]] = {}
  for m in mdls:
    if m.sn not in strain_2_mdl.keys(): strain_2_mdl[m.sn] = [m]
    else: strain_2_mdl[m.sn] += [m]
  for s in strain_2_mdl:
    s_mdls = strain_2_mdl[s]
    vec_mdl = [sm for sm in s_mdls if sm.is_vector]
    if len(vec_mdl):
      vec_mdl = vec_mdl[0]
      if not is_hyb: s_mdls.pop(s_mdls.index(vec_mdl))
      else: continue
    else: vec_mdl = strain_2_mdl[dipify(s)][0]
    s_mdls_2: list[SIR] = []
    for sm in s_mdls:
      if sm not in s_mdls_2: s_mdls_2 += [sm]
    s_mdls = s_mdls_2
  ts_i = np.array(range(int(nt)))
  ps_init = np.empty(shape=(nt, len(mdls), len(mdls[0].pop.
     getAllPop())))
  ps = listify(ps_init)
  pops = [m.pop for m in mdls]
  pops_check = []
  for p in list(pops):
    if hex(id(p)) not in pops_check: pops_check += [hex(id(p))]
    else: pops.remove(p)
  num_pops = len(pops)
  num_Rs = len(mdls[0].Rs)
  num_mdls = len(mdls)
  all_Rs = np.array([0.0 for i in range(num_mdls*num_Rs)])
  vec_pop = [p for p in pops if p.is_vector][0]
  host_pop = [p for p in pops if not p.is_vector][0]
  for p in pops:
    max_inf = -1
    max_strn = ''
    for s in p.inf:
      if p.inf[s] > max_inf:
        max_inf = p.inf[s]
        max_strn = s
    if init_mut_prop:
      num_mut_infs = int(max_inf*init_mut_prop)
      p.inf[max_strn] = num_mut_infs
      p.inf[max_strn.lower()] = max_inf - num_mut_infs
      for ind in p.individuals[:num_mut_infs]:
        ind.genotype_freqs[max_strn.upper()] = ind.pc
        ind.genotype_freqs[max_strn.lower()] = 0
      shuffle(p.individuals)
    else:
      p.inf[max_strn] = 0
      p.inf[max_strn.lower()] = max_inf
  [p.updateSelBiases(alleles) for p in pops]
  for p in pops:
    p.init_pop = p.tot_pop
    p.num_gnts = len(p.gnts)
  for m in mdls: m.bds = m.bd/m.pop.num_gnts

  hists_h = []
  hists_v = []
  hist_tms = []
  for i in ts_i:
    for j in range(num_mdls):
      mdls[j].setRs()
      for k in range(num_Rs):
        all_Rs[j*num_Rs+k] = mdls[j].Rs[k]
    sum_Rs = sum(all_Rs)
    Xs = adaptSim(all_Rs/sum_Rs, sum_Rs, dt)
    for i_m in range(num_mdls):
      for i_r in range(num_Rs):
        rpt = Xs[i_m*num_Rs+i_r]
        if rpt: mdls[i_m].trans(i_r, rpt)
    for i_p in range(num_pops): ps[i][i_p] = pops[i_p].getAllPop()
    for p in pops:
      for indv in p.individuals: indv.simPara()
      p.update()
    vpi = len(vec_pop.individuals)
    hpi = len(host_pop.individuals)
```

```
    print(f'{int(100*i/nt)}%; vec indvs: {vpi}; host indvs: {hpi};
        vec pop: {vec_pop.tot_pop}; host pop: {host_pop.tot_pop}
       ',
         end='\r')
    hist_check = hist_fac*i
    hist_check_2 = int(hist_fac*(i+1))
    hist_check_int = int(hist_check)
    if (hist_check_int == hist_check or hist_check_2 >
       hist_check_int) and num_hist:
      hists_h += [host_pop.getGntDist()]
      hists_v += [vec_pop.getGntDist()]
      hist_tms += [i*dt]
  hists_h += [host_pop.getGntDist()]
  hists_v += [vec_pop.getGntDist()]
  hist_tms += [ts_i[-1]*dt]
  return ts_i*dt, ps, pops, hists_v, hists_h, hist_tms, mdls

def adaptSim(ps: np.ndarray[float], sum_Rs: float, dt: float) ->
   np.ndarray[int]:
  '''
  Adaptively picks the best way to estimate the results of the
     model. Returns an array containing the number of times each
     event
  occurs (ordered the same way as the given probabilities).

  *Note that the adaptive functionality is of limited value
     computationally and can severely undermine accuracy.
     Accordingly, it currently
  defaults to a binomial distribution for choosing the number of
     times each event happens.*

  ### Parameters
  - `ps`: The relative probabilities of each event, as a NumPy
     array.
  - `sum_Rs`: The net rate of all events.
  - `dt`: The size of the time step.
  '''
  Xs = 0*ps
  p_cond = 0
  rng = np.random.default_rng()
  N = int(sum_Rs*dt) # to save on time, the random variable this
     should technically be has been replaced with its avg value
  det_thres = 0.1
  for i in range(len(ps)):
    if p_cond >= 1 or N <= 0: break
    p = ps[i]/(1 - p_cond)
    if p > 1: p = 1
    if p <= 0: continue
    if False: # re-enable this if speed becomes particularly
       necessary again (it makes ensuring accuracy... annoying at
        best)
      if N > 200 and (p > det_thres and p < 1-det_thres): Xs[i] =
         int(N*p)         # Deterministic case
      elif N > 1000:
        if N*p < 25 or N*(1-p) < 25: Xs[i] = rng.poisson(lam=N*p)
              # Large-ish N, p close to 0 or 1
        else: Xs[i] = abs(int(rng.normal(loc=N*p, scale=N*p*(1-p))
          ))  # Large-ish N, p close to neither 0 nor 1
      else: Xs[i] = rng.binomial(n=N, p=p)                  # Small
          N
    else: Xs[i] = rng.binomial(n=N, p=p)
    N -= Xs[i]
    p_cond += ps[i]
  return Xs
```